Imperial College London
Department of Computing

Multi-version Execution for Increasing the Reliability and Availability of Updated Software

Petr Hosek

September 2015

Supervised by Dr Cristian Cadar

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of the Imperial College London, 5 September 2015

DECLARATION OF ORIGINALITY

This thesis is a presentation of my own original research and all the sources have been quoted and acknowledged by means of complete references.

COPYRIGHT DECLARATION

The copyright of this thesis rests with the author and is made available under a Creative Commons Attribution Non-Commercial No Derivatives licence. Researchers are free to copy, distribute or transmit the thesis on the condition that they attribute it, that they do not use it for commercial purposes and that they do not alter, transform or build upon it. For any reuse or redistribution, researchers must make clear to others the licence terms of this work.

COLOPHON

This document was typeset using LuaLaTeX, using a custom style based on classicthesis.[1] The bibliography was processed by Biblatex[2] and Biber.[3]

The default text font is Minion Pro, while the sans-serif text is typeset in Myriad Pro; monospaced text uses Bitstream Vera Mono.

[1] https://code.google.com/p/classicthesis/

[2] http://www.ctan.org/pkg/biblatex

[3] http://biblatex-biber.sourceforge.net/

Abstract

Software updates are an integral part of the software development and maintenance process, but unfortunately they present a high risk, as new releases often introduce new bugs and security vulnerabilities; as a consequence, many users refuse to upgrade their software, relying instead on outdated versions, which often leave them exposed to known software bugs and security vulnerabilities.

In this thesis we propose a novel *multi-version execution* approach, a variant of N-version execution, for improving the software update process. Whenever a new program update becomes available, instead of upgrading the software to the newest version, we run the new version in parallel with the old one, and carefully synchronise their execution to create a more reliable multi-version application.

We propose two different schemes for implementing the multi-version execution technique—via failure recovery and via transparent failover—and we describe two possible designs for implementing these schemes: Mx, focused on recovering from crashes caused by the faulty software updates; and Varan, focused on running a large number of versions in parallel with a minimal performance overhead.

Mx uses static binary analysis, system call interposition, lightweight checkpointing and runtime state manipulation to implement a novel fault recovery mechanism, which enables the recovery of the crashing version using the code of the other, non-crashing version. We have shown how Mx can be applied successfully to recover from real crashes in several real applications.

Varan combines selective binary rewriting with high-performance event streaming to significantly reduce performance overhead, without sacrificing the size of the trusted computing base, nor flexibility or ease of debugging. Our experimental evaluation has demonstrated that Varan can run C10k network servers with low performance overhead, and can be used in various scenarios such as transparent failover and live sanitisation.

To everyone who believed in me when others did not.

Acknowledgments

First and foremost, I would like to thank my supervisor Dr Cristian Cadar for all his support and advice throughout the past four years. He was an excellent advisor and I have thoroughly enjoyed our collaboration, and I have learned more than I could have ever imagined.
Second, I would like to thank Dr Susan Eisenbach for sharing her wisdom and experience. She was a great mentor, especially in the beginning of my studies, and she made me feel like a true part of our department.
I would not be able to pursue my studies without Google's financial support; I was a recipient of the Google Europe Fellowship in Software Engineering, and this research was supported by this Google Fellowship. I cannot thank them enough for their generosity.
Finally, I would like to thank Elena for all her love and support throughout my studies which made the last few years so much more enjoyable.

Contents

1 OVERVIEW 1

1.1 Solution . . . 2
1.2 Organisation . . . 4
1.3 List of publications . . . 4

2 MULTI-VERSION EXECUTION 6

2.1 Real-world scenarios . . . 7
2.1.1 *Vim* double free error on home path completion . . . 7
2.1.2 *Lighttpd 1.4* `mod_compress` segfault with disabled ETag support . . . 8
2.1.3 *Redis* crash on wrong `HMGET` type . . . 9
2.2 Rationale . . . 10
2.2.1 Synchronising the execution of different versions . . . 12
2.2.2 Surviving software errors . . . 14
2.3 Scope . . . 16
2.4 Summary . . . 17

3 SOFTWARE EVOLUTION IN REAL-WORLD 18

3.1 Study infrastructure . . . 18
3.2 Overview . . . 21
3.2.1 Basic characteristics . . . 22
3.2.2 Selection of revisions . . . 23
3.2.3 Revision setup . . . 24
3.3 Code and test evolution . . . 25
3.3.1 Patch characteristics . . . 27
3.4 Code coverage . . . 29
3.4.1 Coverage evolution . . . 31
3.4.2 Patch coverage . . . 33
3.4.3 Bug analysis . . . 36
3.5 External behaviour evolution . . . 37
3.5.1 Trace differences . . . 39
3.5.2 Manual analysis . . . 40
3.6 Threats to validity . . . 40
3.7 Discussion about empirical studies . . . 41

3.8 Summary . . . 42

4 MULTI-VERSION SOFTWARE UPDATES 43
4.1 Prototype . . . 43
4.1.1 System call interposition . . . 44
4.1.2 Runtime state manipulation . . . 47
4.1.3 Binary static analysis . . . 49
4.2 Evaluation . . . 50
4.2.1 Ability to run distant versions . . . 53
4.2.2 Performance overhead . . . 54
4.3 Discussion . . . 56
4.4 Summary . . . 56

5 EFFICIENT MULTI-VERSION EXECUTION 58
5.1 Overview . . . 58
5.1.1 System call interception . . . 58
5.1.2 Event-streaming architecture . . . 59
5.1.3 Rewrite rules for system call sequences . . . 60
5.2 Prototype . . . 61
5.2.1 Setup of address spaces and communication channels . . . 61
5.2.2 Selective binary rewriting . . . 63
5.2.3 Event streaming . . . 65
5.2.4 Rewrite rules for system call sequences . . . 68
5.3 Performance evaluation . . . 68
5.3.1 Microbenchmarks . . . 69
5.3.2 C10k servers . . . 70
5.3.3 Comparison with prior NVX systems . . . 73
5.4 Transparent failover . . . 76
5.4.1 Handling divergent behaviour . . . 77
5.5 Discussion . . . 79
5.6 Summary . . . 80

6 RELATED WORK 81
6.1 *N*-version programming . . . 81
6.1.1 Multi-variant execution . . . 82
6.1.2 Multi-version execution . . . 83
6.2 Software updates . . . 84
6.2.1 Dynamic software updates . . . 84
6.2.2 Update management and distribution . . . 85
6.3 Sandboxing . . . 86

6.3.1 System call interposition . . . 86
6.3.2 Software fault isolation . . . 87
6.4 Record & replay . . . 87
6.5 Software evolution . . . 89

7 OTHER APPLICATIONS AND FUTURE WORK 90
7.1 Regression testing . . . 90
7.2 Live sanitisation . . . 91
7.3 Record-replay . . . 92
7.4 Security honeypot . . . 92
7.5 Future work . . . 94

8 CONCLUSION 96

References 97

List of Tables

Table 3.1 Summary of applications used in our study. *ELOC* represents the number of executable lines of code and *TLOC* the number of lines in test files in the last revision analysed. . . . 22

Table 3.2 Revisions used in our study. *OK*: code compiles and tests complete successfully, *TF*: some tests fail, *TO*: tests time out, *CF*: compilation fails, *Time*: the number of months analysed. . . . 24

Table 3.3 The median number of executable lines, hunks from executable files, and executable files in a patch. Only data from patches which add or modify executable code is considered. . . . 30

Table 3.4 Number of revisions where the test suite non-deterministically succeeds/fails, and the maximum, median and average number of lines which are non-deterministically executed in a revision. . . . 30

Table 3.5 Overall patch coverage bucketed by the size of the patch in ELOC. NP is the number of patches in the bucket and C is their overall coverage. Only patches which add or modify executable code are considered. . . . 35

Table 3.6 Overall latent patch coverage: the fraction of the lines of code in all patches that are only executed by the regression suite in the next 1, 5 or 10 revisions. The overall patch coverage is listed for comparison. . . . 35

Table 3.7 The median coverage and the number of revisions achieving 100% coverage for the revisions containing bug fixes. The overall metrics are included for comparison. . . . 36

Table 4.1 Utilities from *GNU Coreutils*, the crash bugs used, and the versions in which these bugs were introduced and fixed. We group together utilities affected by the same or similar bugs. . . . 50

Table 4.2 The maximum distance in number of revisions, and the time span between the revisions that can be run by Mx for each bug. . . . 53

Table 5.1 C10k server applications used in the evaluation. . . . 71

Table 5.2 Comparison with Mx, *Orchestra* and *Tachyon* on the benchmarks used to evaluate these systems. . . . 73

Table 6.1 Taxonomy of related work which summarises the existing approaches and compares them with respect to each other. . . . 81

List of Figures

Figure 2.1 Two schemes of multi-version execution presented in the thesis. 11
Figure 2.2 A multi-version execution platform running conventional and multi-version applications side by side. 12
Figure 2.3 Three solutions for synchronising two divergent versions of the same application . 15
Figure 2.4 Lockstep execution of two version synchronized by the monitor. 16
Figure 3.1 The architecture of the infrastructure used in the empirical study. 19
Figure 3.2 Evolution of executable lines of code. 25
Figure 3.3 Evolution of textual lines of test code. 26
Figure 3.4 Co-evolution of executable and test code. Each increment represents a change. . . . 27
Figure 3.5 Breakdown of patches by type: affecting executable application code but not test code, affecting both, affecting only test code, and neither. 28
Figure 3.6 Evolution of the overall line and branch coverage. 32
Figure 3.7 Patch coverage distribution. Each colour represents a range of coverage values with the bar size indicating the percentage of patches whose coverage lies in the respective range. 34
Figure 3.8 Co-evolution of executable code and external behaviour. Each increment represents a change. 39
Figure 3.9 Correlation of differences in post-processed system call traces with differences in source code. The named versions are the only ones introducing external behaviour changes. 39
Figure 4.1 Mx system architecture. 44
Figure 4.2 Rem's recovery mechanism uses the code of the non-crashing version to run through the buggy code. 48
Figure 4.3 Normalised execution times for the *SPEC CPU2006* benchmark suite running under Mx. 54
Figure 5.1 The event-streaming architecture of Varan. 60
Figure 5.2 Setup of address spaces and communication channels. 61
Figure 5.3 Event delivery in a multi-threaded NVX program, with the ordering of events enforced using logical clocks. 67
Figure 5.4 System call microbenchmarks. 70

Figure 5.5 Performance overhead for the *Beanstalkd*, *Lighttpd 1.4*, *Memcached*, *Nginx* and *Redis* servers for different number of followers. The client and server are running on the same machine, simulating a worst-case scenario. 71
Figure 5.6 Performance overhead for individual *Redis* operations for different number of followers. The client and server are running on the same machine, simulating a worst-case scenario. 72
Figure 5.7 Performance overhead for the *Apache httpd*, *thttpd*, and *Lighttpd 1.4* servers for different numbers of followers to allow for comparison with existing systems. 74
Figure 5.8 *SPEC CPU2000* performance overhead for different numbers of followers. 75
Figure 5.9 *SPEC CPU2006* performance overhead for different numbers of followers. 75
Figure 5.10 Average number of system calls per second during the execution of the SPEC CPU2006 benchmark suite, measured using the strace tool. 76
Figure 7.1 The use of event-streaming architecture for live sanitisation. 91
Figure 7.2 The architecture of the multi-variant honeypot system. 94

List of Listings

2.1 First (incorrect) fix of `ExpandOne` function in *Vim*. 8
2.2 Second (correct) fix `ExpandOne` function in *Vim*. 8
2.3 Original (correct) version of the `hmgetCommand` function in *Redis*. 10
2.4 Refactored (buggy) version of the `hmgetCommand` function in *Redis*. 10
3.1 Example of an assertion macro used in ØMQ codebase. 33
5.1 *Lighttpd 1.4* SUID bit detection patch. 77
5.2 Example of a BPF rewriting rule. 78

1

Overview

With the growing size of modern computer software systems, the number of errors present in such systems has increased. At the same time, increasing reliance on computers and computer software systems in every aspect of human life implies a strong need for reliable software systems.

Software updates are an integral part of the software maintenance process, but unfortunately present a high risk, with many users and administrators refusing to upgrade their software and relying instead on outdated versions, which often leaves them and their systems exposed to software bugs and security vulnerabilities. For example, a recent survey of 50 system administrators has reported that 70% of respondents refrain from performing a software upgrade, regardless of their experience level [Cra+07]. The Hearbleed bug [Cod14] illustrates this behaviour. A fixed version of OpenSSL was released on the same day the Heartbleed bug was publicly disclosed. However, a month later, 1.5% (12,043) of the 800,000 most popular sites in the world were still using the vulnerable version of OpenSSL, making their servers susceptible to attacks [Ley14].

One of the main reasons for which users hesitate to install updates is that a significant number of them result in failures. It is only too easy to find examples of updates that fix a bug or a security vulnerability only to introduce another problem in the code. For example, a recent study of software updates in commercial and open-source operating systems has shown that at least 14.8% to 25% of fixes are incorrect and have affected end-users [Yin+11]. A new bugs and security issues introduced by software updates may cause disruption and system outages. The incident caused by the air traffic control failure in December 2014, which disrupted hundreds of flights to and from London airports, is a prime example. According to the interim report published by the official enquiry panel [Pan15], this incident was caused by a system update in November 2014, which changed the number of available "Atomic Functions". Due to a latent defect, present in the system since its original development in 1990, this update caused the failure of the entire system following a specific action of the air traffic control, despite existing safety mechanisms.

Another example of such disruption is the recent Verizon outage [Str14] of the billing and activation system, affecting a large number of users which were unable to log into their accounts. After the incident, Verizon issued the following statement:

> The Verizon Wireless billing system was fully restored early today, shortly after midnight. The issue affecting some customer access to account information was an unintended

consequence of a software update performed by the company on its billing system two days ago.

The outage, which according to reports lasted for 48 hours, was so significant that it generated a lot of negative reactions from frustrated customers including its own #verizonfail and #verizonoutage Twitter hashtags. This as well as other similar incidents show the possible consequences of failed software updates.

Nowadays, many services including large-scale ones such as Google, Facebook or Amazon employ *continuous deployment*, whereby new versions are continuously released to users [Joh09]. Multiple versions can be released throughout the day, and A/B testing is often used to prevent complete outage in case of an error caused by one of the versions by making each version accessible only to a fraction of users. While this approach helps minimise the number of users affected by newly introduced bugs, it is not foolproof—bugs introduced by new releases may manifest themselves only in particular cases or following prolonged operation, after a single version has been deployed to the entire user base, as evident from past cases affecting services such as Blogger [Kes11] or Facebook [Joh10]. Failures and outages affecting the majority of the user-base therefore occur even in these applications. Such outages might cause significant financial losses and frustration among users, and it is therefore critical to avoid them.

Our goal is to improve the software update process in such a way as to encourage users to upgrade to the latest software version, without sacrificing the stability or availability of the older version, by taking advantage of the idle resources made available by multi-core platforms.

1.1 SOLUTION

We tackle the problem of faulty software updates using a simple but effective *multi-version execution* based approach. Whenever a new update becomes available, instead of upgrading the software to the new version, we run the new version in parallel with the old one; by carefully coordinating their executions and selecting the behaviour of the more reliable version when they diverge, we create a more secure and dependable multi-version application.

The goal of multi-version execution is to increase availability and reliability of updated software by exploiting the abundance of resources (*e.g.*, idle processor time) made available by modern highly parallel platforms. We believe that multi-version execution updates is a timely solution in the context of today's computing platforms [CPW10]. The last decade has seen the emergence of new hardware platforms, ranging from multi-core processors to large-scale data centres, which provide an abundance of computational resources and a high degree of parallelism. These platforms already benefit applications with a great amount of inherent concurrency. However, there has been relatively little progress in exploiting this abundance of resources to improve software reliability and availability, especially in the context of software updates.

Typical data centres are designed for peak load to ensure service-level agreements are met. Servers are rarely completely idle but they do not operate near their maximum utilisation; most of the time, they operate at between 10% and 50% of their maximum utilisation levels. However, even an energy-efficient server consumes half its full power when doing virtually no work—and for normal servers, this ratio is much worse [BH07]. Therefore, instead of dynamically switching off unused servers which is not effective, we aim to use the abundance of these resources (*i.e.* processing power) to increase software availability and reliability.

In this thesis, we present two different schemes for implementing this multi-version execution technique. The first is a *failure recovery scheme*, which allows program to survive errors introduced by software updates. The mechanism is completely autonomous and operates on the off-the-shelf-binaries making it easy to use and deploy. This scheme is limited to two versions run in parallel. The second is a *transparent failover scheme*, which allows N versions to be run in parallel; if the primary version fails, the mechanism transparently switches to another version without any disruption in service. This scheme is designed to have minimal performance overhead enabling the use of a large number of versions, but offers limited guarantees in case of a failure.

In order for our approach to be successful, the external behaviour of the versions that are run in parallel has to be similar enough to allow us to synchronise their execution. In practice, it means that the external behaviour of all the versions has to be the same. The empirical study presented in this thesis shows that changes to the external behaviour of an application are often minimal, so our approach should work well with versions that are not too distant from one another (*i.e.* several revisions apart).

Similarly, our system relies on the assumption that versions re-converge to the same behaviour after a divergence. Therefore, we believe multi-version execution would be a good fit for applications that perform a series of mostly independent requests, such as network servers. These applications are usually structured around a main dispatch loop, which provides a useful re-convergence point. Our approach is also suitable to local code changes, which have small propagation distances, thus ensuring that the different versions will eventually re-converge to the same behaviour.

Our approach is targeted toward scenarios where the availability and reliability of a software system is more important than strict correctness, high performance and low energy consumption. In terms of correctness guarantees, it is similar to previous approaches such as failure oblivious computing [Rin+04] which may sacrifice strict correctness for increased availability and reliability. We aim to alleviate potential problems by regularly checking if all versions exhibit the same external behaviour and by reverting to running a single version when a non-resolvable divergence is detected making this a best-effort approach.

There are three main challenges that we need to address to make this approach viable. First, we need to implement mechanisms for effectively coordinating the parallel execution of multiple program versions. Second, when executions diverge, we need to select the output of the more reliable one. Finally, we would like to be able to scale the number of program versions run in parallel in order to balance conflicting requirements such as performance, reliability, and energy consumption.

1.2 ORGANISATION

The rest of the thesis is organised as follows:

Chapter 2 gives an overview of the N-version, multi-variant and multi-version execution techniques and sets the background for the rest of the thesis. The chapter also presents several real-world scenarios which are used throughout the thesis as a part of the evaluation.

Chapter 3 presents a study of software evolution in several real-world open-source systems and acts as a motivation for many of the design choices made in the systems presented in this thesis.

Chapter 4 presents Mx, a multi-version execution runtime focused on fail-recovery from crashes caused by bugs introduced in software updates. The chapter gives a detailed overview of the fail-recovery mechanism and explains how Mx survives bugs in several existing open-source applications.

Chapter 5 describes Varan, an efficient N-version monitor designed to run a large number of versions in parallel. The chapter presents the high-level architecture as well as details of the prototype implementation, and includes a thorough evaluation including a comparison with previous systems.

Chapter 6 discusses work related to N-version and multi-version execution in the reliability and availability context.

Chapter 7 introduces other possible applications of multi-version and N-version execution techniques including possible future work, such as live sanitization, record and replay and security honeypots.

Finally, the thesis concludes with Chapter 8, which summarizes the contributions and raises additional research questions.

1.3 LIST OF PUBLICATIONS

The text of this thesis is based on, and borrows heavily from the work and the writing in the following papers:

- Cristian Cadar and Petr Hosek. "Multi-Version Software Updates". In: *Proceedings of the 4th Workshop on Hot Topics in Software Upgrades (HotSWUp'12)*. Zurich, Switzerland, June 2012
- Petr Hosek and Cristian Cadar. "Safe Software Updates via Multi-version Execution". In: *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*. San Francisco, CA, USA, May 2013
- Paul Dan Marinescu, Petr Hosek, and Cristian Cadar. "Covrig: A Framework for the Analysis of Code, Test, an Coverage Evolution in Real Software". In: *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'14)*. San Jose, CA, USA, July 2014

- Petr Hosek and Cristian Cadar. "Varan the Unbelievable: An efficient N-version execution framework". In: *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. Istanbul, Turkey, Mar. 2015

2

Multi-version Execution

An indisputable and widely accepted fact is that all software contains bugs. These bugs cause software failures which can lead to anything from minor discomfort and loss of service to major disasters including loss of lives. Software reliability is the probability of failure-free operation for a specified period of time in a specified environment.

There are two basic approaches to increase software reliability. One is fault avoidance, using formal specification and verification methods, and a rigorous software development process. Another approach is fault tolerance, through replication, redundancy and diversity. One of the popular software fault tolerance methods that uses diversity is *N*-version programming [Avi95].

The idea behind *N*-version programming (NVP) is to have independent teams design and implement the same functionality multiple times from the same specification as different program versions. These versions are then run in parallel using an *N*-version execution environment (NVX) and their results are voted on and the majority of the outputs are selected.

NVP depends on the assumption that software errors in different versions are independent. Otherwise, these versions are not effective at detecting errors as the different versions are likely to contain the same bugs. The experiment done by Knight and Leveson [KL86; KL90] showed that the assumption of independence does not hold and independently developed versions had the same or similar software faults 50% of the time since programmers tended to commit certain classes of mistakes independently.

The problem of NVP is that even a small probability of correlated bugs in different versions significantly reduces the potential reliability improvement, which makes NVP less cost-effective compared to other fault-tolerance techniques due to the need for developing *N* different versions. Sha [Sha01] compared different fault tolerance approaches and suggested that, assuming a finite software development cycle, it is better to invest resources into a single reliable implementation rather than three different implementations which will be run in parallel.

The issue of cost related to the development of multiple independent versions has been partially addressed by the multi-variant program execution research, which can be seen as an evolution of *N*-version programming. Rather than developing different versions of the same application manually, these are generated from a single version using different techniques such as special source code annotations [TXZ09; TG10], code transformations based on the modification of the abstract syntax tree [Sch+14; BAM14], custom compilers producing different binary layouts for different variants [Sal+09; Fra10].

We could also run different programs where one program acts as a specification and other implementations are checked against the specification at runtime. This is the basic idea behind runtime monitoring and checking [Kimo1; Kim+01]. In runtime monitoring and checking, one program is the implementation while another is the specification. The implementation is checked against the specification at runtime, which is easier than proving the equivalence statically (or formally). Furthermore, the specification can be less efficient and thus simple. The example shown by Lui Sha in [Shao1] uses Bubblesort as a runtime specification for Quicksort.

One particularly important class of bugs are bugs introduced by software updates. These bugs often affect existing functionality, degrading the reliability, and potentially also security, of the software. This leads to frustration as users are no longer able to use the functionality they have been able to use in the past; as a consequence, users often refuse applying updates to their software and rely instead on outdated versions, which may contain bugs and security vulnerabilities.

To address this problem, we propose a variant of NVX focused on improving reliability and availability of evolving software: *multi-version execution*. While NVX and multi-version execution techniques have many similarities, there are several key differences. First and foremost, in NVX the different versions are developed by different teams while multi-version execution uses the subsequent revisions, which are the product of natural software evolution. Second, the goal of NVX is to increase the reliability of software by running independently developed, functionally equivalent software variants, whereas multi-version execution assumes that the variants are largely the same except for the parts affected by the software update.

While designed primarily in the context of software updates, the multi-version execution monitors and environments can benefit NVX too. Furthermore, as shown in this thesis, multi-version execution runtimes can be used in other reliability scenarios, such as running expensive error detectors ("sanitisers") during deployment or implementing security honeypots.

2.1 REAL-WORLD SCENARIOS

To motivate the multi-version execution technique, we present three scenarios involving the *Vim* editor,*Lighttpd 1.4* and *Redis* servers. These correspond to two categories of applications that could benefit from our approach: user applications such as text editors for which reliability is a key concern; and network servers, with stringent security and availability requirements.

2.1.1 Vim *double free error on home path completion*

Vim[1] is arguably one of the most popular text editors. While trying to fix a memory leak, *Vim* developers introduced a double `free` bug that caused *Vim* to crash whenever the user tried to use a path completion feature. This bug was introduced in version 7.1.127, released in September 2007. The incorrectly fixed

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

version of the ExpandOne function is shown in Listing 2.1. The newly introduced code frees up the memory used to store the expanded path. Unfortunately, when the path contains the ~ symbol used as an alias for user's home directory, this code path will be invoked twice leading to a crash.

```
@@ -3543,0 +3548,4 @@ ExpandOne(xp, str, orig, options, mode)
+ /* Free "orig" if it wasn't stored in "orig_save". */
+ if (orig != orig_save)
+ vim_free(orig);
+
```

Listing 2.1: First (incorrect) fix of ExpandOne function in *Vim*.

The double free bug was detected and fixed in version 7.1.147, released one month later in October 2007. The correct version of the ExpandOne function is shown in Listing 2.2. In this version, the orig_saved variable type was changed to **int** and is being used as a flag preventing the second free invocation.

```
@@ -3355,0 +3356 @@ ExpandOne(xp, str, orig, options, mode)
+ int orig_saved = FALSE;
@@ -3423,0 +3425 @@ ExpandOne(xp, str, orig, options, mode)
+ orig_saved = TRUE;
@@ -3549 +3551 @@ ExpandOne(xp, str, orig, options, mode)
- if (orig != orig_save)
+ if (!orig_saved)
        vim_free(orig);
```

Listing 2.2: Second (correct) fix ExpandOne function in *Vim*.

Despite the bug being fixed promptly, it made its way into Ubuntu 8.04, affecting a large number of users.[2] Since the 8.04 is a long-term support version, the Ubuntu maintainers avoid incorporating new changes except for critical security bug fixes. This meant that the *Vim* users affected by this bug had to build and install their own version of *Vim* from the source.

2.1.2 *Lighttpd 1.4* mod_compress *segfault with disabled ETag support*

Lighttpd 1.4[3] is a popular open-source web-server used either alone or in conjunction with other web-servers by several high-traffic websites such as Wikipedia and Xkcd. Despite its popularity, crash bugs are still a common occurrence in *Lighttpd 1.4*, as evident from its bug tracking database.[4] Below we discuss one such bug.

In April 2009, a patch was applied[5] to *Lighttpd 1.4*'s code related to the HTTP ETag functionality. An ETag is a unique string assigned by a web server to a specific version of a web resource, which can be used

[2] https://bugs.launchpad.net/ubuntu/+source/vim/+bug/219546
[3] http://www.lighttpd.net/
[4] http://redmine.lighttpd.net/issues/
[5] http://redmine.lighttpd.net/projects/lighttpd/repository/revisions/2438

to quickly determine if the resource has changed. The patch was a one-line change, which discarded the terminating zero when computing a hash representing the ETag. More exactly, line 47 in `etag.c`:

```
for (h=0, i=0; i < etag->used; ++i) h = (h<<5)^(h>>27)^(etag->ptr[i]);
```

was changed to:

```
for (h=0, i=0; i < etag->used-1; ++i) h = (h<<5)^(h>>27)^(etag->ptr[i]);
```

This correctly changed the way ETags are computed, but unfortunately, it broke the support for compression, whose implementation depended on the previous computation. More precisely, *Lighttpd 1.4*'s support for HTTP compression uses caching to avoid re-compressing files which have not changed since the last access. To determine whether the cached file is still valid, *Lighttpd 1.4* internally uses ETags. Unfortunately, the code implementing HTTP compression did not consider the case when ETags are disabled. In this case, `etags->used` is `0`, and when the line above is executed, `etag->used-1` underflows to a very large value, and the code crashes while accessing `etag->ptr[i]`. Interestingly enough, the original code was still buggy (it always returns zero as the hash value, and thus it would never re-compress the files), but it was not vulnerable to a crash.

The segfault was diagnosed and reported in March 2010[6] and fixed at the end of April 2010,[7] more than one year after it was introduced.

2.1.3 *Redis crash on wrong* `HMGET` *type*

Redis is an advanced key-value data structure server,[8] often referred to as one the most popular NoSQL databases. Due to its performance and low-resource requirements, *Redis* is being used by many well-known services such as GitHub. Because the entire dataset is held in memory, reliability is critically important, as the server crash and subsequent restart requires reloading data into memory, which can lead to a prolonged server downtime. Nevertheless, like any other large software system, *Redis* is subject to crash bugs. Issue 344[9] is one such example. This issue causes *Redis* to crash when the `HMGET` command is used with the wrong type. The bug was introduced during a code refactoring applied in revision `7fb16bac`. The original code of the problematic `hmgetCommand` function is shown in Listing 2.3, while the (buggy) refactored version is shown in Listing 2.4.

In the original code, if the lookup on line 1 is successful, but the type is not `REDIS_HASH` (line 9), the function returns after reporting an incorrect type (lines 10–11). However, in the refactored version (Listing 2.4), the **`return`** statement is missing, and after reporting an incorrect type (line 4), the function continues execution and causes segmentation fault inside the `hashGet` function invoked on line 8. This is a critical bug, which may negatively affect the service availability and reliability as described above.

[6] `http://redmine.lighttpd.net/issues/2169`
[7] `http://redmine.lighttpd.net/projects/lighttpd/repository/revisions/2723`
[8] `http://redis.io/`
[9] `http://code.google.com/p/redis/issues/detail?id=344`

```
 1 robj *o = lookupKeyRead(c->db, c->argv[1]);
 2 if (o == NULL) {
 3   addReplySds(c,sdscatprintf(sdsempty(),"*%d\r\n",c->argc-2));
 4   for (i = 2; i < c->argc; i++) {
 5     addReply(c,shared.nullbulk);
 6   }
 7   return;
 8 } else {
 9   if (o->type != REDIS_HASH) {
10     addReply(c,shared.wrongtypeerr);
11     return;
12   }
13 }
14 addReplySds(c,sdscatprintf(sdsempty(),"*%d\r\n",c->argc-2));
```

Listing 2.3: Original (correct) version of the `hmgetCommand` function in *Redis*.

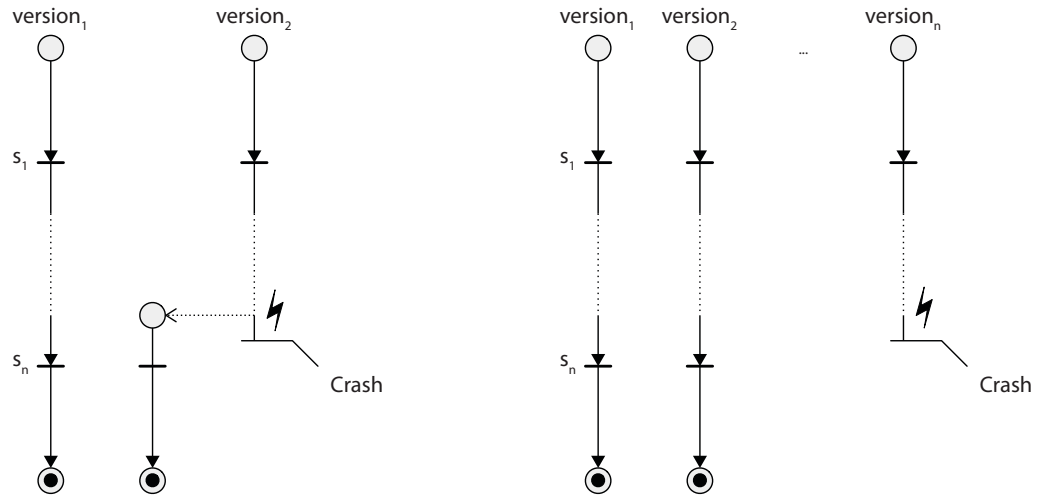
```
 1 robj *o, *value;
 2 o = lookupKeyRead(c->db,c->argv[1]);
 3 if (o != NULL && o->type != REDIS_HASH) {
 4   addReply(c,shared.wrongtypeerr);
 5 }
 6 addReplySds(c,sdscatprintf(sdsempty(),"*%d\r\n",c->argc-2));
 7 for (i = 2; i < c->argc; i++) {
 8   if (o != NULL && (value = hashGet(o,c->argv[i])) != NULL) {
 9     addReplyBulk(c,value);
10     decrRefCount(value);
11   } else {
12     addReply(c,shared.nullbulk);
13   }
14 }
```

Listing 2.4: Refactored (buggy) version of the `hmgetCommand` function in *Redis*.

The bug was introduced in April 2010, diagnosed and reported only half a year later in October 2010 and then fixed after fifteen days. As in the case of the *Lighttpd 1.4* bug, this means that for several months, any *Redis* instance was susceptible to a crash and vulnerable to a possible attack.

2.2 RATIONALE

All the scenarios presented above describe software updates which, while trying to fix existing bugs or refactor the code, also introduced new bugs that caused the code to crash under certain conditions. The bottom line for all these scenarios is that for significant amount of time, *i.e.* about one year in the case of *Lighttpd 1.4*, six months in the case of *Redis*, and one month in the case *Vim*, users affected by buggy patches essentially had to decide between (1) incorporating other security and bug fixes added to the code, but being vulnerable to these crash bugs, and (2) giving up on these security and bug fixes and using an old version of *Lighttpd 1.4* or *Redis*, which are not vulnerable to these newly introduced bugs.



(a) Failure recovery scheme.

(b) Transparent failover scheme.

Figure 2.1: Two schemes of multi-version execution presented in the thesis.

Note that this is particularly true for the period between the time when the bug was introduced and the time it was diagnosed (*i.e.* eleven months in case of *Lighttpd 1.4*, six months in case of *Redis*), since during this period most users would not know how to change the server's configuration to avoid the crash.

Our approach aims to provide users with a third choice: when a new version becomes available, instead of replacing the old version, we will run both versions in parallel; as new versions arrive, we continue to execute them in parallel with the existing ones. Then, by selecting the output of the more reliable version when their executions diverge, we can increase the overall reliability of the software. There are multiple ways to implement this approach. In this thesis, we propose two different schemes:

FAILURE RECOVERY This scheme, depicted in Figure 2.1a, uses two versions run in parallel, and allows the program to survive certain classes of errors introduced by software patches in either of the two versions, as long as the patched regions do not overlap. The trade-off is a large perfomance penalty, making this scheme more suitable for applications with less stringent performance requirements, such interactive applications.

TRANSPARENT FAILOVER This scheme, shown in Figure 2.1b, uses N versions run in parallel, with one of the versions designated as a leader and the others as followers. When one of the versions fails, we transparently discard it, and if necessary re-elect the new leader, without any disruption in service. This scheme cannot recover crashing versions, but offers minimal performance overhead enabling the use of a large number of versions, making it particularly suitable for high-performance servers with strict reliability requirements.

The goal of both schemes is to run multiple versions of an application in parallel, and to synchronise their executions so that (1) users are given the illusion that they interact with a single version of the application, (2) the multi-version application is at least as reliable and secure as one of the individual versions in isolation, and (3) the synchronisation mechanism incurs a reasonable performance overhead.

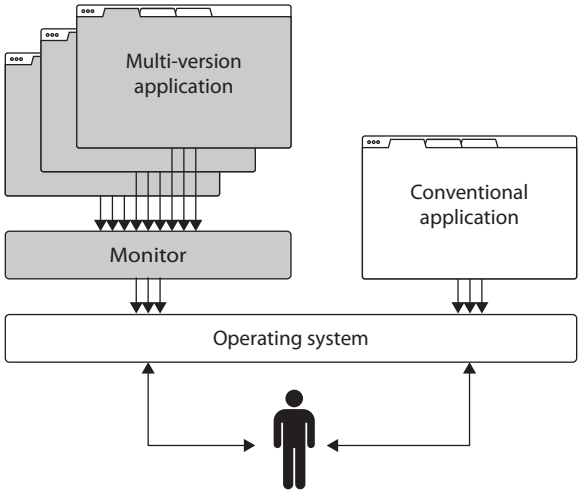


Figure 2.2: A multi-version execution platform running conventional and multi-version applications side by side.

While the first goal applies equally to both schemes, there are differences with respect to the second and third goal. For the failure recovery scheme, we sacrifice some of the performance being more lenient with respect to the third goal, but we can make the second goal more stringent requiring the multi-version application to be as reliable as *any* of the two versions ran in parallel. The transparent failover scheme has minimal performance overhead, but offers limited guarantees in case of failure—we automatically discard failing versions, which means that we can only continue execution as long as there is at least one version alive.

2.2.1 *Synchronising the execution of different versions*

To satisfy the first goal, both schemes require a form of virtualization framework, typically called an *execution monitor*, that would coordinate the execution of multiple application versions, and mediate their interaction with the external environment, as shown in Figure 2.2. Various such frameworks have been designed in the past in the context of running multiple automatically generated variants of a given application in parallel [BZ06; Cox+06; Sal+09], and some of the techniques proposed in prior work can be reused in our context. To be practical, this coordination mechanism has to incur a reasonable overhead on top of native execution and ensure that the overall system is able to scale up and down the number of software versions run in parallel in order to balance conflicting requirements such as performance, reliability, security and energy consumption.

This coordination mechanism can be implemented at multiple levels of abstraction. The simplest, but least flexible approach is to synchronise the different versions at the level of application inputs and outputs. This is particularly suitable for applications that use well-defined protocols such as web servers and web browsers. The main advantage of this option is that it can tolerate large differences between

different software versions (in fact, one could even run different implementations, as in [XDK12]), but the main downside is that it is not applicable in the general case, when the input-output specification is not available.

A more flexible approach, which we adopted, is to synchronise the different versions at the level of system calls. System calls represent the *external behaviour* of an application, and they are the primary way in which application interact with the surrounding environment (the other being shared memory and side channels). While this option allows fewer changes between different application versions, it does not require any a priori knowledge of the application's input/output behaviour, and is particularly suitable in the context of software updates, where the external application behaviour usually remains unchanged. In fact, an empirical study presented in Chapter 3, which we conducted on several real applications, has shown that while the source code of these applications changes on a regular basis, the external behaviour often remains stable when security and bug fixes are introduced.

There are several different ways to implement an effective system call monitor, each with its own set of trade-offs. Solutions targeting binaries are significantly easier to use, but compared to solutions operating at source code level, they give up some of the precision while typically requiring more engineering effort. The most common ways for implementing monitors operating on binaries are dynamic linking, ptrace, kernel extensions and binary instrumentation, while solutions targeting source code typically rely on source code instrumentation.

System calls are rarely performed directly by the application, which rather use the wrapper functions provided by the C library. By linking to a custom library, which provides a custom version of these functions, we could intercept the system calls performed by the application. This could be done either statically at link time [Sea], or dynamically at runtime [LSR14] (*e.g.*, using `LD_PRELOAD` or `DYLD_INSERT_LIBRARIES` environment variables). The major benefit of this approach is efficiency and relative ease of implementation. However, the mechanism could be easily bypassed (*e.g.*, by invoking system calls directly); also the C library ABI has a significantly larger surface compared to the system call interface, requiring more engineering effort.

ptrace is an interface provided by most UNIX-like operating systems, including Linux, providing the means by which a process might observe and control another process. The primary use of ptrace interface is for debugging, but the ease of use makes ptrace a popular choice for implementing system call monitors [Gol+96; Sal+09; MB12]. ptrace-based solutions require relatively low engineering effort, they are easy to deploy and fairly flexible which makes them especially suitable for rapid prototyping [Spi+07].

The use of ptrace has some drawbacks: a significant performance overhead due to the large number of context switches, problematic support for multi-threaded applications, and the lack of a filtering mechanism that would allow interception only of system calls of interest. ptrace-based solutions are also more difficult to debug as the use of ptrace disallows the use of ptrace-based debuggers such as *GDB*. The performance overhead could be partially improved by the use of more efficient mechanism for copying memory from/to the monitored process, such as shared memory [Sal+09], or cross memory attach (§4.1.1). The lack of a filtering mechanism could be addressed by combining ptrace with the seccomp/bpf mechanism [KZ13].

An alternative to ptrace is to implement the system call monitor entirely [Pro02; Cox+06] or partially in kernel space [GPR04]. While this approach has numerous advantages compared to other approaches, such as minimal performance overhead and direct access to the application's execution context and address space, there are several drawbacks. First, this approach requires kernel patches and/or new kernel modules which complicates the development, limits portability across different operating systems or even different kernel versions, and hinders maintainability. Second, the part of the monitor that resides in the kernel (*i.e.* kernel module) must be run in privileged mode, which means that bugs in the implementation may compromise the system stability. Furthermore, it also makes it difficult for regular users to deploy and use such monitors.

Binary rewriting techniques allow transforming the executable (either statically or dynamically) altering its functionality; as such, they can be used to implement system call monitors by rewriting all system call instructions into control flow transfer instructions (*e.g.*, `JMP` instruction). Existing monitors were built either on top of existing binary rewriting systems [TXZ09], or using a purpose-built binary translation mechanism [FC08]. The advantage of binary rewriting-based monitors is relatively low performance overhead, especially in the case of a purposely built rewriting mechanism. The main disadvantage is the complexity of the implementation which requires significant development effort.

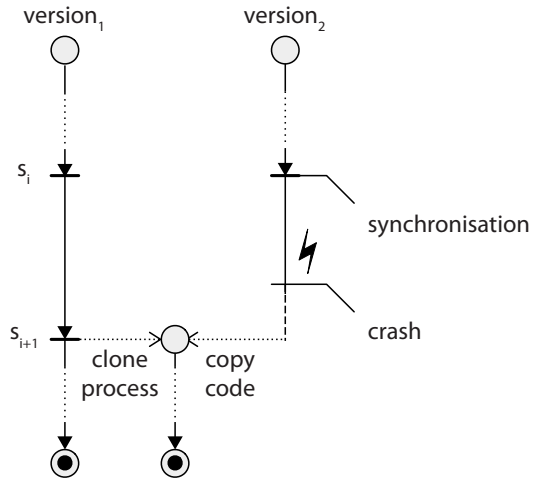
2.2.2 *Surviving software errors*

One particular challenge for our approach is to detect any divergences between different software versions, and resolve them in such a way as to increase the overall reliability of the application. Selecting the "correct" behaviour of an application when different versions disagree is not possible in the general case without having access to a high-level specification. However, one could (1) focus on universal correctness properties, such as the absence of crashes, and (2) use various heuristics such as majority voting and favoring the latest application versions. Our approach is to resolve a divergence by always using the behaviour of the version that has not crashed, and favour the behaviour of the latest version in all other cases.

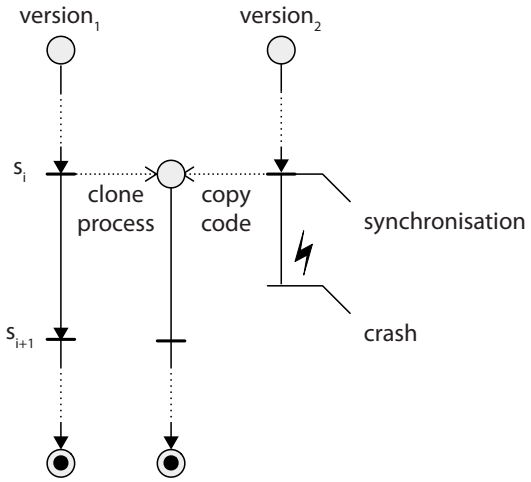
While this approach is sufficient for the transparent failover scheme, the failure recovery scheme requires a mechanism for surviving errors in one of the two versions. Numerous fault recovery techniques have been designed and described in prior work [Qin+05; Vee+11; Rin+04; SK06; Cos+05; Per+09; Can+04; LSR14], but none of these techniques focuses explicitly on faults caused by bugs introduced by software updates.

We assume that the only difference between the two versions are the changes introduced by the patch, and that the patch does not modify the internal application state in a way which would lead to a different behaviour in each version. We define the internal state of the application as data structures managed by the application and stored on the stack or heap.

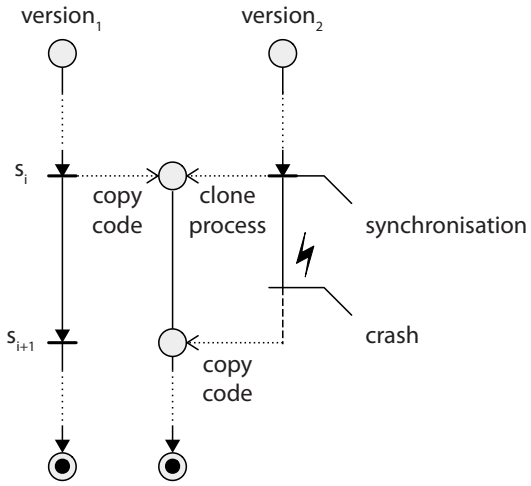
Our solution is based upon the observation that errors in programs are usually located at particular places (*i.e.* specific instructions) in the program's code. Therefore, when one of the versions crashes, we can use the code of the other, non-crashing version to execute over this critical point in that version's



(a) Patch the text after the crash and continue execution



(b) Patch state before the crash and continue execution



(c) Use the code of older version only to run through critical section

Figure 2.3: Three solutions for synchronising two divergent versions of the same application

code. This approach may not work when the memory layout of the two versions differs, as the code of one version may fail to locate the memory structures necessary for its execution in the memory of the other version. Nevertheless, the described approach still works in many cases when the memory layout of the two versions does not differ significantly (§4.2).

In our failure recovery approach, we run the two versions in parallel, monitoring their execution and taking frequent checkpoints at every synchronisation point, which in our case are system calls. When either of the versions fails, there are three possible solutions to synchronise the divergent versions:

1. Clone the correctly executing version to duplicate its state (*e.g.*, memory continents, memory mappings) after the crash and replace its code with the code of the failed version. Then restart both versions and continue their execution (Figure 2.3a).

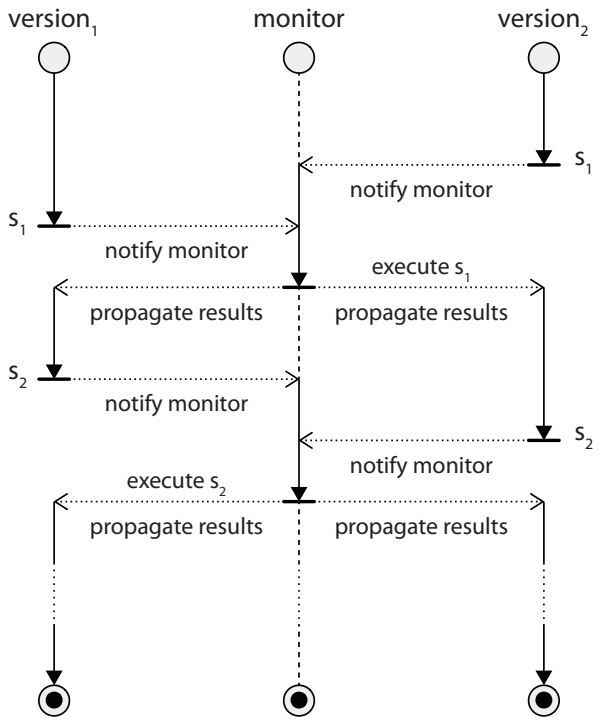


Figure 2.4: Lockstep execution of two version synchronized by the monitor.

2. Clone the correctly executing version to duplicate its state at the last synchronisation point (*i.e.* creating checkpoint). After the crash, replace the code of this cloned version with the code of the failed version. Then restart this cloned version, execute over the *critical section*, and continue execution of both versions (Figure 2.3b).

3. Clone the failing version to duplicate its state at the last synchronisation point. After the crash, replace the code of this cloned version with the code of the correctly executing version. Then restart this cloned version; after the application successfully executes over the *critical section*, replace the code of the cloned version again with the original code, and continue its execution (Figure 2.3c).

While all three solutions can recover the crashing version, in our implementation we use the third solution as it has the key property of fully recovering both the code and the state of the crashing version. This ensures that applications can survive crashes that occur at different points in different versions, but adds the extra challenge of restarting crashed versions. Despite this challenge, we believe this key property is crucial for the practical application of the multi-version execution technique.

2.3 SCOPE

We are restricting our approach to small changes, introduced by software updates, primarily bug and security fixes. The assumption is that the differences between individual versions run in parallel are

small. Since our technique is a runtime technique, we care about differences in behaviour rather than source code. These are not always correlated, and as shown in Chapter 3, the changes to the external behaviour are less common.

We assume that the sequence of system calls performed by all versions, *i.e.* their external behaviour, is the same (although we can relax this assumption, as shown in Section 5.2.4). Given that, we can run all versions in lockstep as shown in Figure 2.4. The monitor runs all versions in parallel. When any of the versions performs a system call, it notifies the monitor, which then waits for other versions to perform the system call as well and checks that all versions performed the same system call by comparing the values of all arguments.

This approach is used by Mx, a prototype system described in Chapter 4, which implements the failure recovery scheme and focuses on surviving the crashes caused by bugs introduced by software updates. To implement this technique, it uses the ptrace interface.

The lockstep execution allows a precise control over the execution of each version. The disadvantage is that the application runs at the speed of the slowest version at any given point, which can have a negative impact on performance, especially when running large number of versions in parallel. To address this issue, we can modify this scheme and allow one of the versions, designated as a leader, to run without waiting for other versions, producing a sequence of system call events, which the other versions designated as followers compare against.

This approach is used by a second system called Varan, described in Chapter 5, aimed towards running a large number of versions with low performance overhead, implementing the transparent failover scheme. To achieve this goal, Varan uses selective binary rewriting.

2.4 Summary

Software updates are an integral part of the software development and maintenance process, but unfortunately they present a high risk, as new releases often introduce new bugs and security vulnerabilities.

In this chapter, we have argued for a new way of performing software updates, in which the new version of an application is run in parallel with old application versions, in order to increase the reliability of the overall system. We believe that multi-version software updates can have a significant impact on current software engineering practices, by allowing frequent software updates without sacrificing the stability of older versions.

We proposed two different multi-version execution schemes: failure recovery and transparent failover, targeting different scenarios with different trade-offs. We also presented two different designs for building monitors suitable for multi-version execution implementing these schemes. The first one, called Mx described in Chapter 4, is focused on surviving crashes caused by bugs introduced in software updates, with the prototype implementation built using the ptrace mechanism. The second one, called Varan described in Chapter 5, is aimed towards running a large number of versions side-by-side with low performance overhead, and uses selective binary rewriting to achieve this goal.

3

Software Evolution in Real-world

The multi-version execution approach is based on two key assumptions. First, that new bugs are being introduced during the software evolution and maintenance process even in well-tested code. Second, that during software evolution, the externally observable behaviour of the applications remains relatively stable, especially between minor revisions (*i.e.* security and bug fixes). While there is a lot of first hand and anecdotal evidence in support of these assumptions, despite the key role that software evolution plays in the application life cycle, it is surprising how few empirical studies one can find in the research literature regarding the evolution of the *execution* of real systems.

Software repositories provide rich information about the construction and evolution of software systems. While static data extracted from software repositories have been extensively studied, dynamic metrics concerning the execution of the software have received much less attention, due to the inherent difficulty of running and monitoring a large number of software versions.

In this chapter, we present an empirical study concerning both static and dynamic metrics which aims to answer some of the questions related to software evolution. To perform this study, we have built a flexible infrastructure that can be used to run each version of a system in isolation and collect static and dynamic software metrics from the test suite execution. We consider the tests to be an (imperfect) proxy for a real-world execution.

We have used this infrastructure to examine how code and tests co-evolve in nine popular open-source systems. We report the main characteristics of software patches, and analyse the program evolution from both the source code and the external behaviour perspective. We assess the impact of non-determinism on the execution of test suites. We also analyse the code and patch coverage, and investigate whether the coverage of code containing bugs and bug fixes is lower or higher than average; the latter of the two would provide evidence that even well tested code still has bugs, in support of our technique.

3.1 STUDY INFRASTRUCTURE

The architecture of the infrastructure used to perform our study is depicted in Figure 3.1. It contains a generic driver which iterates through all the revisions in a given range and invokes routines specific to each system to compile, run, and collect statistics of interest.

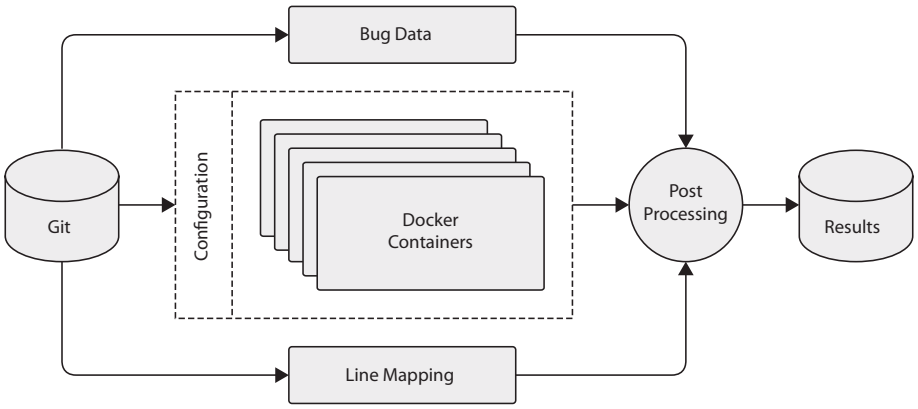


Figure 3.1: The architecture of the infrastructure used in the empirical study.

LIGHTWEIGHT SOFTWARE CONTAINERS. We employed software containers [Sol+07], an operating system-level virtualisation mechanism that provides the ability to run multiple isolated virtual Linux systems ("containers") inside a single host OS. When launched, the driver starts by loading the selected range of revisions from the project's *Git* repository, and for each revision starts a new software container. The use of containers offers increased isolation and reproducibility guarantees by providing a consistent environment in which to run each software revision and ensuring that different revisions do not interfere with each other, *e.g.*, by inadvertently leaving behind lock files or not properly freeing up resources.

The choice of lightweight OS-level virtualisation rather than more traditional virtual machines (*e.g.*, KVM[1] or Xen[2]) reduces the performance penalty associated with spawning and tearing down VMs, operations performed for each revision analysed. To get a sense of this difference, we compared an LXC[3] container, which required under a second for these operations, with a Xen VM, which needed over a minute.

In our implementation, we use Docker[4] to create and manage the lower-level LXC containers, and Vagrant[5] to deploy them on multiple local or cloud machines. Each container is used to configure, compile and test one program revision, as well as collect the metrics of interest, such as code size and coverage. The containers are remotely controlled through SSH using the Fabric[6] framework.

PROJECT CONFIGURATION. We used a modular pipeline architecture, which made it possible to automate the analysis and minimise the manual effort. The analysis is driven by a Python configuration file describing the system. A minimal file provides the name of the system, its *Git* repository location, a method to compile the system, *e.g.*, install dependencies and run the appropriate `make` command, and a method to run the regression tests, *e.g.*, run the `make test` command. Finally, the configuration file

[1] `http://www.linux-kvm.org/`
[2] `http://www.xenproject.org/`
[3] `http://linuxcontainers.org/`
[4] `https://www.docker.io/`
[5] `https://www.vagrantup.com/`
[6] `http://fabfile.org/`

can also specify an *end revision* and a specific number of revisions to analyse. For accurate test suite size measurements, the files or folders which make up the test suite can also be indicated.

For each revision, we collect several static and dynamic metrics. The static metrics are obtained either directly from the version control system (*e.g.*, the number of lines of test code) or after compiling each revision (*e.g.*, the number of executable lines of code). The dynamic metrics require running the regression tests (*e.g.*, the overall line coverage or the regression test success status).

Further information and graphs—including the ones presented in this thesis—are automatically derived in the post-processing stage from these primary metrics using a set of scripts.

BUG DATA. For our study, we mined bug data from both software repositories and, where available, bug tracking systems. We automatically obtained a list of candidate bug-fixing revisions by iterating through the list of commits and checking the commit message for words such as *fix*, *bug* or *issue*, followed by a number representing the bug identifier. For example, a typical *Memcached* bug fix commit message looks like *"Issue 224 - check retval of main event loop"*. The regular expression that we used to identify these commits is similar to the ones used in prior work [Her+13]:

```
(?:bug|issue|fix|resolve|close)\s*\#?\s?(\d+)
```

Where possible, we confirmed that the bug identifier is valid by querying the associated bug tracking system. We further manually checked all reported revisions and confirmed that they included no false positives. While it is impossible to quantify the false negative rate without a knowledgeable developer manually checking all the revisions in a repository, we believe that the automatically obtained bug fixes create a representative subset of the fixes in the repository.

LINE MAPPING. The ability to track how lines move and change across revisions is the cornerstone of many high-level software evolution analyses. A line mapping algorithm improves over the traditional `diff` algorithm by tracking the movement of individual lines rather than hunks. Conceptually, line mapping is a function which takes two revisions, *r1* and *r2*, and a program location described by a pair *(file name 1, line number 1)* associated with *r1*. The output is a pair *(file name 2, line number 2)* identifying the corresponding location in *r2*.

Our implementation of the line mapping algorithm is similar to the algorithms described in previous work [ŚZZ05; Kim+06; CCD07; WS08]. It makes use of the *Levenshtein edit distance* [Lev66] to track line edits, and *tf–idf* [Spa88] and *cosine similarity* [Sin01] to track line movements. It also uses the *Hungarian algorithm* [Kuh55] to find the optimal matching of lines across versions. Compared to previous work, our implementation can also improve precision by using coverage information to filter non-executable lines. In our study, we used line mapping to determine whether patches are tested within the next few revisions after they were created (§3.4.2).

EDIT DISTANCE. Edit distance, also known as Levenshtein edit distance [Lev66] is defined as the minimum number of operations (insertions, deletions, and substitutions) required to transform one string into the other. We use edit distance to measure the similarity of the system call traces (§3.5).

However, these traces are not strings, rather sequences of strings. To allow the use of edit distance on these sequences, we use a hashing scheme, which assigns each string in the sequence a 32-bit number. The algorithm then calculates the edit distance between the integer strings.

Another problem is the efficiency of the implementation. The sequences we are comparing in our study are hundreds of millions entries long. We have implemented the Levenshtein algorithm as a native C Python extension and used OpenMP to parallelise this implementation to take advantage of all available cores. This implementation only keeps one line of the computation matrix, and divides this line into strides which are then processed by different cores.

CLOUD DEPLOYMENT. Since compiling and executing a large number of revisions can be very time-consuming, especially for larger projects, we deployed the infrastructure to our private cloud and used multiple machines to run the analysis in parallel on different subset of revisions (*i.e. sharding*). We have built our system around a standard set of tools: Packer for building custom Docker-enabled machine images, Vagrant for controlling and provisioning the virtual machines based on these images, a private Docker registry for serving Docker containers containing the driver scripts, and a *fabfile* for orchestrating the entire cluster.

3.2 OVERVIEW

We used our infrastructure to understand the evolution of sevelar popular open-source applications written in C/C++, over a combined period of twenty three years. To allow for faithful comparison, we have restricted our selection to applications which have less than 100,000 ELOC and include a regression test suite. The nine evaluated applications are:

GNU Binutils[7] is a set of utilities for inspecting and modifying object files, libraries and binary programs. We selected for analysis the twelve utilities from the `binutils` folder (`addr2line`, `ar`, `cxxfilt`, `elfedit`, `nm`, `objcopy`, `objdump`, `ranlib`, `readelf`, `size`, `strings` and `strip`), which are standard user-level programs under many UNIX distributions.

Beanstalkd[8] is a simple and fast work queue originally designed for reducing the latency of page views in high-volume web applications.

Git[9] is one the most popular distributed version control systems used by the open-source developer community.

Lighttpd 1.4[10] is a lightweight web server optimised for high performance environments.

Lighttpd 2.0[11] is the new major version of the *Lighttpd 1.4* web server developed entirely from scratch by the same team of developers.

Memcached[12] is a general-purpose distributed memory caching system used by several popular sites such as Craigslist, Digg and Twitter.

Table 3.1: Summary of applications used in our study. *ELOC* represents the number of executable lines of code and *TLOC* the number of lines in test files in the last revision analysed.

	Code		Tests	
Application	Language	ELOC	Language	TLOC
Beanstalkd	C	2567	C	2031
Binutils	C	27,029	DejaGnu	5186
Git	C	79,760	C/shell	108,464
Lighttpd 1.4	C	20,487	Perl	3378
Lighttpd 2.0	C	23,884	Python	2440
Memcached	C	4426	C/Perl	4605
Redis	C	18,203	Tcl	7589
Vim	C	78,682	Vim script	13,654
ØMQ	C++	7276	C++	3460

Redis[13] is a popular key-value data store used by many well-known services such as Twitter, GitHub and StackOverflow.

Vim[14] is arguably one of the most popular text editors.

ØMQ[15] is a high-performance asynchronous messaging middleware library used by a number of organisations such as Los Alamos Labs, NASA and CERN.

The nine applications are representative for C/C++ open-source code: GNU *Binutils* are user-level utilities, *Git* is a version control system, *Beanstalkd*, *Lighttpd 2.0*, *Memcached* and *Redis* are server applications, *Vim* is a text editor while ØMQ is a library.

3.2.1 *Basic characteristics*

Table 3.1 shows some basic characteristics of these systems: the language in which the code and tests are written, the number of executable lines of code (ELOC) and the number of lines of test code (TLOC) in the last revision analysed. To accurately measure the number of ELOC, we leveraged the information stored by the compiler in `gcov` graph files, while to measure the number of TLOC we did a simple line count of the test files (using `cloc`, or `wc -l` when `cloc` cannot detect the file types).

The code size for these applications varies from only 2567 ELOC for *Beanstalkd* to 79,760 ELOC for *Git*. The test code is written in a variety of languages and ranges from 2031lines of Python code for *Beanstalkd* to 108464lines of C and shell code for *Git*. The test code is 36% larger than the application code in the case of *Git*, approximately as large as the application code for *Beanstalkd* and *Memcached*, around 40% of the application code for *Redis* and ØMQ, around 19% of the application code for *Binutils*, around 17% of the application code for *Lighttpd 1.4* and *Vim*, and only around 10% of the application code for *Lighttpd 2.0*. Running the test suite on the last version takes only a few seconds for *Binutils*,

Lighttpd 1.4, *Lighttpd 2.0*, *Vim* and ØMQ, 110 seconds for *Memcached*, 315 seconds for *Redis*, and 30 minutes for *Git*, using a four-core Intel Xeon E3-1280 machine with 16 GB of RAM.

The version control system used by the majority of these applications is *Git*. Four of these projects—*Git*, *Memcached*, *Redis*, and ØMQ—are hosted on the GitHub[16] online project site. The other two—*Binutils* and *Lighttpd 2.0*—use their own *Git* hosting. *Lighttpd 1.4* uses Subversion, but the project also provides a *Git* mirror on GitHub. *Vim* uses Mercurial.

3.2.2 *Selection of revisions*

Our goal was to select a comparable number of revisions across applications. The methodology was to start from the current version at the day of our experiments, and select an equal number of previous revisions for all systems. We only counted revisions which modify executable code, tests or both because this is what our analyses look at. We decided to select 250 such revisions from each system because some systems had non-trivial dependency issues further back than this, which prevented us from properly compiling or running them. We still had to install the correct dependencies where appropriate, *e.g.*, downgrade `libev` for older versions of *Lighttpd 2.0* and `libevent` for *Memcached*.

Note that not all revisions compile, either due to development errors or portability issues (*e.g.*, system header files differing across OS distributions). Redis has the largest number of such transient compilation errors—38. The prevailing reasons are missing `#include` directives, *e.g.*, `unistd.h` for the `sleep` function, and compiler warnings subsequently treated as errors. The missing `#include` directives most likely slipped past the developers because on some systems other `libc` headers cause the missing headers to be indirectly included. The compiler warnings were generated because newer compiler versions, such as the one that we used, are more pedantic. Other reasons include forgotten files and even missing semicolons.

We decided to fix the errors which had likely not been seen at the time a particular revision was created, for example by adding the compile flag `-Wno-error` in *Binutils* so that warnings do not terminate the build process. In all situations when we could not compile a revision, we rolled over the changes to the next revisions until we found one where compilation was successful. Revisions which do not successfully compile are not counted towards the 250 limit.

Another important decision concerns the granularity of the revisions being considered. Modern decentralised software repositories based on version control systems such as *Git* do not have a linear structure and the development history is a directed acyclic graph rather than a simple chain. Different development styles generate different development histories; for example, *Git*, *Redis* and ØMQ exhibit a large amount of branching and merging while the other six systems have a mostly linear history. Our decision was to focus on the main branch, and treat each merge into it as a single revision. In other words, we considered each feature branch a single indivisible unit. Our motivation for this decision was twofold: first, development branches are often spawned by individual developers in order to work on a certain issue and are often "private" until they are merged into the main branch. As a result, sub-revisions

[16] `https://github.com/`

Table 3.2: Revisions used in our study. *OK*: code compiles and tests complete successfully, *TF*: some tests fail, *TO*: tests time out, *CF*: compilation fails, *Time*: the number of months analysed.

	OK+TF+TO=250				
Application	OK	TF	TO	CF	Time
Beanstalkd	150	100	0	7	67mo
Binutils	240	10	0	25	35mo
Git	249	0	1	0	5mo
Lighttpd 1.4	235	14	1	14	58mo
Lighttpd 2.0	145	105	0	13	36mo
Memcached	206	43	1	5	47mo
Redis	211	38	1	38	6mo
Vim	250	0	0	0	16mo
ØMQ	171	79	0	11	17mo

in such branches are often unusable or even non-compilable, reflecting work-in-progress. Second, the main branch is generally the one tracked by most users, therefore analysing revisions at this level is a good match in terms of understanding what problems are seen in the field. This being said, there are certainly development styles and research questions that would require tracking additional branches; however, we believe that for our benchmarks and research questions this level of granularity provides meaningful answers.

Table 3.2 summarises the revisions that we selected: they are grouped into those that compile and pass all the tests (*OK*), compile but fail some tests (*TF*), compile but time out while running the test suite (*TO*), and fail to compile (*CF*). The time limit that we enforced was empirically selected for each system such that it is large enough to allow a correct revision to complete all tests. As shown in the table, timeouts were a rare occurrence, with at most one occurrence per application.

Table 3.2 also shows the development time span considered, which ranges from only 5-6 months for *Git* and *Redis*, which had a fast-paced development during this period, to almost five and a half years for *Beanstalkd*. The age of the projects at the first version that we analysed ranges from 5 days for *Beanstalkd*, to 15 years for *Vim*.

3.2.3 *Revision setup*

All the programs analysed were compiled to record coverage information. In addition, we disabled compiler optimisations, which generally interact poorly with coverage measurements. We used existing build targets and configuration options if available, otherwise we configured the application with the flags `CFLAGS='-O0 -coverage'` and `LDFLAGS=-coverage`. All code from the system headers, *i.e.* `/usr/include/` was excluded from the results.

Each revision was run in a virtualised environment based on 64-bit version of Ubuntu 12.10 (12.04.3 for *Git*) running inside an LXC container. To take advantage of the inherent parallelism of this approach,

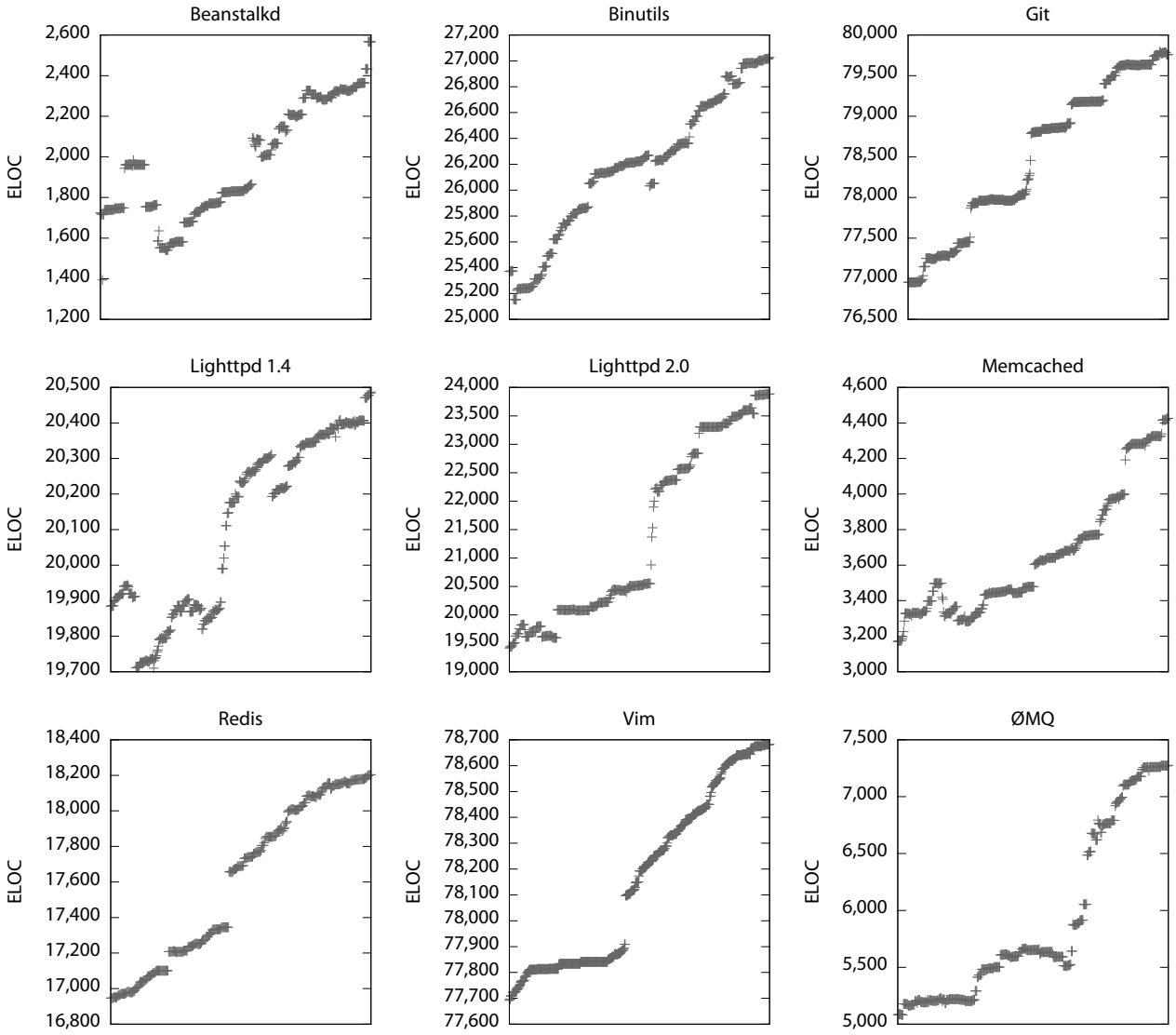


Figure 3.2: Evolution of executable lines of code.

the containers were spawned in one of 28 long-running Xen VMs, each with a 4 Ghz CPU, 6 GB of RAM, and 20 GB of storage, running a 64-bit version of Ubuntu 12.04.3.

3.3 CODE AND TEST EVOLUTION

Figure 3.2 shows the evolution of each system in terms of ELOC. As discussed above, we measured the number of ELOC in each revision by using the information stored in gcov graph files. This eliminates all lines which were not compiled, such as those targeting architectures different from our machine. One of the main reasons for which we have decided to measure ELOC rather than other similar metrics is that they can be easily related to the dynamic metrics, such as patch coverage, presented in Sections 3.4.

As evident from this figure, all nine systems grow over time, with periods of intense development that increase the ELOC significantly, alternating with periods of code tuning and testing, where the code size

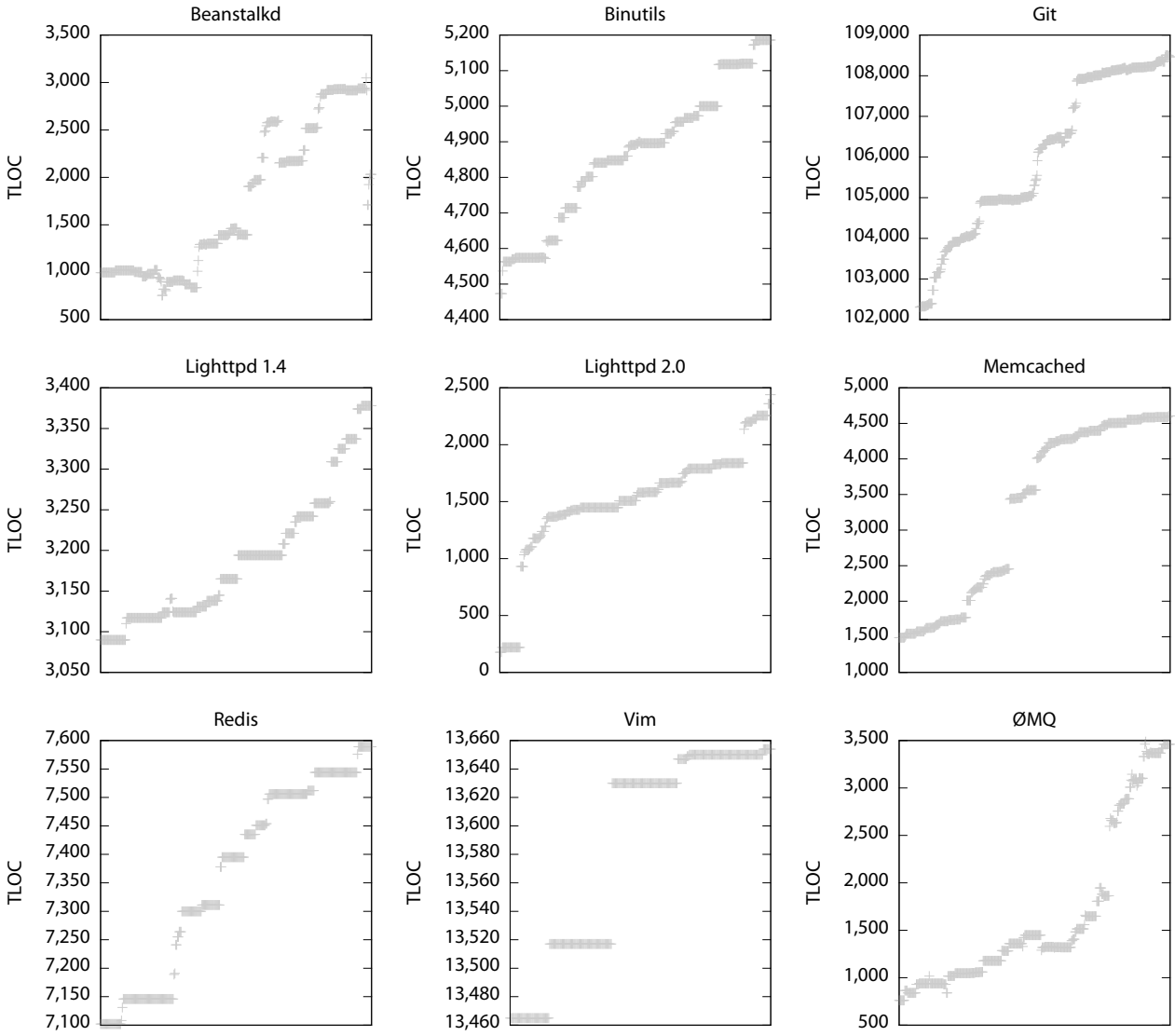


Figure 3.3: Evolution of textual lines of test code.

increases at a slower pace. It is interesting to note that there are also several revisions where the number of ELOC decreases (*e.g.*, in ØMQ): upon manual inspection, we noticed that they relate to refactorings such as using macros or removing duplicate code.

The total number of ELOC added or modified varies between 2,296 for *Redis* and 10,834 for *Lighttpd 2.0*, while the end-to-end difference in ELOC varies between 842 for *Beanstalkd* and 4,466 for *Lighttpd 2.0*.

Figure 3.3 presents the evolution of the size of the test suite in each system, measured in textual lines of test code (TLOC). For each system, we manually identified the files responsible for regression testing and recorded the number of lines contained in them at each revision. It can be seen that test evolution is less dynamic than code evolution, developers adding less test code than regular code. In some cases, developers even remove large parts of the test code. This is most obvious in case of *Beanstalkd* whose test suite has been rewritten from scratch several times over the analysed period.

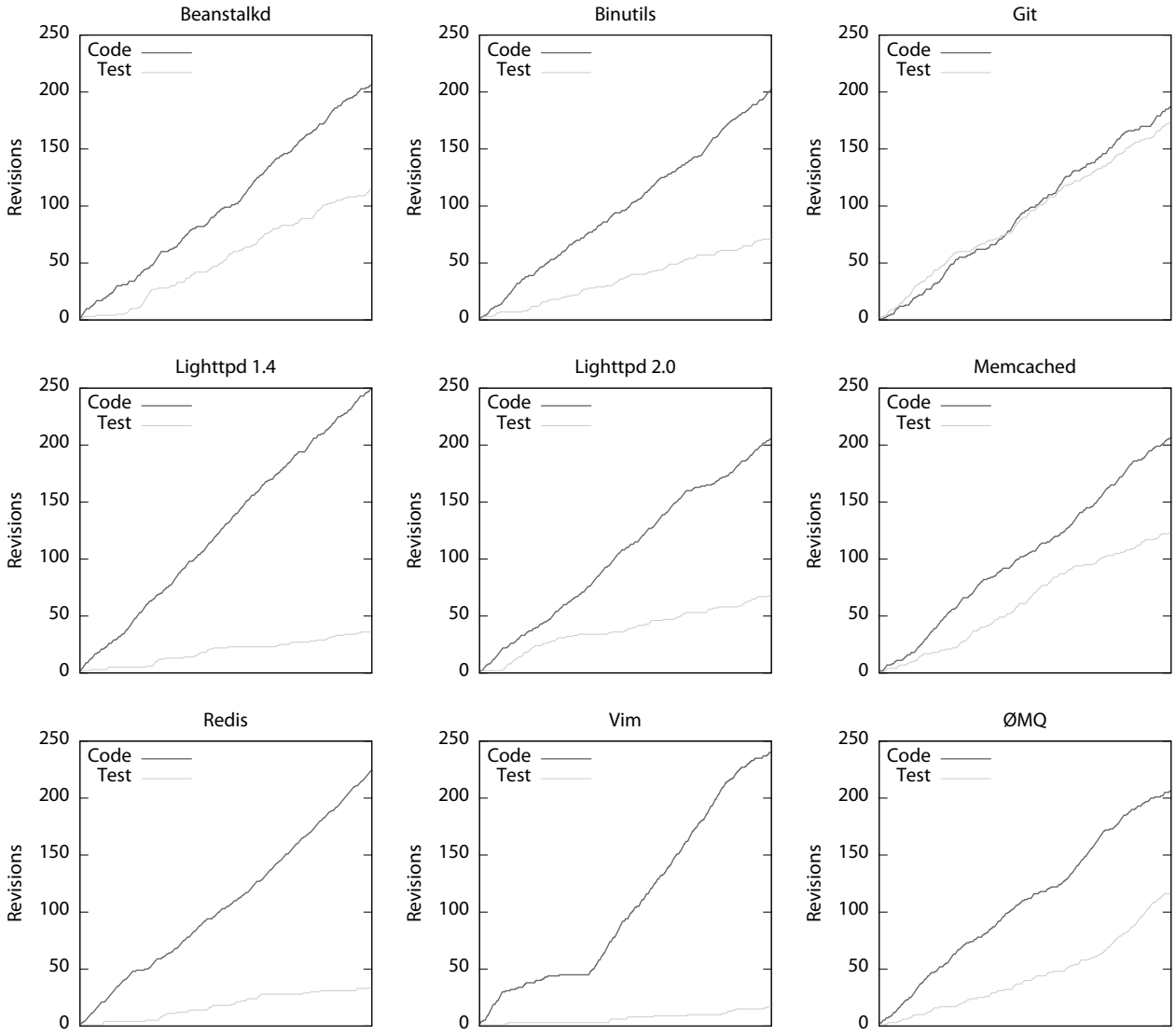


Figure 3.4: Co-evolution of executable and test code. Each increment represents a change.

To better understand the co-evolution of executable and test code, we merged the above data and plotted in Figure 3.4 only whether a revision changes the code or tests: that is, the *Code* and *Test* values increase by one when a change is made to the code, respectively to the tests in a revision, and stay constant otherwise. As it can be seen, while the *Code* line smoothly increases over time, the *Test* line frequently stays constant across revisions, indicating that testing is often a *phased* activity [Zai+11], that takes place only at certain times during development. One exception is *Git*, where code and tests evolve more *synchronously*, with a large number of revisions modifying both code and tests.

3.3.1 *Patch characteristics*

Each revision defines a *patch*, which consists of the totality of changes introduced by that revision. Software patches represent the building blocks of software evolution, and can affect code, regression

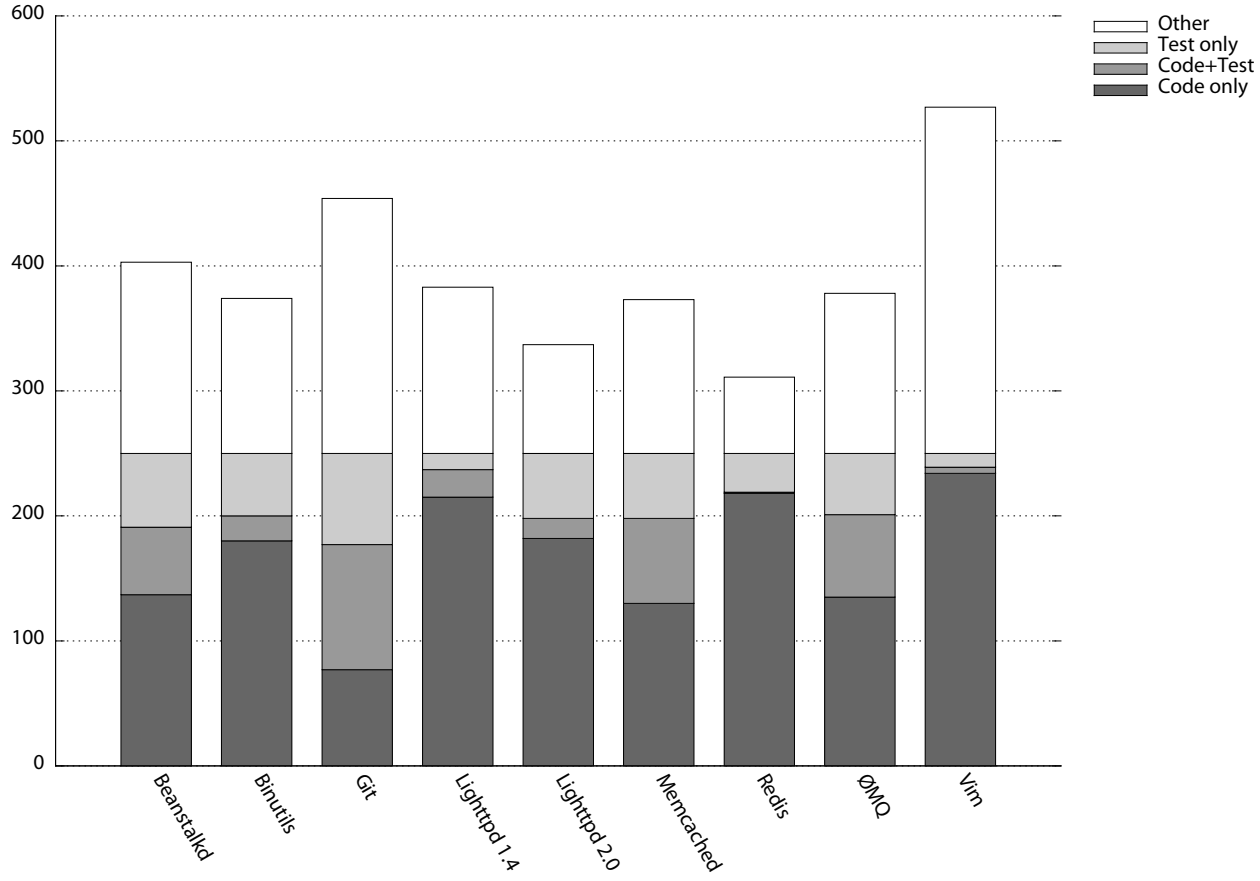


Figure 3.5: Breakdown of patches by type: affecting executable application code but not test code, affecting both, affecting only test code, and neither.

tests, or infrastructure components such as build scripts, and play a variety of roles, including bug fixing, feature addition, and better testing.

Figure 3.5 classifies patches into those that modify executable application code but not the test code (*Code only*), those that modify both executable application code and test code (*Code+Test*), and those that modify test code but not executable application code (*Test only*). Note that for each application, these three values sum to 250, since we only selected revisions which modify executable code and/or tests, as discussed previously. Figure 3.5 also shows the number of patches from the time span analysed that modify neither executable program code nor tests (*Other*).

The first observation is that a substantial amount of time is spent in maintenance activities that do not involve code nor tests. For example, during the period analysed, in addition to the 250 target patches, there were around 120 additional such patches in *Binutils*, *Memcached* and ØMQ, 153 in *Beanstalkd*, 204 in *Git* and 277 in *Vim*. Note that some of these patches may modify code that is excluded during preprocessing on our machine, but most cases involved changes to build scripts, documentation, and other similar software artefacts.

From the 250 patches selected for each application, the majority only modify code, with a relatively small number of patches (73 in *Git*, and under 52 for the others) touching only tests. The number of revisions that modify both code and tests can offer some indication of the development style used: at

one end of the spectrum there is *Redis*, with only one such patch, suggesting that coding and testing are quite separate activities; at the other end there is *Git*, with 100 such patches, suggesting a development discipline in which code changes are frequently accompanied by a test case.

The size of a patch and the number of locations affected by it can provide useful guidance for the external behaviour changes—small, localised changes are likely to introduce small changes in behaviour. The *Lines* column in Table 3.3 provides information about the size of the executable code patches analysed in each system, measured in ELOC. Note that our measurements ignore changes in the amount of whitespace, *e.g.*, whitespace at the end of the line, characters, because our target programming languages, C and C++, are insensitive to such modifications. Most patches are small, with the median number of ELOC ranging from 3 to 7.

To understand the degree to which patches are spread out through the code, we also recorded the number of areas in the code—*hunks* in *Git* terminology—and the number of files containing executable code which suffered changes. More formally, a hunk groups together all the lines added or modified in a patch which are at a distance smaller than the *context size*. We used the default unified diff format with a context size of three lines when computing the hunks.[17] The *Hunks* column in Table 3.3 shows that the median number of hunks varies between 2 and 4.

Finally, the median number of files modified by a patch is only 1 for all benchmarks with the exception of *Vim* and ØMQ, for which it is 2. The fraction of patches that modify a single file is, in increasing order, 4.6% for *Vim*, 48.7% for ØMQ, 58.7% for *Git*, 67.5% in *Beanstalkd*, 65.1% for *Lighttpd 2.0*, 66.6% for *Memcached*, 84.9% for *Redis*, and 88.5% for *Binutils*.

While these results are based on static analysis only, the fact that most patches are small and localised suggests that individual patches are likely to have only a small impact on the external behaviour. This hypothesis is further confirmed in Section 3.5 which analyses the dynamic behaviour.

3.4 CODE COVERAGE

As a part of our study focuses on coverage metrics, we first investigate whether code coverage is deterministic, *i.e.* whether the regression test suite in a given revision achieves the same coverage every time it is executed. As we show, non-determinism has implications in the reproducibility of test results and the fault detection capability of the tests as well as for multi-version execution.

We measured the overall coverage achieved by the regression test suite using `gcov`. Interestingly, we found that most programs from our experiments except *Binutils*, *Lighttpd 1.4* and *Vim* are non-deterministic, obtaining slightly different coverage in each run of the test suite. Therefore, we first quantified this non-determinism by running the test suite five times for each revision and measuring how many revisions obtained mixed results, *i.e.* one run reported success while another reported failure. We have found a number of revisions displaying this behaviour, as listed in Table 3.4 under the column *Nondet Result*.

[17] See `http://www.gnu.org/software/diffutils/manual/html_node/` for more details.

Table 3.3: The median number of executable lines, hunks from executable files, and executable files in a patch. Only data from patches which add or modify executable code is considered.

Application	Lines	Hunks	Files
Beanstalkd	4	2	1
Binutils	5	2	1
Git	7	3	1
Lighttpd 1.4	5	3	1
Lighttpd 2.0	6	3	1
Memcached	6	3	1
Redis	4	2	1
Vim	3	3	2
ØMQ	7	4	2

Table 3.4: Number of revisions where the test suite non-deterministically succeeds/fails, and the maximum, median and average number of lines which are non-deterministically executed in a revision.

	Nondet.	Nondet. ELOC		
Application	Result	Max	Median	Average
Beanstalkd	3	17	0	0.17
Binutils	0	0	0	0
Git	1	23	13	11.80
Lighttpd 1.4	0	93	20.5	27.13
Lighttpd 2.0	1	37	10	13.01
Memcached	21	22	8.5	7.55
Redis	16	71	23	30.98
Vim	0	7	0	0.98
ØMQ	32	47	27	19.52

We further counted for each pair of runs the number of lines whose coverage status differs. We used a 0/1 metric, *i.e.* we only considered a difference when one of the five runs never executes a line and another one executes it. We only did this for revisions in which the test suite completes successfully to avoid spurious results that would occur if we compare a run which completed with one that was prematurely terminated due to a failure. As shown in Table 3.4, *Binutils*, *Lighttpd 1.4* and *Vim* seems to be deterministic with respect to its test suite, while *Redis*, for example, contains on average 30.98 lines that are non-deterministically executed.

When reporting the overall coverage numbers, we accumulated the coverage information across all five runs.[18] Therefore, the results aim to count a line as covered if the test suite *may* execute it. The darker grey (upper) lines in Figure 3.6 plot the overall line coverage for all benchmarks. It can be seen that coverage level varies significantly, with *Beanstalkd* at one end achieving only 16.04% coverage on average, and *Git* at the other achieving 80.74%, while in-between *Lighttpd 2.0* achieves 39.08%, *Redis* 59.97%, ØMQ 66.88%, and *Memcached* 72.98%.

[18]With the exception of *Git*, where for convenience we considered a single run, as the number of lines affected by non-determinism represent less than 0.3% of the total codebase.

We manually investigated the non-determinism and pinpointed three sources: (1) multi-threaded code, (2) ordering of network events, and (3) non-determinism in the test harness. As an example from the first category, the test from ØMQ called `test_shutdown_stress` creates 100 threads to check the connection shutdown sequence. In a small percentage of runs, this test was exposing a race condition.[19] In the third category, some *Redis* tests generate and store random integers, non-deterministically executing the code implementing the internal database data structures. The *Memcached* test `expirations.t` is representative of tests that make assumptions based on hardcoded wall-clock time values, which cause failures under certain circumstances. The test timings were previously adjusted[20] in response to failures under Solaris' `dtrace` and we believe that some of the failures that we encountered were influenced by the Docker environment.

These results show that a multi-version execution environment needs to account for sources of non-determinism to be practical. Varan and Mx intercept and emulate accesses to environment values which are non-deterministic, such as time and date (*e.g.*, through file `/etc/localtime` or system call `gettimeofday()`) or random number generators (*e.g.*, through file `/dev/random` or system call `getrandom()`). The strict system call ordering enforced across versions by both Varan and Mx ensures that all networking events are delivered in the same order in all versions.

For non-determinism related to concurrency, deterministic multi-threading mechanisms (DMT) have been an active area of research for several years, and while there is no practical solution which could handle all possible cases with reasonable overhead, the existing solutions are already usable for many real-world applications [Ber+10; LCB11]. Varan uses logical clocks to enforce ordering across threads. While this approach provides weaker guarantees than those typically provided by other DMT systems, it can still handle most cases, as described in Section 5.2.3.

3.4.1 *Coverage evolution*

One interesting question is whether coverage stays constant over time. As evident from Figure 3.6, for *Binutils*, *Git*, *Memcached*, and *Redis*, the overall coverage remains stable over time, with their coverage changing with less than 2 percentage points within the analysed period. On the other hand, the coverage of *Lighttpd 2.0* and ØMQ increase significantly during the time span considered, with *Lighttpd 2.0* increasing from only 2.02% to 49.37% (ignoring the last two versions for which the regression suite fails), and ØMQ increasing from 62.89% to 73.04%. *Beanstalkd* is the only application whose coverage decreases over time, from 31.36% to 5.25%, as its test suite has been reimplemented several times over the project lifetime (as shown in Figure 3.3) with each implementation achieving different coverage. An interesting observation is that coverage evolution is not strongly correlated to the co-evolution of executable and test code. Even when testing is a phased activity, coverage remains constant because the already existing tests execute part of the newly added code.

[19] `https://github.com/zeromq/zeromq4-x/commit/de239f3`
[20] `https://github.com/memcached/memcached/commit/890e3cd`

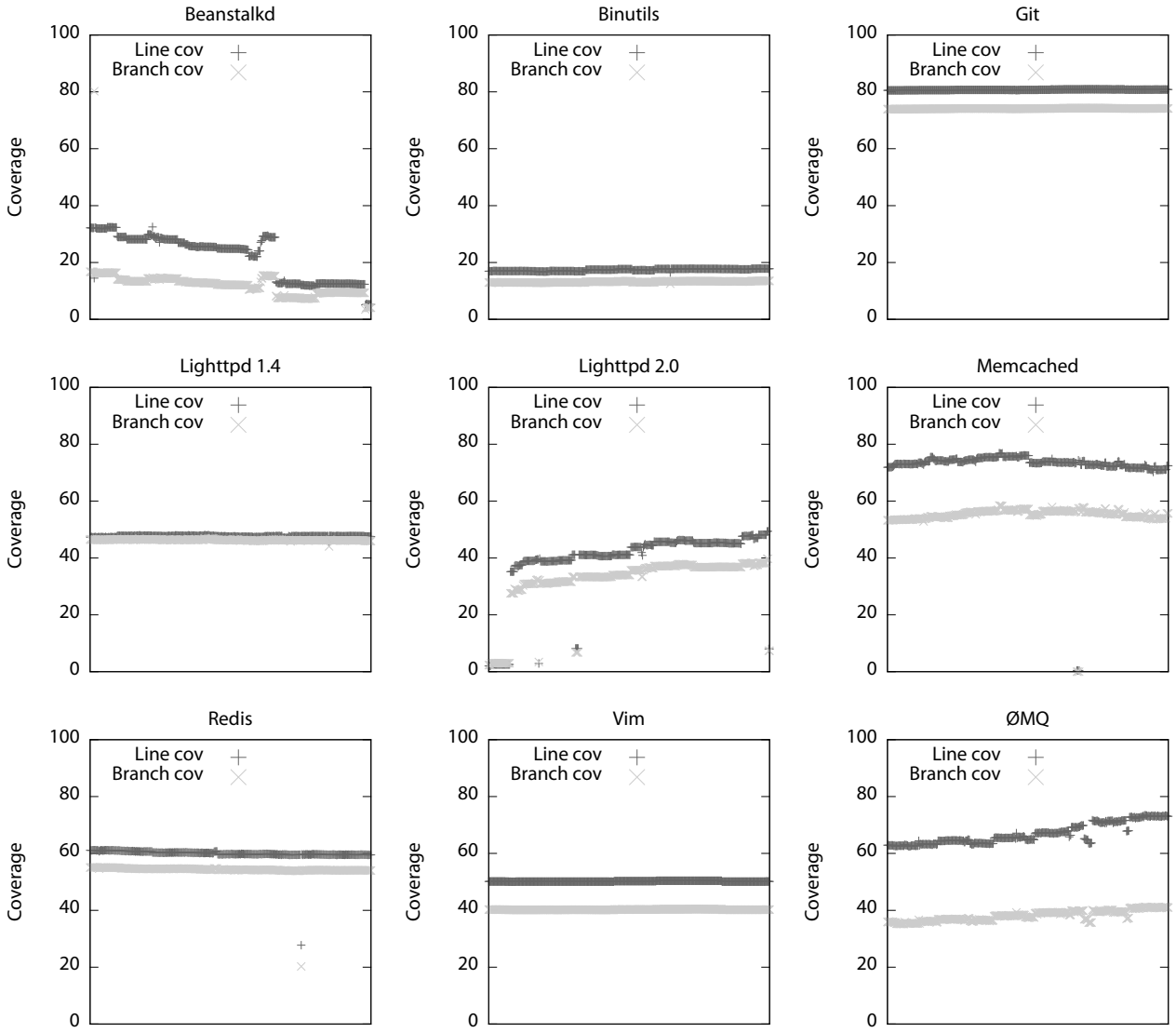


Figure 3.6: Evolution of the overall line and branch coverage.

One may notice that a few revisions from *Lighttpd 2.0*, *Memcached* and *Redis* cause a sudden decrease in coverage. This happens because either bugs in the program or in the test suite prevent the regression tests from successfully running to completion. In all cases, these bugs are fixed after just a few revisions.

Figure 3.6 also shows that branch coverage closely follows line coverage. The difference between line and branch coverage is relatively small, with the exception of *Memcached* and ØMQ. The larger difference is due to the frequent use of certain code patterns which generate multiple branches on a single line, such as the one shown in Listing 3.1, which comes from the ØMQ codebase. The `zmq_assert` macro is expanded into a single line resulting in 100% line coverage, but only 50% branch coverage when executed in a typical run of the program (where assertions do not fail).

The fact that line and branch coverage closely follow one another suggests that in many situations only one of these two metrics might be needed. For this reason, in the remaining of the paper, we report only line coverage.

```
#define zmq_assert(x) \
  do {\
    if (unlikely (!(x))) {\
      fprintf (stderr, "Assertion failed: %s (%s:%d)\n", #x, \
          __FILE__, __LINE__);\
      zmq::zmq_abort (#x);\
    }\
  } while (false)
```

Listing 3.1: Example of an assertion macro used in ØMQ codebase.

Finally, we have looked at the impact on coverage of revisions that only add or modify tests (*Test only* in Figure 3.5). An interesting observation is that many of these revisions bring no improvements to coverage. For example, in *Lighttpd 2.0* only 26 out of 52 such revisions improve coverage. The other 26 either do not affect coverage (18) or decrease it (8). The revisions which do not affect coverage can be a sign of test driven development, *i.e.* the tests are added before the code which they are intended to exercise. The revisions which decrease coverage are either a symptom of non-determinism—six of them, with small decreases in coverage—or expose bugs or bigger changes in the testing infrastructure (the other two). These two revisions exhibit a drop in coverage of several thousands lines of code. In one case, the tests cause *Lighttpd 2.0* to time out, which leads to a forceful termination and loss of coverage data. This problem is promptly fixed in the next revision. In the other case, the new tests require a specific (new) module to be built into the server, terminating the entire test suite prematurely otherwise.

3.4.2 *Patch coverage*

We define *patch coverage* as the ratio between the number of executed lines of code added or modified by a patch and the total number of executable lines in the patch, measured in the revision that adds the patch.

Figure 3.7 shows the distribution of the patch coverage for each system. Each column corresponds to all patches which affect executable code in a system, normalised to 100%. The patches are further grouped into four categories depending on their coverage. As it can be observed, the patch coverage distribution is bi-modal across applications: the majority of the patches in *Git*, *Memcached* and ØMQ achieve over 75% coverage, while the majority of the patches in *Beanstalkd*, *Binutils*, *Lighttpd 2.0* and *Redis* achieve under 25%. One interesting aspect is that for all applications, there are relatively few patches with coverage in the middle ranges: most of them are either poorly ($\leqslant$25%) or thoroughly (>75%) covered.

Table 3.5 presents the same patch coverage statistics, but with the patches bucketed by their size into three categories: less than 10 ELOC, between 11 ELOC to 100 ELOC, and greater than 100 ELOC. For all benchmarks, patches are distributed similarly across buckets, with the majority of patches having $\leqslant$10 ELOC and only a few exceeding 100 ELOC. Except for *Beanstalkd* and *Vim*, the average coverage of

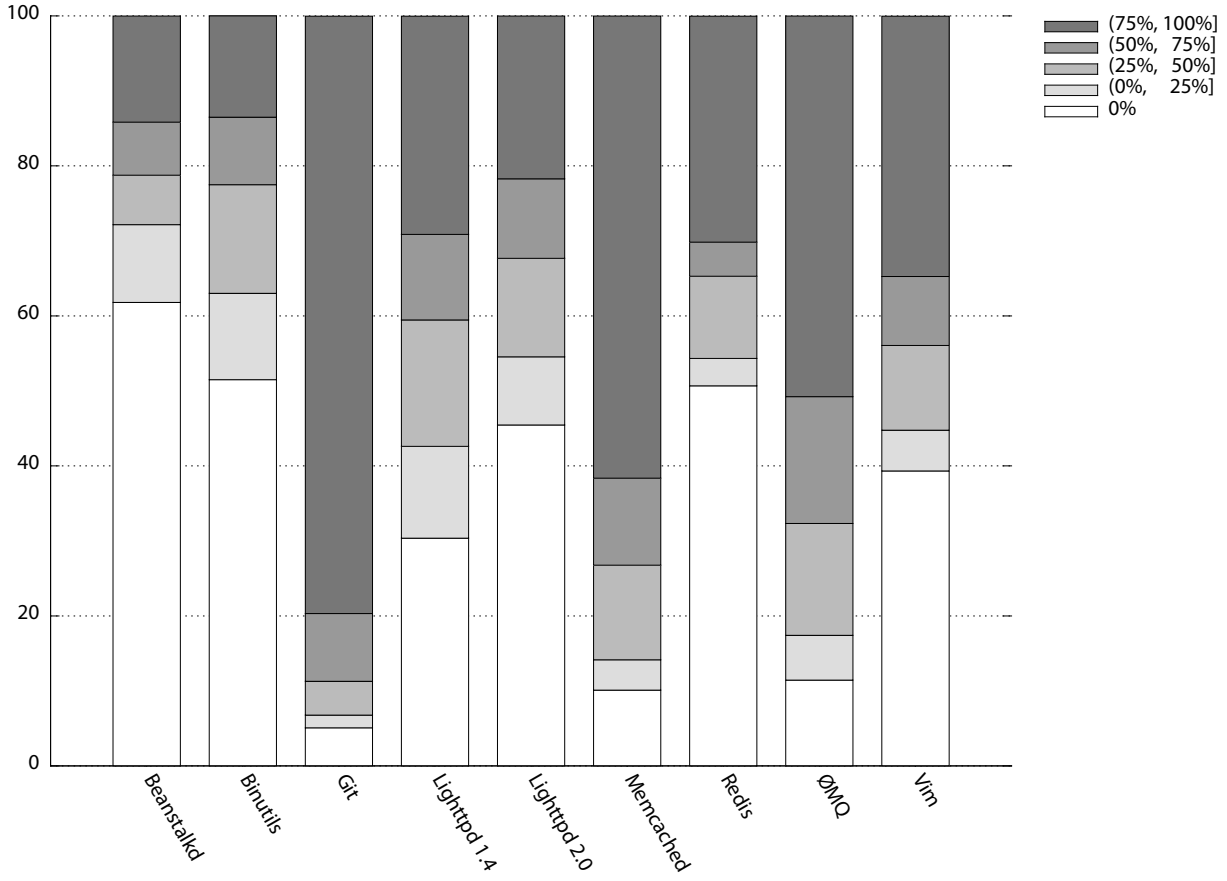


Figure 3.7: Patch coverage distribution. Each colour represents a range of coverage values with the bar size indicating the percentage of patches whose coverage lies in the respective range.

patches with ⩽10 ELOC is higher than for those with >100 ELOC, but the coverage of the middle-size category varies.

Finally, the first column in Table 3.6 shows the overall patch coverage, *i.e.* the percentage of covered ELOC across all patches. For *Beanstalkd*, *Binutils*, *Git*, *Memcached* and *Vim*, it is within five percentage points from the overall program coverage, while for the other benchmarks it is substantially lower—for example, the average overall program coverage in *Redis* is 59.97%, while the overall patch coverage is only 30.4%.

In some projects, tests exercising the patch are added only after the code has been submitted, or the patch is only enabled (*e.g.*, by changing the value of a configuration parameter) after related patches or tests have been added. To account for this development style, we also recorded the number of ELOC in each patch which are only covered in the next few revisions (we considered up to ten subsequent revisions). We refer to the ratio between the number of such ELOC and the total patch ELOC as *latent patch coverage*.

We counted these lines by keeping a sliding window of uncovered patch lines from the past ten revisions and checking whether the current revision covers them. When a patch modifies a source file, all entries from the sliding window associated with lines from that file are remapped if needed, using the line mapping algorithm discussed in Section 3.1.

Table 3.5: Overall patch coverage bucketed by the size of the patch in ELOC. NP is the number of patches in the bucket and C is their overall coverage. Only patches which add or modify executable code are considered.

	⩽10		11-100		>100	
Application	NP	C	NP	C	NP	C
Beanstalkd	139	10.3%	44	17.3%	8	3.2%
Binutils	128	19.5%	63	25.0%	9	16.8%
Git	102	87.4%	65	82.4%	10	87.0%
Lighttpd 1.4	176	43.7%	61	35.8%	0	N/A
Lighttpd 2.0	120	41.9%	58	31.3%	20	30.8%
Memcached	122	73.7%	73	70.8%	3	57.0%
Redis	164	33.8%	51	34.8%	4	21.1%
Vim	199	41.2%	38	44.0%	2	75.8%
ØMQ	119	65.5%	64	68.0%	18	48.9%

Table 3.6: Overall latent patch coverage: the fraction of the lines of code in all patches that are only executed by the regression suite in the next 1, 5 or 10 revisions. The overall patch coverage is listed for comparison.

Application	Overall	+1	+5	+10
Beanstalkd	10.1%	0.5%	0.5%	0.7%
Binutils	21.2%	0.1%	0.3%	0.3%
Git	85.1%	0%	0%	0%
Lighttpd 1.4	37.7%	0.1%	0.4%	0.7%
Lighttpd 2.0	31.3%	0.9%	5.0%	6.1%
Memcached	68.9%	2.1%	3.4%	3.5%
Redis	30.4%	5.2%	5.5%	6.4%
Vim	50.2%	0.1%	0.1%	0.1%
ØMQ	56.9%	0.4%	3.5%	6.0%

Table 3.6 shows the overall latent patch coverage *i.e.* the fraction of patch lines that are covered in the next few revisions after the patch is introduced. We report the results for three sliding window sizes: one, five and ten revisions. The latent patch coverage is significantly smaller compared to the overall patch coverage, accounting at most for 6.4% in *Redis*, where, as previously pointed out, the developers almost never add code and tests in the same revision.

As conjectured, we found two main causes of latent patch coverage: tests being added only after the patch was written (this was the case in *Lighttpd 2.0*, where 12 revisions which only add tests cover an additional 74 ELOC) and patch code being enabled later on. In fact, the majority of latent patch coverage in *Lighttpd 2.0*—337 lines—is obtained by 6 revisions which change no test files. Upon manual inspection, we found that the code involved was initially unused, and only later revisions added calls to it.

Latent patch coverage is important to consider in various coverage analyses. The delay of several revisions until obtaining the patch coverage can be an artefact of the development methodology, in which

Table 3.7: The median coverage and the number of revisions achieving 100% coverage for the revisions containing bug fixes. The overall metrics are included for comparison.

	Coverage (med)		Fully Covered	
Application	Overall	Fix	Overall	Fix
Lighttpd 1.4	37%	71.42%	24.0%	46.1%
Memcached	89.0%	100%	45.4%	53.3%
Redis	0%	94.1%	25.5%	44.0%
ØMQ	76.0%	55.4%	33.3%	28.0%

case it should be assimilated into the normal patch coverage. Furthermore, our results show that in most of the systems analysed, latent patch coverage is small but non-negligible.

3.4.3 *Bug analysis*

For this part of our study, we collected bug data according to the methodology presented in Section 3.1 and we limited our analysis to the four systems which lend themselves to automatic identification of bug fixes based on commit messages: *Lighttpd 1.4*, *Memcached*, *Redis* and ØMQ. The other five systems use non-specific commit messages for bug fixes, requiring an extensive manual analysis or more complex algorithms such as machine learning and natural language processing to understand the contents of a specific revision [Mur+10]. We ignored revisions which do not affect executable files, such as fixes to the build infrastructure or the documentation and then manually confirmed that the remaining revisions are indeed bug fixes [HJZ13] and further removed fixes which modify only non-executable lines (e.g. variable declarations). We thus obtained 13 fixes in *Lighttpd 1.4*, 41 fixes in *Memcached* and 22 fixes each in *Redis* and ØMQ.

We measured the patch coverage of these revisions and report the median values in Table 3.7, together with the corresponding overall metric, for comparison. For *Lighttpd 1.4*, *Memcached* and *Redis*, the coverage for fixes is higher than that for other types of patches. For *Redis*, the median value jumps from 0% to 94.1%, while for *Lighttpd 1.4*, the median value increases from 37% to 71.42%; for *Memcached* the difference is less pronounced. On the other hand, the fixes in ØMQ are covered less than on average. The fraction of fixes which have 100% coverage follows the same trend.

To try to understand whether buggy code is less thoroughly tested than the rest of the code, we started from the observation that bug-fixing revisions are usually only addressing the bug, without touching unrelated code. Because of this, we can identify the code responsible for the bugs by looking at the code which is removed or modified by bug-fixing revisions and compute its coverage in the revision before the fix. The coverage for this code is 24.4% for *Lighttpd 1.4*—slightly lower than the overall patch coverage, 72.7% for *Memcached*—roughly the same as the overall patch coverage, 65.2% for *Redis*—much larger than the overall patch coverage, and 35.8% for ØMQ—significantly lower.

While these numbers cannot be used to infer the correlation between the level of coverage and the occurrence of bugs—the sample is too small, and the bugs collected are biased by the way they

are reported—they suggest the limitations of line coverage as a testing metric, with bugs still being introduced even by patches which are fully covered by the regression test suite. Therefore, even well-tested code may contain bugs, which can manifest themselves after prolonged operation in the production environment, and runtime technologies like multi-version execution can be useful in those scenarios.

3.5 EXTERNAL BEHAVIOUR EVOLUTION

Another major part of our study is the analysis of the *external behaviour* of the application. In the context of Linux applications, we define the external behaviour of an application as its sequence of system calls, which are the primary mechanism for an application to change the state of its environment. Note that the key insight here is that we are only concerned with externally observable behaviour, and are oblivious to the way the external behaviour is generated.

Our assumption is that the changes to the external behaviour of an application are relatively small. To verify this assumption, we used our infrastructure to trace the external behaviour of two applications from our set across the selected range of revisions: *Lighttpd 1.4* and *Lighttpd 2.0*. This choice is pragmatic, other applications either ran excessively long when the system call tracing was enabled, their traces contained a lot of non-determinism which made the comparison impossible (this would not be a problem in the multi-version execution context as shown in Section 4.1.1 and Section 5.2.3), or their test suite required extensive modifications to allow for tracing.

The infrastructure used for system call tracing is conceptually similar to the one used for collecting coverage information, but there are some notable differences. We used LXC containers directly rather than through Docker to allow the use of the Linux perf tool,[21] which was used for recording the application's execution and obtaining the system call traces. These containers were controlled by an Ansible framework[22] using a custom LXC connector.

To collect the external behaviour information, we built each revision and recorded its execution while running the standard test suite. The use of perf has some advantages compared to other similar tools such as strace,[23] most notably the performance overhead, which is significantly lower as perf is implemented largely in the kernel while strace uses the ptrace interface. This minimises the potential side-effects of the tracing infrastructure on the test suite execution. The downside is the size of the recorded data which can be considerable, especially for I/O intensive applications such as *Redis*, as the perf tool uses a custom binary format which contains a large number of additional information compared to the simple textual output of strace.

All tests were executed in an LXC container based on the Ubuntu 12.04 template. To eliminate the possible sources of non-determinism, we have disabled address-space layout randomisation while running the tests (this was only for the purpose of the study, both Mx and Varan support address-space

[21] https://perf.wiki.kernel.org/index.php/Main_Page
[22] http://www.ansible.com/
[23] http://sourceforge.net/projects/strace/

layout randomisation as shown in Section 4.1.1 and Section 5.1.1). To account for the remaining non-determinism, we have repeated the tracing five times for each revision, as we did in the code coverage study (§3.4).

System call traces generated by perf were normalised and post-processed. We first split the original trace on a per-process basis so that the trace for each process ended up stored in a separate file. Moreover, we used the order in which processes were started as a basis for the naming scheme to allow comparison of the traces across different runs and versions: the main thread of the first process would be stored in the file `000-00.trace`, the first thread spawned by the main thread would be stored in the file `000-01.trace`, the main thread of the second process would use the file `001-00.trace`, *etc.*

Trace files were then post-processed by eliminating return values and arguments of system calls which are known to be non-deterministic, *i.e.* memory addresses, process identifiers, and timing information. This post-processing step might reduce the precision of our comparison, but we performed it for two different reasons: (1) many system calls have indirect arguments, accepting addresses of data structures residing in the process address space, and these addresses may differ across individual executions; (2) the return value of some system calls depends on the current system resource usage (*e.g.*, number of running processes and threads, amount of available memory) which could differ from one run to another. However, this sort of non-determinism would not show up in a multi-version execution context as explained in Section 4.1.1.

We then compared the post-processed traces of consecutive revisions using the edit distance. To distinguish the differences in the external behaviour introduced by the changes in the code from differences introduced by changes in the test suite, we have only compared consecutive revisions which do not modify tests. To determine the differences between the two revisions, we calculated the edit distance for every pair of traces between the two revisions. Since we traced each revision five times, this gives us 25 edit distance values for every pair of revisions.

The next step was to determine whether the observed differences are caused by actual changes in the code or by non-determinism. To do so, we calculated the mean and the variance for each vector of edit distance values; these were used to calculate the *index of dispersion*, which is defined as a ratio of the variance to the mean and is often used as a normalised measure of the dispersion of a probability distribution. Obtaining an index value greater than one is an indicator of over-dispersion, meaning that the variability present in the given data set is greater than what would be expected based on a given statistical model. Since the frequencies of each sequence of system calls in all possible traces are not known a priori, we must choose a frequency distribution for abnormal sequences. There are several possibilities for choosing this distribution, the simplest of which is to assume that the abnormal distribution is uniform [HB97]. We then interpret the presence of over-dispersion as "noise" due to non-determinism, and as such we discard that measurement. Otherwise, we use the mean value as the actual value for the set. Through this process, for each pair of revisions considered, we obtain a vector of edit distances where individual elements are edit distances of individual processes.

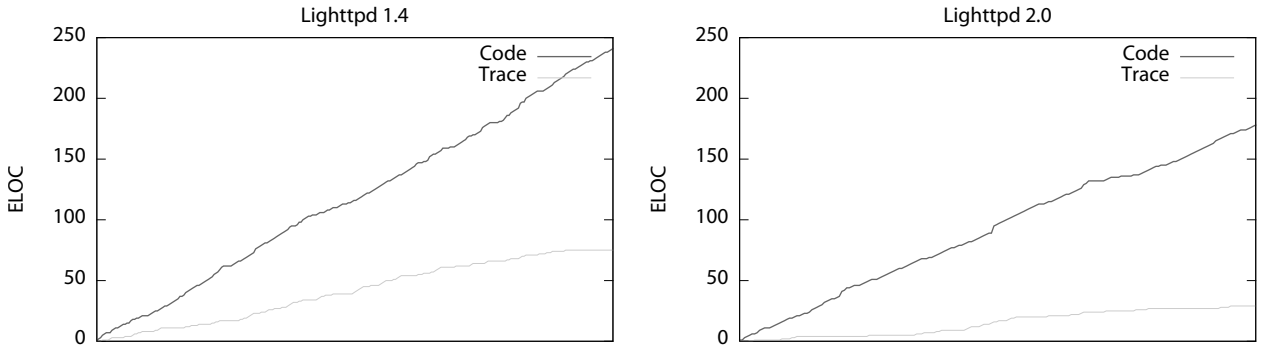


Figure 3.8: Co-evolution of executable code and external behaviour. Each increment represents a change.

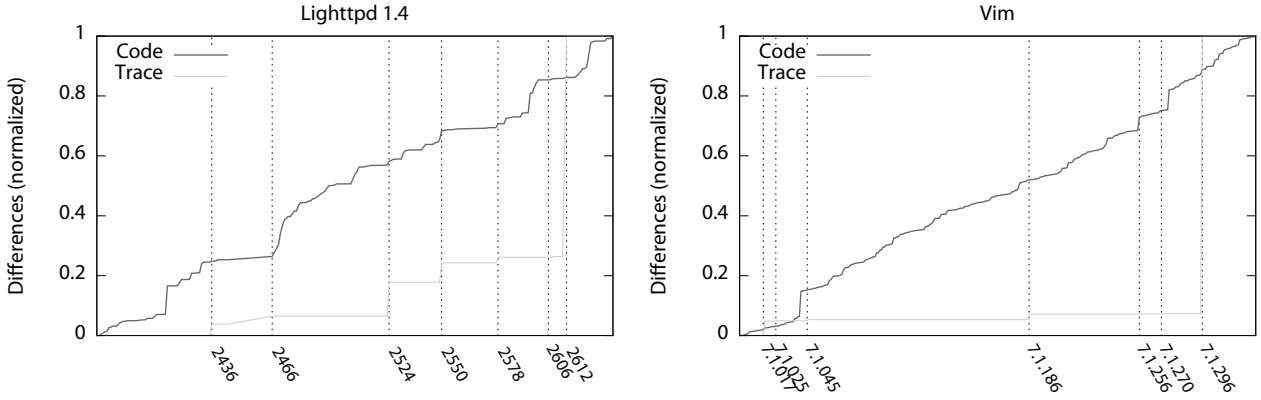


Figure 3.9: Correlation of differences in post-processed system call traces with differences in source code. The named versions are the only ones introducing external behaviour changes.

3.5.1 *Trace differences*

Our results are shown in Figure 3.8, which correlates the differences in post-processed system call traces with the source code changes. Out of 274 *Lighttpd 1.4* revisions considered, 199 introduced no changes in external behaviour, which accounts for 72.6 % of revisions considered. In case of *Lighttpd 2.0*, this number is even higher, with 180 out 209 revisions introducing no changes accounting for 86.1 % of revisions.

These results show that less than a quarter of revisions considered in our study introduced changes in external behaviour. Even more importantly, the revisions which introduced the bugs described in Section 2.1 are among the revisions which did not introduce any external behaviour changes, yet these revisions are responsible for critical crash bugs. This confirms that multi-version execution technique could be applied in these cases.

3.5.2 *Manual analysis*

To get more data points, we also did a manual analysis on a subset of revisions for two applications—*Lighttpd 1.4* and *Vim*. In the case of *Lighttpd 1.4*, we compared 164 successive revisions, namely revisions in the range 2379–2635 of branch `lighttpd-1.4.x`, which were developed and released over a span of approximately ten months, from January to October 2009. In the case of *Vim*, we compared 330 successive versions, namely versions 7.1–7.1.330, which were developed and released in approximately 13 months, from May 2007 to June 2008.

For the manual analysis, we traced the system calls made by these revisions using the `strace`[24] tool, while running a subset of the tests from the *Lighttpd 1.4* regression suite targeting the core functionality (a total of seven tests, but each test contains a large number of test cases issuing HTTP requests).

As in the case of the automated analysis, the recorded system call traces were normalised and post-processed. We split the traces on a per-process basis, and normalised all differences caused by timing, *e.g.*, we collapsed all sequences of `accept–poll` and `select–wait` system calls, which represent repeated polling operations. We have also eliminated any `brk` system calls from traces, as these calls are usually used for managing heap memory and their occurrence is often dependent on process scheduling. Finally, for the manual analysis, we also eliminated individual system call arguments and return values. This post-processing step might reduce the precision of our comparison, but we performed to make the manual analysis actually feasible. This would not be an issue in the multi-version execution context as the multi-version execution environment would intercept all sources of non-determinism (§4.1.1).

Our results are shown in Figure 3.9, which correlates the differences in post-processed system call traces with the source code changes. The graph shows that changes in externally observable behaviour occur only sporadically. In fact, 156 out of 163 revisions of *Lighttpd 1.4* and 322 out 329 revisions of *Vim* (which account for more than 95 % of all the revisions considered in both cases) introduced no changes in external behaviour.

3.6 THREATS TO VALIDITY

The main threat to validity in our study regards the generalisation of our results. The patterns we have observed in our data may not generalise to other systems, or even to other development periods for the same systems. However, we regard the selected systems to be representative for open-source C/C++ code, and the analysis period was chosen in an unbiased way, starting with the current version at the time of our experiments.

Errors in the software underlying our framework could have interfered with our experiments. Both Docker and Linux Containers were under development and not recommended for use in production systems at the time of our study. Furthermore, in the case of some applications, we have observed test failures caused by the AUFS filesystem used by Docker. However, we have thoroughly investigated these

[24] `http://sourceforge.net/projects/strace/`

failures and we believe they did not affect the results presented in our study. Given the large quantity of data that we collected from a large number of software revisions, errors in our scripts cannot be excluded.

3.7 DISCUSSION ABOUT EMPIRICAL STUDIES

Our experience has revealed two main types of challenges for conducting similar or larger studies that involve running a large number of program revisions. The first category relates to the inherent difficulty of running revisions:

1. Decentralised repositories have non-linear histories, so even defining what a revision is can be difficult, and should be done with respect to the research questions being answered. In our case, we chose a granularity at the level of commits and merges to the main branch.

2. Older revisions have undocumented dependencies on specific compiler versions, libraries, and tools. We found it critical to run each revision in a separate virtualised environment as provided by LXC or Docker, to make it easy to install the right dependencies, or adjust build scripts.

3. Some revisions do not compile. This may be due to errors introduced during development and fixed later, or due to incompatible dependencies. The execution infrastructure has to be flexible in tolerating such cases, and one needs a methodology for dealing with non-compilable revisions. In our case, we have skipped over the non-compilable revisions and incorporated their changes into the next compilable one.

4. The execution of the regression test suite is often non-deterministic; the test suite may randomly fail and some lines may be non-deterministically executed. Studies monitoring program execution need to take non-determinism into account.

The second category of challenges relates to reproducibility and performance. To address these challenges, we have built an infrastructure that enables reproducibility through the use of software containers. Performance has two different aspects: at the level of an individual revision, we have found it essential to use a form of operating system-level virtualisation (in our case, LXC and Docker), in order to minimise the time and space overhead typically associated with hardware virtualisation. Across revisions, we found it necessary to provide the ability of running our set of revisions on multiple local and cloud machines. For example, running the *Git* regression suite took in our case 26 machine days (250 revisions $\times$ 30 min/revision $\times$ 5 runs), which would have been too expensive if we used a single machine, especially since we also had to repeat some runs during our experimentation.

3.8 SUMMARY

In this chapter, we have presented a comprehensive study of the evolution of nine popular open-source systems. We believe the presented results provide evidence in support of our assumptions. We have shown that: (1) software patches are small and fairly localized, (2) changes to the external behaviour are infrequent and rather small, and (3) even well tested and fully covered code may contain bugs. Furthermore, we have revealed that intercepting sources of non-determinism (*e.g.*, time and date, random number generators or network events) is critically important for the viability of multi-version execution. The results presented in this chapter are encouraging for the applicability of our technique.

4

Multi-version Software Updates

Multi-version execution can be used to improve the reliability of updated software by running multiple different versions (revisions) in parallel, discarding one (or more) versions in case of failure. The potential problem of such an approach is that we might eventually run out versions given multiple crash bugs located at different points across versions. We tackle this problem by combining multi-version execution with a failure recovery mechanism, introduced in Section 2.2, which takes advantage of the similarity between consecutive software versions.

We implemented this approach in a prototype system called Mx, which targets crash bugs in Linux applications. Mx allows a new and an old version of an application to run concurrently, without requiring any modifications to the application itself or the operating system, nor any input from the user. To achieve this goal, Mx combines static and dynamic techniques: it uses static analysis at the binary-level to construct mappings between the old and the new versions, and synchronises the execution of the two versions at the system call level, using system call interposition and synchronisation. When one of the versions crashes, Mx transparently restarts it via a lightweight checkpointing mechanism and often allows it to survive the bug by using the code of the other version.

4.1 PROTOTYPE

We have implemented our approach in a prototype system called Mx, targeted at multi-core processors running Linux. Currently, Mx supports two versions run in parallel. The system works directly on application binaries, making it easy to deploy it and possibly integrate it with existing software package managers.

On a platform using Mx, conventional (*i.e.* unmodified) applications and multi-version (MV) applications run side by side. The key property that must hold on such a platform is that without purposely trying to do so, applications should not be able to distinguish between conventional and MV applications running on the platform. In particular, the multiple versions of an MV application should appear as one to any other entity interacting with them (*e.g.*, user, operating system, other machines). Furthermore, MV applications should be more reliable than their component versions, and their performance should not be significantly degraded.

To achieve these goals, our prototype Mx employs several different components, as shown in the architectural overview of Figure 4.1. The input to Mx consists of the binaries of two versions of an

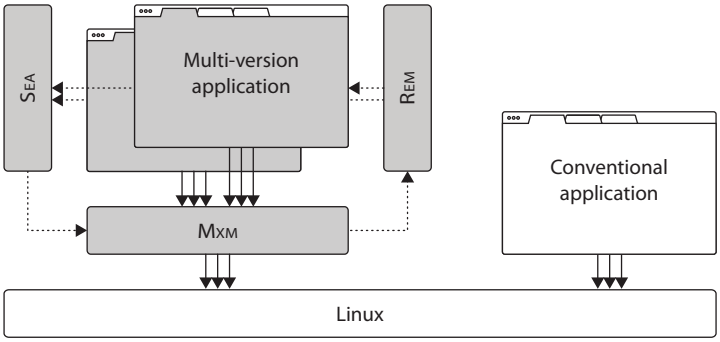


Figure 4.1: Mx system architecture.

application, which we will refer to as *the old version*—the one already running on the system, and *the new version*—the one newly released.

These two binaries are first statically analysed by the Sea (Static Executable Analyser) component, which constructs a mapping from the control flow graph (CFG) of the old version to the CFG of the new version (§4.1.3). The two versions are then passed to Mxm (Multi-eXecution Monitor), whose job is to run the two versions in parallel, synchronise their execution, virtualise their interaction with the outside environment, and detect any divergences in their external behaviour (§4.1.1). Once a divergence is detected, it is resolved by Rem (Runtime Execution Manipulator), which selects between the available behaviours, and resynchronises the two versions after the divergence (§4.1.2).

The system prototype has been implemented in C with a small amount of assembly, and the current version has approximately 28,200 LOC, with majority being the code for handling individual system calls and their arguments. The implementation currently supports Linux kernels 3.2.0 and above, running x86 and x86-64 architectures.

The rest of this section describes the main Mx system components and their implementation in more detail, and discusses how they work together to support safe software updates.

4.1.1 *System call interposition*

One of the main components of our multi-version execution environment is the Mxm monitor. Mxm's main jobs are to run the two versions concurrently, mediate their interaction with the outside world, synchronise their executions, and detect any divergences in their external behaviour. Mxm works by intercepting all system calls issued by each application version, and manipulating them to ensure that the two versions are executed in a synchronised fashion.

Mxm provides functionality similar to conventional virtual machine monitors. Whenever the MV application is executed, the Mxm connects to the two application versions running in parallel, intercepting their kernel system calls. Mxm ensures that the two versions act as one to the outside world by mediating access to the underlying operating system to achieve complete isolation of the running application from

other application instances as well as from the external environment. The environment controlled by the monitor consists mainly of a restricted file system access, socket interceptors and signal handlers.

System call interception

Mx is implemented using the ptrace interface provided by the Linux kernel. This interface, often used for application debugging, allows simple deployment (without any need for compile-time instrumentation) and makes the monitor itself lightweight since it is running as a regular unprivileged process. Mx is similar in operation to previous monitors whose goal is to synchronise applications at the level of system calls such as *Orchestra* [Sal+09], PLR [Shy+09] or *Tachyon* [MB12].

Mx runs each version in a separate child process, intercepting all their system calls. When a system call is intercepted in one version, Mx waits until the other version also performs a system call. With a pair of system calls in hand (one executed by the old version, and one by the new version), Mx compares their types and arguments. If they differ, Mx has detected a divergence and invokes the Rem component to resolve it (§4.1.2).

Otherwise, if the two versions perform the same system call with the same arguments, Mx virtualises their interaction with the environment. If the operation performed by the system call has no side effects and does not involve virtualised state (*e.g.*, `sysinfo`), Mx allows both processes to execute it independently. Otherwise, it executes the system call on their behalf and copies its results into the address spaces of both versions.

Mx must also enforce deterministic execution across versions. This consists mainly of intercepting instructions that may produce non-deterministic results, and returning the same result in both versions. Examples of such non-deterministic operations include random number generators (*e.g.*, read calls to `/dev/[u]random`), date and time (*e.g.*, read calls to `/etc/localtime`), and access to files and network (*e.g.*, file descriptor consistency). Note that non-deterministic effects resulting from allocating memory objects at different addresses in memory or randomly arranging memory areas via address space layout randomisation (ASLR) do not pose any problems: Mx understands the semantics of individual system calls and rather than directly comparing memory addresses (which might be different in each executed version), it compares the actual values stored at those memory locations. Mx supports both memory buffers (by comparing the actual buffer content) as well as data structures referenced by pointers (including nested ones).

Since Mx fully controls executing programs intercepting all their system calls, it can ensure that both programs have the same view of their environment. Whenever the monitored process makes a system call, Mx is notified twice—first before, then after the call has been executed. When a ptrace event is raised (*e.g.*, a new child process has been started), the monitor is notified as well. Due to internal limitations of the ptrace interface, once the system call has been made, it cannot be skipped, so when Mx wants to execute the call on behalf of its child processes, it simply replaces it with a system call that does not change the system state (`getpid` in our case).

There are several challenges that we encountered while implementing Mx. First, Mx must partly understand the semantics of system calls. For example, many system call parameters use complex (often

nested) structures with complicated semantics to pass values to the operating system kernel, as in the case of `ioctl` or `futex`. To be able to compare the parameters of these system calls and copy back their results, Mxm needs to understand the semantics of these structures. However, there are only a relatively small number of system calls in Linux, and once the support for handling them is implemented, it can be reused across applications. Mxm currently implements 131 system calls (out of the 311 provided by Linux x86-64 3.1.9), which was enough to allow us to run Mx on our benchmarks (§4.2).

Second, the arguments of a system call are often passed through pointers, which are only valid in the application address space, which is not directly available to Mxm. Therefore, Mxm needs to copy the contents pointed to by these structures to its own address space in order to perform their comparison. The ptrace interface on only allows to copy a single doubleword on x86 and a single quadword on x86-64 per system call, which is very expensive. Previous approaches either used various ad-hoc optimisations such as named pipes or shared memory with custom shellcode [Sal+09], or a modified kernel [MB12] to overcome this limitation. Instead, Mxm uses *cross memory attach*, a new mechanism for fast interprocess communication which has been recently added to the Linux kernel [Cor10]. This mechanism provides two new system calls—`process_vm_readv` and `process_vm_writev`—which allows the tracer to directly access the memory space of the tracee using an interface similar to the `readv` and `writev` system calls without any additional overhead.

Third, because the structures passed as arguments to system calls often have variable size, Mxm also needs a fast way to allocate and deallocate memory for them in order to minimise the overall overhead imposed by our system. For this purpose, Mxm uses a region-based memory allocator [Ros67], namely the obstack library,[1] which is part of the *GNU* C Library. Each monitored process has its own region, which is used for allocating memory to store the copy of the process' system call arguments. After the comparison of arguments between the versions, the entire region can be disposed at once.

Multi-process and multi-threaded applications

Finally, a particular challenge arises in the context of multi-process and multi-threaded applications. Using a single monitor instance to intercept both versions and their child processes (or threads) would eliminate any advantage that these applications derive from using concurrency. Therefore, Mxm uses a new monitor thread for each set of child processes (or threads) spawned by the application. For instance, if the old and new versions each have a parent and a child process, then Mxm will use two threads: one to monitor the parent processes, and one to monitor the child processes in each version.

Due to limitations of the ptrace interface (which was not designed to be used in a multi-process or multi-threaded environment), handing the control of any child processes being spawned by the application over to a new monitoring thread is somewhat complicated. In Mxm we adopt a solution similar to Orchestra [Sal+09]. When a new child process is spawned, we let the parent monitoring thread to supervise its execution until the first system call. Then, we replace this system call with a `pause` system call, disconnect the parent monitor (which causes a `SIGCONT` signal to be sent to all new child

[1] `http://www.gnu.org/software/hello/manual/libc/Obstacks.html`

processes), and spawn a new monitoring thread which immediately reconnects to the new child process, restores its original system call, and resumes its execution.

Mxm does not enforce deterministic execution across multiple versions of multi-threaded programs (which may diverge if race conditions can lead to different external behaviour across executions), although we could overcome this limitation by adopting Varan's solution (§5.2.3).

Environment virtualisation

To improve I/O performance and decrease virtualisation overhead, processes are allowed to open files with read-only permissions directly, while files with write permissions are opened by the monitor itself. This imposes another problem as file descriptors assigned to these files are not necessarily the same in each version (*e.g.*, due to scheduling non-determinism). Therefore, Mxm needs to virtualise these file descriptors.

Whenever the monitored process opens a file with read-only permissions, a new virtual file descriptor is assigned to this file together with the mapping to a real file descriptor for each version. This virtual file descriptor is then sent to each version. When a system call is made using this virtual file descriptor, Mxm replaces it with the real file descriptor before executing the system call. The actual file operation is then executed by the process itself avoiding any memory copying by Mxm.

A similar approach is also used for virtualisation of process, group, parent and child identifiers. Whenever a process tries to obtain the actual ID, Mxm replaces this with a virtual ID and keeps the mapping between the real and the virtual ID. When a process invokes a system call using this ID as an argument (*e.g.*, `kill`), the virtual ID is replaced with the actual ID before executing the system call.

4.1.2 *Runtime state manipulation*

At the core of our system lies the Rem component, which is invoked by Mxm whenever a divergence is detected. Rem has two main jobs: (1) to decide whether to resolve the divergence in favour of the old or the new version; and (2) to allow the other version to execute through the divergence and resynchronise the execution of the two versions after the divergence. As discussed in Section 2.2, in Mx we focus our attention on surviving crash errors, so the key challenge is to allow the crashing version to survive the crash. This is essential to the success of our approach, which relies on having both versions alive at all times, so that the overall application can survive any crash bugs that happen in either the old or the new version (although not in both at the same time).

We emphasise that we apply our approach only to crash errors (those raising a `SIGSEGV` signal), and not to other types of program termination, such as `abort`. This is important from a security perspective, because when a vulnerability is discovered, but a proper solution is not yet known, developers often[2] fail-stop the program rather than letting it continue and allowing the attacker to compromise the system.

[2]For example, see the patch in `json-cpp` http://jsoncpp.svn.sourceforge.net/viewvc/jsoncpp/trunk/jsoncpp/include/json/assertions.h?r1=247&r2=246&pathrev=247

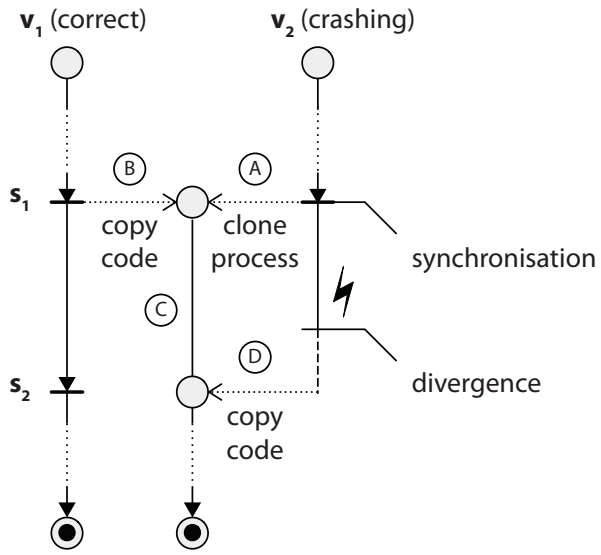


Figure 4.2: Rem's recovery mechanism uses the code of the non-crashing version to run through the buggy code.

Suppose that one of the versions has crashed between the execution of system call s_1 and the execution of system call s_2. Then, in many common scenarios, the code executed between the two system calls is responsible for the crash (*e.g.*, the old version crashes because it doesn't incorporate a bug fix present in the new version, or the new version crashes because its code was patched incorrectly). Therefore, our strategy is to do a form of *runtime code patching*, in which we use the code of the non-crashing version to execute over the buggy code in the crashing version.

Our exact recovery mechanism is illustrated in Figure 4.2. At each system call, Mx creates a lightweight checkpoint of each version. This is implemented using the `clone` system call in Linux, which internally uses a copy-on-write strategy.

As shown in Figure 4.2, suppose that the crash happens in version v_2, between system calls s_1 and s_2. Then, Rem first restores v_2 at point s_1 (Ⓐ), copies v_1's code into v_2's code segment (Ⓑ), executes over the buggy code using v_1's code (Ⓒ, note that we are still using v_2's memory state), and then restore v_2's code at point s_2 (Ⓓ).

There are several challenges in implementing this functionality. First, Rem needs the ability to read and write the application's code segment. In the current implementation, we bypass this by linking together the two application versions after renaming all the symbols in one of the versions using a modified version of the `objcopy` tool.[3] However, in the future we plan to implement this transparently by using the cross-memory attach mechanism used by Mxm.

Second, Rem needs to modify the contents of the stack in v_2. This is necessary because the return addresses on the stack frames of v_2 still point to v_2's original code, which was now replaced by v_1's code. Without also modifying v_2's stack, any function **RET** instruction executed between s_1 and s_2 would most likely veer execution to an incorrect location, since function addresses are likely to be different across

[3] `http://sourceware.org/binutils/docs/binutils/objcopy.html`

different versions. Thus, after Rem replaces v_2's code, it also updates the return addresses on v_2's stack with the corresponding return addresses in v_1, which are obtained via static analysis (§4.1.3). Because system calls are invoked via wrapper functions in C library, this ensures that when v_2 resumes execution, it will immediately return to the code in v_1. To implement this functionality, Rem makes use of the `libunwind` library,[4] which provides a portable interface for accessing the program stack, for both x86 and x86-64 architectures. To actually modify the execution stack of v_2, Rem uses again the ptrace interface.

Unfortunately, updating the stack return addresses is not sufficient to ensure that v_2 uses v_1's code between s_1 and s_2, as v_2 may also use function pointers to make function calls. To handle such cases, Rem inserts breakpoints to the first instruction of every function in v_2's original code. Then, when a breakpoint is encountered, Rem is notified via a `SIGTRAP` signal, and redirects execution to the equivalent function in v_1's code (which is obtained from the Sea component) by simply changing the instruction pointer.

Finally, after executing through the buggy code, Rem performs the same operations in reverse: it redirects execution to v_2's original code, changes the return addresses on the stack to point to v_2's functions, and disables all breakpoints inserted in v_2's code. The one additional operation that is done at this point is to copy all the global data modified by v_1's code into the corresponding locations referenced by v_2's code.

Note that Mx cannot currently handle major modifications to the layout of the data structures used by the code, including individual stack frames. While this still allows us to support several common software update scenarios, in future work we plan to improve the system with the ability to perform full stack reconstruction [MB09] and automatically infer basic data structure changes at the binary-level [Coz+08].

Our approach of using the code of the non-crashing version to survive failures in the crashing version may potentially leave the recovered version in an inconsistent state. However, Mx is able to discover most internal state inconsistencies by checking whether the two versions have the same external behaviour after recovery. When the behaviour of the recovered version starts to differ, Mx will immediately discard it and continue with only one version. The discarded version can be later restarted at a convenient synchronisation point. This restarting functionality is not currently implemented in Mx, but we plan to add it as a future extension.

4.1.3 *Binary static analysis*

The Sea component statically analyses the binaries of the two versions to obtain information needed at runtime by the Rem component. Sea is invoked only once, when the multi-version application is assembled from its component versions.

The main goal of Sea is to create several mappings from the code of one version to the code of the other. First, Sea extracts the addresses of all function symbols in one version and maps them to the

[4] `http://www.nongnu.org/libunwind/`

Table 4.1: Utilities from *GNU Coreutils*, the crash bugs used, and the versions in which these bugs were introduced and fixed. We group together utilities affected by the same or similar bugs.

Utility	Bug description	Bug span
`md5sum` `sha1sum`	Buffer underflow	v5.1 – v6.11

`mkdir`
`mkfifo`
`mknod`
	`NULL`-pointer dereference	v5.1 – v6.11

`cut`	Buffer overflow	v5.3 – v8.11

addresses of the corresponding functions in the other version. This mapping is used by Rem to handle calls performed via function pointers (§4.1.2).

Second, Sea computes a mapping from all possible return addresses in one version to the corresponding return addresses in the other version. In order to allow for code changes, this mapping is done by computing an ordered list of all possible return addresses in each function. For example, if function `foo` in v_1 performs call instructions at addresses `0xabcd0000` and `0xabcd0100`, and function `foo` in v_2 performs call instructions at addresses `0xdcba0000` and `0xdcba0400`, then Sea will compute the mapping `{0xabcd0005 → 0xdcba0005, 0xabcd0105 → 0xdcba0405}` (assuming each call instruction takes 5 bytes). This mapping is then used by Rem to rewrite return addresses on the stack.

To construct these tables, Sea first needs to extract the addresses of all function symbols and then disassemble the code for each individual function in order to locate the call instructions within them. The implementation is based on the `libbfd` and `libopcodes` libraries, which are part of the *GNU Binutils* suite.[5] To obtain the addresses of all function symbols defined by the program, Sea uses `libbfd` to extract the static and dynamic symbol tables and relocation tables. To disassemble individual functions, Sea uses the `libbf` library [Kwa12], built on top of `libopcodes`.

4.2 Evaluation

To evaluate our approach, we show that Mx can survive crash bugs in several applications. We have evaluated Mx using a set of bugs in three applications: *GNU Coreutils*, *Redis* and *Lighttpd 1.4*. We discuss each application in turn below.

GNU Coreutils

The *GNU Coreutils* utility suite[6] provides the core user-level environment on most UNIX systems. We have selected a number of bugs reported on the *Coreutils* mailing list, all of which trigger segmentation

[5] `http://www.gnu.org/software/binutils/`
[6] `http://www.gnu.org/software/coreutils/`

faults. The bugs are described in Table 4.1, together with the utilities affected by each bug and the versions in which they were introduced and fixed.

The bug affecting both `md5sum` and `sha1sum` utilities, introduced in version 5.1 and later fixed in version 6.11, caused a crash due to a buffer underflow when checking an invalid BSD-style input. Another bug we have considered affected `mkdir` , `mkfifo` and `mknod` utilities; this bug, which was reported in 6.10 and fixed in 6.11 resulted in a crash when accessing an invalid context. Finally, the bug affecting the `cut` utility, introduced in 5.3 and later fixed in 8.11, resulted in a segmentation fault when using a large unbounded range.

For all these bugs, we configured Mx to run the version that fixed the bug together with the one just before. We could have also run the version that introduced the bug with the one just before, but we could not immediately tell where the bug was introduced, and we cannot build versions earlier than 6.10 due to changes in GCC and GNU C library. Mx successfully intercepted the crash and recovered the execution by using the strategy described in Section 4.1.2.

We discuss below the bug in the `cut` utility (used to remove sections from each line of a file), triggered by the following invocation:

```
cut -c1234567890- --output-d=: foo
```

This bug is triggered by the conditional statement on line 516 in the file `src/cut.c`:

```
516 if (output_delimiter_specified
517     && !complement
518     && eol_range_start && !is_printable_field (rsi_candidate))
```

This code uses the lower bound of the size of the printable field vector; however, when calculating the size of this vector, ranges going to the end of line (*i.e.* `0-`) are not considered. The cause of this bug is an incorrectly calculated size of a dynamically allocated buffer, which is used to allocate the buffer on line 495:

```
495 printable_field = xzalloc (max_range_endpoint / CHAR_BIT + 1);
```

When Mx intercepts this bug, it uses the last checkpoint to recover the execution of the crashing version. This checkpoint is taken after the `brk` system call triggered by the `malloc` function call that allocates the buffer in function `bindtextdomain` on line 755:

```
755 bindtextdomain (PACKAGE, LOCALEDIR);
```

The recovered process uses the code of the other version to correctly calculate the size of the field vector and switches back to the original code during the allocation of this buffer, as function `xzalloc` triggers a `mmap64` system call just before executing the conditional statement on line 516, which originally triggered the bug.

Redis

Below, we describe how Mx can survive the *Redis* bug presented in Section 2.1; this bug, introduced during a code refactoring, causes *Redis* to crash when the `HMGET` command is used with the wrong type.

We are running in parallel the revision `a71f072f` (*the old version*, just before the bug was introduced) with revision `7fb16bac` (*the new version*, which contains the bug). Mx first invokes Sea to perform a static analysis of the two binaries and construct the mappings described in §4.1.3. Then, Mx invokes the Mxm monitor, which executes both versions as child processes and intercepts their system calls.

When the new version crashes after issuing the problematic `HMGET` command, Mxm intercepts the `SIGSEGV` signal which is sent to the application by the operating system. At this point, Rem starts the recovery procedure. First, Rem sends a `SIGKILL` signal to the new version to terminate it. It then restores the last checkpoint of the new version, which was taken at the point of the last invoked system call, which in this case is an `epoll_ctl` system call. Then, Rem uses the information provided by Sea to rewrite the stack of the new version, as detailed in §4.1.2. In particular, Rem replaces the return addresses of all functions in the new version with the corresponding addresses from the old version. The stack rewriting itself however is not enough. The newer version can still use function pointers, which are part of the replicated state, to invoke the original code. To prevent this situation, Rem inserts breakpoints at the beginning of all the functions in the code of the new version (to intercept indirect calls via function pointers), and then finally restores the original processor registers of the checkpointed process and restarts the execution of the (modified) new version.

Since the checkpoint was performed right after the execution of the system call `epoll_ctl`, the first thing that the code does is to return from the C library wrapper that performed this system call. This in turn will return to the corresponding code in the old version that invoked the wrapper, since all return addresses on the stack have been rewritten. From then on, the code of the old version is executed (but in the state of the new version), until the first system call is intercepted. In our example, the old and the new versions perform the same system call (and with the same arguments), so Rem concludes that the two processes have re-converged, and thus restores back the code of the new version by performing the steps above in reverse, plus the additional step of synchronising their global state (see §4.1.2). Finally, the control is handed back to the Mxm monitor, which continues to monitor the execution of the two versions.

Lighttpd 1.4

To evaluate Mx on *Lighttpd 1.4*, we have used two different crash bugs, #1601 and #2140. The first bug is the one described in detail in Section 2.1, related to the ETag and compression functionalities. As previously discussed, the crash is triggered by a very small change, which decrements the upper bound of a **`for`** loop by one. Mx successfully protects the application against this crash, and allows the new version to survive it by using the code of the old version.

The other crash bug we reproduced affects the URL rewrite functionality.[7] This is also caused by an incorrect bound in a **`for`** loop. More precisely, the loop:

```
for (k=0; k < pattern_len; k++)
```

should have been:

[7] `http://redmine.lighttpd.net/projects/lighttpd/issues/2140`

Table 4.2: The maximum distance in number of revisions, and the time span between the revisions that can be run by Mx for each bug.

Application Bug	Max distance	Time span
Lighttpd 1.4 #2169	87	2 months 2 days
Lighttpd 1.4 #2140	12	2 months 1 day
Redis #344	27	6 days

```
for (k=0; k+1 < pattern_len; k++)
```

The bug seems to have been present since the very first version added to the repository. It was reported in December 2009, and fixed one month later. As a result, we are running Mx using the last version containing the bug together with the one that fixed it. While this bug does not fit within the pattern targeted by Mx (where a newer revision introduces the bug), from a technical perspective it is equally challenging. Mx is able to successfully run the two versions in parallel, and help the old version survive the crash bug.

Both *Lighttpd 1.4* bugs #1601 and #2140 are very simple—their fix consist of a single character—yet still they made the *Lighttpd 1.4* server vulnerable to a potential attack. Mx can not only handle the crash, but also successfully recover the failing version in both cases.

4.2.1 *Ability to run distant versions*

In the previous sections, we have shown how Mx can help software survive crash bugs, by running two *consecutive* versions of an application, one which suffers from the bug, and one which does not. One important question is how far apart can be the versions run by Mx. To answer this question, we determined for each of the bugs discussed above the most distant revisions that can be run together to survive the bug as shown in Table 4.2.

For the *Coreutils* benchmarks, we are able to run versions which are hundreds of revisions apart: 1124 revisions (corresponding to over one year and seven months of development time) for the `md5sum /sha1sum` bug; 2937 revisions (over four years of development time) for the `mkdir /mkfifo /mknod` bug; and 1201 revisions (over two years and three months of development time) for the `cut` bug.

The most distant versions for the first *Lighttpd 1.4* bug are approximately two months apart and have 87 revisions in-between, while the most distant versions for the second *Lighttpd 1.4* bug are also approximately two months apart but have only 12 revisions in-between. Finally, the most distant versions for the *Redis* bug are 27 revisions and 6 days apart.

Of course, it is difficult to draw any general conclusions from only this small number of data points. Instead, we focus on understanding the reasons why Mx could not run farther apart versions for the bugs in *Lighttpd 1.4* and *Redis* (we ignore *Coreutils*, for which we can run very distant versions). For *Lighttpd 1.4* issue #2169, the lower bound is defined by a revision in which a pair of `geteuid` and `getegid` calls are replaced with a single call to `issetugid` to allow *Lighttpd 1.4* to start for a non-root user with

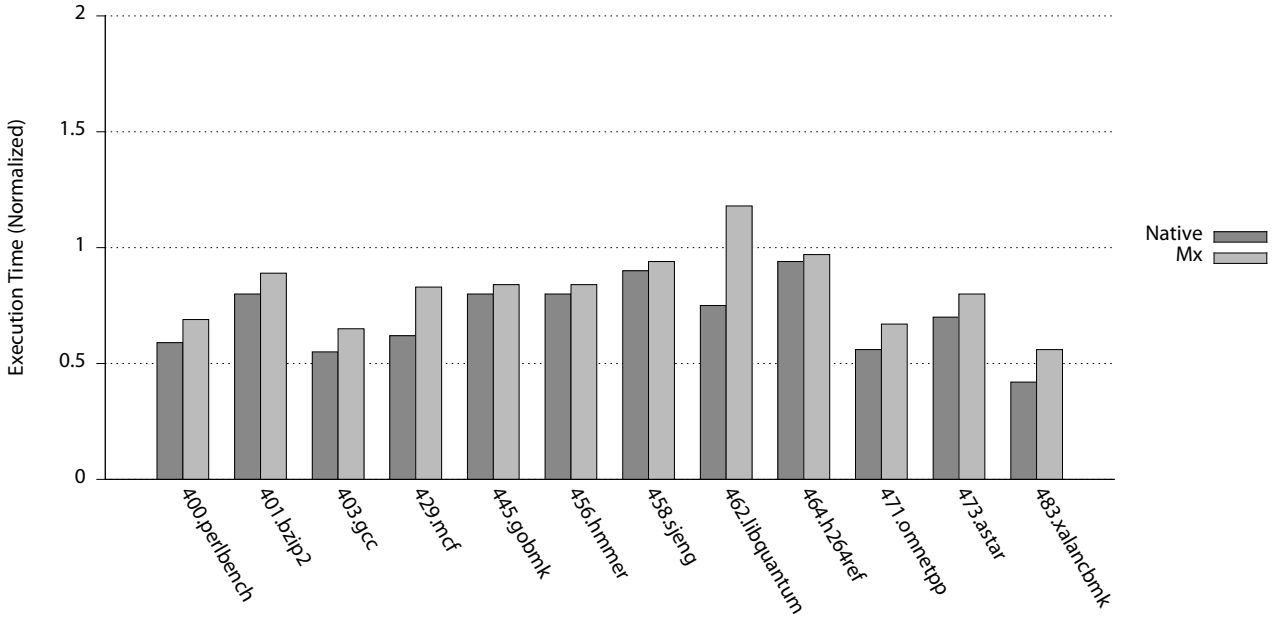


Figure 4.3: Normalised execution times for the *SPEC CPU2006* benchmark suite running under Mx.

GID `0`. Mx currently does not support changes to the order of system calls, but this limitation could be overcome by using the rewrite rules for system call sequences supported by Varan (§5.2.4). This would allow Mx to recognise that the pair `geteuid` and `getegid` could be matched with the call to `issetugid` (§5.4).

The upper bound for *Lighttpd 1.4* issue `#2169` adds a `read` call to `/dev/[u]random`, in order to provide a better entropy source for generating HTTP cookies. This additional `read` call changed the sequence of system calls, which Mx cannot handle, but which again could be handled by Varan's rewriting rules.

For *Lighttpd 1.4* issue `#2140`, both the lower and the upper bounds are caused by a change in a sequence of `read` system calls. We believe this could be optimised by allowing Mx to recognise when two sequences of read system calls are used to perform the same overall read.

For the *Redis* bug, the lower bound is given by the revision in which the `HMGET` command was first implemented. Since there was no support for `HMGET` before that version, Mx has no way to survive the crash caused by invoking `HMGET` with a wrong type (see §4.2). The upper bound is defined by a revision which changes the way error responses are being constructed and reported, which results in a different sequence of system calls.

4.2.2 *Performance overhead*

We ran our experiments on a four-core server with 3.50 GHz Intel Xeon E3-1280 and 16 GB of RAM running 64-bit Linux v3.1.9.

SPEC CPU2006 To measure the performance overhead of our prototype, we first used the standard *SPEC CPU2006*[8] benchmark suite. Figure 4.3 shows the performance of Mx running two instances of the same application in parallel, compared to a native system. The execution time overhead of Mx varies from 3.43 % to 105.16 % compared to executing just a single version, with the geometric mean across all 29 benchmarks at 17.91 %. This result is comparable with previous work using multi-variant execution that used *SPEC CPU* to measure performance [Sal+09] (even though that work used *SPEC CPU2000*).

GNU Coreutils The six *GNU Coreutils* applications discussed in §4.2 are mostly used in an interactive fashion via the command-line interface (CLI). In this context, a high performance overhead is acceptable as long as it is not perceptible to the user; prior studies have shown that response times of less than 100 ms typically feel instantaneous [CMN86]. In many common use cases (*e.g.*, creating a directory, or using `cut` on a small text file), the overhead of Mx was imperceptible—creating a directory takes around 1 ms natively and 4 ms with Mx. For the three utilities that process files, we calculated the maximum file size for which the response time with Mx stays under the 100ms threshold. For `cut` , the maximum file size is 1.10MB (with an overhead of 14.08×), for `md5sum` 1.25MB (16.23× overhead), and for `sha1sum` 1.22MB (12.00× overhead).

Lighttpd 1.4 We used the `http_load` [9] multiprocessing test client that is also used by the *Lighttpd 1.4* developers. This benchmarks runs multiple fetches in parallel, to test the throughput of a web server. To measure the performance overhead, we performed 10,000 fetches using 5 parallel connections serving 1000 files, and we used three different file sizes—1 kB, 10 kB and 100 kB. fetches When run on top of Mx, the overhead for *Lighttpd 1.4* ranges from 2.60× to 3.49× depending on the file size. This result is comparable to existing ptrace-based multi-variant systems that used *Lighttpd 1.4* to measure performance [MB12].

Redis To measure the performance overhead for *Redis*, we used the `redis-benchmark` [10] utility, which is part of the standard *Redis* distribution and simulates different types of operations supported by *Redis* being issued concurrently by N clients. We used the default workload of 50 clients issuing 10,000 requests and calculated the average overhead across all commands. The overhead for *Redis* varies, depending on the operation being performed, from 3.74× to 16.72×.

Given the results presented hereinbefore, it is clear that while Mx might be suitable for interactive applications such as *GNU Coreutils* or *Vim* where the increased performance overhead does not degrade the user experience or CPU-bound compute kernels such as *SPEC CPU2006*, it is clearly unsuitable for high-performance, heavily I/O-bound server appications such as *Lighttpd 1.4* or *Redis*. This is what motivated us to design the high-performance event-streaming mechanism of VARAN, which is a more practical and more readily deployable solution.

[8] `http://www.spec.org/cpu2006/`
[9] `http://www.acme.com/software/http_load/`
[10] `http://redis.io/topics/benchmarks`

4.3 DISCUSSION

This section discusses in more detail the scope of our approach, its limitation and the different trade-offs involved.

CPU UTILISATION AND MEMORY CONSUMPTION Mx incurs a performance overhead, as discussed in §4.2.2. In our experience, Mx is applicable to interactive applications such as command-line utilities and interactive applications, where the performance degradation is not noticeable to the user. Mx is not applicable to servers requiring high-throughput or to patches that fix performance bugs, as the system runs no faster than the slowest version. Both of of these issues are addressed by VARAN.

Our approach of using idle CPU time to run additional versions also increases energy consumption and therefore might not be applicable to energy-constrained devices such as smartphones. However, it is interesting to note that idle CPUs are not "free" either: even without considering the initial cost of purchasing the, an energy-efficient server consumes half its full power when doing virtually no work—and for other servers, this ratio is usually much worse [BH07].

Similarly, Mx also doubles the memory consumption, which in practice is less of an issue but could still lead to a performance degradation due to increased memory pressure for some applications.

MEMORY-BASED COMMUNICATION. The current Mx prototype does not support shared memory, which is often used for efficient communication between processes, as its use would be invisible to our system call interposition mechanism. One way we could handle this scenario is by serialising all accesses to shared memory using using a repeatable deterministic task scheduler [RC96]. This approach has been used by Paranoid Android [Por+10] in the same context.

DEPLOYMENT STRATEGY While our approach eases the decision of applying a software update—as incorporating a new version would never decrease the reliability of the multi-version application—the number of versions that can be run in parallel is limited, being dictated by the number of available resources (*e.g.*, the number of available CPU cores). As a result, we need a deployment strategy to decide what versions to use. For example, we could always run the last N released versions (where N is the number of available resources), or we could always keep a one-year old version, etc. This thesis focuses on designing and implementing multi-version execution techniques, but in future work we plan to explore deployment strategies in more detail.

4.4 SUMMARY

Software updates are an important part of the software development and maintenance process. Unfortunately, they also present a high failure risk, and many users refuse to upgrade their software, relying instead on outdated versions, which often leave them exposed to known software bugs and security vulnerabilities.

In this chapter, we have presented Mx, a multi-version execution system for improving the software update process. Whenever a new program update becomes available, instead of upgrading the software to the newest version, we run the new version in parallel with the old one, and carefully synchronise their execution to create a more secure and reliable multi-version application.

Mx supports off-the-shelf Linux applications and our evaluation has shown that it can be applied successfully to several real applications such as *GNU Coreutils*, *Lighttpd 1.4*, and *Redis*.

5

Efficient Multi-version Execution

The main challenges in implementing a system call monitor relate to performance, flexibility and ease of debugging. Many implementations, including Mx, use the ptrace mechanism offered by most UNIX-based operating systems [Sal+09; HC13; BCL07]. While ptrace has its advantages as shown in Chapter 4, namely ease-of-use and not requiring kernel modifications, ptrace introduces large overhead, and these systems see performance degradations of up to two orders of magnitude. A much faster approach is to implement the monitor in kernel space [Cox+06], but this requires kernel patches and/or new kernel modules, and the monitor must be run in privileged mode. Furthermore, none of these approaches scales well with the number of variants (as the monitor is both a communication and synchronisation bottleneck), none are debug-friendly (ptrace disallows the use of *GDB*, while kernel debugging has its well-known set of limitations) and none of them have been designed to be flexible with respect to small variations in system call sequences (which sometimes occur across program versions as shown in Section 3.5).

In this chapter, we propose VARAN,[1] a novel architecture for implementing multi-version monitor. VARAN monitor operate at the system call level, run in user space (and therefore in unprivileged mode), introduce a small performance overhead for popular C10k network servers and often a negligible overhead for CPU-bound applications, scale well with the number of versions, and provide a flexible mechanism for handling small divergences in the system call sequences issued across versions.

5.1 OVERVIEW

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

5.1.1 *System call interception*

The biggest downside of existing system call monitors based on the ptrace interface is the high performance overhead [Sal+09; MB12]. For each system call performed by each version, execution must switch to the monitor process, which has to perform several additional system calls in order to

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

copy buffers to and from the version being monitored, nullify the system call, *etc.* as shown earlier in Section 4.1.1.

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

Whenever code is loaded into memory, VARAN scans each code page to selectively rewrite all system calls with jump instructions to dedicated handlers. Section 5.2.2 discusses in detail the main steps and challenges associated with this binary rewriting approach.

To eliminate the need for additional system calls during interception, VARAN uses a shared ring buffer to communicate between versions. This ring buffer is heavily optimised for performance: it is stored in memory, allows largely lock-free communication, and does not require the dispatch of events to different queues. These aspects are discussed in detail in Section 5.2.3.

5.1.2 *Event-streaming architecture*

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

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

Second, this approach is totally inflexible to any divergence in the sequence of system calls executed across versions. This is an issue when running different software versions, where changes in the sequences of system calls can occur between versions.

To address these limitations, VARAN uses a new approach which we call *event streaming*. In this decentralised architecture, depicted in Figure 5.1, one of the versions is designated as the *leader*, while the others are *followers*. During execution, the leader records all events into a shared ring buffer, which are later read by followers to mimic the leader's external behaviour (§5.2.3). Events consist primarily of regular system call invocations, but also of signals, process forks (*i.e.* `clone` and `fork` system calls) and exits (*i.e.* `exit` and `exit_group` system calls).

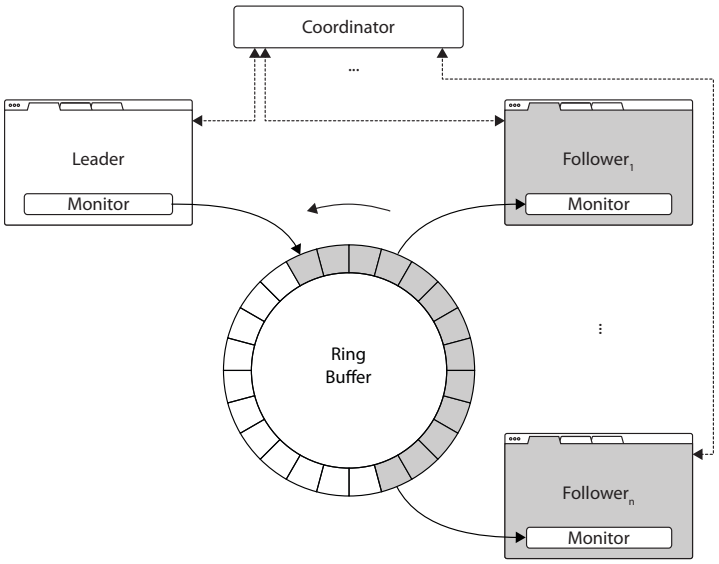


Figure 5.1: The event-streaming architecture of VARAN.

In general, any version can be the leader, although in some situations some may be a better choice than others, *e.g.*, when running multiple software revisions in parallel, one might prefer to designate the newest one as leader. However, the leader can be easily replaced if necessary, *e.g.*, if it crashes (§5.2.3).

The only centralised component in this architecture is the *coordinator*, whose main job is to prepare the versions for execution and establish the necessary communication channels. At a high level, the coordinator first loads the variants into memory, injects several special handlers and memory objects into their address spaces, rewrites any system calls in their code with jumps to the special handlers and then starts executing the variants (§5.2.1) in a decentralised manner.

5.1.3 *Rewrite rules for system call sequences*

In addition to eliminating the central monitor bottleneck, our event-streaming architecture also supports (small) divergences between the system call sequences of different variants. In particular, different software versions can be run inside a system like Mx only as long as they all issue the same sequence of system calls as we showed in Section 4.2.1. However, software patches sometimes change the external behavior of an application. In particular, many divergences in system call traces fall into the following two categories: (i) *addition/removal*, characterising situations when one of the versions performs (or conversely does not perform) an additional system call, typically as a consequence of an additional check, and (ii) *coalescing*, covering the situations when a (repeated) sequence of system calls is executed a different number of times in each version (*e.g.*, one version might execute two `write` system calls, while another version executes only one `write` system call to write the same bytes because extra buffering is used).

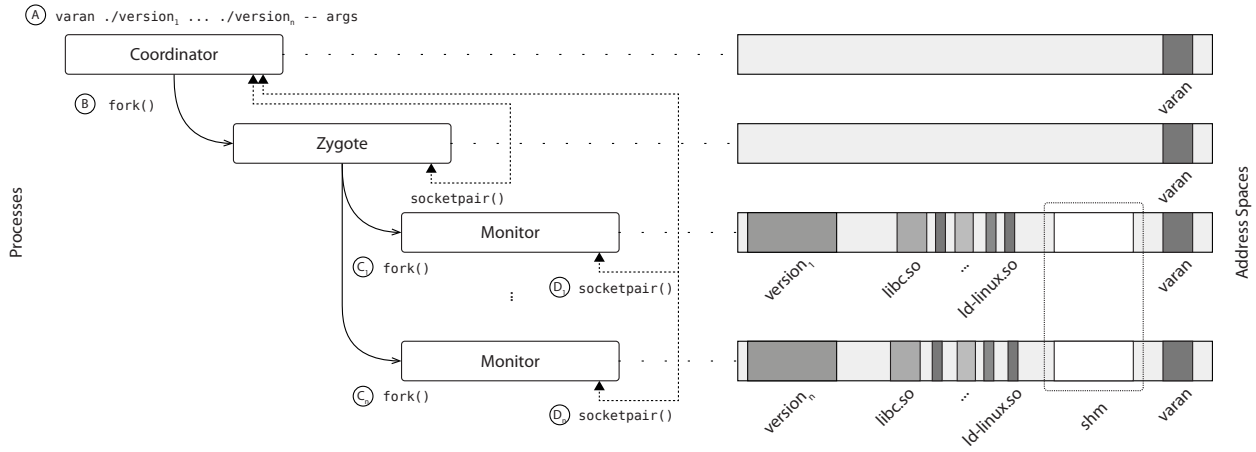


Figure 5.2: Setup of address spaces and communication channels.

Varan is the first NVX system that is able to deal with such changes. When followers process the event sequence streamed by the leader, they can rewrite it to account for any such differences: *e.g.*, they can skip and merge system calls, or perform some calls themselves. We provide a flexible implementation of such rewrite rules using Berkeley Packet Filters (§5.2.4).

5.2 Prototype

We have implemented our approach in a prototype, targeted at multi-core x86 and x86-64 processors running Linux. The system has been implemented in C and Assembly and the current prototype has approximately 37,100 LOC. Varan works on off-the-shelf binaries (both stripped and unstripped) and supports single- as well as multi-threaded applications.

When it starts, Varan first sets up the address spaces of all program versions and establishes the needed communication channels (§5.2.1). It then performs selective binary rewriting to replace all system calls with jump instructions (§5.2.2). After these initial stages, the event streamer component of Varan ensures the coordination of the leader and its followers (§5.2.3).

5.2.1 *Setup of address spaces and communication channels*

The main steps involved in the setup of version address spaces and the needed communication channels are shown in Figure 5.2. To run multiple versions in parallel, the user launches Varan's *coordinator* providing the paths to all versions, together with any command line arguments required to start them (Step Ⓐ in Figure 5.2).

The *coordinator* first creates the shared memory segment used for communication among versions, and then spawns the *zygote* process (Ⓑ), which is responsible for starting the individual versions. The coordinator communicates with the zygote via a UNIX domain socket. For each version *i* that needs to

be spawned, the coordinator sends a fork request to the zygote over this socket pair, which includes the path to that version executable, the command line arguments, and the end-point of a socket pair which will be used for the subsequent communication between the coordinator and that version (Ⓒ$_i$). After receiving this request, the zygote spawns a new process, which first finalises the communication with the coordinator (Ⓓ$_i$). The coordinator then sends the shared memory segment descriptor to this process, which maps it inside its address space.

In the final step, the new process starts executing inside the monitor code, which loads the specified ELF executable and sets up the initial address space as described in the ELF headers. If the program requires a dynamic linker, VARAN loads the linker image specified in the header as well. The text segments of both the application and the dynamic linker are then processed by the binary rewriter (§5.2.2). Finally, VARAN jumps to the application entry point as specified in the ELF header, starting the execution of the application version.

The right-hand side of Figure 5.2 shows the address spaces of the coordinator, zygote, and program versions. When run with VARAN, program versions have two new segments mapped into their address spaces: the shared memory segment used for communication among versions ("shm") and the VARAN statically-linked library ("varan"). Note that VARAN does not prevent address-space layout randomisation schemes to be used by the operating system, because the VARAN library is compiled as position independent code (PIC).

Coordinator

To set up the address spaces of the versions, the coordinator acts as a specialised preloader, inspired by *rtldi*.[2] However, the coordinator does not attempt to replace the existing dynamic linker, which would be unnecessarily complex and may affect compatibility with existing applications. Instead, it simply intercepts the system calls performed by the linker to enable the binary rewriter (§5.2.2) to rewrite the code of dynamically-linked shared libraries.

One important advantage of our interception mechanism is that we do not make use of ptrace to intercept calls to the dynamic linker—instead, the binary rewriter is used to rewrite all the system calls done by the linker with jumps into the coordinator code. As a result, VARAN can be used in combination with existing ptrace-based tools such as *GDB* or strace, which greatly simplifies debugging.

This architecture has several advantages over other possible approaches. VARAN does not require the use of dynamic linker, as required by `LD_PRELOAD`, and supports even statically-linked applications. This also has the advantage of supporting arbitrary dynamic linkers, not just the one provided by *GNU* C library, albeit it is going to be the most common target.

Zygote

The role of the zygote is to spawn new processes on request from the coordinator. Zygote processes are already used in systems such as *Android* and *Chrome* [Chr]—in this paper, we use the term to refer to the architectural pattern rather than a particular implementation, as VARAN provides its own clean-slate

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

implementation. While it would be technically possible for the coordinator to create the processes in which versions run, this would bring some complications regarding the communication channels: for example, the second version spawned would inherit the communication channel between the first version and the coordinator, which would be undesirable.

Monitor

The monitor code is built as a statically-linked, position-independent library, to make sure it does not stand in the way of any segments which have to be loaded by the application at fixed addresses. To ensure that the code can be compiled like this, we must avoid using any global state (*i.e.* any data that would be placed in the ELF `.data` section). One consequence is that Varan cannot use any of the existing C libraries such as *GNU C Library*, as these are not typically built to support this requirement. Instead, Varan provides its own implementation of the necessary C library functions based on the *Bionic C library*.[3] To support the use of Linux system calls, Varan uses a modified version of the `linux_syscall_support.h` header.[4]

5.2.2 *Selective binary rewriting*

To intercept system calls, Varan uses selective binary rewriting [Nan+06]. Unlike traditional dynamic binary rewriting implemented by tools like DynamoRIO [KBA02] or Pin [Luk+05], where the entire process image is being rewritten, often introducing a significant performance overhead, Varan only replaces the instructions for performing system calls (*i.e.* `int $0x80` on x86 and `syscall` on x86-64).

The rewriting itself is done when a segment is mapped into memory with executable permissions, or an existing memory segment is marked as executable. During rewriting, Varan scans the segment searching for system call instructions using a simple x86 disassembler. Every system call found is rewritten with a jump to an internal system call entry point. This process is complicated by the fact that while a system call instruction is only one byte long, a jump instruction requires five bytes. Therefore, in order to rewrite the system call with a jump, we also need to relocate some of the instructions surrounding the system call—*i.e.* perform binary detouring via trampolines [HB99]. On the rare occasions when this is not possible (*e.g.*, because the surrounding instructions are potential branch targets), we replace the system call with an interrupt (*i.e.* `INT 0x0`) which has the same size as system call instruction (*i.e.* `INT 0x80`) on x86-32 and `SYSCALL` on x86-64). This interrupt is handled by Varan through a signal handler installed during initialisation, which redirects the control flow to the system call entry point as for other system calls.

The system call entry point first saves all registers, and then consults an internal system call table to check whether there is a handler installed for that particular system call; if so, it calls that handler, otherwise it invokes the default handler. After processing the system call, the entry point handler restores all registers and returns to the original caller (using `sigreturn` in the case of system calls intercepted

[3] `https://android.googlesource.com/platform/bionic`
[4] `https://code.google.com/p/linux-syscall-support/`

via an interrupt). The system call entry point also implements support for restarting system calls (*i.e.* signaled by the `-ERESTARTSYS` error code). This is used in certain scenarios supported by VARAN including transparent failover (§5.4).

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

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

Virtual system calls

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

The *vsyscall* page was introduced first, but is being deprecated in favor of the *vDSO* segment.[5] The main reason for this development is that the *vsyscall* page is mapped to a fixed address, making it susceptible to return-oriented programming attacks [Roe+12]. To address this issue, the vDSO segment is mapped to a random address. Since the segment is dynamically allocated, it can also support an arbitrary number of virtual system calls (currently `clock_gettime`, `getcpu`, `gettimeofday` and `time`).

On x86-64 system, Linux would map a vDSO segment into each process' address space. This segment contains a fully formed ELF image with function symbols prefixed with `__vdso_`, one for each virtual system call. Applications can then call these functions and avoid performing a full context switch into kernel associated with a regular systems call. However, in practice applications are more likely to use a wrapper provided by their C library such as *GNU* C library, which checks whether the vDSO page is available and if so uses the virtual system call, or alternatively falls back to the regular system call implementation.

Virtual system calls represents one of the major limitations of ptrace-based monitors. Since these system calls are entirely implemented in user space, they cannot be intercepted via ptrace. This is an important limitation: as these system calls provide access to timing information, they are often used as a source of non-determinism (*e.g.*, for random number generators) and their handling is therefore important to ensure deterministic execution across versions. Previous systems either omit discussion on virtual system call handling [HC13; Sal+11] or explicitly mention their inability to handle them [MB12].

[5] We are referring to the x86-64 version, on x86 systems the vDSO segment contains a function used to determine the preferred method of making a system call.

Handling calls made via the *vsyscall* page is easier because the function symbols are always mapped to the same address. To handle *vDSO* calls, we first need to determine the base address of the *vDSO* segment; this address is passed by the kernel in the ELF auxiliary vector via the `AT_SYSINFO_EHDR` flag.[6] Second, we need to examine the ELF headers of the *vDSO* segment to find all symbols. Identifying calls to these symbols is more complicated than in the *vsyscall* case because these symbols are allocated at arbitrary addresses. Instead, we replace the entry point of each function with a jump to dynamically generated code which sets up the stack and then issues a call to the VARAN system call entry point as in the case of regular system calls. Furthermore, we provide a trampoline, which allows the invocation of the original function, by moving the first few instructions of each function to a new place, followed by a jump to the original code. This allows VARAN to take advantage of the virtual system call mechanism to further improve performance.

5.2.3 *Event streaming*

As we discussed briefly in Section 5.1 and illustrated graphically in Figure 5.1, the leader records all external events into a shared ring buffer, while the followers replay them to mimic the leader's behavior. The leader is the only version interacting with the environment, *i.e.* executing the system calls, with the exception of system calls which are local to the process (*e.g.*, `mmap`).

As in MX (§4.1.1), VARAN has to be aware of the system call semantics, in order to transfer the arguments and results of each system call. VARAN currently implements 86 system calls, which were all the system calls encountered across our benchmarks.[7]

Shared ring buffer

For fast communication, the leader and its followers share a common ring buffer of fixed size, which is held entirely in memory. Our initial solution used a separate shared queue for each process [GMV08; LBC10], with the coordinator acting as an event pump—reading events from the leader's queue and dispatching them into followers' queues. This approach worked well for a low system call rate, but at higher rates the event pump quickly became a bottleneck.

As a result, we have instead opted for a design based on the Disruptor pattern [Tho+11], which uses a shared ring buffer allowing concurrent access by multiple producers and consumers, eliminating the need to dispatch events among queues, and thus improving both performance and memory consumption. Our implementation uses C11 atomics, in combination with cache aligning to achieve maximum performance with minimal use of locking (locks are used only during memory allocation and deallocation).

The size of the ring VARAN uses is configurable and has a default value of 256 events. Each event has a fixed size of 64 bytes; the size has been deliberately chosen to fit into a single cache line on modern x86

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

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

CPUs. This is sufficient for sending signals and system calls for which all arguments are passed by value (on x86-64, a system call can have up to six arguments of eight bytes, to fit into general purpose registers). However, for system call arguments passed by reference, the payload might have variable size and can be potentially larger than the event itself. In this case, we use events only to transfer shared pointers, which identify memory shared across versions.

The use of a shared memory buffer may result in a waste of system resources when the leader process performs a system call which blocks for a long period of time, if the followers used busy waiting to check for new events. To address this problem, we have introduced the concept of a *waitlock*. Whenever a follower makes a blocking system call, it acquires the waitlock. If there is no event available, the thread will block until the leader wakes up and notifies it. The waitlocks are efficiently implemented using a combination of C11 atomics and futexes [FRK02].

Transferring file descriptors and leader replacement

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

Multi-process and multi-threaded applications

Handling processes and threads is crucial in supporting many modern applications. We have opted to have separate ring buffers for each tuple of processes or threads in the system: for instance, when a process forks, the parent processes in the leader and all followers form one tuple, and the child processes another, with a process in each tuple acting as the leader. More exactly, when a new process is forked, a new socket pair is established between the process and the coordinator and a new ring buffer is allocated. The leader then continues execution, but the coordinator waits until all followers fork a new process, establishing appropriate socket pairs for communication, and setting the child processes to read events from the newly-allocated ring buffer.

To alleviate non-determinism issues due to scheduling, VARAN enforces system call ordering across all tuples using Lamport's *happens-before* relation [Lam78]. Currently, this is only implemented for multi-threaded applications, which make intensive use of synchronisation primitives, but the same solution could be employed for multi-process applications too.

Each variant has an internal Lamport clock, shared by all threads, and each event e_i sent through the ring buffer is annotated with a timestamp $C(e_i)$. Then, when replaying events from the buffer, each thread checks the timestamp of every new event and only receives the event if it does not violate the happens-before relation. This scenario is depicted in Figure 5.3. If $e_1 \rightarrow e_2$ (e_1 happens before e_2), then $C(e_1) < C(e_2)$ and VARAN enforces $e'_1 \rightarrow e'_2$. Without the ordering, there could be a situation where

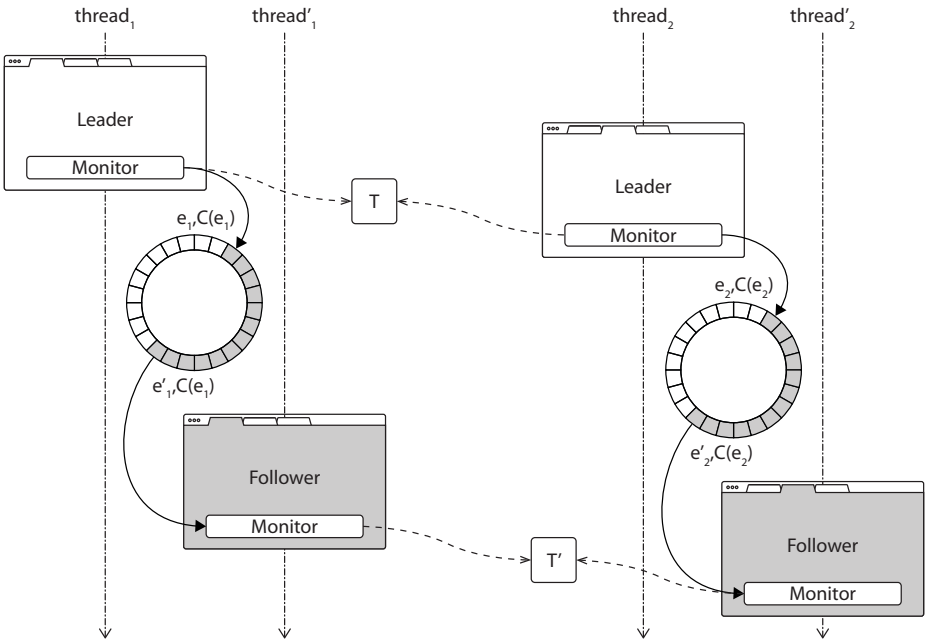


Figure 5.3: Event delivery in a multi-threaded NVX program, with the ordering of events enforced using logical clocks.

$e_1 \rightarrow e_2$, but $e'_1 \not\rightarrow e'_2$, which could lead to a divergence. A similar approach has been proposed in the past for record-replay in shared-memory systems [LAV94].

To implement the internal clocks shared by the threads of a variant (T and T' in Figure 5.3), we use an atomic counter allocated in the shared memory space and updated using C11 atomics for efficiency. When the leader thread writes a new event into the ring buffer, it increments its variant's clock value and attaches it to the event. When a follower thread reads an event from the ring buffer, it compares its variant's clock value with the event's timestamp. If they are equal, the thread increments its variant's clock value and processes the event, otherwise it continues waiting. Our current implementation uses busy waiting, as the wait times are expected to be small. However, shall this become a problem in the future, it is possible to use blocking wait instead (*e.g.*, a futex).

Our solution resembles existing deterministic multi-threading (DMT) mechanisms [Ber+10; LCB11]. The guarantees provided by Varan are weaker than those typically provided by these systems as we do not enforce ordering across atomics-based synchronisation primitives. We have not detected any system call divergences caused by related data races in our benchmarks, which include multi-threaded applications (*e.g.*, *Redis*), similar to the experience reported for prior NVX systems. However, shall this become a problem, we could address it by employing a stronger form of determinism similar to existing DMT systems.

Memory allocation scheme

Efficient shared memory allocation plays an important role in a system like Varan. We use a custom shared memory pool allocator implementation. The allocator has the notion of buckets for different

allocation sizes, where each bucket holds a list of segments, and each segment is divided into chunks of the same size; each bucket holds a free list of chunks. When there are no more unused chunks in a bucket, the allocator requests a new segment from the memory pool, and divides it into chunks which are then added to the free list. Each bucket also has a lock associated with it which has to be held prior to an allocation from that bucket.

5.2.4 *Rewrite rules for system call sequences*

Varan uses Berkeley Packet Filters (BPF) [MJ93] to implement the system call rewrite rules introduced in Section 5.1.3. BPF is a machine language for writing rules and an interpreter shipped with many UNIX implementations, including Linux and BSD. BPF filters have been traditionally used to filter network packets, but recently also for system call filtering as a part of seccomp "mode 2" (also known as seccomp/bpf).

We have integrated a BPF interpreter in Varan to allow for system call rewrite rules. Our implementation is based on the Linux kernel code which was ported to user space and extended for NVX execution. Varan provides BPF extensions on top of the instruction set used by seccomp/bpf.[8] The `event` extension allows access to the event stream, which can be used to compare the system calls executed across versions, as we will show in Section 5.4.1.

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

We expect that these rules will be written by the application developer and distributed alongside the new version. Alternatively, they could be also written by administrators after deployment when a divergence is detected. For the prototype implementation, we have opted for the classic BPF (cBPF) filters, which use an Assembly-like instruction set, as they are easier to implement and are already used by seccomp-bpf [Cor12]. However, they could be also tedious to write and have limited expressivity. Hence, in the future we plan to replace the cBPF with extended BPF (eBPF) [Cor14], which allow filters to be written in C and translated into eBPF instructions using Clang/LLVM.

5.3 Performance evaluation

One of the main contributions of Varan is a significantly lower performance overhead compared to existing state-of-the-art NVX systems. Therefore, we have conducted an extensive performance evaluation, using microbenchmarks (§5.3.1), high-performance C10k servers (§5.3.2) and applications used to evaluate prior NVX systems (§5.3.3).

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

The benchmarks were run on a four-core/eight-thread machine with a 3.50 GHz Intel Xeon E3-1280 CPU and 16 GB RAM running 64-bit Ubuntu 14.04 LTS. Both the server and client application on the same machine using the loopback device to ensure that CPU is fully saturated.

5.3.1 *Microbenchmarks*

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

1. `close(-1)` is representative of an inexpensive system call, which returns immediately.
2. `write(DEV_NULL, ..., 512)` is representative of system calls which involve expensive I/O, but whose result can be sent entirely as a single event in the ring buffer.
3. `read(DEV_NULL, ..., 512)` is representative of system calls which involve expensive I/O, and whose result cannot be fully included in the associated event in the ring buffer. Instead, it has to be copied via additional shared memory (§5.2.3).
4. `open("/dev/null", O_RDONLY)` is representative of system calls that require transferring file descriptors (§5.2.3).
5. `time(NULL)` is a virtual system call implemented via the *vDSO* segment (§5.2.2). It internally calls `__vdso_time` (since glibc 2.15). We could not measure the overhead of using the *vsyscall* page, because it is deprecated on our system (and all recent versions of Linux), with all *vsyscalls* now redirected to their *syscall* versions.

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

Figure 5.4 shows the results. The first set of bars labeled *native* shows the execution time without VARAN. The second set of bars labeled *intercept* shows the execution time with interception, measuring the cost of binary rewriting: for these experiments, the intercepted system call is immediately executed, without any additional processing. As it can be seen, the interception cost is small, at under 15 % in all cases except for `time`. The overhead of intercepting virtual system calls is high in relative terms, but low in absolute ones: 122 cycles vs 49 cycles for native execution for `time`.

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

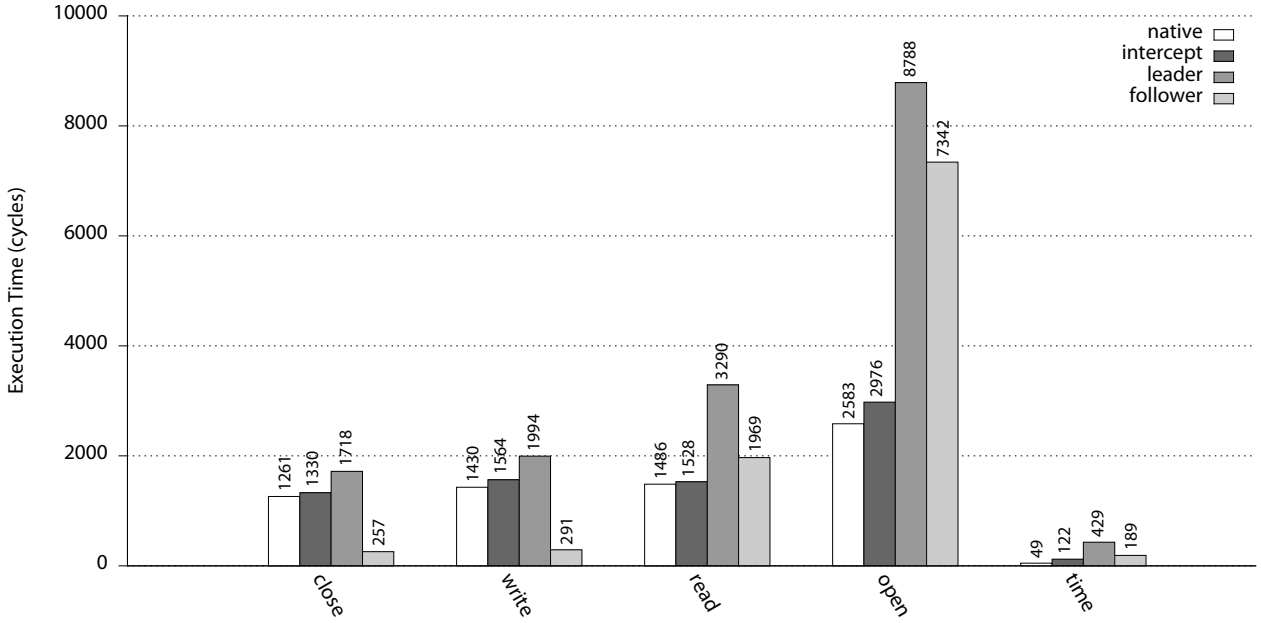


Figure 5.4: System call microbenchmarks.

memory. Finally, the cost for `open` is the highest, since it also involves the slower transfer of the returned file descriptor via a UNIX domain socket.

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

5.3.2 *C10k servers*

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

The five server applications used in our evaluations are summarised in Table 5.1 (the size is measured in lines of code, as reported by the *cloc* tool). For our performance experiments, we ran multiple instances of the same version of each application. Each experiment was performed six times, with the first measurement used to warm up the caches and discarded. The overhead was calculated as the median of the remaining five measurements. To measure the CPU utilisation, we used the mpstat utility.

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

Table 5.1: C10k server applications used in the evaluation.

Application	Size	Threading
Beanstalkd	6,365	single-threaded
Lighttpd 2.0	47,873	multi-threaded
Memcached	9,779	multi-threaded
Nginx	101,852	multi-process
Redis	34,625	multi-threaded

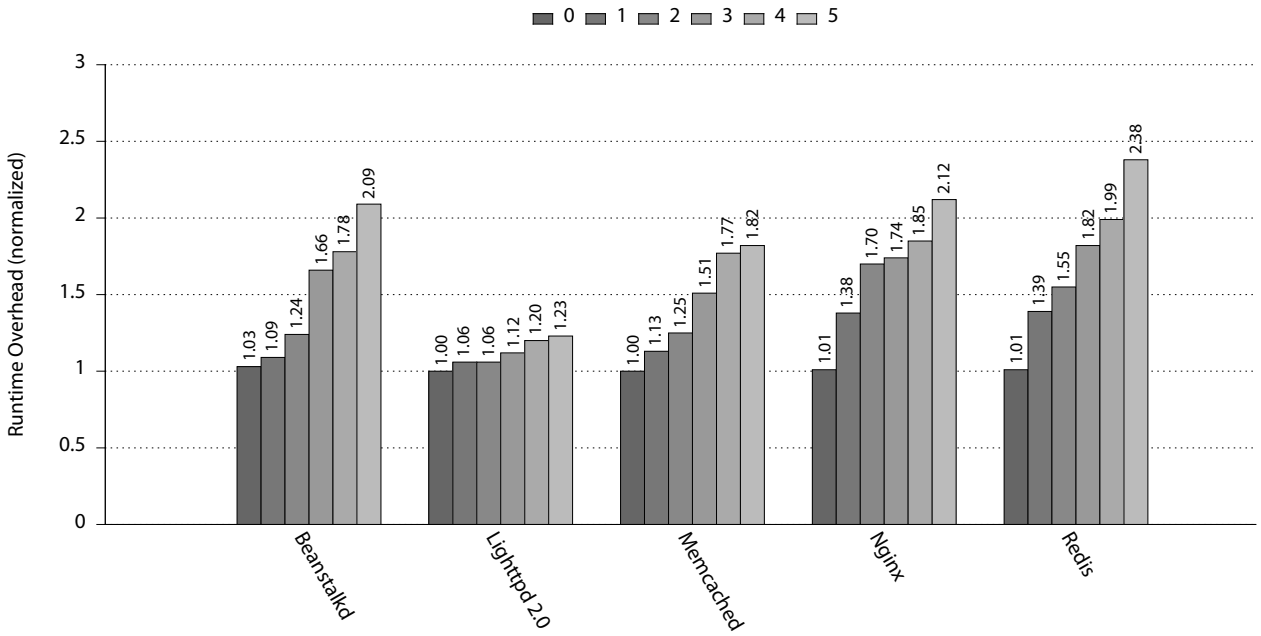


Figure 5.5: Performance overhead for the *Beanstalkd*, *Lighttpd 1.4*, *Memcached*, *Nginx* and *Redis* servers for different number of followers. The client and server are running on the same machine, simulating a worst-case scenario.

Beanstalkd[9] is a simple and fast work queue, used by many websites to distribute jobs among workers. We used revision `157d88b` from the official *Git* repository, the latest revision at the time of writing. To measure performance, we used `beanstalkd-benchmark` [10] with 10 concurrent workers each performing 10,000 push operations using 256 B of data per operation.

Lighttpd 2.0[11] is a lightweight web server optimised for high performance environments, reimplemented entirely from scratch in the version 2.0. We used revision `93d04a3` from the official *Git* repository, the latest revision at the time of writing. We measured the performance of serving a 4 kB page using `wrk` ,[12] which was run for 10 s with 10 clients.

Memcached[13] is a high-performance, distributed memory object caching system, used by many high-profile websites to alleviate database load. We used revision 1.4.17, the latest at the time of writing.

[10] `https://github.com/fangli/beanstalkd-benchmark`
[12] `https://github.com/wg/wrk`

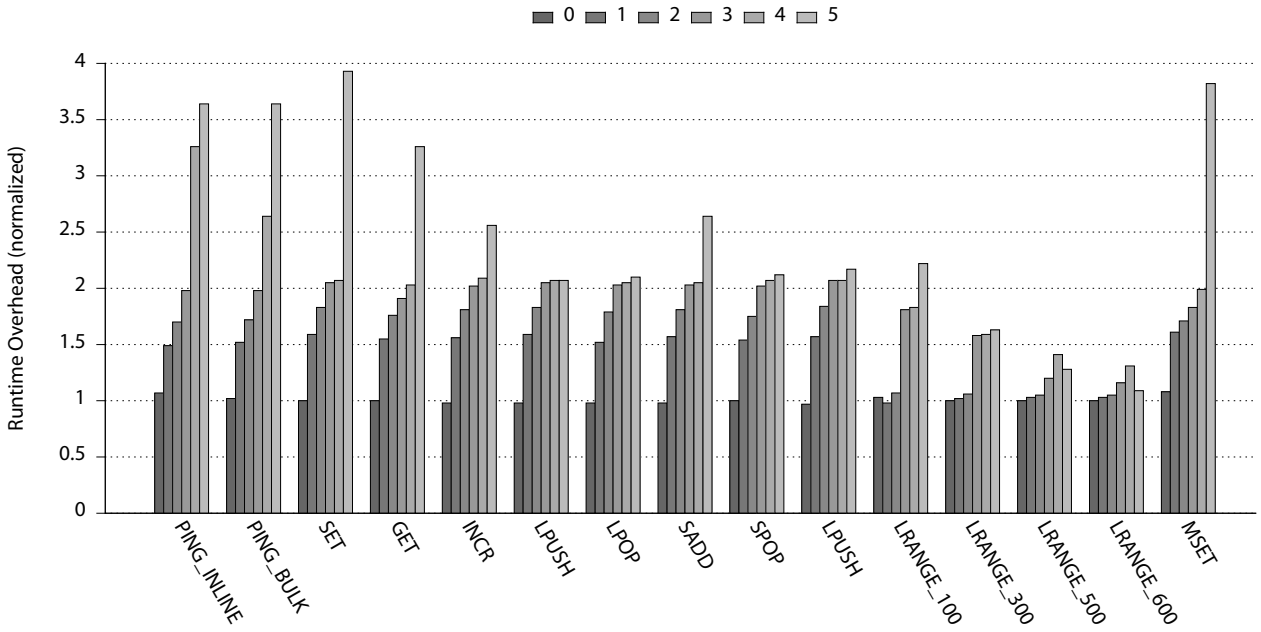


Figure 5.6: Performance overhead for individual *Redis* operations for different number of followers. The client and server are running on the same machine, simulating a worst-case scenario.

To measure the performance overhead, we used the `memslap` benchmark,[14] part of the *libMemcached* library. We used the default workload, *i.e.* an initial load of 10,000 key pairs and 10,000 test executions.

Nginx[15] is a highly popular reverse proxy server often used as an HTTP web server, load balancer or cache. We used version 1.5.12, the latest at the time of writing. We measured performance using the same workload as for *Lighttpd 2.0*.

Redis[16] is a high-performance in-memory key-value data store, used by many well-known services. We used version 2.9.11 in our experiments. To measure performance, we used `redis-benchmark`,[17] distributed as part of *Redis*. The benchmark issues different types of commands supported by *Redis* and measures both the throughput and the latency for each type. We used the default workload, *i.e.* 50 clients issuing 10,000 requests and calculated the average overhead across all commands.

Figure 5.5 shows the results for all servers and Figure 5.6 for individual *Redis* operations. All performance numbers are obtained using the client-side tools presented above. Since both the client and the server run on the same machine using the loopback device, these numbers represent a worst-case scenario, as the network throughtput and latency could hide some of the overhead.

For each benchmark, we show one bar, normalised relative to native execution, showing the performance of VARAN using a given number of followers. We stop at five followers, because our machine has four cores/eight threads, and we also needed additional thread for the leader and coordinator as well as for the client application itself.

[14] `http://libmemcached.org/`

[17] `http://redis.io/topics/benchmarks`

Table 5.2: Comparison with Mx, *Orchestra* and *Tachyon* on the benchmarks used to evaluate these systems.

System	Mechanism	Benchmarks	Overhead	Varan
Mx	ptrace	*Lighttpd 1.4* (`http_load`)	249 %	1 %
		Redis (`redis-benchmark`)	1572 %	39 %
		SPEC CPU2006	17.9 %	14.2 %
Orchestra [Sal+09]	ptrace	*Apache httpd* (`ApacheBench`)	50 %	2.4 %
		SPEC CPU2000	17 %	11.3 %
Tachyon [MB12]	ptrace	*Lighttpd 1.4* (`ApacheBench`)	272 %	0 %
		thttpd (`ApacheBench`)	17 %	0 %

The set of bars for zero followers measure the interception overhead of Varan using binary rewriting. This overhead is negligible for *Lighttpd 2.0*, *Memcached* and most *Redis* operations, 4 % for *Nginx*, and 10 % for *Beanstalkd*.

For all benchmarks, we see that the performance overhead increases slightly with the number of followers. For instance, the overhead for *Beanstalkd* increases from 9 % for one follower to 109 % for five followers, while the overhead for *Memcached* increases from 13 % to 82 %.

The figure also shows that there is a significant difference across benchmarks: the worst performer is *Redis*, which sees performance degradations in the range of 39 % to 138 %, while the best performers are *Lighttpd 2.0*, with only 6 % to 23 % overhead and some operations in *Redis* with under 3 % overhead.

The CPU utilisation is dependent on the number of versions run in parallel and varies slightly for different servers. For one follower it is around 17 % for *Memcached*, 29 % for *Lighttpd 2.0* and *Nginx*, 30 % for *Redis* and 31 % for *Beanstalkd*. For five followers, the CPU utilisation increases to 49 % for *Memcached*, 71 % for *Beanstalkd*, 73 % for *Redis*, 78 % for *Nginx* and 81 % for *Lighttpd 2.0*.

5.3.3 *Comparison with prior NVX systems*

While Sections 5.3.1 and 5.3.2 illustrate some of the worst-case synthetic and real-world scenarios for a system call monitor, in order to compare Varan directly with prior NVX systems, we have also run it on the same set of benchmarks used to evaluate prior systems. In particular, we chose to compare against two state-of-the-art NVX systems—*Orchestra* [Sal+09] and *Tachyon* [MB12]—as well as Mx. These systems and their benchmarks are briefly described in the first three columns of Table 5.2. To our knowledge, we are the first to perform an extensive performance comparison of existing NVX systems.

To allow for more faithful comparison, the benchmarks were run on a pair of machines, one running the server under Varan and the other the client, as this was the setup used by both *Orchestra* and *Tachyon* as well as Mx. The machines were located in the same rack, connected by a 1 Gb Ethernet link.

The last two columns of Table 5.2 show the cumulative results. Since prior systems only handle two versions, the comparison is done against Varan configured in the same way. However, we remind the reader that one of the strengths of Varan's decentralised architecture is that it can often handle

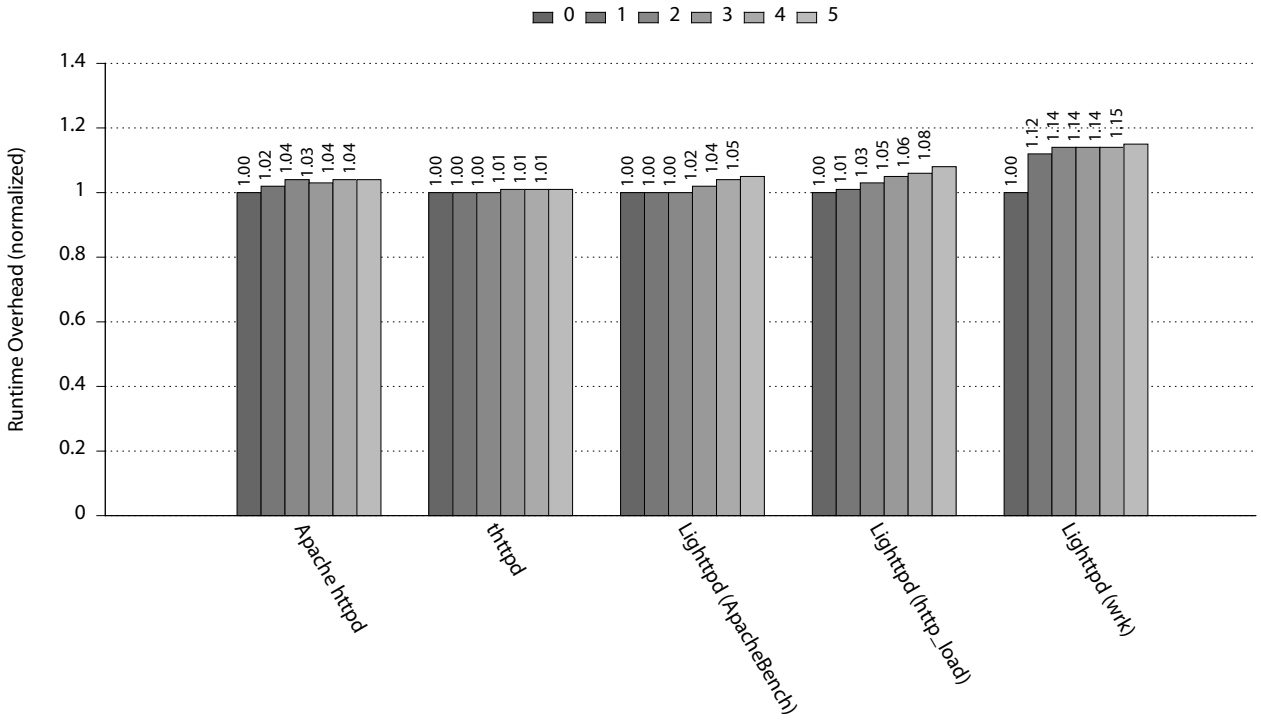


Figure 5.7: Performance overhead for the *Apache httpd*, *thttpd*, and *Lighttpd 1.4* servers for different numbers of followers to allow for comparison with existing systems.

multiple versions with small additional overhead, and below we also show how Varan performs on these benchmarks when more than two versions are used.

Apache httpd[18] was used by *Orchestra*. We used version 1.3.29, the same as in the original work [Sal+09]. The overhead reported for *Orchestra* is 50 % using the `ApacheBench` benchmark. Varan introduces 2.4 % overhead using the same benchmark, which is a significant improvement. Figure 5.7 shows the overhead introduced by Varan for *Apache httpd* (and the other servers used to evaluate prior work) with different numbers of followers. As it can be seen, Varan scales very well with increasing numbers of followers for these benchmarks.

Lighttpd 1.4[19] has been used to evaluate both Mx (§4.2.2) and *Tachyon*. We used version 1.4.36 as in case of Mx (*Tachyon* did not report the version used). For Mx we used the `http_load` benchmark and reported up to 249 % overhead while *Tachyon* used the `ApacheBench` benchmark and reported a up to 272 % overhead. When benchmarked using `http_load` , Varan introduced at most 1 % overhead, while with `ApacheBench` it introduced no noticeable overhead. This marks a significant improvement over previous work.

thttpd[20] was shown to introduce 17 % overhead when run on top of *Tachyon* using the `ApacheBench` benchmark. When run on top of Varan using the same settings as in [MB12], we have not measured any noticeable overhead.

Redis[21] was used in the evaluation of Mx (§4.2.2), in particular the version 1.3.8. The performance overhead incurred by Mx is up to 1572 % using the `redis-benchmark` utility. When run with

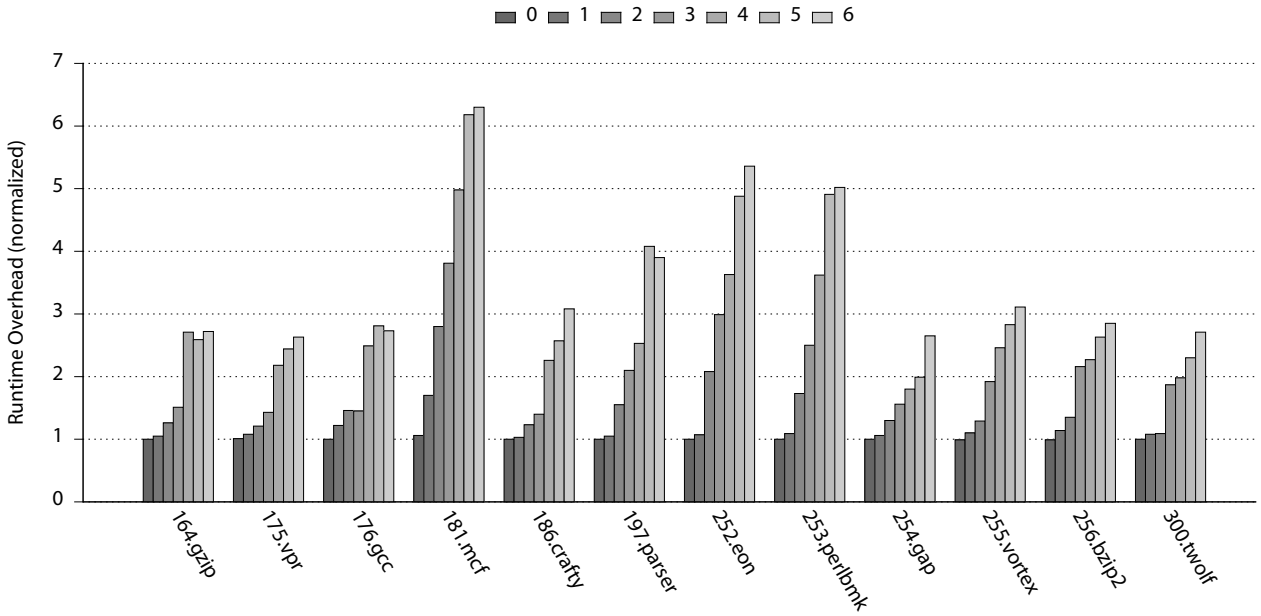


Figure 5.8: *SPEC CPU2000* performance overhead for different numbers of followers.

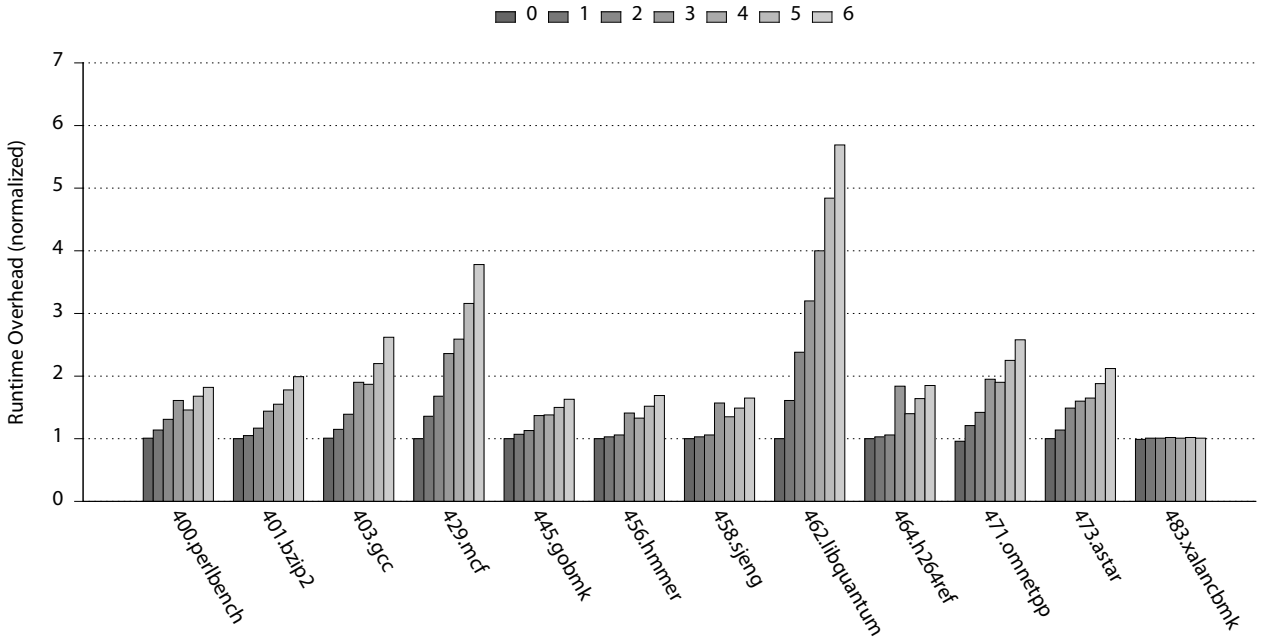


Figure 5.9: *SPEC CPU2006* performance overhead for different numbers of followers.

VARAN using the same benchmark and the same workload, the overhead we measured was no more than 39 %.

SPEC CPU2000[22] was used to evaluate *Orchestra*. We used the latest available version 1.3.1. *Orchestra* reported a 17 % average overhead, while VARAN introduced only a 11.3 % average overhead. The results for the individual applications contained in the *SPEC CPU2000* suite and for different numbers of followers can be seen in Figure 5.8.

SPEC CPU2006[23] was used in the Mx evaluation (§4.2.2). The average overhead of Mx is 17.9 %, while the average overhead of Varan is 14.2 %. Individual results can be seen in Figure 5.9.

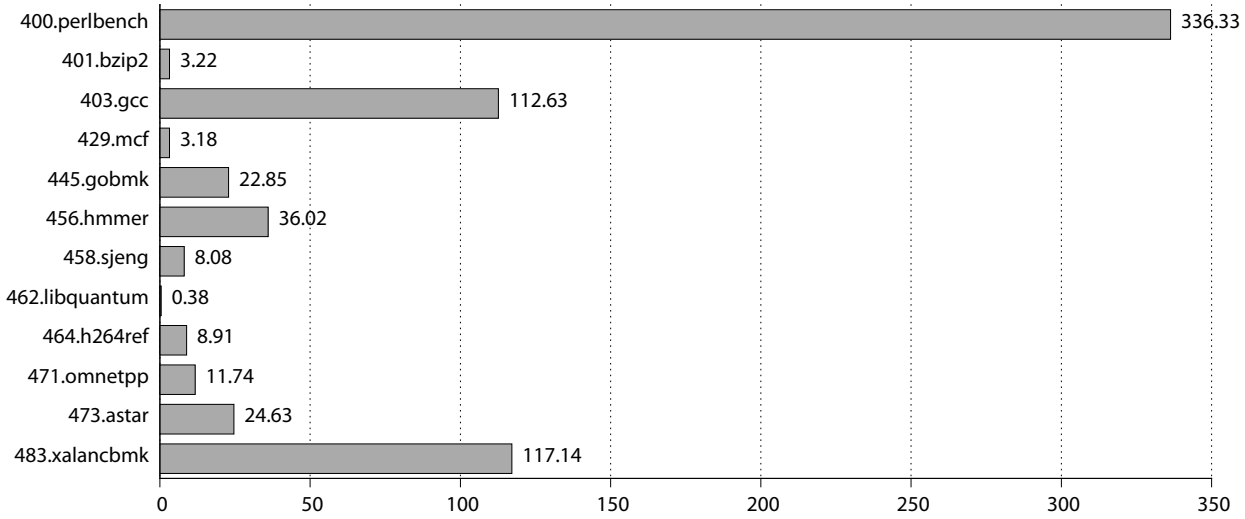


Figure 5.10: Average number of system calls per second during the execution of the SPEC CPU2006 benchmark suite, measured using the strace tool.

The reason why some of the *SPEC CPU* applications scale poorly with the number of followers is likely due to memory pressure and caching effects [Jal07], and to the fact that our machine has only four physical cores (with two logical cores each). To get a better insight into this issue, we have also examined how the frequency of system calls affects the performance overhead. Figure 5.10 shows the average number of system calls during the execution of *SPEC CPU2006*. As it can be observed, 452.libquantum has the lowest average number of system calls per second, but the largest runtime overhead. For comparison, 400.perlbench has the highest average number of system calls per second, but bellow average runtime overhead. This confirms that the increased overhead is not caused by the system call handling mechanism and can be likely attributed to memory effects.

5.4 Transparent failover

The primary application of Varan is *transparent failover*, *i.e.* the ability to transparently switch over to the non-crashing version in the event of failure without any disruption in service. Varan makes it easy to implement transparent failover. When one of the versions crashes, the `SIGSEGV` signal handler installed in each version notifies the coordinator, which decides what restart strategy to use. When one of the followers crashes, the coordinator unsubscribes it from the list of currently-running followers, and discards it without affecting other followers. When the leader crashes, it designates one of the followers as the new leader (currently the one with the smallest internal ID), and notifies it to switch its system call

```
@@ -66,0 +67,11 @@
+#ifdef HAVE_GETUID
+# ifndef HAVE_ISSETUGID
+
+static int l_issetugid() {
+ return (geteuid() != getuid() || getegid() != getgid());
+}
+
+# define issetugid l_issetugid
+# endif
+#endif
+
@@ -592 +603 @@ int main (int argc, char **argv) {
- if (!i_am_root && (geteuid() == 0 || getegid() == 0)) {
+ if (!i_am_root && issetugid()) {
```

Listing 5.1: *Lighttpd 1.4* SUID bit detection patch.

table (§5.2.2) to that of the leader, and to restart the last system call while discarding the old (crashed) leader.

To demonstrate support for transparent failover, we reproduced the *Redis* bug described in Section 2.1. We ran in parallel eight consecutive revisions of *Redis* from the range `9a22de8` to `7fb16ba`, where the last revision introduced a bug which crashes the server by causing a segmentation fault. We then set up a client to send an `HMGET` command that triggers the bug, and measured the increase in latency for that command. When the buggy version is a follower, we do not observe any increase in latency, as expected. When the buggy version is the leader, the latency increases from 42.36 µs to 122.62 µs. In both cases, we observed no extra degradation in throughput for the commands that follow.

As an additional experiment, we ran revisions `2437` and `2438` of *Lighttpd 1.4* (also introduced earlier), the latter of which introduced a crash bug. We then set up a client that triggers the bug and measured the latency for that request. Both when the buggy version was the leader or a follower, there was no significant increase in latency, which remained at around 5 ms.

These examples demonstrate the difference between Mx and Varan. Unlike Varan, which discards the crashing version, Mx can recover from the crash (§4.2). On the other hand, Mx can only run two versions in parallel and incurs relatively high performance overhead, while Varan can run an arbitrary number of versions with low performance overhead as shown by the comparison in Section 5.3.3.

5.4.1 *Handling divergent behaviour*

Multiple different software versions (revisions) can be run inside an NVX system only as long as they all issue the same sequence of system calls. This limitation is due to the fact that prior NVX systems, either run versions in lockstep as in case of Mx, or do not support rewriting system call traces in case of divergence.

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

Listing 5.2: Example of a BPF rewriting rule.

Because Varan does not run the versions in lockstep and can use system call rewrite rules, it can often overcome this limitation. To illustrate, we used several *Lighttpd 1.4* revisions from our evolution study (§3.5) which introduced new system calls and as such cannot be run in parallel by prior NVX systems that rely on lockstep execution.

As a first experiment, we ran revision `2435` as leader together with revision `2436` as follower. The patch is shown in Listing 5.1. Revision `2436` introduces two additional checks using the `getuid` and `getgid` system calls. More precisely, revisions until and including `2435` used `geteuid()` and `getegid()` C library functions to check the user account under which the server is being run, before issuing an `open` system call. This resulted in a sequence of `geteuid`, `getegid` and `open` system calls. Revision `2436` replaced the use of the aformentioned functions with `issetugid()` which changed the system call sequence to `geteuid`, `getuid`, `getegid`, `getgid`, followed by `open` as before.

To allow for this divergence, we used the custom BPF filter shown in Listing 5.2. The filter is executed by the follower whenever a divergence is detected. In our experiment, this happens when the follower executes the newly introduced `getuid` system call. The filter first loads the system call number executed by the leader into the implicit BPF accumulator (line 1) and checks whether the call is either `getegid` (line 2) or `open` (line 3). The former will be true in this case, so control will transfer to line 6, which loads the system call number executed by the follower into the accumulator, checks whether it is `getuid` (line 7) and finally transfers control to line 12 returning the value `SECCOMP_RET_ALLOW`, which instructs Varan to execute the additional system call (*i.e.* `getuid`) in the follower. Any other combination of system calls would have killed the follower (line 11). After executing the `getuid` system call and replaying the execution of `getegid` (which the leader also executed), Varan would detect a second divergence when the follower tries to execute `getgid` instead of `open`. This divergence would be resolved using the same filter, taking the path on lines 3, 9, 10 and 12.

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

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

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

5.5 DISCUSSION

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

CPU UTILISATION AND MEMORY CONSUMPTION. The performance evaluation reported in Section 5.3 considers the overhead in terms of throughput or clock time. However, an NVX framework introduces a CPU utilisation overhead linear in the number of versions. While this might be a serious concern in some scenarios, leaving cores idle has a cost as well [BH07] and in many cases idle cores can be profitably used to increase software reliability and security [Cox+06; CPW10; Sal+09; BZ06; CH12; HBT13].

Furthermore, the CPU utilisation overhead is only relevant for the user space CPU time, since the system call is only going to be executed by the leader. Many C10k servers spend a large amount of their execution time in the kernel—*e.g.*, as shown in [Cle15], 84% of a single-threaded 1KB write in *Redis* is spent in the kernel. This means that the total CPU utilisation of the multi-version application is often going to be significantly less than the number of versions run in parallel.

Similarly, the memory overhead imposed by Varan is linear in the number of versions, as in prior NVX systems. This can lead to degradations in performance due to memory pressure and caching effects, as we have observed in Section 5.3.3.

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

SYNCHRONISATION. While Varan supports multi-threaded and multi-process applications (§5.2.3), there is a potential issue with synchronisation primitives implemented entirely in user space, as these primitives will be invisible to Varan. While it is possible to use entirely user-space synchronisation

primitives, in our experience, they are not that frequent and standard synchronisation primitives combine atomics with system calls (*i.e.* futex). We have not observed any related problems in our concurrent benchmarks (§5.3.2, §5.3.3).

SECURITY. Although our focus with VARAN has been on improving software reliability, VARAN could be also used to implement existing NVX security defences [Cox+06; Sal+09]. However, there are two additional problems that VARAN introduces, as discussed below.

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

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

5.6 SUMMARY

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

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

Our experimental evaluation has demonstrated that VARAN can run C10k network servers with low performance overhead and can be used successfully as a multi-version execution monitor.

6

Related Work

This chapter provides an overview of previous work in the area. We first discuss other scenarios which involve the execution of multiple program versions, and then discuss related work in area of *N*-version programming, dynamic software updates, sandboxing and record & replay. These approaches and their common use cases are summarized in Table 6.1.

Table 6.1: Taxonomy of related work which summarises the existing approaches and compares them with respect to each other.

Summary	Use case	Characteristic
N-version programming Running multiple software versions in parallel	Availability Reliability	Number of possible applications with higher resource usage
Dynamic software updates Applying software updates at runtime	Reliability Maintenance	Resource efficient operation but requires intrusive changes
Sandboxing Isolating software and its effects from the system	Security	Simple and effective solution but limited to fault isolation
Record & replay Recording the execution for future replay	Debugging Testing	Asynchronous in nature restricting the possible scenarios

6.1 *N*-version programming

The original idea of concurrently running multiple versions of the same application was first explored in the context of *N*-version programming, a software development approach introduced in the 1970's in which multiple teams of programmers independently develop functionally-equivalent versions of the same program in order to minimise the risk of having the same bugs in all versions [CA78]. During runtime, these versions are executed in parallel on top of an *N*-version execution (NVX) environment in order to improve the fault tolerance of the overall system. Both the version generation and the synchronisation mechanism required manual effort.

A possible realisation of the original *N*-version programming paradigm is Cocktail [XDK12], which proposes the idea of running different web browsers in parallel under the assumption that any two of them are unlikely to be vulnerable to the same attacks. Compared to Mx and Varan, Cocktail's task is

simplified by exclusively targeting web browsers, which implement common web standards, while both Mx and Varan support arbitrary off-the-shelf applications.

6.1.1 *Multi-variant execution*

Cox *et al.* [Cox+06] proposed a general framework for increasing application security by running in parallel several automatically-generated diversified variants of the same program—when a divergence is detected, an alarm is raised. The technique was implemented in two prototypes, one in which the variants are run on different machines, and one in which they are run on the same machine and synchronised at the system call level, using a modified Linux kernel. By using multiple variants with potentially disjoint exploit sets, the proposed approach helps applications survive certain classes of security vulnerabilities such as buffer overflows, as attackers would need to simultaneously compromise all variants.

Berger *et al.* [BZ06] described a different approach which uses heap over-provisioning and full randomisation of object placement and memory reuse to run multiple replicas and reduce the likelihood that a memory error will have any effect. The goal of their work is to increase reliability by tolerating certain memory errors in exchange for space cost and execution time.

Within this paradigm, the Orchestra framework [Sal+09] used a modified compiler to produce two versions of the same application with stacks growing in opposite directions, runs them in parallel on top of an unprivileged user-space monitor, and raises an alarm if any divergence is detected to protect against stack-based buffer overflow attacks. This work has been further improved in [Sal+11] with a new method for signal delivery accompanied with detailed analysis of the benchmark characteristics. However, their approach is still limited to two versions running in lockstep with the only difference being the opposite growing stack.

Trachsel *et al.* [TG10] used the multi-variant execution approach to increase performance, where program variants are obtained by using different (compiler) optimisations and algorithms. The goal is to increase the overall system performance by always selecting the variant which finishes its execution first. Thereby, no synchronisation across variants is needed.

More recently, researchers have proposed additional multi-variant techniques that fit within the N-version programming paradigm, *e.g.*, by employing complementary thread schedules to survive concurrency errors [Vee+11], or using genetic programming to automatically generate a large number of application variants that can be combined to reduce the probability of failure or improve various non-functional requirements [Har+12]. Yumerefendi *et al.* [YMC07] run multiple sandboxed copies of the same process to detect potential privacy leaks and to enforce access control and privacy management; Shye *et al.* [Shy+09] used multiple instances of the same application in order to overcome transient hardware failures. Cadar *et al.* have also argued that automatically generated software variants are a good way for exploiting the highly parallel nature of modern hardware platforms [CPW10].

There are two key differences between our approach and the work discussed above. First, we do not rely on automatically-generated variants, but instead run in parallel existing software versions, which raises a number of different technical challenges. This means that in our case, the variants are not semantically

equivalent. This eliminates the challenge of generating diversified variants and creates opportunities in terms of recovery from failures, but also introduces additional challenges in terms of synchronising the execution of the different versions. Second, prior work has mostly focused on detecting divergences, while our main concern is to handle them, either by recovering as in case of Mx or by discarding the crashing version as in case of Varan, in order to increase the availability of the overall multi-version application.

6.1.2 *Multi-version execution*

Cook and Dage [CD99] proposed a multi-version framework for upgrading components. Users formally specify the specific input subdomain that each component version should handle, after which versions are run in parallel and the output of the version whose domain includes the current input is selected as the overall output of the computation. The system was implemented at the level of leaf procedures in the Tcl language. The key difference with Mx and Varan is that this framework requires a formal description of what input domain should be handled by each version; in comparison, Mx and Varan target crash bugs and are fully automatic. Moreover, Mx's goal is to have all versions alive at all times, so crash recovery plays a key role. Finally, both Mx and Varan have to carefully synchronise access to shared state, which is not an issue at the level of Tcl leaf procedures.

The multi-version execution approach has been in the past successfully used for testing. Back-to-back testing [Vou90], where the same input is sent to different variants or versions of an application and their outputs compared for equivalence, has been used since the 1970s. Band-aid patching [SIK07] proposed an online patch testing system that splits execution before a patch, and then retroactively selects one code version based on certain criteria. More recently, delta execution [TXZ09] proposed to run two different versions of a single application, splitting the execution at points where the two versions differ, and comparing their behaviour to test the patch for errors and validate its functionality. Similarly, Tachyon [MB12] is an online patch testing system in which the old and the new version of an application are run concurrently; when a divergence is detected, the options are to either halt the program, or to create a manual rewrite rule specifying how to handle the divergence.

The idea of running multiple executions concurrently has also been used in an offline testing context. For instance, d'Amorin *et al.* [dLM07] optimise the state-space exploration of object oriented code by running the same program on multiple inputs simultaneously, while Kim *et al.* [KKB12] improve the testing of software product lines by sharing executions across a program family.

By comparison with this body of work, our focus is on managing divergences across software versions at runtime in order to keep the application running, and therefore runtime deployment and automatic crash recovery play a central role in both Mx and Varan.

Closely related to the execution of multiple versions is the management of multiple versions and software updates, such as deciding when to upgrade, applying updates, *etc.* Previous work on improving the software update process has also looked at different aspects related to managing and deploying new software versions.

6.2.1 *Dynamic software updates*

Dynamic software updating (DSU) systems are concerned with the problem of updating programs while they are running. These systems have been implemented both in kernel-space [Bau+05; MR07; AK09; GKT13] and in user-space [Alt+05; Nea+06; Che+07; MB09; Hay+11; Hay+12].

The dynamic update process typically consists of two stages—code update and state transfer—and existing systems employed different approaches to handle these.

Ginseng [Nea+06] and K42 [Bau+05] employ indirection to enable the code update. This simplifies the code update mechanism, but the indirection introduces performance overhead during normal execution. Ginseng uses a specialised compiler which creates a "dynamically updatable program" where all function symbols are replaced with function pointers and all direct function calls are replaced with indirect call through these pointers. During update, these pointers are updated to point to the new version of the code, which is loaded using `dlopen`. K42 operating system has a per-address-space object translation table used for all object invocations. When the new kernel module is loaded, the translation table entries are updated to point to the new version.

OPUS [Alt+05], DynAMOS [MR07], POLUS [Che+07] and Ksplice [AK09] use binary rewriting instead of indirection to replace the entry point of the functions being updated with a jump to a trampoline. Compared to indirection, this approach does not require the use of a specialised compiler, but it is highly platform-dependent. We use the same approach in VARAN to handle virtual system calls (§5.2.2).

The aforementioned systems treat individual functions or objects as the unit of code for updates. Such systems are incapable of handling functions that rarely end (*e.g.*, `main` or functions that contain event-handling loops). This was addressed by systems such as UpStare [MB09], Ekiden [Hay+11] and Kitsune [Hay+12], which update the code by loading an entirely new program instead of replacing individual functions. After the new version has been loaded, either through `fork-exec` or through `dlopen`, the execution needs to be restarted from the previous point. UpStare uses stack reconstruction to achieve this by replacing the stack frames for old functions with their new versions. Ekiden and Kitsune on the other hand rely on manual approach requiring the programmer to direct the execution into the equivalent update point in the new version.

PROTEOS [GKT13] and TTST [Giu+13] use a custom instrumentation component implemented on top of LLVM compiler framework to automate state transfer and state management to a large degree, only requiring programmer-provided state transfer callback for complex state changes (and cases of

pointer ambiguity). Unlike previous systems which use in-place live update strategies, Proteos and TTST confine different program versions into separate processes which simplifies state managements and enables simple fail-over in case of a failed update. While Proteos is a new research operating system, TTST is implemented entirely in Linux user-space.

Compared to Mx and Varan, in DSU systems the two versions co-exist only for the duration of the software update, but DSU and the Rem component of Mx face similar challenges when switching execution from one version to another. We hope that some of the technique developed in DSU research will also benefit the recovery mechanism of Mx and vice versa.

6.2.2 *Update management and distribution*

Prior work on improving software updating has looked at different aspects related to managing and deploying new software versions.

Beattie *et al.* [Bea+02] considered the issue of timing the application of security updates—patching too early could result in breaking the system by applying a broken patch, patching too late could on the other hand lead to the risk of penetration by an attacker exploiting a well known security issue. Using the cost functions of corruption and penetration, based upon real world empirical data, they have shown that 10 and 30 days after the patch's release date are the optimal times to apply the patch to minimise the risk of a defective patch. Such a delay still opens a lot of opportunities for potential attackers.

Crameri *et al.* [Cra+07] proposed a framework for staged deployment. This framework integrates upgrade deployment, user-machine testing and problem reporting into the overall upgrade process. The framework itself clusters user machines according to their environment and software updates are tested across clusters using several different strategies allowing for the deployment of complex upgrades.

A solution inspired by N-version programming was proposed by Michael Franz [Fra10]. Instead of executing multiple versions in parallel, the author suggested distributing a unique version of every program to every user. Such versions should be created automatically by a "multicompiler", and distributed to users through "App Store". This would increase security as it would be much more difficult to generate attack vectors by reverse engineering of security patches for these diversified versions.

In relation to this work, Mx and Varan try to ease the decision of applying a software update, because incorporating a new version should only increase the reliability and availability of the overall multi-version application. However, in practice the number of versions that can be run in parallel is dictated by the number of available resources (*e.g.*, the number of available CPU cores), so effective update strategies are still needed in this context, and work in this area could provide helpful solutions to this problem.

Many large-scale services, such as Facebook and Flickr use a *continuous deployment* approach, where new versions are continuously released to users [Joh09; Har09], but each version is often made accessible only to a fraction of users to prevent complete outage in case of newly introduced errors. While this approach helps minimise the number of users affected by new bugs, certain bugs may manifest themselves only following prolonged operation, after the release has been deployed to the entire user

base. We believe our proposed approach is complementary to continuous deployment, and could be effectively combined with it.

6.3 SANDBOXING

N-version execution environment typically uses some form of application sandboxing. The goal is to isolate the running application from the underlying system and especially, to prevent software failures in any of the versions being executed from affecting the rest of the system including other versions.

Prior sandbox architectures include both kernel-based mechanisms [BBS95; BGM02; Cow+00; BFM99] and system call interposition monitors [Gol+96; AR00; JS99; Pro02; KZ13]. Other mechanisms have been also used to build sandboxes, such as binary rewriting [FC08; PHG12], system transactions [JPS11] and virtual machine introspection [GR03].

6.3.1 *System call interposition*

Regardless of the application, most operating system resources such as files and sockets can only be accessed through the system call interface. Therefore, this interface is often a target for monitoring and regulation. Both VARAN and MX operate at the system call interface layer, synchronizing system calls performed by different application versions. VARAN draws inspiration from the Ostia delegating architecture [GPR04], and from the selective binary rewriting approach implemented by *BIRD* [Nan+06] and seccompsandbox.[1] MX on the other hand uses the ptrace interface same as many of the existing monitors such as Orchestra [Sal+09] or Tachyon [MB12].

System call interposition has been an active area of research and there are there many different ways in which the system call interception mechanism can be implemented, including dynamic libraries loaded through the preloading mechanism (*i.e.* using `LD_PRELOAD` or `DYLD_INSERT_LIBRARIES` variables) and custom C libraries [Sea], facilities provided by the operating system (*i.e.* ptrace interface or the `/proc` subsystem), kernel extensions and binary rewriting.

Janus [Gol+96] was one of the first systems to implement a confinement mechanism by interposing on the system calls made by the application via the standard tracing mechanisms, in particular ptrace. A similar architecture has been later adopted by MAPbox [AR00] and Consh [AKS98]. However, all of these systems suffered from numerous limitations posed by the ptrace interface [Wag99]. Jain and Sekar [JS99] tried to address some of these issues to build a more efficient user-space confinement mechanism. A later Janus [Wag99] evolution implemented a tracing mechanism for Solaris using the `/proc` subsystem. To address some of the ptrace shortcomings, Systrace [Pro02] used a loadable kernel module.

Some system call interposition mechanisms were implemented entirely in kernel [Cow+00; BFM99]. Ostia [GPR04] has shown how to implement a secure and efficient system call interposition mechanism

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

by combing a kernel extension with a custom user-space binary rewriting mechanism avoiding the use of ptrace achieving lower performance overhead compared to ptrace-based sandboxing mechanisms. More recently, MBOX [KZ13] implemented a sandboxing mechanism by combining ptrace and seccomp/bpf to allow for selective system call filtering.

Orchestra [Sal+09] and Tachyon [MB12] used system call interposition through ptrace interface to implement *N*-version runtimes. To improve the performance overhead when reading tracee memory, one of the ptrace shortcomings, Orchestra uses shared memory injected into the application's address space to read the values of indirect system call arguments, while Tachyon uses cross memory attach, same as Mx (§4.1.1), for the same purpose.

Due to their importance, especially when used for application sandboxing, system call monitors themselves can also become targets for attackers. Garfinkel [Gar03] and Watson [Wat07] described common vulnerabilities in system call monitors and the ways in which these could be exploited.

Numerous intrusion detection mechanisms rely on the analysis of system call sequences looking for potential anomalies [HFS98; WDD00; Sek+01; GRS04; BCS06]. These systems rely on system call tracing, which is similar to system call interposition, although there is no need for modifying or denying system calls.

6.3.2 *Software fault isolation*

The use of software-based fault isolation for executing untrusted code has been first described in [Wah+93] for RISC machines. However, the first effective implementation of software fault isolation for the CISC architecture has been shown only much later in [MM06].

These concepts have been later used by several sandboxes designed for isolation of plugins on the web. Xax [Dou+08] uses system call interposition together with address space isolation to leverage existing libraries and programs on the web. Native Client [Yee+09] requires the code to be recompiled to a restricted subset of the x86/ARM ISA which can be checked prior to running and confines the application using segmentation and memory protection. The problem of protecting trusted code from the untrusted code addressed by both of these sandboxes is orthogonal to traditional system-level sandboxing and these systems rely on separate sandboxing mechanism for that purpose (*e.g.*, Native Client uses either seccompsandbox or seccomp/bpf as an "outer" sandbox layer).

6.4 RECORD & REPLAY

Event streaming in VARAN can be seen as a variant of record & replay. Record & replay technique has been an active research topic for more than forty years and different approaches have been developed over this period. However, unlike traditional record & replay systems that require a persistent log, VARAN keeps the shared ring buffer in memory, and deallocates events as soon as they are not needed, which minimises performance overhead and space requirements in the NVX context.

BugNet [NPC05], Jockey [Sai05], liblog [Gee+06], and R2 [Guo+08] use a library approach where the replay tool is provided as a library which is inserted into a target application to allow for record & replay. Jockey and liblog try to minimise the interference with the application being recorded to ensure that the application behaves the same with and without the tool, and rely solely on intercepting of a set of C library calls. BugNet provides a special API applications must use to support record & replay while R2 requires developers to annotate functions which can be recorded and replayed correctly. RecPlay [RD99] also uses a library approach, but focuses on debugging of non-deterministic parallel programs. The approach combines the record & replay mechanism with automatic data race detection implemented using vector clocks.

Flashback [Sri+04] has a similar focus, enabling deterministic debugging, but rather than library, it is implemented as a kernel extension. The key concept is the use of shadow processes, which duplicate the state of a process in memory allowing fast rollback of the debugged program with small overhead. Scribe [LVN10] is another kernel-based solution focused on transparent, low-overhead record & replay with the ability to switch from replayed to live execution. While the kernel-based implementation results in lower performance overhead, it makes these systems less easy to develop and debug, and less safe to use compared to library based approaches.

RR [BSV07] is similar to VARAN by replicating an application into multiple instances for fault tolerance. As VARAN, it also uses a variant of record & replay to synchronise the primary replica with others. Unlike VARAN, RR is focused on fail-stop scenarios and does not support running different versions of the same application. The preliminary prototype has been implemented as a Linux kernel extension, with all the associated limitations discussed above, and evaluated using only a single benchmark.

ReVirt [Dun+02], SMP-ReVirt [Dun+08] and iDNA [Bha+06] implement record & replay mechanism as part of a virtual machine hypervisor, allowing to record and replay the entire system, including the operating system and the target applications. ReVirt is built on top of UML, which lacks support for multiprocessor virtual machines. This limitation has been addressed in SMP-ReVirt which builds on top of Xen. iDNA uses a custom virtualisation engine called Nirvana. The virtualisation-based record & replay mechanism allow faithful replay of every aspect of the application's environment, including the scheduling decisions of the operating system, with a small performance overhead. On the other hand, these systems are more difficult to deploy and their use is more expensive as they require an entire operating system to record a single application.

Aftersight [CGC08] uses virtualisation-based record & replay mechanism to enable running heavy-weight analyses on production workloads by decoupling the analysis from normal execution. The system supports three different modes of operation: in the synchronous safety and best-effort safety modes, the analysis is being run in parallel with the workload, while in the offline mode, the execution log is stored for later analysis akin to traditional record & replay systems. The synchronous safety model resembles the Mx's mode of operation where the workload and the analysis are run in lockstep. The best-effort safety mode is similar to VARAN, and in particular the live sanitisation deployment model described in Section 7.2, where there is no synchronisation between the output of the workload and the analysis.

Paranoid Android [Por+10] uses a similar approach for protecting mobile phones—rather than running the attack detection mechanism directly on the device, negatively affecting the performance and the battery life, the phone records a minimal execution trace, and sends it to the server, where it is replayed inside a virtual machine. This enables the use of multiple attack detection techniques that can be all run in parallel. Same as Mx, Paranoid Android also uses ptrace mechanism for system call interception. Similarly to Varan's event streaming mechanism, Paranoid Android uses asynchronous record & replay mechanism to decouple the execution of the monitored application from its replicas.

6.5 SOFTWARE EVOLUTION

Despite the significant role that coverage information plays in software testing, there are relatively few empirical studies on this topic. We discuss some representative studies below.

Early work on this topic was done by Elbaum *et al.* [EGR01], who have analysed how the overall program coverage changes when software evolves, using a controlled experiment involving the space program, and seven versions of the Bash shell. One of the key findings of this study was that even small changes in the code can lead to large differences in program coverage, relative to a given test suite. This is a different finding from previous work, such as that by Rosenblum and Weyuker [RW97], which has found that coverage remains stable over time for the KornShell benchmark. In this thesis, we have looked at a related question, of whether overall coverage remains stable over time, taking into consideration the changes to the evolving test suite as well.

Zaidman *et al.* [Zai+11] have examined the co-evolution of code and tests on two open-source and one industrial Java applications. The study looks at the evolution of program coverage over time, but only computes coverage for the major and minor releases of each system, providing around ten data points for each system. By looking at the co-evolution of code and tests, the analysis can infer the development style employed by each project: one key finding is that code and tests co-evolve either *synchronously*, as when agile methods are used; or *phased*, with periods of intense coding followed by periods of intense testing. In our experiments presented in Chapter 3, we have observed both development styles.

7

Other applications and future work

Both Mx and Varan are designed as flexible frameworks which could be extended to support a variety of application scenarios involving NVX systems. In this chapter, we discuss four such scenarios: regression testing (§7.1), live sanitization (§7.2), record-replay (§7.3), and security honeypots (§7.4). We also present other possible future extensions to both Mx and Varan.

7.1 REGRESSION TESTING

By default, whenever Mx detects a divergence between the two versions which are being run in parallel, it either discard one of the versions or, if the divergence is caused by the segmentation fault, it tries to recover the failing version (§4.1.2). This mode of operation is suitable as a way for improving reliability of updated software, as shown in Section 4.2.

However, Mx can be also used in a different mode, in which the application is fail-stopped whenever a divergence is detected. This mode of operation is suitable for regression testing. When working on the new version of the application, developers would typically use the regression suite to ensure that the new version does not break the existing functionality. When such a situation occurs, the first step is typically to determine where the bug was introduced. While this may be easy for simple sequential applications, such task could be challenging for applications which use non-deterministic inputs such as network events or random number generators.

One of the main goals of Mx is to intercept all sources of non-determinism and provide both versions with the same input. This allows developers to run both versions in parallel, and see whether the patch introduces any divergence by running both applications in lockstep. When a divergence is detected, Mx can also provide developers with additional information such as the stack trace obtained by the Rem component.

While conceptually simple, we believe this mechanism could be helpful in many development scenarios. Furthermore, we believe it could be integrated with other mechanisms such as `git bisect`, which normally relies on the regression testing suite, but might be too coarse grained in some scenarios; when combined with Mx it can also detect more subtle differences (*e.g.*, due to non-determinism).

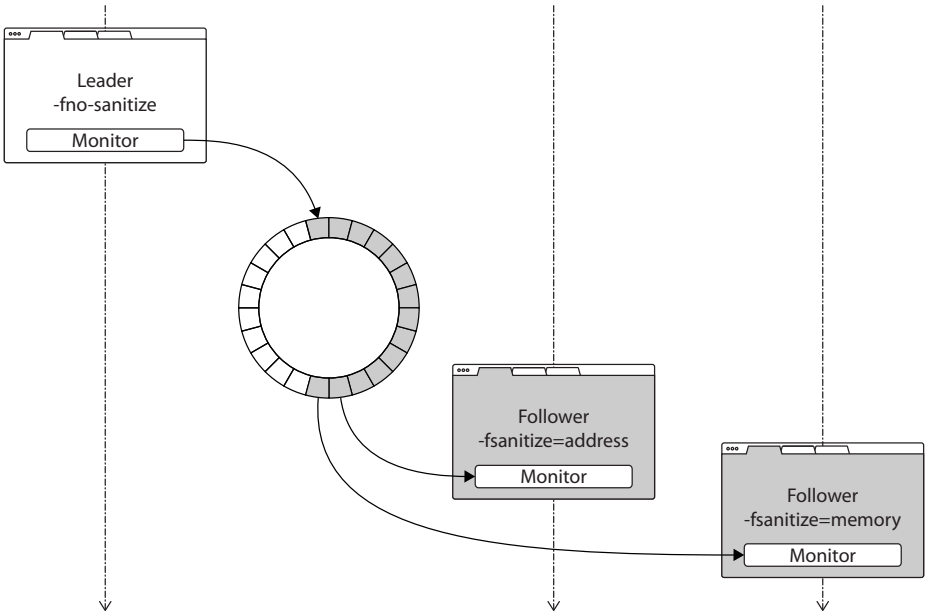


Figure 7.1: The use of event-streaming architecture for live sanitisation.

7.2 LIVE SANITISATION

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

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

With VARAN, we can perform live sanitisation by running the native unsanitised version as the leader, with sanitised versions as followers, as shown in Figure 7.1. While sanitisation itself introduces a performance overhead, since followers do not need to execute any I/O operations and merely replay them, they can often keep up with the leader, allowing users to run sanitised versions in production without introducing any significant overhead.

To demonstrate this, we build revision `7f77235` of *Redis* twice: once using Clang without any sanitisation, once with ASan enabled. We then ran both versions in parallel using Varan and used the same benchmark with the same settings as for our performance evaluation (§5.3.2). As expected, we have not measured any additional slowdown in the leader compared to the scenario with two non-sanitised versions being run in parallel. To get a better insight into the effect of running the sanitised version with Varan, we have measured the median length of the log, *i.e.* the distance between the leader and the follower. With sanitisation, this length increases from two to six, which does not impose any problems.

7.3 RECORD-REPLAY

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

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

We have implemented a simple prototype of the two aforementioned clients on top of Varan and compared its performance against *Scribe* [LVN10], a state-of-the-art record-replay system implemented in the kernel. Unfortunately, because *Scribe* is implemented in the kernel and is only maintained for an old 32-bit Linux kernel (2.6.35), we had to run our experiments inside a virtual machine (kindly provided to us by *Scribe*'s authors, as the source tree was broken at the time of our experiments). To allow for a more faithful comparison, we ran Varan inside the same virtual machine.

We used *Redis* as a benchmark, running the same workload as before, and configured both systems to record the execution to persistent storage. We recorded an overhead of 53 % for *Scribe*,[1] compared to 14 % for Varan.

7.4 SECURITY HONEYPOT

The ability to perform parallel multi-version execution could be also used to deploy high-interaction honeypots that can offer a detailed account of the attackers' activities. These honeypots could run vulnerable versions of security critical services, and compare their behaviour, in real time, against that

[1] The overhead we measured for *Scribe* is higher than the overhead reported in [LVN10]; however, note that the original work used less aggressive benchmarks such as *Apache httpd*. The use of a virtual machine also affected the result.

of a known secure version. Any divergence detected could be resolved in favour of the secure version and logged for further analysis. This would provide a fine-grained understanding of the anatomy of an attack, which is typically not possible with today's honeypot deployments. The construction of such a honeypot system using an existing multi-variant execution environment has been already proposed in the past [JWF10] but, to our knowledge, no implementation of such mechanism exists.

We have implemented this security mechanism in a simple prototype system based on top of VARAN [Pes13]. This system runs two variants of an application—*public* and *private*—in lockstep and monitors their behavior. A monitor synchronises the execution of the two variants and grants or denies access to operating system resources. Any divergence in the variant behavior might be a sign of an ongoing attack. While this prototype reuses a large part of the VARAN's codebase, namely the preloader, the monitor and the system call interception mechanism, there are several important differences.

In a regular multi-variant or multi-version application, all versions run on the same machine under the same security restrictions. However, in the honeypot system, each variant has a different access to operating system resources such as file system and network.

For example, consider a web service running on top of the multi-variant honeypot (MVH). A public variant is granted access to network resources while the private variant can only access the file system. Every request received by the public variant is forwarded to the private variant through the monitor, and after being processed, the result is sent back to the public version. When the public version is compromised due to an existing (potentially unknown) vulnerability, it might issue an additional system call in an attempt to access local resources; this will result in a divergence which will be detected and recorded by the monitor. The monitor will also close the connection to the private variant to protect any critical resources.

An altenative approach has been used in the past to implement shadow honeypots [Ana+05], where both the vulnerable and secure version share the same process and address space and the runtime monitor switches between the two versions depending on the request. This approach can reduce the resource usage, but significantly increases the potential attack surface thereby reducing the security compared to our solution where both versions are stricly isolated.

The architecture of our prototype system is depicted in Figure 7.2. To enforce strong separation, each variant can be run on a different machine. Even if attackers manage to subvert our monitoring infrastructure and gain full access to the operating system, the impact of such an intrusion can be minimal as the machine which runs the public version would not contain any critical resources and would ideally be running inside a demilitarised zone (DMZ) to prevent access to other machines on the network. Such an architecture, however, precludes the use of the shared ring buffer and the event streaming (§5.2.3). Instead, in our prototype, we use network sockets to transfer events between variants and the monitor. This has a negative impact on the performance; we have observed a per-system call overhead of 18 to 25× in our microbenchmarks. However, we believe that in the case of honeypot systems, performance overhead is not the most important factor, as the applications running on top of such systems are not expected to be used in performance critical scenarios.

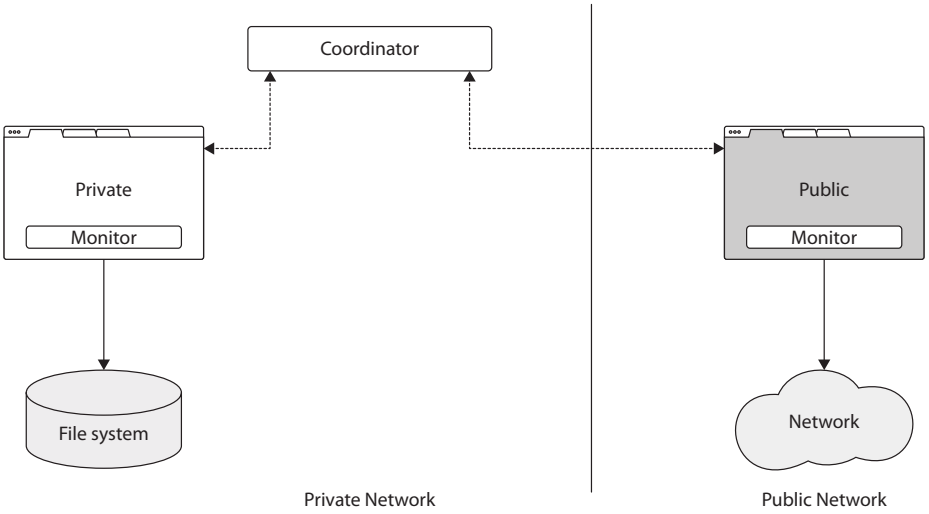


Figure 7.2: The architecture of the multi-variant honeypot system.

We have tested our prototype by running two variants of *Lighttpd* 1.4.28 on top of our honeypot system, where the private variant has been compiled with all the security mechanisms provided by GCC enabled, while the public variant did not use any of these. We then used an existing exploit [Erio8] to attack the public variant and inject shellcode[2] which tries to add a root user with a custom password to the system. This attack has been successfully detected and prevented by the honeypot monitor.

7.5 FUTURE WORK

There are a number of other ways in which both Mx and Varan could be extended in the future. In general, we would like to increase their flexibility, in order to expand the range of versions we could run in parallel. This means both tolerance to a wider set of changes as well as ability to detect different types of divergences.

The recovery mechanism in Mx is based on the ability of restarting the crashed version using a lightweight checkpointing mechanism. In our current implementation, we only keep the latest checkpoint for performance reasons. However, this means that Mx cannot recover a fault caused by a change prior to the last checkpoint. We could raise this limitation by keeping more checkpoints and iteratively retrying them if we fail to recover using a more recent one. There is a trade-off involved: keeping more checkpoints incurs a higher performance overhead and we would also need to keep the log of all the system calls between the checkpointed state and the point of failure, akin to Varan's record & replay mechanism. Other option is to improve our checkpointing mechanism by implementing it as a loadable kernel module that only stores the part of the state needed for recovery [Sri+04].

While Varan allows executing a large number of versions in parallel, the number of versions is currently set in the beginning. We would like enhance our prototype implementation with the ability to

[2] http://www.shell-storm.org/shellcode/files/shellcode-658.php

dynamically adjust the number of versions that are run concurrently. This will ensure that multi-version applications will be able to utilise all available resources (*i.e.* idle processor time) without affecting the overall system performance during peak load.

While scaling down the number of versions is straightforward, to scale up we would need the ability to start new versions and allow them to catch up with the leader. There are different ways in which such a support could be implemented. For applications structured around a central dispatch loop (*e.g.*, network servers or applications with graphical user interface), we could infer the loop headers (either statically [SGL96; Hav97] or dynamically [SIN11]) and allow the new versions to "catch up" with the leader at the beginning of the dispatch loop. Some DSU systems [Hay+12] use a similar approach, but rather than inferring the loop headers, they require the programmer to annotate the application. We could combine Varan with one of these DSU systems to take advantage of these annotations.

8

Conclusion

Software updates present a difficult challenge to the software maintenance process. Too often, updates result in failures, and users face the uncomfortable choice between using an old stable version which misses recent bug and security fixes, and using a new version which improves the software in certain ways, only to introduce other bugs and security vulnerabilities.

We tackle this problem using a simple but effective *multi-version execution* based approach. Whenever a new update becomes available, instead of upgrading the software to the new version, we run the new version in parallel with the old one; by carefully coordinating their executions and selecting the behaviour of the more reliable version when they diverge, we create a more secure and dependable multi-version application. With the widespread availability of multi-core processors, multi-version execution might become a viable approach for increasing the reliability and availability of updated software systems.

In this thesis, we have proposed two different schemes for implementing the multi-version execution technique. The first is a failure recovery scheme, which allows programs to survive errors introduced by software updates. The second is a transparent failover scheme, which allows N versions to be run in parallel; if one these versions fails, the mechanism transparently switches to another version without any disruption in service.

To enable these scenarios, we need a runtime monitor that enables the execution of multiple versions in parallel, coordinates and synchronises their execution, making them appear as a single application to any outside entities. Unfortunately, existing monitors impose a large performance overhead and none of them scales well with the number of versions, since the runtime monitor usually acts as a performance bottleneck. Moreover, none of the existing monitors handles the case when the execution of versions run in parallel diverges. We have presented two different designs for building runtime monitor which address these drawbacks; the first, called Mx is focused on surviving crashes caused by bugs introduced in software updates and the second, called Varan is aimed at running a large number of versions side-by-side with low overhead.

Mx is a multi-version execution system implementing a novel fault recovery mechanism targeting application updates which might introduce potential crash bugs. We have shown how Mx can be successfully applied in practice by recovering from real crashes in several real applications such as *GNU Coreutils*, a set of user-level UNIX applications; *Lighttpd*, a popular web server used by several high-traffic websites; and *Redis*, an advanced key-value data store server used by many well-known services.

VARAN is a multi-version execution framework that combines binary rewriting with a novel event-streaming architecture to significantly reduce performance overhead and scale well with the number of versions, without relying on intrusive kernel modifications. Our evaluation shows that VARAN can run multi-version applications based on popular C10k network servers with only a modest performance overhead, and can be effectively used to increase software reliability.

Bibliography

[AK09] Jeff Arnold and M. Frans Kaashoek. "Ksplice: Automatic Rebootless Kernel Updates". In: *Proceedings of the 4th European Conference on Computer Systems (EuroSys'09)*. Nuremberg, Germany, Mar. 2009.

[AKS98] Albert Alexandrov, Paul Kmiec, and Klaus Schauser. *Consh: Confined Execution Environment for Internet Computations*. `http://itslab.inf.kyushu-u.ac.jp/ssr/Links/alexandrov98consh.pdf`. Dec. 1998.

[Alt+05] Gautam Altekar et al. "OPUS: Online Patches and Updates for Security". In: *Proceedings of the 14th USENIX Security Symposium (USENIX Security'05)*. Baltimore, MD, USA, Aug. 2005.

[Ana+05] K. G. Anagnostakis et al. "Detecting targeted attacks using shadow honeypots". In: *Proceedings of the 14th USENIX Security Symposium (USENIX Security'05)*. Baltimore, MD, USA, Aug. 2005.

[AR00] Anurag Acharya and Mandar Raje. "MAPbox: Using Parameterized Behavior Classes to Confine Applications". In: *Proceedings of the 9th USENIX Security Symposium (USENIX Security'00)*. Denver, CO, USA, Aug. 2000.

[Avi95] Algirdasi A. Avižienis. In: *Software Fault Tolerance*. Ed. by Michael R. Luy. John Wiley & Sons, 1995. Chap. The Methodology of N-Version Programming, pp. 23–46.

[BAM14] Benoit Baudry, Simon Allier, and Martin Monperrus. "Tailored Source Code Transformations to Synthesize Computationally Diverse Program Variants". In: *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'14)*. San Jose, CA, USA, July 2014.

[Bau+05] Andrew Baumann et al. "Providing dynamic update in an operating system". In: *Proceedings of the 2005 USENIX Annual Technical Conference (USENIX ATC'05)*. Anaheim, CA, USA, Apr. 2005.

[BBS95] Andrew Berman, Virgil Bourassa, and Erik Selberg. "TRON: Process-specific File Protection for the UNIX Operating System". In: *Proceedings of the 1995 USENIX Technical Conference (TCON'95)*. New Orleans, LA, USA, Jan. 1995.

[BCL07] Danilo Bruschi, Lorenzo Cavallaro, and Andrea Lanzi. "Diversified Process Replicae for Defeating Memory Error Exploits". In: *Proceedings of the International Performance, Computing, and Communications Conference (IPCCC'07)*. New Orleans, LA, USA, Apr. 2007.

[BCS06] Sandeep Bhatkar, Abhishek Chaturvedi, and R. Sekar. "Dataflow Anomaly Detection". In: *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P'06)*. Oakland, CA, USA, May 2006.

[Bea+02] Steve Beattie et al. "Timing the Application of Security Patches for Optimal Uptime". In: *Proceedings of the 16th USENIX Conference on System Administration (LISA'02)*. Philadelphia, PA, USA, Nov. 2002.

[Ber+10] Tom Bergan et al. "CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution". In: *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'10)*. Pittsburgh, PA, USA, Mar. 2010.

[BFM99] Lee Badger, Mark Feldman, and Glenwood Md. "Hardening COTS Software with Generic Software Wrappers". In: *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P'99)*. Oakland, CA, USA, May 1999.

[BGM02] Massimo Bernaschi, Emanuele Gabrielli, and Luigi V. Mancini. "Remus: A Security-enhanced Operating System". In: *ACM Transactions on Information and System Security (TISSEC)* 5.1 (Feb. 2002), pp. 36–61. ISSN: 1094-9224.

[BH07] Luiz André Barroso and Urs Hölzle. "The Case for Energy-Proportional Computing". In: *Computer* 40 (12 2007), pp. 33–37.

[Bha+06] Sanjay Bhansali et al. "Framework for Instruction-level Tracing and Analysis of Program Executions". In: *Proceedings of the 2006 Virtual Execution Environments (VEE'06)*. Ottawa, Canada, Mar. 2006.

[BSV07] P. Bergheaud, D. Subhraveti, and M. Vertes. "Fault Tolerance in Multiprocessor Systems Via Application Cloning". In: *Proceedings of the 27th IEEE International Conference on Distributed Computing Systems (ICDCS'07)*. Toronto, Canada, June 2007.

[BZ06] Emery D. Berger and Benjamin G. Zorn. "DieHard: probabilistic memory safety for unsafe languages". In: *Proceedings of the Conference on Programing Language Design and Implementation (PLDI'06)*. Ottawa, Canada, June 2006.

[CA78] Liming Chen and Algirdas Avizienis. "N-version programming: A Fault-tolerance approach to reliability of software operation". In: *Proceedings of the 8th IEEE International Symposium on Fault Tolerant Computing (FTCS'78)*. Toulouse, France, June 1978.

[Can+04] George Candea et al. "Microreboot—A technique for cheap recovery". In: *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI'04)*. San Francisco, CA, USA, Dec. 2004.

[CCD07] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. "Identifying Changed Source Code Lines from Version Repositories". In: *Proceedings of the 4th International Workshop on Mining Software Repositories (MSR'07)*. Minneapolis, MN, USA, May 2007.

[CD99] Jonathan E. Cook and Jeffrey A. Dage. "Highly Reliable Upgrading of Components". In: *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*. Los Angeles, CA, USA, May 1999.

[CGC08] Jim Chow, Tal Garfinkel, and Peter M. Chen. "Decoupling Dynamic Program Analysis from Execution in Virtual Environments". In: *Proceedings of the 2008 USENIX Annual Technical Conference (USENIX ATC'08)*. Boston, MA, USA, June 2008.

[CH12] Cristian Cadar and Petr Hosek. "Multi-Version Software Updates". In: *Proceedings of the 4th Workshop on Hot Topics in Software Upgrades (HotSWUp'12)*. Zurich, Switzerland, June 2012.

[Che+07] Haibo Chen et al. "POLUS: A POwerful Live Updating System". In: *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*. Minneapolis, MN, USA, May 2007.

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

[Cle15] Alex Clemmer. *84% of a single-threaded 1KB write in Redis is spent in the kernel*. Feb. 2015.

[CMN86] Stuart Card, Thomas Moran, and Allen Newell. "The model human processor: an engineering model for human performance". In: *Handbook of Perception and Human Performance* 2 (1986), pp. 1–35.

[Cod14] Codenomicon. *The Heartbleed Bug*. http://heartbleed.com/. Apr. 2014.

[Cor10] Jonathan Corbet. *Cross Memory Attach*. http://lwn.net/Articles/405346/. 2010.

[Cor12] Jonathan Corbet. *Yet another new approach to seccomp*. http://lwn.net/Articles/475043/. Jan. 2012.

[Cor14] Jonathan Corbet. *Extending extended BPF*. https://lwn.net/Articles/603983/. July 2014.

[Cos+05] Manuel Costa et al. "Vigilante: end-to-end containment of Internet worms". In: *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*. Brighton, UK, Oct. 2005.

[Cow+00] Crispin Cowan et al. "SubDomain: Parsimonious Server Security". In: *Proceedings of the 14th USENIX Conference on System Administration (LISA'00)*. New Orleans, LA, USA, Dec. 2000.

[Cox+06] Benjamin Cox et al. "N-variant systems: a secretless framework for security through diversity". In: *Proceedings of the 15th USENIX Security Symposium (USENIX Security'06)*. Vancouver, BC, Canada, July 2006.

[Coz+08] Anthony Cozzie et al. "Digging for data structures". In: *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*. San Diego, CA, USA, Dec. 2008.

[CPW10] Cristian Cadar, Peter Pietzuch, and Alexander L. Wolf. "Multiplicity Computing: A Vision of Software Engineering for Next-Generation Computing Platform Applications". In: *Proceedings of the FSE/SDP workshop on the Future of Software Engineering Research (FoSER'10)*. Santa Fe, NM, USA, Nov. 2010.

[Cra+07] Olivier Crameri et al. "Staged deployment in Mirage, an integrated software upgrade testing and distribution system". In: *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*. Stevenson, WA, USA, Oct. 2007.

[dLM07] Marcelo d'Amorim, Steven Lauterburg, and Darko Marinov. "Delta execution for efficient state-space exploration of object-oriented programs". In: *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'07)*. London, UK, July 2007.

[Dou+08] John R. Douceur et al. "Leveraging legacy code to deploy desktop applications on the web". In: *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*. San Diego, CA, USA, Dec. 2008.

[Dun+02] George W. Dunlap et al. "ReVirt: Enabling Intrusion Analysis Through Virtual-machine Logging and Replay". In: *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI'02)*. Boston, MA, USA, Dec. 2002.

[Dun+08] George W. Dunlap et al. "Execution Replay of Multiprocessor Virtual Machines". In: *Proceedings of the 2008 Virtual Execution Environments (VEE'08)*. Seattle, WA, USA, Mar. 2008.

[EGR01] Sebastian Elbaum, David Gable, and Gregg Rothermel. "The impact of software evolution on code coverage information". In: *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*. Florence, Italy, Nov. 2001.

[Eri08] Jon Erickson. In: *Hacking: The Art of Exploitation*. No Starch Press, 2008. Chap. 0x400 Networking, pp. 195–280.

[FC08] Bryan Ford and Russ Cox. "Vx32: Lightweight User-level Sandboxing on the x86". In: *Proceedings of the 2008 USENIX Annual Technical Conference (USENIX ATC'08)*. Boston, MA, USA, June 2008.

[Fra10] Michael Franz. "E unibus pluram: massive-scale software diversity as a defense mechanism". In: *Proceedings of the 2010 Workshop on New Security Paradigms (NSPW'10)*. Concord, MA, USA, Sept. 2010.

[FRK02] Hubertus Franke, Rusty Russell, and Matthew Kirkwood. "Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux". In: *Proceedings of the 2002 Ottawa Linux Symposium (OLS'02)*. Ottawa, Canada, June 2002.

[Gar03] Tal Garfinkel. "Traps and Pitfalls: Practical Problems in in System Call Interposition based Security Tools". In: *Proceedings of the 10th Network and Distributed System Security Symposium (NDSS'03)*. San Diego, CA, USA, Feb. 2003.

[Gee+06] Dennis Geels et al. "Replay debugging for distributed applications". In: *Proceedings of the 2006 USENIX Annual Technical Conference (USENIX ATC'06)*. Boston, MA, USA, May 2006.

[Giu+13] Cristiano Giuffrida et al. "Back to the Future: Fault-tolerant Live Update with Time-traveling State Transfer". In: *Proceedings of the 27th USENIX Conference on System Administration (LISA'13)*. Washington, D.C., USA, Nov. 2013.

[GKT13] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. "Safe and Automatic Live Update for Operating Systems". In: *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. Houston, TX, USA, Mar. 2013.

[GMV08] John Giacomoni, Tipp Moseley, and Manish Vachharajani. "FastForward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue". In: *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*. Salt Lake City, UT, USA, Feb. 2008.

[Gol+96] Ian Goldberg et al. "A Secure Environment for Untrusted Helper Applications Confining the Wily Hacker". In: *Proceedings of the 6th USENIX Security Symposium (USENIX Security'96)*. San Jose, CA, USA, July 1996.

[GPR04] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. "Ostia: A Delegating Architecture for Secure System Call Interposition". In: *Proceedings of the 11th Network and Distributed System Security Symposium (NDSS'04)*. San Diego, CA, USA, Feb. 2004.

[GR03] Tal Garfinkel and Mendel Rosenblum. "A Virtual Machine Introspection Based Architecture for Intrusion Detection". In: *Proceedings of the 10th Network and Distributed System Security Symposium (NDSS'03)*. San Diego, CA, USA, Feb. 2003.

[GRS04] Debin Gao, Michael K. Reiter, and Dawn Song. "Gray-box Extraction of Execution Graphs for Anomaly Detection". In: *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS'04)*. Washington, D.C., USA, Oct. 2004.

[Guo+08] Zhenyu Guo et al. "R2: An Application-Level Kernel for Record and Replay". In: *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*. San Diego, CA, USA, Dec. 2008.

[Har+12] Mark Harman et al. "The GISMOE challenge: constructing the pareto program surface using genetic programming to find better programs". In: *Proceedings of the 27th IEEE International Conference on Automated Software Engineering (ASE'12)*. Essen, Germany, Sept. 2012.

[Har09] Ross Harmess. *Flipping Out*. `http://code.flickr.com/blog/2009/12/02/flipping-out/`. 2009.

[Hav97] Paul Havlak. "Nesting of Reducible and Irreducible Loops". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19.4 (July 1997), pp. 557–567. ISSN: 0164-0925.

[Hay+11] Christopher M. Hayden et al. "State Transfer for Clear and Efficient Runtime Updates". In: *Proceedings of the 3rd Workshop on Hot Topics in Software Upgrades (HotSWUp'11)*. Hannover, Germany, Apr. 2011.

[Hay+12] Christopher M. Hayden et al. "Kitsune: Efficient, General-purpose Dynamic Software Updating for C". In: *Proceedings of the 27th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'12)*. Tucson, AZ, USA, Oct. 2012.

[HB97] P. Helman and J. Bhangoo. "A statistically based system for prioritizing information exploration under uncertainty". In: *IEEE Transactions on Systems, Man and Cybernetics* 27.4 (July 1997), pp. 449–466. ISSN: 1083-4427. DOI: 10.1109/3468.594912.

[HB99] Galen Hunt and Doug Brubacher. "Detours: Binary Interception of Win32 Functions". In: *Proceedings of the 3rd USENIX Windows NT Symposium (USENIX NT'99)*. Seattle, WA, USA, July 1999.

[HBT13] Tomas Hruby, Herbert Bos, and Andrew S. Tanenbaum. "When Slower is Faster: On Heterogeneous Multicores for Reliable Systems". In: *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC'13)*. San Jose, CA, USA, June 2013.

[HC13] Petr Hosek and Cristian Cadar. "Safe Software Updates via Multi-version Execution". In: *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*. San Francisco, CA, USA, May 2013.

[HC15] Petr Hosek and Cristian Cadar. "Varan the Unbelievable: An efficient N-version execution framework". In: *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. Istanbul, Turkey, Mar. 2015.

[Her+13] Kim Herzig et al. "Predicting Defects Using Change Genealogies". In: *Proceedings of the 24th International Symposium on Software Reliability Engineering (ISSRE'13)*. Pasadena, CA, USA, Nov. 2013.

[HFS98] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. "Intrusion Detection Using Sequences of System Calls". In: *Journal of Computer Security* 6.3 (Aug. 1998), pp. 151–180. ISSN: 0926-227X.

[HJZ13] Kim Herzig, Sascha Just, and Andreas Zeller. "It's Not a Bug, It's a Feature: How Misclassification Impacts Bug Prediction". In: *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*. San Francisco, CA, USA, May 2013.

[Jal07] Aamer Jaleel. *Memory Characterization of Workloads Using Instrumentation-Driven Simulation*. Tech. rep. Intel Corporation, 2007.

[Joh09] Robert Johnson. "OOPSLA keynote: Moving Fast at Scale - Lessons Learned at Facebook". In: *Proceedings of the 24th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'09)*. Orlando, FL, USA, Oct. 2009.

[Joh10] Robert Johnson. *More Details on Today's Outage*. https://www.facebook.com/notes/facebook-engineering/more-details-on-todays-outage/431441338919. Facebook, Sept. 2010.

[JPS11] Suman Jana, Donald E. Porter, and Vitaly Shmatikov. "TxBox: Building Secure, Efficient Sandboxes with System Transactions". In: *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P'11)*. Oakland, CA, USA, May 2011.

[JS99] K. Jain and R. Sekar. "User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement". In: *Proceedings of the 6th Network and Distributed System Security Symposium (NDSS'99)*. San Diego, CA, USA, Feb. 1999.

[JWF10] Todd Jackson, Christian Wimmer, and Michael Franz. "Multi-Variant Program Execution for Vulnerability Detection and Analysis". In: *Proceedings of the 6th Cyber Security and Information Intelligence Research Workshop (CSIIRW'10)*. Oak Ridge, TN, USA, Apr. 2010.

[KBA02] Vladimir Kiriansky, Derek Bruening, and Saman P Amarasinghe. "Secure Execution via Program Shepherding." In: *Proceedings of the 11th USENIX Security Symposium (USENIX Security'02)*. San Francisco, CA, USA, Aug. 2002.

[Kes11] Eddie Kessler. *Blogger Incident Report*. Google, May 2011. URL: http://status.blogger.com/2011/05/blogger-incident-report-by-eddie.html.

[Kim+01] Moonjoo Kim et al. "Java-MaC: a Run-time Assurance Tool for Java Programs". In: *Proceedings of the 2001 Runtime Verification (RV'01)*. Paris, France, July 2001.

[Kim+06] Sunghun Kim et al. "Automatic Identification of Bug-Introducing Changes". In: *Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*. Tokyo, Japan, Sept. 2006.

[Kim01] Moonjoo Kim. "Information extraction for run-time formal analysis". PhD thesis. University of Pennsylvania, 2001.

[KKB12] Chang Hwan Peter Kim, Sarfraz Khurshid, and Don Batory. "Shared Execution for Efficiently Testing Product Lines". In: *Proceedings of the 23rd International Symposium on Software Reliability Engineering (ISSRE'12)*. Dallas, TX, USA, Nov. 2012.

[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 (TSE)* 12.1 (Jan. 1986), pp. 96–109. ISSN: 0098-5589.

[KL90] John C. Knight and Nancy G. Leveson. "A Reply to the Criticisms of the Knight &Amp; Leveson Experiment". In: *ACM SIGSOFT Software Engineering Notes* 15.1 (Jan. 1990), pp. 24–35. ISSN: 0163-5948.

[Kuh55] Harold W. Kuhn. "The Hungarian Method for the assignment problem". In: *Naval Research Logistics Quarterly* 2 (1955), pp. 83–97.

[Kwa12] Michael Kwan. "Detouring program execution in binary applications". MA thesis. Imperial College London, 2012.

[KZ13] Taesoo Kim and Nickolai Zeldovich. "Practical and effective sandboxing for non-root users". In: *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC'13)*. San Jose, CA, USA, June 2013.

[Lam78] Leslie Lamport. "Time, clocks, and the ordering of events in a distributed system". In: *Communications of the Association for Computing Machinery (CACM)* 21.7 (July 1978), pp. 558–565. ISSN: 0001-0782.

[LAV94] L.J. Levrouw, K.M.R. Audenaert, and J.M. Van Campenhout. "A New Trace And Replay System For Shared Memory Programs Based On Lamport Clocks". In: *Proceedings of the 6th IEEE International Parallel & Distributed Processing Symposium (IPDPS'94)*. Dallas, TX, USA, Oct. 1994.

[LBC10] Patrick Pak-Ching Lee, Tian Bu, and Girish Chandranmenon. "A lock-free, cache-efficient multi-core synchronization mechanism for line-rate network traffic monitoring". In: *Proceedings of the 24th IEEE International Parallel & Distributed Processing Symposium (IPDPS'10)*. Atlanta, GA, USA, Apr. 2010.

[LCB11] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. "DTHREADS: Efficient Deterministic Multithreading". In: *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*. Cascais, Portugal, Oct. 2011.

[Lev66] Vladimir I Levenshtein. "Binary codes capable of correcting deletions, insertions and reversals". In: *Soviet physics doklady*. Vol. 10. 1966, p. 707.

[Ley14] John Leyden. *AVG on Heartbleed: It's dangerous to go alone. Take this (an AVG tool)*. http://www.theregister.co.uk/2014/05/20/heartbleed_still_prevalent/. AVG, May 2014.

[LSR14] Fan Long, Stelios Sidiroglou-Douskos, and Martin Rinard. "Automatic Runtime Error Repair and Containment via Recovery Shepherding". In: *Proceedings of the Conference on Programing Language Design and Implementation (PLDI'14)*. Edinburgh, UK, June 2014.

[Luk+05] Chi-Keung Luk et al. "Pin: building customized program analysis tools with dynamic instrumentation". In: *Proceedings of the Conference on Programing Language Design and Implementation (PLDI'05)*. Chicago, IL, USA, June 2005.

[LVN10] Oren Laadan, Nicolas Viennot, and Jason Nieh. "Transparent, Lightweight Application Execution Replay on Commodity Multiprocessor Operating Systems". In: *Proceedings of the ACM SIGMETRICS 2010 (SIGMETRICS'10)*. New York, NY, USA, June 2010.

[MB09] Kristis Makris and Rida A. Bazi. "Immediate Multi-Threaded Dynamic Software Updates Using Stack Reconstruction". In: *Proceedings of the 2009 USENIX Annual Technical Conference (USENIX ATC'09)*. San Diego, CA, USA, June 2009.

[MB12] Matthew Maurer and David Brumley. "TACHYON: Tandem Execution for Efficient Live Patch Testing". In: *Proceedings of the 21st USENIX Security Symposium (USENIX Security'12)*. Bellevue, WA, USA, Aug. 2012.

[MHC14] Paul Dan Marinescu, Petr Hosek, and Cristian Cadar. "Covrig: A Framework for the Analysis of Code, Test, an Coverage Evolution in Real Software". In: *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'14)*. San Jose, CA, USA, July 2014.

[MJ93] Steven McCanne and Van Jacobson. "The BSD Packet Filter: A New Architecture for User-level Packet Capture". In: *Proceedings of the 1993 Winter USENIX Conference*. San Diego, CA, USA, Jan. 1993.

[MM06] Stephen McCamant and Greg Morrisett. "Evaluating SFI for a CISC architecture". In: *Proceedings of the 15th USENIX Security Symposium (USENIX Security'06)*. Vancouver, BC, Canada, July 2006.

[MR07] Kristis Makris and Kyung Dong Ryu. "Dynamic and Adaptive Updates of Non-quiescent Subsystems in Commodity Operating System Kernels". In: *Proceedings of the 2nd European Conference on Computer Systems (EuroSys'07)*. Lisbon, Portugal, Mar. 2007.

[Mur+10] Alessandro Murgia et al. "A machine learning approach for text categorization of fixing-issue commits on CVS". In: *Proceedings of the 4th International Symposium on Empirical Software Engineering and Measurement (ESEM'10)*. Bolzano, Italy, Sept. 2010.

[Nan+06] Susanta Nanda et al. "BIRD: Binary Interpretation using Runtime Disassembly". In: *Proceedings of the 4th International Symposium on Code Generation and Optimization (CGO'06)*. New York, NY, USA, Mar. 2006.

[Nea+06] Iulian Neamtiu et al. "Practical dynamic software updating for C". In: *Proceedings of the Conference on Programing Language Design and Implementation (PLDI'06)*. Ottawa, Canada, June 2006.

[NPC05] Satish Narayanasamy, Gilles Pokam, and Brad Calder. "BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging". In: *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA'05)*. Washington, D.C., USA, June 2005.

[Pan15] Independent Enquiry Panel. *NATS System Failure 12 December 2014 – Interim Report*. Jan. 2015.

[Per+09] Jeff H. Perkins et al. "Automatically patching errors in deployed software". In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP'09)*. Big Sky, MT, USA, Oct. 2009.

[Pes13] Giuseppe Pes. "Design of a Honeypot via System Call Interposition". MA thesis. Imperial College London, 2013.

[PHG12] Mathias Payer, Tobias Hartmann, and Thomas R. Gross. "Safe Loading – A Foundation for Secure Execution of Untrusted Programs". In: *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P'12)*. San Francisco, CA, USA, May 2012.

[Por+10] Georgios Portokalidis et al. "Paranoid Android: Versatile Protection for Smartphones". In: *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC'10)*. Austin, TX, USA, Dec. 2010.

[Pro02] Niels Provos. "Improving host security with system call policies". In: *Proceedings of the 11th USENIX Security Symposium (USENIX Security'02)*. San Francisco, CA, USA, Aug. 2002.

[Qin+05] Feng Qin et al. "Rx: treating bugs as allergies—a safe method to survive software failures". In: *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*. Brighton, UK, Oct. 2005.

[RC96] Mark Russinovich and Bryce Cogswell. "Replay for Concurrent Non-deterministic Shared-memory Applications". In: *Proceedings of the Conference on Programing Language Design and Implementation (PLDI'96)*. Philadelphia, PA, USA, May 1996.

[RD99] Michiel Ronsse and Koen De Bosschere. "RecPlay: A Fully Integrated Practical Record/Replay System". In: *ACM Transactions on Computer Systems (TOCS)* 17.2 (May 1999), pp. 133–152. ISSN: 0734-2071.

[Rin+04] Martin Rinard et al. "Enhancing Server Availability and Security Through Failure-Oblivious Computing". In: *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI'04)*. San Francisco, CA, USA, Dec. 2004.

[Roe+12] Ryan Roemer et al. "Return-Oriented Programming: Systems, Languages, and Applications". In: *ACM Transactions on Information and System Security (TISSEC)* 15.1 (Mar. 2012), 2:1–2:34.

[Ros67] Douglas T. Ross. "The AED free storage package". In: *Communications of the Association for Computing Machinery (CACM)* 10 (8 Aug. 1967), pp. 481–492.

[RW97] D.S. Rosenblum and E.J. Weyuker. "Using coverage information to predict the cost-effectiveness of regression testing strategies". In: *IEEE Transactions on Software Engineering (TSE)* 23.3 (1997), pp. 146–156.

[Sai05] Yasushi Saito. "Jockey: a user-space library for record-replay debugging". In: *Proceedings of the 6th International Workshop on Automated Debugging (AADEBUG'05)*. Monterey, CA, USA, Sept. 2005.

[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 European Conference on Computer Systems (EuroSys'09)*. Nuremberg, Germany, Mar. 2009.

[Sal+11] Babak Salamat et al. “Run-Time Defense against Code Injection Attacks using Replicated Execution”. In: *IEEE Transactions on Dependable and Secure Computing (TDSC)* 8 (4 2011), pp. 588–601.

[Sch+14] Eric Schulte et al. “Software Mutational Robustness”. In: *Genetic Programming and Evolvable Machines* 15.3 (Sept. 2014), pp. 281–312. ISSN: 1389-2576.

[Sea] Mark Seaborn. *Plash*. http://plash.beasts.org/.

[Sek+01] R. Sekar et al. “A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors”. In: *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P'01)*. Oakland, CA, USA, May 2001.

[SGL96] Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee. “Identifying Loops Using DJ Graphs”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 18.6 (Nov. 1996), pp. 649–658. ISSN: 0164-0925.

[Sha01] Lui Sha. “Using Simplicity to Control Complexity”. In: *IEEE Software* 18.4 (July 2001), pp. 20–28. ISSN: 0740-7459.

[Shy+09] Alex Shye et al. “PLR: A Software Approach to Transient Fault Tolerance for Multicore Architectures”. In: *IEEE Transactions on Dependable and Secure Computing (TDSC)* 6 (2 2009), pp. 135–148.

[SIK07] Stelios Sidiroglou, Sotiris Ioannidis, and Angelos D. Keromytis. “Band-aid patching”. In: *Proceedings of the 3rd Workshop on Hot Topics in System Dependability (HotDep'07)*. Edinburgh, UK, June 2007.

[Sin01] Amit Singhal. “Modern information retrieval: a brief overview”. In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 24 (2001), pp. 35–43.

[SIN11] Yukinori Sato, Yasushi Inoguchi, and Tadao Nakamura. “On-the-fly Detection of Precise Loop Nests Across Procedures on a Dynamic Binary Translation System”. In: *Proceedings of the 8th ACM International Conference on Computing Frontiers (CF'11)*. Ischia, Italy, May 2011.

[SK06] Stelios Sidiroglou and Angelos D. Keromytis. “Execution Transactions for Defending Against Software Failures: Use and Evaluation”. In: *Springer International Journal of Information Security (IJIS)* 5.2 (2006), pp. 77–91.

[Sol+07] Stephen Soltesz et al. “Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors”. In: *Proceedings of the 2nd European Conference on Computer Systems (EuroSys'07)*. Lisbon, Portugal, Mar. 2007.

[Spa88] Karen Sparck Jones. "Document Retrieval Systems". In: 1988. Chap. A Statistical Interpretation of Term Specificity and Its Application in Retrieval, pp. 132–142. ISBN: 0-947568-21-2.

[Spi+07] Richard P. Spillane et al. "Rapid file system development using ptrace". In: *Proceedings of the 2007 Experimental Computer Science (ExpCS'07)*. San Diego, CA, USA, June 2007.

[Sri+04] Sudarshan M. Srinivasan et al. "Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging". In: *Proceedings of the 2004 USENIX Annual Technical Conference (USENIX ATC'04)*. Boston, MA, USA, June 2004.

[Str14] Adario Strange. *Verizon Services Back Up, But Outage Prompts Fail Hashtags on Twitter*. http://mashable.com/2014/06/28/verizon-back-up/. Verizon, June 2014.

[ŚZZ05] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. "When Do Changes Induce Fixes?" In: *Proceedings of the 2005 International Workshop on Mining Software Repositories (MSR'05)*. St. Louis, MO, USA, May 2005.

[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 (CF'10)*. Bertinoro, Italy, May 2010.

[Tho+11] Martin Thompson et al. *Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads*. Tech. rep. LMAX, 2011. URL: http://lmax-exchange.github.io/disruptor/files/Disruptor-1.0.pdf.

[TXZ09] Joseph Tucek, Weiwei Xiong, and Yuanyuan Zhou. "Efficient Online Validation with Delta Execution". In: *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*. Washington, D.C., USA, Mar. 2009.

[Vee+11] Kaushik Veeraraghavan et al. "Detecting and surviving data races using complementary schedules". In: *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*. Cascais, Portugal, Oct. 2011.

[Vou90] Mladen A. Vouk. "Back-to-back testing". In: *Information and Software Technology (IST)* 32 (1 Jan. 1990), pp. 34–45.

[Wag99] David A. Wagner. *Janus: an Approach for Confinement of Untrusted Applications*. Tech. rep. UCB/CSD-99-1056. University of California at Berkley, 1999. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/1999/5271.html.

[Wah+93] Robert Wahbe et al. "Efficient software-based fault isolation". In: *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP'93)*. Asheville, NC, USA, Dec. 1993.

[Wat07] Robert N. M. Watson. "Exploiting Concurrency Vulnerabilities in System Call Wrappers". In: *Proceedings of the 1st USENIX Workshop on Offensive Technologies (WOOT'07)*. Boston, MA, USA, Aug. 2007.

[WDD00] Andreas Wespi, Marc Dacier, and Hervé Debar. "Intrusion Detection Using Variable-Length Audit Trail Patterns". In: *Proceedings of the 3th International Symposium on Recent Advances in Intrusion Detection (RAID'00)*. Toulouse, France, Oct. 2000.

[WS08] Chad Williams and Jaime Spacco. "SZZ Revisited: Verifying When Changes Induce Fixes". In: *Proceedings of the 2008 International Workshop on Defects in Large Software Systems (DEFECTS'08)*. Seattle, WA, USA, July 2008.

[XDK12] Hui Xue, Nathan Dautenhahn, and Samuel T. King. "Using Replicated Execution for a More Secure and Reliable Web Browser". In: *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS'12)*. San Diego, CA, USA, Feb. 2012.

[Yee+09] Bennet Yee et al. "Native Client: A Sandbox for Portable, Untrusted x86 Native Code". In: *Proceedings of the IEEE Symposium on Security and Privacy (IEEE S&P'09)*. Oakland, CA, USA, May 2009.

[Yin+11] Zuoning Yin et al. "How Do Fixes Become Bugs?" In: *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'11)*. Szeged, Hungary, Sept. 2011.

[YMC07] Aydan R. Yumerefendi, Benjamin Mickle, and Landon P. Cox. "TightLip: Keeping Applications from Spilling the Beans". In: *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI'07)*. Cambridge, MA, USA, Apr. 2007.

[Zai+11] Andy Zaidman et al. "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining". In: *Empirical Software Engineering (EMSE)* 16.3 (2011), pp. 325–364.