Safe and Automatic
Live Update

Ph.D. Thesis

Cristiano Giuffrida

VU University Amsterdam, 2014

vrije Universiteit amsterdam

This work was funded by European Research Council under ERC Advanced Grant 227874.

Advanced School for Computing and Imaging

This work was carried out in the ASCI graduate school.
ASCI dissertation series number 300.

ISBN 978-90-5383-072-7

Cover design by Dirk Vogt.
Printed by Wöhrmann Print Service.

VRIJE UNIVERSITEIT

SAFE AND AUTOMATIC LIVE UPDATE

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. F.A. van der Duyn Schouten,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de Faculteit der Exacte Wetenschappen
op donderdag 10 april 2014 om 13.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

CRISTIANO GIUFFRIDA

geboren te Rome, Italië

promotor: prof.dr. A.S. Tanenbaum

“All problems in computer science can be solved
by another level of indirection.”

Butler Lampson, quoting David Wheeler.

Acknowledgements

"Yesterday I learned that I received a 3-million euro grant from the EU—all for myself [. . .]. Might you be interested?". This is the email that started it all, on the now-distant date of August 15, 2008. I cannot recall many more details from that email, but I do recall that glancing over its first and last sentence was enough for me to conclude that my spam filter was far less aggressive than I had hoped for. I was literally about to discard the message, when the signature at the bottom caught my attention. It read *"Andy Tanenbaum"*, which initially only reinforced my belief that targeted phishing attacks had grown to become astonishingly common and sophisticated. After a sudden epiphany, I finally decided to keep on reading.

That was the beginning of my Ph.D. journey at the Vrije Universiteit Amsterdam. A challenging and rewarding journey which ultimately culminated in this dissertation, but which would not have been possible without the help of others. First and foremost, I would like to thank my supervisor, Andy Tanenbaum, for his constant guidance and support during all these years. Not only did he teach me how to do research and write papers, but he helped me develop passion for simple and elegant solutions and showed me that critical thinking, dedication, and perseverance can take you a long way. I am deeply indebted to him for providing me with an excellent topic, while allowing me, at the same time, to independently explore many exciting research directions that have gradually shaped my scientific interests over the years.

Next, I would like to express my gratitude to all the members of my Ph.D. thesis committee: Herbert Bos, Cristian Cadar, Bruno Crispo, Orran Krieger, and Liuba Shira. Their valuable comments greatly contributed to improving the quality of this dissertation. I am especially grateful to Orran Krieger, for his extensive and insightful comments on the text, and Cristian Cadar, for fruitful early discussions and feedback on the techniques presented in this dissertation. A special mention is in order for my internal referees. I am extremely grateful to Herbert Bos, for his constant support and excellent suggestions throughout my research work, and Bruno Crispo, for strongly encouraging me to expand my research interests into systems security.

I would also like to acknowledge the many other people I had the pleasure to work or spend time with at the Vrije Universiteit Amsterdam. First, I would like to thank those who shared their Ph.D. journeys with me, especially Stefano Ortolani, for his friendship and engagement in many joint *"not-so-side"* research projects, and my P4.69 office mates, Raja Appuswamy, Jorrit Herder, Tomas Hruby, Erik van der Kouwe, David van Moolenbroek, and Dirk Vogt, for creating an inspiring and enjoyable work environment. Next, I would like to thank all the other members of the MINIX 3 team, including Ben Gras, Philip Homburg, Lionel Sambuc, Arun Thomas, and Thomas Veerman, for their ability to challenge my most ambitious ideas and provide very valuable support on short notice. Needless to say, I will never forget the movie nights and the other team events we shared together. I am also grateful to the many excellent students who contributed to our research projects, especially Calin Iorgulescu and Anton Kuijsten, with whom I shared many sleepless nights before a paper deadline. Finally, I would like to thank all the other people from the Computer Systems group, who substantially contributed to making my doctoral years special both professionally and personally, Willem de Bruijn, Albana Gaba, Ana Oprescu, Georgios Portokalidis, Asia Slowinska, and Spyros Voulgaris, in particular.

I am also grateful to Divyesh Shah, for inviting me to join the kernel memory team at Google and for providing me with all the necessary support during my internship time in Mountain View, California. I would also like to thank all the other members of the kernel memory team, especially Ying Han and Greg Thelen, for their dedication and support. My experience at Google has further strengthened my interest in operating systems and memory management, while providing much inspiration for the RCU-based quiescence detection protocol presented in Chapter 7.

I would also like to extend my gratitude to the many friends who provided the much needed distraction and support during my Ph.D. journey, including: all my fellow *"Djangonians"*, especially the pioneers, Albana, Christian, Dirk, Stefano, and Steve, for the unforgettable moments shared together; Raja, for the way-too-many *"Lost"* evenings; Flavio, for being always there for me in the difficult times.

Last but not least, I would like to thank my family for supporting me all these years. My mother, among so many other things, for being a constant source of inspiration throughout my life, encouraging me to pursue my passion for research, and getting me interested in computers at a young age—despite her passion for archeology. My uncle and aunt, for guiding me through important decisions in my life. My cousins, Chiara and Matteo, for joining me on so many adventures. Finally, Laura, for her love, support, and much needed patience throughout this endeavor.

Cristiano Giuffrida
Amsterdam, The Netherlands, December 2013

Contents

List of Figures

List of Tables

Publications

This dissertation includes a number of research papers, as appeared in the following conference proceedings (or as recently submitted for review)[1]:

Cristiano Giuffrida, Calin Iorgulescu, and Andrew S. Tanenbaum. Mutable Checkpoint-Restart: Automating Live Update for Generic Long-running C Programs[2]. *Under review*.

Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. EDFI: A Dependable Fault Injection Tool for Dependability Benchmarking Experiments[3]. In *Proceedings of the Pacific Rim International Symposium on Dependable Computing (PRDC '13)*, pages 1–10. December 2-4, 2013, Vancouver, BC, Canada.

Cristiano Giuffrida, Calin Iorgulescu, Anton Kuijsten, and Andrew S. Tanenbaum. Back to the Future: Fault-tolerant Live Update with Time-traveling State Transfer[4]. In *Proceedings of the 27th USENIX Systems Administration Conference (LISA '13)*, pages 89–104. November 3-8, 2013, Washington, D.C., USA.
Awarded Best Student Paper.

Cristiano Giuffrida, Lorenzo Cavallaro, and Andrew S. Tanenbaum. Practical Automated Vulnerability Monitoring Using Program State Invariants[5]. In *Proceedings of the 43rd International Conference on Dependable Systems and Networks (DSN '13)*, pages 1–12. June 24-27, 2013, Budapest, Hungary.

[1] The text differs from the original version in minor editorial changes made to improve readability.

[2] Appears in Chapter 7.

[3] Appears in Chapter 5.

[4] Appears in Chapter 6.

[5] Appears in Chapter 4.

Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Safe and Automatic Live Update for Operating Systems[6]. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*, pages 279–292. March 16-20, 2013, Houston, TX, USA.

Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization[7]. In *Proceedings of the 21st USENIX Security Symposium (USENIX Security '12)*, pages 40–55. August 8-10, 2012, Bellevue, WA, USA.

Related publications not included in the dissertation are listed in the following:

Cristiano Giuffrida, and Andrew S. Tanenbaum. Safe and Automated State Transfer for Secure and Reliable Live Update. In *Proceedings of the Fourth International Workshop on Hot Topics in Software Upgrades (HotSWUp '12)*, pages 16–20. June 3, 2012, Zurich, Switzerland.

Cristiano Giuffrida, Lorenzo Cavallaro, and Andrew S. Tanenbaum. We Crashed, Now What? In *Proceedings of the Sixth Workshop on Hot Topics in System Dependability (HotDep '10)*, pages 1–8. October 3, 2010, Vancouver, BC, Canada.

Cristiano Giuffrida, and Andrew S. Tanenbaum. A Taxonomy of Live Updates. In *Proceedings of the 16th Annual ASCI Conference (ASCI '10)*, pages 1–8. November 1-3, 2010, Veldhoven, The Netherlands.

Cristiano Giuffrida, and Andrew S. Tanenbaum. Cooperative Update: A New Model for Dependable Live Update. In *Proceedings of the Second International Workshop on Hot Topics in Software Upgrades (HotSWUp '09)*, pages 1–6. October 25, 2009, Orlando, FL, USA.

Publications not related to this dissertation, but published in refereed conferences, journals, or workshops are listed in the following:

Erik van der Kouwe, Cristiano Giuffrida, and Andrew S. Tanenbaum. On the Soundness of Silence: Investigating Silent Failures Using Fault Injection Experiments. In *Proceedings of the Tenth European Dependable Computing Conference (EDCC '14)*. May 13-16, 2014, Newcastle upon Tyne, UK.

Erik van der Kouwe, Cristiano Giuffrida, and Andrew S. Tanenbaum. Evaluating Distortion in Fault Injection Experiments. In *Proceedings of the 15th IEEE Symposium on High-Assurance Systems Engineering (HASE '14)*, pages 1–8. January 9-11, 2014, Miami, FL, USA.
Awarded Best Paper.

[6] Appears in Chapter 2.

[7] Appears in Chapter 3.

Dirk Vogt, Cristiano Giuffrida, Herbert Bos, and Andrew S. Tanenbaum. Techniques for Efficient In-Memory Checkpointing. In *Proceedings of the Ninth Workshop on Hot Topics in System Dependability (HotDep '13)*, pages 1–5. November 3, 2013, Farmington, PA, USA.

Stefano Ortolani, Cristiano Giuffrida, and Bruno Crispo. Unprivileged Black-box Detection of User-space Keyloggers. In *IEEE Transactions on Dependable and Secure Computing (TDSC)*, Volume 10, Issue 1, pages 40–52, January-February 2013.

Cristiano Giuffrida, Stefano Ortolani, and Bruno Crispo. Memoirs of a Browser: A Cross-browser Detection Model for Privacy-breaching Extensions. In *Proceedings of the Seventh ACM Symposium on Information, Computer and Communications Security (ASIACCS '12)*, pages 132–152. May 1-3, 2012, Seoul, South Korea.

Ciriaco Andrea D'Angelo, Cristiano Giuffrida, and Giovanni Abramo. A Heuristic Approach to Author Name Disambiguation in Bibliometrics Databases for Large-scale Research Assessments. In *Journal of the American Society for Information Science and Technology (JASIST)*, Volume 62, Issue 2, pages 257–269, February 2011.

Stefano Ortolani, Cristiano Giuffrida, and Bruno Crispo. KLIMAX: Profiling Memory Write Patterns to Detect Keystroke-Harvesting Malware. In *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (RAID '11)*, pages 81–100. September 20-21, 2011, Menlo Park, CA, USA.

Stefano Ortolani, Cristiano Giuffrida, and Bruno Crispo. Bait your Hook: A Novel Detection Technique for Keyloggers. In *Proceedings of the 13th International Symposium on Recent Advances in Intrusion Detection (RAID '10)*, pages 198–217. September 15-17, 2010, Ottawa, ON, Canada.

Cristiano Giuffrida, and Stefano Ortolani. A Gossip-based Churn Estimator for Large Dynamic Networks. In *Proceedings of the 16th Annual ASCI Conference (ASCI '10)*, pages 1–8. November 1-3, 2010, Veldhoven, The Netherlands.

1 Introduction

“*In a system made up of many modules, it is often necessary to update one of the modules so as to provide new features or an improvement in the internal organization [. . .]. If the module does manage permanent data structures which must be modified and the system is one which is expected to continue operation throughout the change, the problem is more difficult, but it can be solved.*” With these words, in 1976, Robert Fabry introduced the first known *live update* solution able to deploy software updates on a running system without discontinuing its operations or disrupting its internal state [99]. Since then, many research and commercial live update solutions have been proposed over the years. As Fabry originally envisioned, live updates are nowadays used to improve the general organization of a running software system—for example, to fix bugs—without compromising its availability guarantees.

What Fabry could have hardly envisioned at the time is the importance of live update technologies in today’s always-on society, with the need for continuous operation rapidly emerging in many application areas with different levels of impact. Mass market software, not initially conceived with high availability in mind, attracts more and more consumers who expect nonstop operation. For instance, many unsophisticated users find it very annoying to reboot their system after a software update. In addition, with the advent of ubiquitous computing, software tailored to ordinary devices will become more sophisticated, thus naturally incurring frequent maintenance updates. It is very unlikely that most people will understand the need to turn off their car as a consequence of a high-priority security update. Even worse, one could envision legal consequences when a user is forced to update and restart his TV set while watching an expensive pay-per-view sporting event. For workstation users in companies, in turn, reduced availability directly translates to productivity loss.

In industrial systems, the need for continuous operation is even more evident. In many cases, high availability is required by design. As an example, the telephone network, a 99.999% availability system, can tolerate at most five minutes of downtime per year [117]. In other applications, such as factories and power plants, availability constraints are even more tight.

In business systems, absence of service leads to revenue loss. It has been estimated that the cost of an hour of downtime can be as high as hundreds of thousands of dollars for e-commerce service providers like Amazon and eBay and millions of dollars for brokerages and credit card companies [234]. In addition, long-lasting periods of downtime for popular services are newsworthy and affect user loyalty and investor confidence. For example, when eBay suffered its longest (22-hour) outage in 1999, the impact on the public image of the company caused 26% decline in its stock price and an estimated revenue loss between $3 million and $5 million [102].

In mission-critical and safety-critical systems, downtime or unexpected behavior can lead to catastrophic consequences. For example, unplanned downtime in a widely deployed energy management system caused a blackout affecting 50 million people in U.S. and Canada in August 2004 [242]. Another famous episode relates to the Patriot missile defense system used during the Gulf War. A software flaw prevented the interception of an incoming Scud missile, leading to the death of 28 American soldiers [52]. Many other examples of mission-critical systems can be found in aerospace, telecommunication, transportation, and medical applications.

The growing need for high availability in several real-world software systems is, however, on a collision course with the vertiginous evolution of modern software. Despite decades of research and advances in technology and software engineering, the majority of cost and effort spent during software lifetime still goes to maintenance [118]. The introduction of new features, enhancements, bug fixes, and security patches are the norm rather then the exception in widely adopted software solutions in use nowadays. In addition, current trends in software development suggest that this tendency will likely grow in the future. The complexity of modern software systems is increasing dramatically, and so are the number of bugs, security vulnerabilities, and unanticipated changes. Prior studies have determined that the number of bugs in commercial software ranges from 1 bug per 1000 lines of code to 20 bugs per 1000 lines of code [227]. As a result, software vendors continuously release new updates and publish recommendations.

With more and more frequent software releases, the traditional halt-update-restart cycle constitutes a major problem for the management of systems that must provide strong availability guarantees. A common alternative—often termed ”*rolling upgrades*” [91]—is to replicate local installations across multiple machines and deploy updates on one node at the time. This approach, however, is only applicable to particular categories of systems and cannot alone preserve nonpersistent system state. In addition, a long-lasting incremental update process may expose different service requests to different—and possibly incompatible—system versions, potentially introducing logical inconsistencies or silent data corruption [93]. Even in distributed systems, where replication naturally occurs, more compelling alternatives to rolling upgrades have been proposed in the literature [93]. Note, however, that the focus of this dissertation is specifically on live update in a *local* context and we refer the interested reader to [26; 279; 173; 91; 25] for *distributed* live update systems.

In practice, replication and downtime are ill-affordable options for many real-world systems. Not surprisingly, studies have shown that many organizations choose to forego installing updates or not to install them at all [245]. This strategy, in turn, increases the risk of unplanned downtime caused by bugs or security exploits, a concrete concern given that studies show that 75% of the attacks exploit known security vulnerabilities [190]. Emblematic is the case of SCADA systems in the post-Stuxnet era, with organizations struggling to reconcile their annual maintenance shutdowns and the need to promptly patch discovered security vulnerabilities [156].

In this complex landscape, live update technologies are increasingly gaining momentum as the most promising solution to support software evolution in high-availability environments. A major challenge to foster widespread adoption, however, is to build trustworthy live update systems which come as close to the usability and dependability guarantees of regular software updates as possible. Unfortunately, existing live update solutions for widely deployed systems programs written in C [32; 68; 214; 194; 69; 213; 36; 193; 132] fall short in both regards.

Usability is primarily threatened by the heroic effort required to support live update functionalities over time. Prior solutions largely delegate challenging tasks like state transfer and update safety to the user. Another threat stems from the update mechanisms adopted in prior solutions, which typically advocate patching changes *in place*—directly into the running version [32; 43; 69; 68; 194; 36; 214; 213]. This strategy may significantly complicate testing and debugging, other than potentially introducing address space fragmentation and run-time overhead over time [214].

Dependability is primarily threatened by the lack of mechanisms to safeguard and validate the live update process. Existing solutions make no effort to recover from programming errors at live update time or minimize the amount of trusted code. Another threat stems from the mechanisms adopted to determine safe update states. Prior solutions suffer from important limitations in this regard, ranging from low predictability to poor convergence guarantees or deadlock-prone behavior.

Current practices to mitigate these problems are largely unsatisfactory. An option is to limit live update support to simple and small security patches [36; 32], which can drastically reduce the engineering effort required to support live update functionalities over time and the probability of programming errors plaguing the live update code. This strategy, however, has no general applicability and still fails to provide strong dependability guarantees at live update time. Another option is to limit live update functionalities to programs written in safe languages [90; 269; 285], which naturally provide a somewhat simpler and safer programming environment. Unfortunately, while much modern software is now written in safe languages like Java, C still dominates the scene in systems programming, largely due to versatility, performance concerns, and direct control over low-level resources. In addition, the abundance of legacy C and C++ code discourages porting, which also suggests that a complete switch to safe higher-level languages is not on the horizon.

Safe and Automatic Live Update

In this dissertation, we argue that live update solutions should be truly *safe* and *automatic*, with general applicability to several possible classes of updates—ranging from security patches to complex version updates—and generic systems software written in C. Our investigation focuses on important system properties largely neglected by prior work in the area, including security, reliability, performance, integrity, and maintainability. To substantiate our claims, we present several major contributions and demonstrate the viability of the proposed techniques in practice. In particular, this dissertation introduces the first end-to-end live update design for the entire software stack, with support for safely and automatically updating both the operating system as well as long-running legacy programs at the application level.

At the heart of our live update mechanisms lies a novel *process-level update* strategy, which confines the different program versions in distinct processes and transfers the execution state between them. When an update becomes available, our framework allows the new version to start up and connect to the old version to request all the information from the old state it needs (e.g., data structures, even if they have changed between versions). When all the necessary data have been transferred over and the old state completely remapped into the new version (including pointers and the data pointed to), our framework terminates the old version and allows the new version to atomically and transparently resume execution. This approach allows users to install version updates in the most natural way and without affecting the original internal representation of the program. Compared to prior in-place live update solutions the tradeoff lies in generally higher memory usage and longer update times for simple updates—since we replace entire processes instead of individual functions or objects—which is arguably a modest price to pay given the increasingly low cost of RAM and the relatively infrequent occurrence of software updates.

Our *state transfer framework* automates the entire state transfer process, inspecting the state of the two program versions and seamlessly operating the necessary type and memory layout transformations, even in face of complex state changes. This strategy is empowered by a new instrumentation pass in the LLVM compiler framework [179] that maintains metadata about all the data structures in memory. Compared to prior solutions, our state transfer strategy dramatically reduces the engineering effort required to adopt and support live update functionalities over time.

Our *hot rollback* support, in turn, safeguards the live update process, allowing the system to recover from common update-time errors. When an error is detected during the update process, our framework automatically rolls back the update, terminating the new version and allowing the old version to resume execution normally, similar to an aborted atomic transaction. Thanks to our process-level update strategy, update-time errors in the new version are structurally prevented from propagating back to the old version and hindering its integrity, allowing for safe and automatic error recovery at live update time. This is in stark contrast with prior live update solutions, which offer no fallback to a working version should the update fail.

To identify update-time errors, our framework incorporates three error detection mechanisms. *Run-time error detection* relies on hardware/software exceptions to detect crashes, panics, and other abnormal run-time events. *Invariants-based detection* can automatically detect state corruption from violations of static *program state invariants* recorded by our LLVM-based instrumentation. *Time-traveling state transfer*, finally, can automatically detect generic memory errors from state differences between distinct process versions known to be equivalent by construction. When any of these errors are detected, the update is immediately aborted, preventing the new version from resuming execution in an invalid state. Instead, the system continues executing normally as if no update had been attempted. Compared to prior solutions, our error detection techniques provide a unique fault-tolerant design, able to safeguard the live update process against a broad range of update-time errors.

At the OS level, we demonstrate the effectiveness of our techniques in PROTEOS, a new research operating system designed with live update in mind. PROTEOS natively supports process-level updates on top of a multiserver OS architecture based on MINIX 3 [141], which confines all the core operating system components in separate, event-driven processes, each individually protected by the MMU. The rigorous event-driven programming model adopted in the individual operating system components allows updates to happen only in predictable and controllable system states.

Our event-driven design integrates support for *state quiescence*, a new technique that gives users fine-grained control over the live update transaction, with the ability to specify safe update state constraints on a per-update basis. This strategy dramatically simplifies reasoning about update safety, reducing the effort required to support complex updates. Thanks to our design, PROTEOS can atomically replace as many OS components as necessary in a single fault-tolerant live update transaction, safely and automatically supporting very large updates with no user-perceived impact.

Furthermore, PROTEOS combines our live update design with a compiler-based fine-grained address space randomization strategy to support *live rerandomization*, a new technique that allows individual operating system components to periodically rerandomize their memory layout at run time. This strategy seeks to harden the operating system against memory error exploits that rely on known memory locations. Live rerandomization is a novel application of live update to systems security.

At the application level, we demonstrate the effectiveness of our techniques in *Mutable Checkpoint-Restart* (*MCR*), a new live update framework for generic (multiprocess and multithreaded) long-running C programs. MCR extends our instrumentation techniques to allow legacy user programs to support safe and automatic live update with little manual effort. In particular, MCR addresses three major concerns: (i) how to enable legacy user programs to automatically reach and detect safe update states; (ii) how to enable legacy user programs to automatically remap their own state and data structures; (iii) how to enable legacy user programs to automatically reinitialize the individual execution threads in the new version.

To address the first concern, MCR relies on standard profiling techniques to instrument all the quiescent points [135] in the application and export their properties to the runtime. Our *profile-guided quiescence detection* technique uses this information to control all the in-flight events processed by the application, automatically detecting a safe update state when no event is in progress. This approach reduces the annotation effort and provides strong convergence guarantees to safe update states.

To address the second concern, MCR relies on garbage collection techniques to conservatively trace all the data structures in memory with no user annotations required. Our *mutable GC-style tracing* technique uses this information to formulate constraints on the memory layout of the new version for the benefit of our automated state transfer strategy. This approach drastically reduces the annotation effort to implement state transfer and allows our framework to automatically remap the state across versions even with partial knowledge on global pointers and data structures.

To address the third concern, MCR allows the new version to (re)initialize in a natural but controlled way, with predetermined events replayed, as needed, from the prior initialization of the old version. Our *state-driven mutable record-replay* technique relies on this strategy to prevent the new version from disrupting the old state at initialization time, while exploiting existing code paths to automatically reinitialize the individual execution threads correctly. This approach reduces the annotation effort to implement control migration functionalities, allowing our framework to restore a quiescent thread configuration in the new version with little user intervention.

We have evaluated our live update techniques both at the operating system and at the application level, focusing our experimental investigation on performance, memory usage, fault tolerance, and engineering effort. Our results evidence important limitations in prior approaches and confirm the effectiveness of our techniques in building *safe* and *automatic* live update solutions for the entire software stack.

Organization of the Dissertation

This dissertation makes several contributions, with results published in refereed conferences and workshops (Page xix). The remainder is organized as follows:

- Chapter 2 presents PROTEOS, a new research operating system designed with live update in mind. In PROTEOS, process-level updates are a first-class abstraction implemented on top of a multiserver microkernel-based operating system architecture. PROTEOS combines our live update techniques with an event-driven programming model adopted in the individual operating system components, simplifying state management and live update safety. Chapter 2 appeared in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)* [113].

- Chapter 3 presents ASR$_3$, a third-generation address space randomization (ASR) technique which significantly improves the security and performance of prior solutions. At the heart of ASR$_3$ lies live rerandomization, a technique which combines fine-grained link-time ASR strategies with our live update design to enable process-level randomization (and rerandomization) at runtime. We demonstrate the benefits of live rerandomization in the context of OS-level address space randomization and evaluate our techniques in PROTEOS. Chapter 3 appeared in *Proceedings of the 21st USENIX Security Symposium (USENIX Security '12)* [108].

- Chapter 4 presents RCORE, an efficient dynamic program monitoring infrastructure to perform automated security vulnerability monitoring. This chapter details the program state invariants adopted by our live update techniques and generalizes their application to concurrent memory error monitoring. The key idea is to perform our invariants analysis concurrently to the execution of the target program and rely on invariants violation to detect memory errors that affect the global application state. Chapter 4 appeared in *Proceedings of the 43rd International Conference on Dependable Systems and Networks (DSN '13)* [109].

- Chapter 5 presents EDFI, a new tool for general-purpose fault injection experiments which improves the dependability of prior tools. In detail, EDFI relies on program instrumentation strategies to perform *execution-driven fault injection*, a technique which allows realistic software faults to be injected in a controlled way at runtime. Our focus on fault injection is motivated by our interest in evaluating the fault-tolerance properties of the proposed live update techniques. In particular, we used EDFI to conduct the first fault injection campaign on live update code (Chapter 6). Chapter 5 appeared in *Proceedings of the Pacific Rim International Symposium on Dependable Computing (PRDC '13)* [112].

- Chapter 6 presents time-traveling state transfer, a technique to strengthen the fault-tolerance properties of process-level updates and significantly reduce the amount of trusted live update code involved. The key idea is to rely on semantics-preserving state transfer transactions across multiple past, future, and reversed versions to detect memory errors introduced in the process using a simple state differencing strategy. We implemented and evaluated time-traveling state transfer on popular server programs on Linux, conducting fault injection experiments to assess the effectiveness of our techniques. Chapter 6 appeared in *Proceedings of the 27th USENIX Systems Administration Conference (LISA '13)* [110].

- Chapter 7 presents Mutable Checkpoint-Restart (MCR), a new technique to support live update in generic legacy C programs with very low engineering effort. This chapter introduces profile-guided quiescence detection, state-driven mutable record-replay, and mutable GC-style tracing. We implemented and evaluated MCR on popular server programs on Linux, assessing the overhead and the engineering effort required to deploy our techniques on long-running C programs found "*in the wild*". Chapter 7 is currently submitted for review [111].

- Chapter 8 concludes the dissertation, summarizing key results, analyzing current limitations, and highlighting opportunities for future research directions.

2

Safe and Automatic Live Update for Operating Systems

Abstract

Increasingly many systems have to run all the time with no downtime allowed. Consider, for example, systems controlling electric power plants and e-banking servers. Nevertheless, security patches and a constant stream of new operating system versions need to be deployed without stopping running programs. These factors naturally lead to a pressing demand for live update—upgrading all or parts of the operating system without rebooting. Unfortunately, existing solutions require significant manual intervention and thus work reliably only for small operating system patches.

In this paper, we describe an automated system for live update that can safely and automatically handle major upgrades without rebooting. We have implemented our ideas in PROTEOS, a new research OS designed with live update in mind. PROTEOS relies on system support and nonintrusive instrumentation to handle even very complex updates with minimal manual effort. The key novelty is the idea of *state quiescence*, which allows updates to happen only in safe and predictable system states. A second novelty is the ability to automatically perform transactional live updates at the *process level*, ensuring a safe and *stable* update process. Unlike prior solutions, PROTEOS supports automated state transfer, state checking, and *hot rollback*. We have evaluated PROTEOS on 50 real updates and on novel live update scenarios. The results show that our techniques can effectively support both simple and complex updates, while outperforming prior solutions in terms of flexibility, security, reliability, and stability of the update process.

2.1 Introduction

Modern operating systems evolve rapidly. Studies on the Linux kernel have shown that its size has more than doubled in the last 10 years, with a growth of more than 6 MLOC and over 300 official versions released [231]. This trend leads to many frequently released updates that implement new features, improve performance, or fix important bugs and security vulnerabilities. With today's pervasive demand for 24/7 operation, however, the traditional patch-install-reboot cycle introduces unacceptable downtime for the operating system (OS) and all the running applications. To mitigate this problem, enterprise users often rely on "*rolling upgrades*" [91]—upgrading one node at a time in a highly replicated software system—which, however, require redundant hardware (or virtualized environments), cannot normally preserve program state across system versions, and may introduce a very large update window with high exposure to mixed version races [93].

Live update (sometimes also called *hot* or *dynamic* update) is a potential solution to this problem, due to its ability to upgrade a running system on the fly with no service interruption. To reach widespread adoption, however, a live update solution should be practical and trustworthy. We found that existing solutions for operating systems [42; 43; 194; 36; 68] and generic C programs [214; 193; 32; 69; 193; 213] meet these requirements only for simple updates. Not surprisingly, many live update solutions explicitly target small security patches [36; 32]. While security patches are a critical application for live update—as also demonstrated by the commercial success of solutions like Ksplice [36]—we believe there is a need for solutions that can effectively handle more complex updates, such as upgrading between operating system versions with hundreds or thousands of changes.

We note two key limiting factors in existing solutions. First, they scale poorly with the size and complexity of the update. This limitation stems from the limited system support to ensure update safety and transfer the run-time state from one system version to another. Existing solutions largely delegate these challenging tasks to the programmer. When applied to updates that install a new OS version with major code and data structure changes, this strategy requires an unbearable effort and is inevitably error prone.

Second, they scale poorly with the number of live updates applied to the system. This limitation stems from the update mechanisms employed in existing solutions, which assume a rigid address space layout, and glue code and data changes directly into the running version. This strategy typically leads to an *unstable* live update process, with memory leakage and performance overhead growing linearly over time (§2.2.3). We found that both limiting factors introduce important reliability and security issues, which inevitably discourage acceptance of live update.

This paper presents PROTEOS, a new research operating system designed to safely and automatically support many classes of live updates. To meet this challenge, PROTEOS introduces several novel techniques. First, it replaces the long-used notion of *function* [36] and *object* [43] *quiescence* with the more general idea of *state quiescence*, allowing programmer-provided *state filters* to specify constraints on the update state. The key intuition is that operating systems quickly transition through many states with different properties. Restricting the update to be installed only in specific states dramatically simplifies reasoning on update safety.

Further, PROTEOS employs transactional *process-level* live updates, which reliably replace entire processes instead of individual objects or functions. To increase the update surface and support complex updates, we explore this idea in a design with all the core OS subsystems running as independent event-driven processes on top of a minimal message-passing substrate running in kernel mode. Using processes as updatable units ensures a *stable* update process and eliminates the need for complex patch analysis and preparation tools. In addition, PROTEOS uses hardware-isolated processes to sandbox the state transfer execution in the new process version and support *hot rollback* in case of run-time errors.

Finally, PROTEOS relies on compiler-generated instrumentation to automate state transfer (migrating the state between processes), state checking (checking the state for consistency), and tainted state management (recovering from a corrupted state), with minimal run-time overhead. Our state transfer framework is designed to *automatically* handle common structural state changes (e.g., adding a new field to a `struct`) and recover from particular tainted states (i.e., memory leakage), while supporting a convenient programming model for extensions. As an example, programmers can register their own callbacks to handle corrupted pointers or override the default transfer strategy for state objects of a particular type.

Our current PROTEOS implementation runs on the x86 platform and supports a complete POSIX interface. Our state management framework supports C and assembly code. Its instrumentation component is implemented as a link-time pass using the LLVM compiler framework [179]. We evaluated PROTEOS on 50 real updates (randomly sampled in the course of over 2 years of development) and novel live update scenarios: *online diversification*, *memory leakage reclaiming*, and *update failures* (§2.6.2). Our results show that: (i) PROTEOS provides an effective and easy-to-use update model for both small and very complex updates. Most live updates required minimal effort to be deployed, compared to the "*tedious engineering effort*" reported in prior work [68]; (ii) PROTEOS is reliable and secure. Our state transfer framework reduces manual effort to the bare minimum and can safely rollback the update when detecting unsafe conditions or run-time errors (e.g., crashes, timeouts, assertion failures). Despite the complexity of some of the 50 updates analyzed, live update required only 265 lines of custom state transfer code in total. (iii) The update techniques used in PROTEOS are *stable* and efficient. The run-time overhead is well isolated in allocator operations and only visible in microbenchmarks (6-130% overhead on allocator operations). The service disruption at update time is minimal (less than 5% macrobenchmark overhead while replacing an OS component every 20s) and the update time modest (3.55s to replace all the OS components). (iv) The update mechanisms used in PROTEOS significantly increase the update surface and enable novel live update scenarios. In our experiments, we were able to update *all* the OS components in a single fault-tolerant transaction and completely *automate* live update of as many as 4,873,735 type transformations throughout the entire operating system (§2.6.2).

2.1.1 Contribution

This paper makes several contributions. First, we identify the key limitations in existing live update solutions and present practical examples of reliability and security problems. Second, we introduce a new update model based on *state quiescence*, which generalizes existing models but allows updates to be deployed only in predictable system states. Third, we introduce transactional process-level updates, which allow safe *hot rollback* in case of update failures, and present their application to operating systems. Fourth, we introduce a new reliable and secure state transfer framework that automates state transfer, state checking, and tainted state management. Finally, we have implemented and evaluated these ideas in PROTEOS, a new research operating system designed with live update in mind. We believe our work raises several important issues on existing techniques and provides effective solutions that can drive future research in the field.

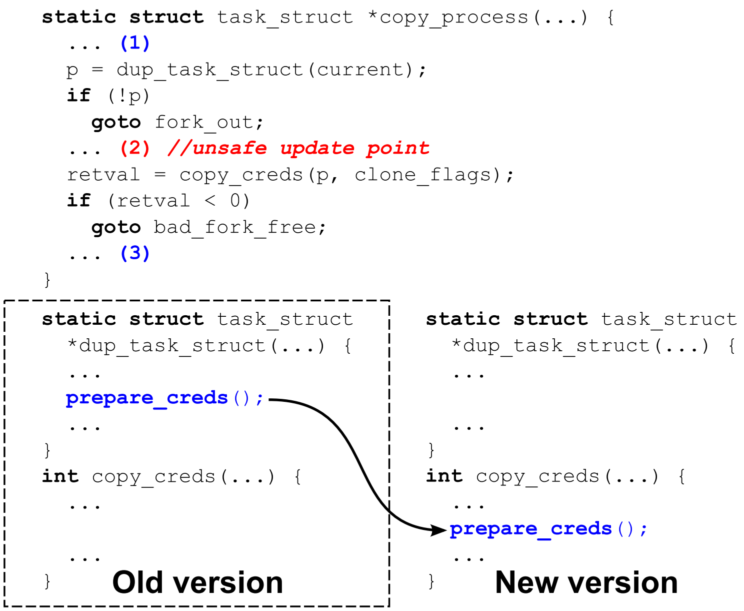
2.2 Background

Gupta has determined that the validity of a live update applied in an arbitrary state S and using a state transfer function T is undecidable in the general case [125]. Hence, system support and manual intervention are needed. Unfortunately, existing solutions offer both limited control over the update state S and poor support to build the state transfer function T.

2.2.1 Safe Update State

Prior work has generally focused on an update-agnostic definition of a safe update state. A number of solutions permit both the old and the new version to coexist [194; 69; 68], many others disallow updates to active code [103; 124; 32; 43; 36]. The first approach yields an highly unpredictable update process, making it hard to give strong safety guarantees. The second approach relies on the general notion of *function* (or *object*) *quiescence*, which only allows updates to functions that are not on the call stack of some active thread.

Figure 2.1 shows that quiescence is a weak requirement for a safe update state. The example proposed (inspired by real code from the Linux `fork` implementation) simply moves the call `prepare_creds()` from the function `dup_task_struct` to the function `copy_creds`. Since `copy_process` is unchanged, function quiescence would allow the update to happen at any of the update points (1, 2, 3). It is easy to show, however, that the update point (2) is unsafe, since it may allow a single invocation of the function `copy_process()` to call (i) the old ver-



```
static struct task_struct *copy_process(...) {
  ... (1)
  p = dup_task_struct(current);
  if (!p)
    goto fork_out;
  ... (2) //unsafe update point
  retval = copy_creds(p, clone_flags);
  if (retval < 0)
    goto bad_fork_free;
  ... (3)
}
```

```
static struct task_struct
  *dup_task_struct(...) {
  ...
  prepare_creds();
  ...
}
int copy_creds(...) {
  ...


  ...
}
```

```
static struct task_struct
  *dup_task_struct(...) {
  ...


  ...
}
int copy_creds(...) {
  ...
  prepare_creds();
  ...
}
```

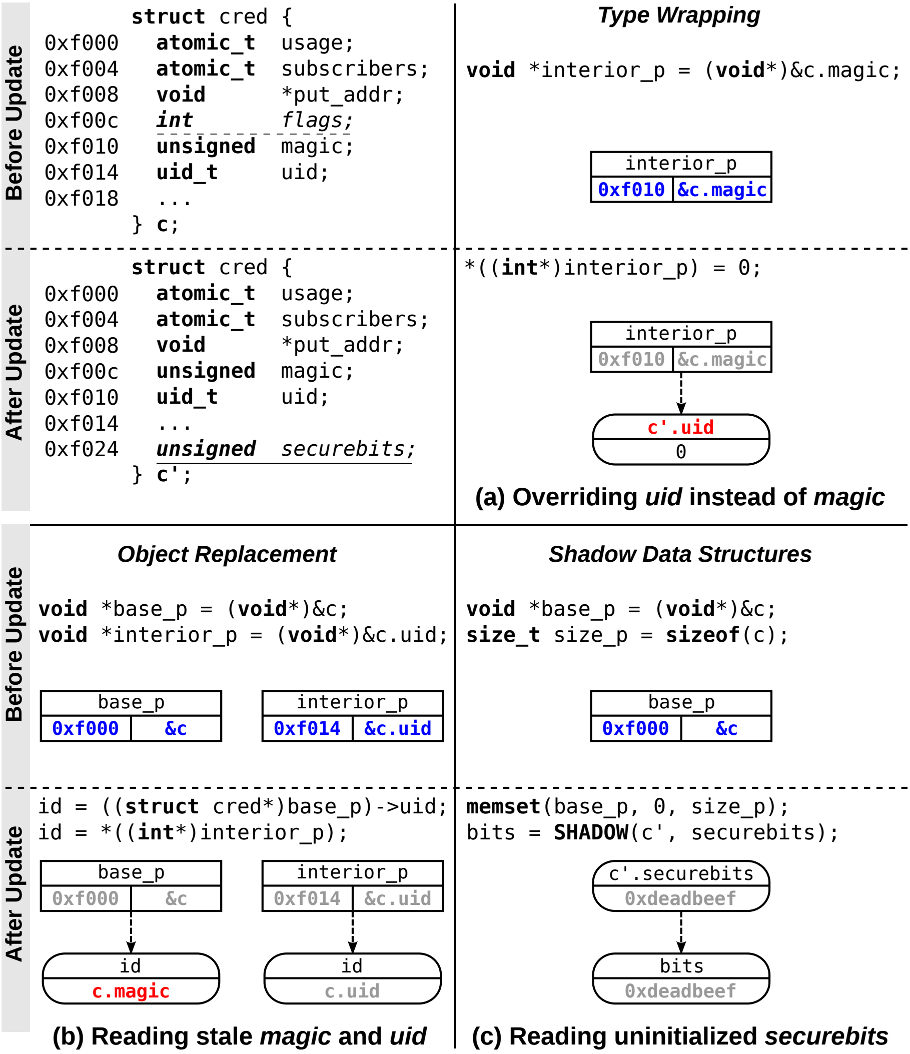
Figure 2.1: An unsafe live update using function quiescence.

sion of the function **`dup_task_struct()`** and (ii) the new version of the function **`copy_creds()`**. Due to the nature of the update, the resulting execution would *incorrectly* call **`prepare_creds()`** twice—and not once, as expected during normal update-free execution.

To address this problem, prior live update solutions have proposed pre-annotated transactions [215], update points [214], or static analysis [213]. These strategies do not easily scale to complex operating system updates and always expose the programmer to the significant effort of manually verifying update correctness in all the possible system states. PROTEOS addresses this problem using our new notion of *state quiescence*, which generalizes prior update safety mechanisms and allows the programmer to dynamically express update constraints on a per-update basis. In the example, the programmer can simply specify a *state filter* (§2.4.3) requesting no **`fork`** to be in progress at update time.

2.2.2 State Transfer

Prior work has generally focused on supporting data type transformations in a rigid address space organization. Three approaches are dominant: *type wrapping* [214; 213], *object replacement* [43; 69; 68; 193], and *shadow data structures* [194; 36]. Type wrapping instruments data objects with extra padding and performs in-place type transformations. Object replacement dynamically loads the new objects into the address space and transfers the state from the old objects to the new ones.



```
        struct cred {
0xf000    atomic_t  usage;
0xf004    atomic_t  subscribers;
0xf008    void      *put_addr;
0xf00c    int       flags;
0xf010    unsigned  magic;
0xf014    uid_t     uid;
0xf018    ...
        } c;
```

```
        struct cred {
0xf000    atomic_t  usage;
0xf004    atomic_t  subscribers;
0xf008    void      *put_addr;
0xf00c    unsigned  magic;
0xf010    uid_t     uid;
0xf014    ...
0xf024    unsigned  securebits;
        } c';
```

```
void *interior_p = (void*)&c.magic;
```

```
*((int*)interior_p) = 0;
```

```
void *base_p = (void*)&c;
void *interior_p = (void*)&c.uid;
```

```
id = ((struct cred*)base_p)->uid;
id = *((int*)interior_p);
```

```
void *base_p = (void*)&c;
size_t size_p = sizeof(c);
```

```
memset(base_p, 0, size_p);
bits = SHADOW(c', securebits);
```

(a) Overriding *uid* instead of *magic*

(b) Reading stale *magic* and *uid*

(c) Reading uninitialized *securebits*

Figure 2.2: Examples of live update security vulnerabilities introduced by unhandled pointers into updated data structures: (a) Type-unsafe memory writes; (b) Misplaced reads of stale object data; (c) Uninitialized reads.

Shadow data structures are similar, but preserve the old objects and only load the new fields of the new objects. While some have automated the generation of type transformers [214; 213], *none* of the existing live update solutions for C provides automated support for transforming pointers and reallocating dynamic objects. Figure 2.2 demonstrates that failure to properly handle pointers into updated objects can introduce several problems, ranging from subtle logical errors to security vulnerabilities. Type wrapping may introduce type-unsafe memory reads/writes for stale interior pointers into updated objects. This is similar to a typical dangling pointer vulnerability [27], which, in the example, causes the pointer `interior_p` to erroneously write into the field `uid` instead of the field `magic`. Object replacement may introduce similar vulnerabilities for stale base pointers to updated objects. In the example, this causes the pointer `base_p` to erroneously read from the field `magic` in the old object instead of the field `uid` in the new one. It may also introduce misplaced reads/writes for stale interior pointers into updated objects. In the example, this causes the pointer `interior_p` to read the field `uid` from the old object instead of the new one. Finally, shadow data structures may introduce missing read/write errors for nonupdated code accessing updated objects as raw data. This may, for example, lead to uninitialized read vulnerabilities, as shown in the example for the field `securebits`.

Prior live update solutions have proposed static analysis to identify all these cases correctly [214]. This strategy, however, requires sophisticated program analysis that scales poorly with the size of the program, limits the use of some legal C idioms (e.g., `void*` pointers), and only provides the ability to *disallow* updates as long as there are some live pointers into updated objects. Thus, extensive *manual* effort is still required to locate and transfer all the pointers correctly in the common case of long-lived pointers into updated objects. In our experience, this effort is unrealistic for nontrivial changes. PROTEOS addresses this problem by migrating the *entire state* from one process version to another, automating pointer transfer and object reallocation with none of the limitations above. This is possible using our run-time state introspection strategy implemented on top of LLVM-based instrumentation (§2.5.2).

2.2.3 Stability of the update process

We say that a live update process is *stable* if version τ of the system with no live update applied behaves no differently than version $\tau - k$ of the same system after k live updates. This property is crucial for realistic long-term deployment of live update. Unfortunately, the update mechanisms used in existing live update solutions for C repeatedly violate the stability assumption. This is primarily due to the rigid address space organization used, with every update loading new code and data directly into the running version. This *in-place* strategy typically introduces memory leakage (due to the difficulties to reclaim dead code and data) and poorer spatial locality (due to address space fragmentation). For example, prior work on server applications reported 40% memory footprint increase and 29% performance overhead after 10 updates [214]. Further, solutions that redirect execution to the new code via binary rewriting [194; 36; 69; 32] introduce a number of trampolines (and thus overhead) that grows linearly with the number and the size of the updates. Finally, shadow data structures change the code representation and force future updates to track all the changes previously applied to the system, complicating version management over time. PROTEOS' *process-level* updates eliminate all these issues and ensure a *stable* live update process (§2.5).

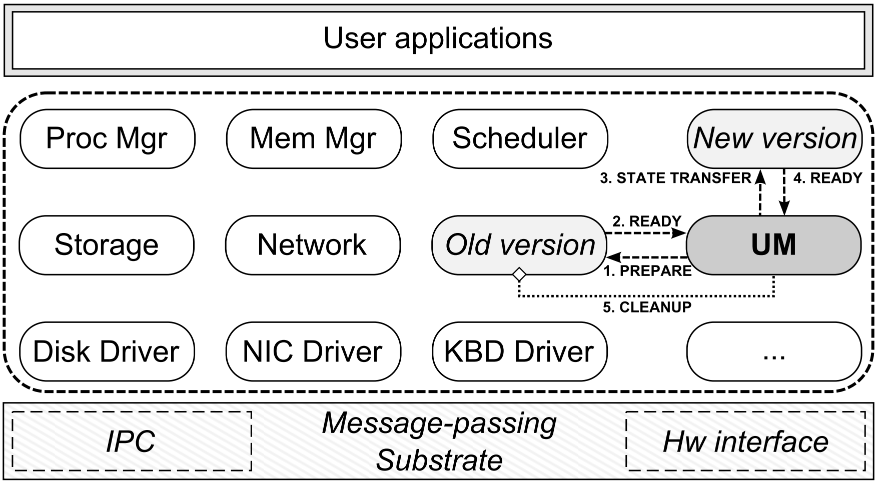


Figure 2.3: The architecture of PROTEOS.

2.3 Overview

Our design adheres to 3 key principles: (i) *security and reliability*: updates are only installed in predictable system states and the update process is safeguarded against errors and unsafe conditions; (ii) *large update surface*: no constraints on the size, complexity, and number of updates applied to the system; (iii) *minimal manual effort*: state filters minimize code inspection effort to ensure safety; automated state transfer minimizes programming effort for the update; process-level updates make deploying live updates as natural as installing a new release, with no need for specialized toolchains or complex patch analysis tools.

2.3.1 Architecture

Figure 2.3 shows the overall architecture of PROTEOS. Our design uses a minimalistic approach with a thin kernel only managing the hardware and providing basic IPC functionalities. All the core operating system subsystems are confined into hardware-isolated processes, including drivers, scheduling, process management, memory management, storage, and network stack. The OS processes communicate through message passing and adhere to a well-defined event-driven model. This design is advantageous for a number of reasons. First, it introduces clear module boundaries and interfaces to simplify updatability and reasoning on update safety. Second, live updates are installed by replacing entire processes, with a new code and data representation that is no different from a freshly installed version of the system. This strategy fulfills the stability requirement and simplifies deployment of live updates. Third, the MMU-based isolation sandboxes the execution of the *entire* state transfer code in the new version, simplifying detection and isolation of run-time errors and allowing for safe *hot rollback* and no memory leakage. Finally, our event-driven update model facilitates state management and allows the system to actively *cooperate* at update time, a strategy which translates to a much more predictable and controllable live update process [105].

The update process is orchestrated by the *update manager* (UM), which provides the interface to deploy live updates for all the OS processes (including itself). When an update is available, the update manager loads the new process instances in memory and requests all the processes involved in the update to converge to the required update state. When done, every process reports back to UM and blocks. At the end of the *preparation phase*, UM *atomically* replaces all the processes with their new counterparts. The new processes perform state transfer and report back to the update manager when done. At the end of the *state transfer phase*, the old processes are cleaned up and the new processes are allowed to resume execution. Synchronization between the update manager and the OS processes is entirely based on message passing. Live updates use atomic transactions: the update manager can safely abort and rollback the update during any stage of the update process, since *no* changes are made to the original process. Figure 2.3 depicts the steps of the update process for single-component live updates (multicomponent updates are discussed in §2.4.5).

2.3.2 Update example

In PROTEOS, building a live update is as simple as recompiling all the updated components using our LLVM compiler plugin. To apply the update, programmers use `prctl`, a simple command-line utility that interfaces with the update manager. For example, the following command instructs the update manager to install a new version of the memory manager in the default live update state (*no event in progress*):

```
prctl update mm /bin/mm.new
```

In our evaluation, we used this update to apply important changes to page fault handling code. An example of a multicomponent update is the following:

```
prctl mupdate net /bin/net.new \
    -state ’num_pending_writes == 0’
prctl mupdate e1000 /bin/e1000.new
prctl mupdate-start
```

In our evaluation, we used this update to change the interface between the network stack and the network drivers. The state filter for the variable `num_pending_writes` is used to ensure that no affected interface interactions are in progress at update time. In our experiments, this change was applied automatically, with no manual effort required. Without the filter, the change would have required several lines of manual and error-prone state transfer code. Since interface changes between versions are common in modern operating systems [229; 230], we consider this an important improvement over the state of the art. While it should be clear that not all the updates can be so smoothly expressed with a simple state filter, this example does show that, when the state is well-captured in the form of global data structures, programmers can much more easily reason on update safety in terms of state quiescence, which frees them from the heroic effort of validating the update in many transient (and potentially unsafe) system states. In our model, identifying a single and well-defined safe update state is sufficient to guarantee a predictable and reliable update process.

2.3.3 Limitations

The OS design adopted in PROTEOS is not applicable as-is to commodity operating systems. Nonetheless, our end-to-end design can be easily applied to: (i) microkernel architectures used in common embedded OSes, such as L4 [178], Green Hills Integrity [8], and QNX [143]; (ii) research OSes using process-like abstractions, such as Singularity [153]; (iii) commodity OS subsystems running in user space, such as filesystems [7] and user-mode drivers in Windows [202] or Linux [60]; (iv) long-running user-space C programs. We make no claim that our OS design is the *only* possible design for a live update system. PROTEOS merely illustrates one way to implement several novel techniques that enable *truly* safe and automatic live updates. For instance, our single-component live update strategy could be also applied to monolithic OS architectures, using shadow kernel techniques [85] to enable state transfer between versions. The reduced modularity, however, would complicate reasoning on update safety for nontrivial updates. Failure to provide proper process-like isolation for the state transfer code, in turn, would lower the dependability of our hot rollback strategy.

We stress that the individual techniques described in the paper (e.g., state quiescence, automated state transfer, and automated state checking) have general applicability, and we expect existing live update solutions for commodity OSes or user-space programs to directly benefit from their integration. To encourage adoption and retrofit existing OSes and widely deployed applications, we explicitly tailored our techniques to the C programming language.

A practical limitation of our approach is the need for annotations to handle ambiguous pointer transfer scenarios (§2.5.3). Our experience, however, shows that the impact of these cases is minimal in practice (§2.6.1). Moreover, we see this as a feature rather than a limitation. Annotations compensate for the effort to manually perform state transfer and readjust all the pointers. Failing to do so leads to the reliability and security problems pointed out earlier.

Finally, a limitation of our current implementation is the inability to live update the message-passing substrate running in kernel mode. Given its small size and relatively stable code base, we felt this was not a feature to particularly prioritize. The techniques presented here, however, are equally applicable to the kernel code

```
static int my_init() {
    ... // initialization code
    return 0;
}
int main() {
    event_eh_t my_ehs = { init : my_init };
    sys_startup(&my_ehs);
    while(1) { // event loop
        msg_t m;
        sys_receive(&m);
        process_msg(&m);
    }
    return 0;
}
```

Listing 2.1: The event-driven programming model.

itself. We expect extending our current implementation to pose no more challenges than enabling live update for the update manager, which PROTEOS already supports in its current form (§2.4.5).

2.4 Live Update Support

This section describes the fundamental mechanisms used to implement safe and automatic live update in PROTEOS.

2.4.1 Programming model

Figure 2.1 exemplifies the event-driven model used in our OS processes. The structure is similar to a long-running server program, but with special *system events* managed by the run-time system—implemented as a library transparently linked against every OS process as part of our instrumentation strategy. At startup, each process registers any custom *event handlers* and gives control to the runtime (i.e., `sys_startup()`).

At boot time, the runtime transparently invokes the *init* handler (`my_init` in the example) to run regular initialization code. In case of live update, in contrast, the runtime invokes the *state transfer* handler, responsible for initializing the new process from the old state. The default *state transfer* handler (also used in the example) *automatically* transfers all the old state to the new process, following a default state transfer strategy (§2.5.4). This is done by applying LLVM-based state instrumentation at compile time and automatically migrating data between processes at runtime.

After startup, each process enters an endless *event loop* to process IPC messages. The call `sys_receive()` dispatches regular messages to the loop, while transparently intercepting the special system events part of the update protocol and handling all the interactions with the update manager. The event loop is designed to be short lived, thanks to the extensive use of asynchronous IPC. This ensures scalability and fast convergence to the update state. Fast *state quiescence* is important to replace many OS processes in a single atomic transaction, eliminating the need for unsafe cross-version execution in complex updates. Note that this property does not equally apply to function *quiescence*, given that many OS subsystems never quiesce [194]. In PROTEOS, *all* the nonquiescent subsystems are isolated in *event loops* with well-defined mappings across versions. This makes it possible to update *any* nonquiescent part of the OS with no restriction. The top of the loop is the only possible *update point*, with an update logically transferring control flow from an invocation of `sys_receive()` in the old process to its counterpart in the new process (and back in case of *hot rollback*).

2.4.2 Virtual IPC endpoints

Two properties make it possible to support transactional process-level updates for the entire OS. First, updates are *transparent* to any nonupdated OS process or user program. Second, updates are *atomic*: only one version at the time is logically visible to the rest of the system. To meet these goals, PROTEOS uses *virtual endpoints* in its IPC implementation. A virtual endpoint is a unique version-agnostic IPC identifier assigned to the *only* active instance of an OS process. At update time, the kernel atomically rebinds all the virtual endpoints to the new instances. The switchover, which occurs at the end of the preparation phase, transparently redirects all the IPC invocations to the new version.

2.4.3 State filters

Unlike prior solutions, PROTEOS relies on *state quiescence* to detect a safe update state. This property allows updates to be installed only when particular constraints are met by the global state of the system. *State filters* make it possible to specify these constraints on a per-update basis. A state filter is a generic boolean expression written in a C-like language and evaluated at runtime. Our state filter evaluator supports the arithmetic, comparison, and logical operators allowed by C. It can also handle pointers to dynamically allocated objects, compute the value of any global/static variable (and subelements), and invoke read-only functions with a predetermined naming scheme. State filters reflect our belief that specifying a safe update state should be as easy as writing an assertion to check the state for consistency. Our evaluator is implemented as a simple extension to our state management framework (§2.5), which already provides the ability to perform run-time state introspection.

At the beginning of the preparation phase, every to-be-updated OS process receives a string containing a state filter, which is installed and transparently evaluated at the end of every following event loop iteration. When the process transitions to the required state, the expression evaluates to `true`, causing the process to report back to the update manager and block at the top of the event loop. The default state filter forces the process to block immediately. To support complex state filters that cannot be easily specified in a simple expression, PROTEOS can automatically compile generic state filter functions (written in C) into binary form. This is simply accomplished by generating *intermediate* process versions that only differ from the old ones by a new filter function `sf_custom`. Since the change is semantics-preserving, the intermediate versions can be automatically installed in the default update state before the actual update process takes place. State filter functions give the programmer the flexibility to express complex state constraints using any valid C code. On the other hand, regular state filters, are a simpler and smoother solution for online development and fast prototyping. They are also safer, since the state expression is checked for correctness by our state transfer framework.

2.4.4 Interface filters

Our short-lived event loop design is not alone sufficient to guarantee convergence to the update state in the general case, especially when the system is under heavy load. To give stronger convergence guarantees in particular scenarios, PROTEOS supports (optional) *interface filters* for every to-be-updated OS process. Each filter is transparently installed into the kernel at the beginning of the preparation phase. Its goal is to monitor the incoming IPC traffic and temporarily block delivery of messages that would otherwise delay state quiescence. Programmers can specify *filtering rules* similar to those used in packet filters [130], to selectively blacklist or whitelist delivery of particular IPC messages by source or type.

2.4.5 Multicomponent updates

Changes that affect IPC interactions require the system to atomically update multiple processes in a single update transaction. To support multicomponent updates, the update manager orderly runs the preparation protocol with every to-be-updated OS process. The overall preparation phase is strictly *sequential*, namely the process i in the update transaction is only requested to start the preparation phase after the process $i - 1$ has already reached state quiescence and blocked. The state transfer phase is, in contrast, completely *parallel*. Parallelism is allowed to avoid placing any restrictions on state transfer extensions that require updated processes to initialize some mutual state. Our sequential preparation strategy, in turn, ensures a predictable live update process and gives the programmer full control over the update transaction, while preserving the ability to safely and automatically rollback the update in case of programming errors (i.e., deadlocks or other synchronization issues). Our design introduces a new structured definition of the live update problem: a live update is feasible if it is possible to identify a sequence of state and interface filters able to drive the system into a state with a valid mapping—and state transfer function—in the new version. Our experience shows that this approach is effective and scales to complex updates. For instance, following a top-down update strategy, we were successfully able to implement a *fault-tolerant* update transaction that atomically replaces *all* the OS processes, *including the update manager itself*.

To update the update manager, PROTEOS uses two simple ideas. First, the update manager is constrained to be the last process in the update transaction to obey the semantics of the update process. At the end of the preparation phase, kernel support allows the update manager to block and atomically yield control to its new process version. Second, the new version completes the update process as part of its own state transfer phase. Once the automated state transfer process completes (§2.5.1), the new manager updates its state to account for its own update and normally waits for the other OS processes to synchronize. This simple strategy added less than 200 lines of code to our original update manager implementation.

2.4.6 Hot rollback

In case of unexpected errors, *hot rollback* enables the update manager to abort the update process and safely allow the old version to resume execution. Our manager can detect and automatically recover from the following errors: (i) timeouts in the preparation phase (e.g., due to broken dependencies in the update transaction or poorly designed state/interface filters which lead to deadlocks or other synchronization errors); (ii) timeouts in the state transfer phase (e.g., due to synchronization errors or infinite loops); (iii) fatal errors in the state transfer phase (e.g., due to crashes, panics, or error conditions automatically detected by our state checking framework). The MMU-based protection prevents any run-time errors from propagating back to the old version. Fatal errors are ultimately intercepted by the kernel, which simply notifies the update manager—or its old instance, which is automatically revived by the kernel when the update manager itself is updating—to perform rollback. To atomically rollback the update during the state transfer phase, the update manager simply requests the kernel to freeze all the new instances, rebind all the virtual endpoints to the old instances, and unblock them. The new instances are cleaned up next in *cooperation* with the old version of the system.

2.5 State Management

To automate process-level updates, PROTEOS needs to automatically migrate the state between the old and the new process. Our migration strategy makes no assumptions about compiler optimizations or number of code or data structures changed between versions. In other words, the two processes are allowed to have *arbitrarily different* memory layouts. This allows us to support arbitrarily complex state changes with no impact on the stability of the update process. To address this challenge, PROTEOS implements *precise* run-time state introspection, which makes it possible to automate pointer transfer and dynamic object reallocation even in face of type changes. Our goal is to require help from the programmers only in the *undecidable* cases, for example, ambiguous pointer scenarios (§2.5.3), semantic changes that cannot be automatically settled by our state mapping and migration strategy

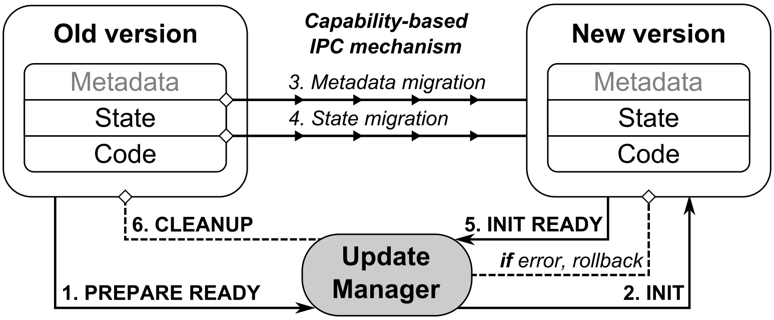


Figure 2.4: The state transfer process.

(e.g., an update renumbering the error codes stored in global variables), and changes that also require updating external state (e.g., an update modifying the representation of some on-disk data structures).

2.5.1 State transfer

To support run-time state introspection, every OS process is instrumented using an LLVM link-time pass, which embeds *state metadata* in a predefined section of the final ELF binary. The metadata contains the relocation and type information required to introspect all the state objects in the process at runtime. The metadata structures use a fixed layout and are located in a randomized location only known to the process and the kernel.

Figure 2.4 depicts the state transfer process. The migration phase starts with the state transfer framework transferring all the metadata from the old version to the new version (local address space). This is done using a capability-based design, with the kernel granting (only) the new process read-only access to the address space of the old process. At the end of the metadata migration phase, both the old and the new metadata are available locally. This allows the framework to introspect both the old and the new state and remap all the state objects across versions. The mapping relies on a version-agnostic *naming scheme* established at compile time. This enables the framework to unambiguously pair functions, variables, strings, and dynamic objects across versions.

At the end of the pairing phase, all the paired objects are scheduled for transfer by default. Programmers can register extensions to change the pairing rules (e.g., in case of variable renaming) or instruct the framework to avoid transferring particular objects (§2.5.4).

In the data migration phase, the framework traverses all the old state objects (and their inner pointers) scheduled for transfer and ordinately migrates the data to their counterparts in the new version. Our traversal strategy is similar, in spirit, to a precise garbage collector that relocates objects [243]. There are, however, important differences to point out. First, all the dynamic (and static) objects are reallocated (loaded) in the new process. Second, our event loop design allows no state objects on the stack at update time. This eliminates the need to create dynamic metadata for all the local variables, which would degrade performance. Note that, to encourage adoption of our state transfer framework in other update and execution contexts (e.g., multithreaded server applications), however, our instrumentation can already support dynamic metadata generation for local variables (disabled in PROTEOS), using stack instrumentation strategies similar to those adopted by garbage collectors [243]. Finally, objects are possibly reallocated (or loaded) with a different run-time type. Unlike prior solutions, our framework applies type transformations (for both objects and pointers) on-the-fly, analyzing the type differences between paired objects at runtime. This eliminates the need for complex patch analysis tools and exposes a powerful programming model to state transfer extensions. We clarify this claim with an example. To deploy a live update that added a new field in the middle of the `struct buf_desc` (a core data structure of the buffer cache) in our evaluation, we only had to write a simple type-based state transfer callback (§2.5.4) that reinitialized the new field in every object in the new version. The latter is a programmer-provided function automatically invoked by the framework on every transferred object of the requested type (e.g., `struct buf_desc`). This allows the programmer to focus on the data transformation logic while the framework automatically performs dynamic object reallocation and updates all the live pointers into the new objects. Since this `struct` was used in complex data structures like hash tables and linked lists (chained together by several base and inner pointers), this is a significant improvement over existing techniques, which would have required extensive and error-prone manual effort to implement state transfer.

2.5.2 Metadata instrumentation

Our LLVM transformation pass operates at the LLVM IR level and generates metadata for global/static variables (and constants), functions, and strings. Although functions and strings need not be normally transferred to the new version, their metadata is necessary to transfer pointers correctly. For each object, the pass records information on the address, the name, and the type. To create unique and version-coherent identifiers to pair static state objects across versions, our pass uses both *naming* (e.g., global variable name) and *contextual* (e.g., module name for static functions/variables) information derived from debug symbols. Note that this strategy does not prevent debug symbols from being completely stripped with no restriction from the final binary. To create metadata for dynamically allocated objects, in turn, the pass analyzes and instruments each allocation site found in the original code. Our static analysis can automatically identify `malloc/free` and `mmap/munmap` allocator abstractions, which PROTEOS natively supports for every OS process. For each allocation site, the pass records the name (derived from the caller and the allocation variable), the allocator name, and the static type. The name and the allocator name are used to pair (and reallocate) allocation sites across versions. The static type is used to dynamically determine the run-time type of every allocated object. For example, an allocation of the form `ptr = malloc(sizeof(msg_t)*4)` will be associated a static type `msg_t` and a run-time type `[4 x msg_t]`. The pass replaces every allocation/deallocation call with a call to a wrapper function responsible to dynamically create/destroy metadata for every dynamic object. To minimize the performance impact, the wrappers normally use in-band descriptors to store the metadata for the dynamic state objects. The allocators, however, support special flags to let the programmer control the allocation behavior (e.g., use out-of-band metadata for special I/O regions, or remap DMA buffers at state transfer time instead of explicitly reallocating them).

2.5.3 Pointer transfer

Pointers pose a fundamental challenge to automating state transfer for C programs. To transfer base and interior pointers correctly, our framework implements dynamic *points-to* analysis on top of the precise type information provided by our instrumentation. Our analysis is cast-insensitive and does not forbid or limit the use of any legal C programming idiom (e.g., `void*`), a problem in prior work [214; 213]. Our pointer transfer strategy follows 5 steps (an example is presented in Figure 2.5): (i) locate the target object (and the inner element, for interior pointers); (ii) locate the target object counterpart in the new version according to the output of the pairing phase; (iii) remap the inner element counterpart in case of type changes; (iv) reinitialize the pointer according to the target object (and element) counterpart identified; (v) schedule the target object for transfer. The last step is necessary to preserve the shape of arbitrarily complex data structures. In addition, the traversal allows our framework to structurally prevent any memory leakages (i.e., unreachable dynamic objects) in the old version from propagating to the new version. Note that our pointer traversal strategy relies only on the run-time type of the target object (and element), with no assumptions on the original pointer type. This strategy can seamlessly support generic `void*` pointers and eliminates the need to explicitly deal with pointer casting. Our framework can also automatically handle pointers with special integer values (e.g., `NULL` or `MAP_FAILED` (-1)) and guard pointers that mark buffer boundaries. Uninitialized pointers are structurally prevented in the allocators and dangling pointers disallowed by design. While our pointer analysis can handle all these common scenarios automatically, we have identified practical cases of *pointer ambiguity* that always require (typically one-time) user intervention, pointers stored as integers and `unions` with inner pointers, in particular. Manually handling these cases via annotations or callbacks (§2.5.4) is necessary to ensure precise pointer analysis. More details on our points-to analysis and our pointer transfer strategy are published elsewhere [104].

```
struct s { //old version
    int flags;
    char string[3];
    short id;
    union IXFER(my_u) u;
    void *userdata;
    PXFER(int) addr;
} my_s;
short *p = &my_s.id;
```

```
struct s { //new version
    int flags;
    int id;
    char string[2];
    union IXFER(my_u) u;
    PXFER(int) addr;
    int newfield;
} my_s;
int *p = &my_s.id;
```

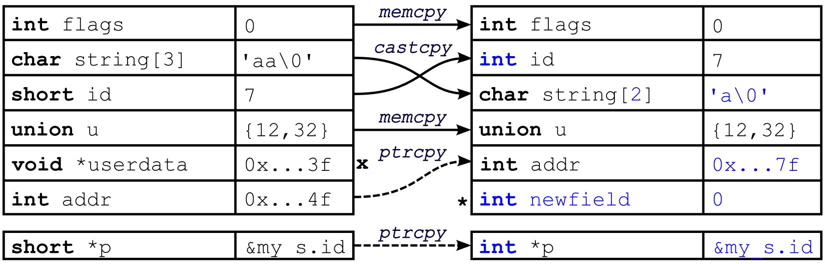


Figure 2.5: Automating type and pointer transformations.

2.5.4 Transfer strategy

Our framework follows a well-defined default state transfer strategy, while allowing programmer-provided extensions to arbitrarily override the default behavior.

Figure 2.5 shows an example of the transfer strategy followed by our framework for a simple update. All the objects and the pointers are automatically transferred (and reallocated on demand) to the new version in spite of type changes. Our default transfer strategy automates state transfer for many common structural changes, such as: (i) primitive type transformations, (ii) array truncation/expansion, and (iii) addition/deletion/reordering of `struct` fields. Extensions can be used to handle more complex state changes (and cases of pointer ambiguity) with minimal effort. The latter are supported in the form of *type-* or *object-based* annotations or callbacks, evaluated every time the framework traverses or remaps the intended type (or object). Annotations are implemented at the preprocessor level with no changes in the compiler. Figure 2.5 shows an example, with the `IXFER` and `PXFER` type-based annotations forcing the framework to `memcpy` the `union u` (without introspecting it) and perform pointer transfer of the integer `addr`.

Programmer-provided state transfer callbacks, in turn, provide a more generic extension mechanism to override the default state transfer behavior during any stage of the state transfer process and at several possible levels of abstraction. For instance, programmers can register object-level and element-level callbacks—evaluated when the framework performs a particular action on a given object or an element part of an object, respectively. To specify the trigger entity in the most flexible way, callbacks can be registered by object/element storage (e.g., data, heap), object/element name (e.g., `my_var_namespace_*`), and object/element type (e.g., `struct` `my_struct_s`), or using any combination thereof. To support many possible trigger events, programmers can register object/element pairing callbacks (to override the default name-based pairing strategy adopted by the framework), object/element transfer callbacks (to override the default transfer strategy or selectively schedule individual objects for transfer), and pointer transfer callbacks (to override the default pointer transfer strategy). Note that the callbacks are automatically and logically chained together by the framework. For example, a user-defined element pairing callback that remaps a `struct` field in a nonstandard way in the new version is automatically invoked by the framework when either transferring the original field to the new version or remapping an inner pointer to the field into an updated object. The callbacks all run in the context of the new process version after completing the metadata migration phase, allowing the programmer to seamlessly access objects in the old and the new version (and their metadata information) with no restriction. The callbacks are written directly in C, providing the ability to operate arbitrary transformations in the state transfer code—even changing external state on the disk, for example. In addition, this allows the programmer to remap complex data structures that significantly change their representation across versions (e.g., a hash table transformed into multiple balanced BSTs) and cannot be automatically paired (nor transferred) by our framework. Even in such complex state transformation scenarios, our programming model can provide a generic callback-driven interface to locate and traverse all the objects (and pointers) to transfer, allowing the programmer to select the best level of abstraction to operate and concentrate on data transformations rather than on manual and error-prone state introspection.

2.5.5 State checking

Our state management framework supports automated state checking using generic *state invariants*. The idea is to detect an invalid state when conservatively defined invariants are violated. *Target-based* invariants are naturally enforced by our *points-to* analysis (i.e., a pointer not pointing to any valid object is invalid). Other invariants are determined by static analysis. We support *value-based* invariants (derived from value set analysis of integer variables) and *type-based* invariants, which verify that a pointer points to a target of a valid type at runtime. This is done by recording metadata on all the valid implicit and explicit pointer casts (i.e., `bitcast` and `inttoptr` LLVM instructions). State checking is performed on the old version before the transfer and on the new version after the transfer. In both cases, the transfer is atomically aborted when invariants violations are found (unless extensions change the default behavior). Checking both the old and the new state allows the framework to detect: (i) a tainted state in the old version (i.e., arbitrary memory corruption) and possibly let extensions recover from it; (ii) corruption in the new state introduced by the state transfer code itself; (iii) violating assumptions in the state transfer process. An example in the latter category is the attempt to transfer a pointer to an old object that no longer exists (or no longer has its address taken) in the new version.

2.6 Evaluation

We have implemented PROTEOS on the x86 platform. The current PROTEOS implementation is a major rewrite of the original MINIX 3 microkernel-based operating system, which only provided process-based isolation for all the core OS components and restartability support for stateless device drivers [141]. Our current prototype includes 22 OS processes (8 drivers and 14 servers) and supports a complete POSIX interface. The static instrumentation is implemented as an LLVM pass in 6550 LOC [1]. The state management framework is implemented as a static library written in C in 8840 LOC. We evaluated PROTEOS on a workstation equipped with a 12-core 1.3Ghz AMD Opteron processor and 4GB of RAM. For evaluation purposes, we ported the C programs in the SPEC CPU 2006 benchmark suite to PROTEOS. We also put together an *sdtools* macrobenchmark, which emulates a typical syscall-intensive workload with common development operations (compilation, text processing, copying) performed on the entire OS source tree. We repeated all our experiments 21 times and reported the median. Our evaluation focuses on 4 key aspects: (i) *Experience*: Can PROTEOS support both simple and complex updates with minimal effort? (ii) *Performance*: Do our techniques yield low run-time overhead and realistic update times? (iii) *Service disruption*: Do live updates introduce low service disruption? (iv) *Memory footprint*: How much memory do our techniques use?

2.6.1 Experience

To evaluate the effort in deploying live updates, we randomly sampled 50 real updates produced by the team of core MINIX 3 developers in the course of over 2 years. The live update infrastructure, in turn, was developed independently to ensure a fair and realistic update evaluation. We carefully analyzed each update considered, prepared it for live update, and finally deployed it online during the execution of our SPEC and *sdtools* benchmarks. We successfully deployed all the live updates considered and checked that the system was fully functional before and aftereach experiment. In 4 cases, our first update attempt failed due to bugs in the state transfer code. The resulting (pointer) errors, however, were immediately detected by our state transfer framework and the update safely rolled back with no consequences for the system. We also verified that the update process was stable (no performance/space overhead increase over time) and that our live update infrastructure could withstand arbitrary compiler optimization changes between versions (e.g., from `-O1` to `-O3`). Table 2.1 presents our findings.

The first three grouped columns provide an overview of all the updates analyzed, with the number of updates considered per category. The *New features* category has the highest number of updates, given that MINIX 3 is under active development. Of the 50 updates considered, 16 involved multiple OS processes. This confirmed the importance of supporting multicomponent live updates. The second group of

[1] Source lines of code reported by David Wheeler's SLOCCount.

			Update LOC			Changes			Manual effort			Time (ms)
Category	**#**	**Multi**	**Total**	**Median**	**90thP**	**Fun**	**Var**	**Ty**	**Ann**	**SF**	**ST LOC**	**Med Upd**
Bug fixes	15	4	1593	18	1231	27	2	2	-	2	55	397
Maintenance	12	5	2206	62	872	16	7	8	-	1	16	230
New features	19	6	10122	195	2435	199	45	101	-	1	63	202
Performance	4	1	652	179	291	10	2	7	-	0	131	358
Total	50	16	14573	63	709	252	56	118	14	4	265	272

Table 2.1: Overview of all the updates analyzed in our evaluation.

columns shows the number of lines of code (total, median, 90th percentile) changed across all the updates, with a total of nearly 15,000 LOC. The third group shows the number of functions, variables, and types changed (i.e., added/deleted/modified). For example, *New features* updates introduced 199 function changes, 45 variable changes, and 101 type changes. The fourth group, in turn, shows the manual effort required in terms of annotations, state filters, and lines of code for state transfer extensions. We only had to annotate 14 declarations (using 3 object-based annotations, 10 type-based annotations, and 1 type-based callback) throughout the entire PROTEOS source code. Encouragingly, this was a modest *one-time* effort that required less than 1 man week. The annotations were only necessary for `unions` with inner pointers and special nontransferrable state objects (e.g., allocator variables). Custom state filters, in turn, were only required for 4 updates. We found that, for most updates, our event loop design gave sufficient predictability guarantees in the default update state. In the remaining cases, however, we faced complex interface changes that would have required extensive manual effort without support for custom state filters. From empirical evidence, we also believe that more than half of the updates would have been extremely hard to reason about using only function quiescence. Despite the many variable and type changes, all the updates required only 265 LOC of state transfer extensions. We found that our state transfer framework was able to fully automate most data structure changes (i.e., addition/removal). In addition, our type-based state transfer callbacks minimized the effort to handle cross-cutting type changes. Finally, the last column reports the median update time, with a value of 272ms across all the updates. We also measured a maximum update time of 3550ms for cross-cutting updates that replaced *all* the OS processes (individually or in a single multicomponent and fault-tolerant transaction).

We now compare our results with prior solutions. Before ours, Ksplice was the only OS-level live update solution evaluated with a comprehensive list of updates over a time interval [36]. Compared to ours, however, their evaluation is based on security patches of much smaller size. Their median patch size is less than 5 LOC, and the 90th percentile less than 30 LOC. As Table 2.1 demonstrates, PROTEOS was evaluated with much more complex updates, while only requiring 265 LOC for state transfer extensions (compared to 132 LOC for Ksplice's 64 small patches [9]). Many other live update solutions for C present only case studies [42; 68; 194; 43] or lack a proper quantitative analysis of the manual effort required [32; 69; 193]. Many researchers, however, have reported "*tedious implementation of the transfer code*" [42], "*tedious engineering efforts*" [68], "*tedious work*" [69], and "*an arduous testing process that spanned several weeks of concentrated work*" [32]. In contrast, we found our techniques to reduce the live update effort to the bare minimum. In particular, our entire update evaluation required only 10 man days. Ginseng [214] and Stump [213] are the only prior solutions for C that provide quantitative measurements for the manual effort. Ginseng (*Stump*) required 140 (*186*) source changes and 336 (*173*) state transfer LOC to apply 30 (*13*) server application updates introducing 21919 (*5817*) new LOC in total. While it is difficult to directly compare their

	PROTEOS	Linux
`malloc`	2.30	1.41
`free`	1.19	1.09
`mmap`	1.41	1.77
`munmap`	1.06	1.42

Table 2.2: Execution time of instrumented allocator operations normalized against the baseline.

results on server applications with ours, we believe that our techniques applied to the same set of updates would have significantly reduced the effort, avoiding manual inspection or code restructuring to eliminate unsupported C idioms, posing no restriction on the nature of the data structure changes, and assisting the programmer in challenging tasks like heap traversal, pointer transfer, and state checking.

2.6.2 Performance

We evaluated the run-time overhead imposed by the update mechanisms used in PROTEOS. Virtual endpoints introduce only update-time costs and no extra run-time overhead on IPC. Transparent interception of special system events introduces only 3 additional cycles per event loop iteration. An important impact comes also from the microkernel-based design itself. Much prior work has been dedicated to improving the performance of IPC [186] and microkernel-based systems in general [185; 149]. Our focus here is on the update techniques rather than on microkernel performance. For instance, our current measurements show that the `gettimeofday`, `open`, `read`, `write`, `close` system calls are 1.05-8.27x slower than on Linux due to our microkernel design. These numbers are, however, pessimistic, given that we have not yet operated many optimizations described in the literature [186; 185; 149].

Much more critical is to assess the cost of our state instrumentation, which directly affects the applicability of our techniques to other OS architectures or user-space applications. To this end, we first ran our SPEC and *sdtools* macrobenchmarks to compare the base PROTEOS implementation with its instrumented version. Our repeated experiments reported no noticeable performance degradation. This is expected since static metadata, used only at update time, is isolated in a separate ELF section with no impact on spatial locality. The use of in-band descriptors to generate dynamic metadata, in turn, minimizes the run-time overhead on allocator operations. To isolate this overhead, we measured the cost of our instrumentation on 10,000 `malloc/free` and `mmap/munmap` repeated allocator operations. We ran the experiments for multiple allocation sizes (0-16MB) and reported the median overhead for `malloc/free` (the overhead does not generally depend on the allocation size) and the maximum overhead for `mmap/munmap` (the overhead generally decreases with the allocation size). For comparison, we also ported our instrumentation to Linux user-space programs and ran the same microbenchmarks on Ubuntu 10.04 LTS 32-bit (`libc` allocators). Table 2.2 depicts our results, with a different (but comparable)

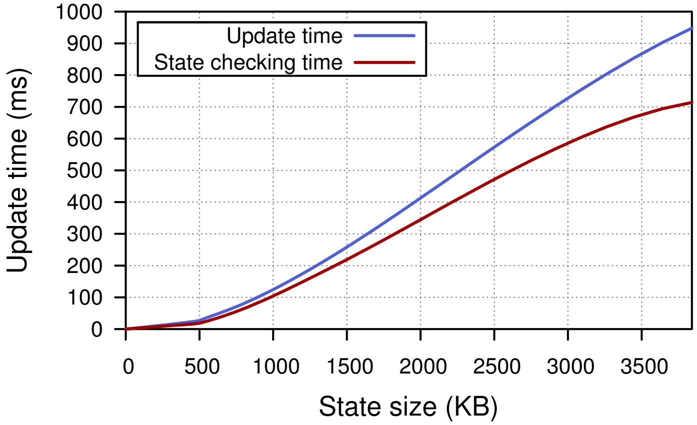
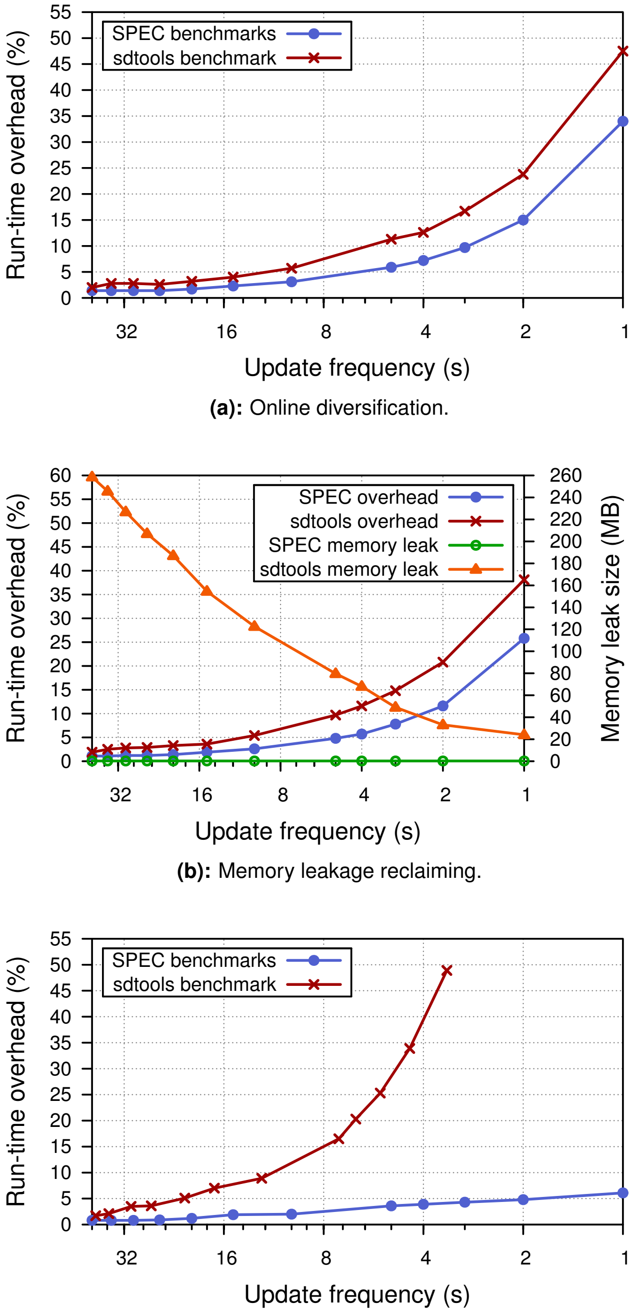


Figure 2.6: Update time vs. run-time state size.

impact of our state instrumentation in the two allocator implementations. The highest overheads in PROTEOS and Linux are incurred by `malloc` (130%) and `mmap` (77%), respectively. Note that these results are overly pessimistic, since common allocation patterns typically yield poorer spatial locality, which will likely mask the overhead on allocator operations further.

We now compare our results with prior techniques. Live update solutions based on (more intrusive) instrumentation strategies have reported macrobenchmark results with worst-case overheads of 6% [214], 6.71%[213], and 96.4% [193]. Solutions based on binary rewriting, in turn, have reported microbenchmark results with 1%-8% invocation overhead [194] for updated functions. Unlike all the prior techniques, our overhead is well-isolated in allocator operations and never increases with the number and the size of the updates applied to the system (stability assumption).

To assess the impact of live updates on the system, we analyzed the distribution of the update time in more detail. Figure 2.6 depicts the update time (the time from the moment the update is signaled to the moment the new version resumes execution) as a function of the run-time state size (total size of all the static and dynamic state objects). These (interpolated) results reflect average measurements obtained during the execution of our macrobenchmarks for all the single-component updates used in our evaluation. The figure shows that the update time grows approximately linearly with the state size. This behavior stems from the fact that the update time is heavily dominated by state transfer and state checking (isolated in the figure). The time to load the new processes in memory and complete the preparation phase is normally marginal. We experimented with many state filters to quiesce all the common OS process interactions and found that the time to reach state quiescence was only a few milliseconds in the worst case. This property makes any overhead associated to evaluating state and interface filters in the preparation phase *marginal*. While our overall update times are generally higher than prior solutions for simple updates (since we



(a): Online diversification.

(b): Memory leakage reclaiming.

(c): Update failures.

Figure 2.7: Run-time overhead vs. update frequency for our benchmarks.

replace entire processes instead of individual functions), the resulting impact is still orders of magnitude shorter than any reboot and bearable for most systems.

2.6.3 Service disruption

To substantiate our last claim, we evaluated the service disruption caused by live update. Figure 2.7 shows the run-time overhead incurred by our macrobenchmarks when periodically updating OS processes in a round-robin fashion. The overhead increases for shorter update intervals, with more disruption incurred by *sdtools*. The figures present three novel live update scenarios. Figure 2.7a presents results for an *online diversification* scenario—an idea we have also developed further in [108]. We implemented a source-to-source transformation able to *safely* and automatically change 3872 type definitions (adding/reordering `struct` elements and expanding arrays/primitive types) throughout the entire operating system. The changes were randomized for each generated OS version, introducing a heavily diversified memory layout across updates. This scenario stressed the capabilities of our state transfer framework, introducing an average of 4,873,735 type transformations at each update cycle and approximating an upper bound for the service disruption. Figure 2.7b presents a *memory leakage reclaiming* scenario. Updates were performed between identical OS versions (approximating a lower bound for the service disruption), but we deliberately introduced a memory leak bug (similar to one found during development) that caused the virtual filesystem not to free the allocated memory at `exec()` time. Shorter update intervals increase the overhead but allow our state transfer framework to automatically repair leaks more quickly. The tradeoff is evident for *sdtools*, which `exec()`ed several programs during the experiment. Figure 2.7c presents an *update failures* scenario. We deliberately simulated state transfer crashes or 2-second timeouts (in equal measure) for each update, resulting in more severe service disruption for the syscall-intensive benchmark *sdtools*, but with no system-perceived impact. This scenario stressed the *unique* fault-tolerant capabilities of our live update infrastructure, able to withstand significant update failures and automatically rollback the entire update transaction with no consequences for the operating system and all the running programs. Overall, our experiments reported a negligible overhead for update intervals larger than 20s. Given that updates are relatively rare events, we expect the update-induced service disruption to be minimal in practice.

2.6.4 Memory footprint

Our metadata instrumentation naturally leads to a larger memory footprint at runtime. Our current implementation required an average of 65 bytes for each type, 27 extra bytes for each variable/constant/string, 38 extra bytes for each function with address taken, 10 extra bytes for each allocation, and 38 bytes for each allocation site. During the execution of our macrobenchmarks, we measured an average state overhead (i.e., metadata size vs. run-time state size) of 18% and an overall memory overhead of 35% across all the OS processes. While comparable to prior instrumentation-based techniques [214; 213; 194], our memory overhead never increases with the number and the size of the updates applied to the system. This is ensured by our stable live update strategy. For comparison, we also ported the Linux ACPI driver to PROTEOS. Despite the very complex code base, adding updatability only required 2 type-based callbacks for 2 `unions`. In this case, the state overhead and the overall memory overhead measured were 41% and 37%, respectively.

2.7 Related work

Several live update solutions are described in the literature, with techniques targeting operating systems [42; 43; 194; 68; 36], C programs [214; 213; 69; 32; 193], object-oriented programs [147; 269], programming languages [267; 90; 34], database systems [59], and distributed systems [53; 54; 25; 26; 30; 279; 173; 92; 91]. We focus here on live update for operating systems and generic C programs, but we refer the interested reader to [142; 24; 123; 257] for more complete surveys.

K42 [265; 42; 43] is a research OS that supports live update functionalities using object-oriented design patterns. To update live objects, K42 relies on system-enforced quiescence, transparently blocking all the threads calling into updated objects. Unfortunately, this strategy leads to a poorly predictable update process, with hidden thread dependencies potentially leading to *unrecoverable* deadlocks. In contrast, PROTEOS gives programmers full control over the update process and can automatically recover from synchronization errors (e.g., deadlocks) introduced by poorly designed update transactions using a predefined timeout. In addition, K42 provides *no* support for automated state transfer. Unlike existing live update techniques for C, however, their object-oriented approach offers a solution to the stability problem. The downside is that their techniques are limited to object-oriented programs. In contrast, the techniques we propose have more general applicability. For instance, state filters and our state management framework can be used to improve existing live update solutions for the Linux kernel [194; 68; 36] and generic C programs [214; 213; 69; 32; 193]. Our framework could be, for example, integrated in existing solutions to automatically track pointers to updated data structures or abort the update in case of unsafe behavior. Our process-level updates, in turn, are an elegant solution to the stability problem for user-level live update solutions. Also note that, while explicitly conceived to simplify state management—no need for explicit update points and stack instrumentation—and minimize manual effort—simpler to reason on update safety and control multicomponent live update transactions—our event-loop design is not strictly required for the applicability of our techniques. For instance, our event-driven model can be easily extended to multithreaded execution using annotated per-thread update points, as previously suggested in [213]. When backward compatibility is not a primary concern, however, we believe our event-driven strategy to offer a superior design for safe and automatic live update. For this reason, we opted for a pure event-driven model for our current PROTEOS implementation.

DynaMOS [194] and LUCOS [68] are two live update solutions for the Linux kernel. They both apply code updates using binary rewriting techniques. To handle data updates, DynaMOS relies on shadow data structures, while LUCOS relies on virtualization to synchronize old and new data structure versions at each write access. Both solutions advocate running the old and the new version in parallel. Unlike ours, their cross-version execution strategy leads to a highly unpredictable update process. In addition, state transfer is delegated entirely to the programmer.

Ksplice [36] is an important step forward over its predecessors. Similar to DynaMOS [194], Ksplice uses binary rewriting and shadow data structures to perform live updates. Unlike all the other live update solutions for C, however, Ksplice prepares live updates at the object code layer. This strategy simplifies patch analysis and does not inhibit any compiler optimizations or language features. Process-level updates used in PROTEOS take these important guarantees one step further. Not only are the two versions allowed to have arbitrarily different code and data layout, but patch analysis and preparation tools are no longer necessary. The new version is compiled and deployed as-is, with changes between versions automatically tracked by our state transfer framework at runtime. Moreover, Ksplice does not support update states other than function quiescence and provides no support for automated state transfer, state checking, or hot rollback.

Related to OS-level live update solutions is also work on extensible operating systems [76; 47; 259; 258] (which only allow predetermined OS extensions), dynamic kernel instrumentation [273; 203] (which is primarily concerned with debugging and performance monitoring), microkernel architectures [178; 263; 82; 140] (which can replace OS subsystems but not without causing service loss [82]), and online maintenance techniques [189; 241] (which require virtualization and domain-specific migration tools).

Similar to OS-level solutions, existing live update techniques for user-space C programs all assume an in-place update model, with code and data changes loaded directly into the running version. Redirection of execution is accomplished with compiler-based techniques [214; 213], binary rewriting [32; 69], or stack reconstruction [193]. Some techniques assume quiescence [32], others rely on predetermined update points [214; 213; 193] or allow unrestricted cross-version execution [69]. Unlike PROTEOS, these solutions offer no support to specify safe update states on a per-update basis, do not attempt to fully automate state transfer or state checking, and fail to ensure a transactional and stable update process.

Recent efforts on user-space C programs by Hayden et al. [132], developed independently from our work, also suggest using entire programs as live updatable units. Unlike our process-level updates, however, their update strategy encapsulates every program version inside a shared library and allows the old and the new version to share the same process address space with no restriction at live update time. This strategy requires every program to be compiled with only position-independent code—which may be particularly inefficient on some architectures—and also fails to properly isolate, detect, and recover from errors in the state transfer code. In addition, their state transfer strategy does not support interior pointers and unrestricted use of `void*` pointers, nor does it attempt to automate pointer transfer for heap-allocated objects with no user intervention. Finally, their system includes xfgen, a tool to generate state transformers using a domain-specific language. While a high-level language may reduce the programming effort, we found much more natural to express state transfer extensions for C programs directly in C, using a convenient and well-defined callback interface.

The techniques used in PROTEOS draw inspiration from prior work in different research areas. Our state filters are inspired by DYMOS [181], an early dynamic modification system that allowed programmers to specify procedures required to be inactive at update time. State filters are more general and easier to use, allowing programmers to specify safe update states in the most natural way. The idea of state transfer between processes was first explored by Gupta [124], but his work assumed a fixed memory layout and delegated state transfer entirely to the programmer. Our state introspection strategy is inspired by garbage collector-style object tracking, a technique also explored in live update solutions for managed languages like Java [269]. Similarly, our update-time memory leakage reclaiming strategy is inspired by prior precise garbage collection techniques for C programs [243]. Finally, state checking is inspired by invariants-based techniques to detect anomalous program behavior [97; 127; 300; 89; 22; 233]. Unlike prior techniques, our state invariants are conservatively derived from static analysis, eliminating false positives that arise from learning *likely* invariants at runtime.

2.8 Conclusion

In this paper, we presented PROTEOS, a new research OS designed with live update in mind. Unlike existing solutions, the techniques implemented in PROTEOS can efficiently and reliably support several classes of updates with minimal manual effort. State and interface filters allow updates to happen only in predictable system states and give programmers full control over the update process. Process-level updates completely eliminate the need for complex toolchains, enable safe hot rollback, and ensure a stable update process. Our state management framework reduces the state transfer burden to the bare minimum, fully automating state transfer for common structural state changes and exposing a convenient programming model for extensions. Finally, our state checking framework can automatically identify errors in a tainted state and detect violating assumptions in the state transfer process itself.

2.9 Acknowledgments

We would like to thank the anonymous reviewers for their comments. This work has been supported by European Research Council under ERC Advanced Grant 227874.

3

Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization

Abstract

In recent years, the deployment of many application-level countermeasures against memory errors and the increasing number of vulnerabilities discovered in the kernel has fostered a renewed interest in kernel-level exploitation. Unfortunately, no comprehensive and well-established mechanism exists to protect the operating system from arbitrary attacks, due to the relatively new development of the area and the challenges involved.

In this paper, we propose the first design for fine-grained address space randomization (ASR) inside the operating system (OS), providing an efficient and comprehensive countermeasure against classic and emerging attacks, such as return-oriented programming. To motivate our design, we investigate the differences with application-level ASR and find that some of the well-established assumptions in existing solutions are no longer valid inside the OS; above all, perhaps, that information leakage becomes a major concern in the new context. We show that our ASR strategy outperforms state-of-the-art solutions in terms of both performance and security without affecting the software distribution model. Finally, we present the first comprehensive *live rerandomization* strategy, which we found to be particularly important inside the OS. Experimental results demonstrate that our techniques yield low run-time performance overhead (less than 5% on average on both SPEC and syscall-intensive benchmarks) and limited run-time memory footprint increase (around 15% during the execution of our benchmarks). We believe our techniques can greatly enhance the level of OS security without compromising the performance and reliability of the OS.

3.1 Introduction

Kernel-level exploitation is becoming increasingly popular among attackers, with local and remote exploits surfacing for Windows [11], Linux [10], Mac OS X [14], BSD variants [157; 20], and embedded operating systems [98]. This emerging trend stems from a number of important factors. First, the deployment of defense mechanisms for user programs has made application-level exploitation more challenging. Second, the kernel codebase is complex, large, and in continuous evolution, with many new vulnerabilities inevitably introduced over time. Studies on the Linux kernel have shown that its codebase has more than doubled with a steady fault rate over the past 10 years [231] and that many known but potentially critical bugs are at times left unpatched indefinitely [121]. Third, the number of targets in large-scale attacks is significant, with a plethora of internet-connected machines running the same kernel version independently of the particular applications deployed. Finally, an attacker has generally more opportunities inside the OS, for example the ability to disable in-kernel defense mechanisms or the option to execute shellcode at the user level (similar to classic application-level attacks) or at the kernel level (approach taken by kernel *rootkits*).

Unfortunately, existing OS-level countermeasures fail to provide a comprehensive defense mechanism against generic memory error exploits. A number of techniques aim to thwart code injection attacks [260; 116; 246], but are alone insufficient to prevent *return-into-kernel-text* attacks [236] and *return-oriented programming* (ROP) in general [152]. Other approaches protect kernel hooks or generally aim at preserving control-flow integrity [281; 291; 183; 237]. Unfortunately, this does not prevent attackers from tampering with noncontrol data, which may lead to privilege escalation or allow other attacks. In addition, most of these techniques incur high overhead and require virtualization support, thus increasing the size of the trusted computing base (TCB).

In this paper, we explore the benefits of address space randomization (ASR) inside the operating system and present the first comprehensive design to defend against classic and emerging OS-level attacks. ASR is a well-established defense mechanism to protect user programs against memory error exploits [49; 167; 50; 288; 289]; all the major operating systems include some support for it at the application level [2; 274]. Unfortunately, the OS itself is typically not randomized at all. Recent Windows releases are of exception, as they at least randomize the base address of the text segment [236]. This randomization strategy, however, is wholly insufficient to counter many sophisticated classes of attacks (e.g., noncontrol data attacks) and is extremely vulnerable to information leakage, as better detailed later. To date, no strategy has been proposed for comprehensive and fine-grained OS-level ASR. Our effort lays the ground work to fill the gap between application-level ASR and ASR inside the OS, identifying the key requirements in the new context and proposing effective solutions to the challenges involved.

3.1.1 Contributions

The contributions of this paper are threefold. First, we identify the challenges and the key requirements for a comprehensive OS-level ASR solution. We show that a number of assumptions in existing solutions are no longer valid inside the OS, due to the more constrained environment and the different attack models. Second, we present the first design for fine-grained ASR for operating systems. Our approach addresses all the challenges considered and improves existing ASR solutions in terms of both performance and security, especially in light of emerging ROP-based attacks. In addition, we consider the application of our design to component-based OS architectures, presenting a fully fledged prototype system and discussing real-world applications of our ASR technique. Finally, we present the first generic *live rerandomization* strategy, particularly central in our design. Unlike existing techniques, our strategy is based on run-time state migration and can transparently rerandomize arbitrary code and data with no state loss. In addition, our rerandomization code runs completely sandboxed. Any run-time error at rerandomization time simply results in restoring normal execution without endangering the reliability of the OS.

3.2 Background

The goal of address space randomization is to ensure that code and data locations are unpredictable in memory, thus preventing attackers from making precise assumptions on the memory layout. To this end, fine-grained ASR implementations [50; 167; 288] permute the order of individual memory objects, making both their addresses and their relative positioning unpredictable. This strategy attempts to counter several classes of attacks.

3.2.1 Attacks on code pointers

The goal of these attacks is to override a function pointer or the return address on the stack with attacker-controlled data and subvert control flow. Common memory errors that can directly allow these attacks are buffer overflows, format bugs, use-after-free, and uninitialized reads. In the first two cases, the attack requires assumptions on the relative distance between two memory objects (e.g., a vulnerable buffer and a target object) to locate the code pointer correctly. In the other cases, the attack requires assumptions on the relative alignment between two memory objects in case of memory reuse. For example, use-after-free attacks require control over the memory allocator to induce the allocation of an object in the same location of a freed object still pointed by a vulnerable dangling pointer. Similarly, attacks based on stack/heap uninitialized reads require predictable allocation strategies to reuse attacker-controlled data from a previously deallocated object. All these attacks also rely on the absolute location of the code the attacker wants to execute, in order to adjust the value of the code pointer correctly. In detail, code injection attacks rely on the location of attacker-injected shellcode. Attacks using *return-into-libc* strategies [86] rely on the location of a particular function—or multiple functions in case of chained *return-into-libc* attacks [216]. More generic attacks based on *return-oriented programming* [261] rely on the exact location of a number of *gadgets* statically extracted from the program binary.

3.2.2 Attacks on data pointers

These attacks commonly exploit one of the memory errors detailed above to override the value of a data pointer and perform an arbitrary memory read/write. Arbitrary memory reads are often used to steal sensitive data or information on the memory layout. Arbitrary memory writes can also be used to override particular memory locations and indirectly mount other attacks (e.g., control-flow attacks). Attacks on data pointers require the same assumptions detailed for code pointers, except the attacker needs to locate the address of some data (instead of code) in memory.

3.2.3 Attacks on nonpointer data

Attacks in this category target noncontrol data containing sensitive information (e.g., `uid`). These attacks can be induced by an arbitrary memory write or commonly originate from buffer overflows, format bugs, integer overflows, signedness bugs, and use-after-free memory errors. While unable to directly subvert control flow, they can often lead to privilege escalation or indirectly allow other classes of attacks. For example, an attacker may be able to perform an arbitrary memory write by corrupting an array index which is later used to store attacker-controlled data. In contrast to all the classes of attacks presented earlier, nonpointer data attacks only require assumptions on the relative distance or alignment between memory objects.

3.3 Challenges in OS-level ASR

This section investigates the key challenges in OS-level address space randomization, analyzing the differences with application-level ASR and reconsidering some of the well-established assumptions in existing solutions. We consider the following key issues in our analysis.

3.3.1 W⊕X

A number of ASR implementations complement their design with W⊕X protection [274]. The idea is to prevent code injection attacks by ensuring that no memory page is ever writable and executable at the same time. Studies on the Linux kernel [184], however, have shown that enforcing the same property for kernel pages introduces implementation issues and potential sources of overhead. In addition, protecting kernel pages in a combined user/kernel address space design does not prevent an attacker from placing shellcode in an attacker-controlled application and redirecting execution there. Alternatively, the attacker may inject code into $W \wedge X$ regions with double mappings that operating systems share with user programs (e.g., `vsyscall` page on Linux) [236].

3.3.2 Instrumentation

Fine-grained ASR techniques typically rely on code instrumentation to implement a comprehensive randomization strategy. For example, Bhaktar et al. [50] heavily instrument the program to create self-randomizing binaries that completely rearrange their memory layout at load time. While complex instrumentation strategies have been proven practical for application-level solutions, their applicability to OS-level ASR raises a number of important concerns. First, heavyweight instrumentation may introduce significant run-time overhead which is ill-affordable for the OS. Second, these load-time ASR strategies are hardly sustainable, given the limited operations they would be able to perform and the delay they would introduce in the boot process. Finally, complex instrumentation may introduce a lot of untrusted code executed with no restriction at runtime, thus endangering the reliability of the OS or even opening up new opportunities for attack.

3.3.3 Run-time constraints

There are a number of constraints that significantly affect the design of an OS-level ASR solution. First, making strong assumptions on the memory layout at load time simplifies the boot process. This means that some parts of the operating system may be particularly hard to randomize. In addition, existing rerandomization techniques are unsuitable for operating systems. They all assume a stateless model in which a program can gracefully exit and restart with a fresh rerandomized layout. Loss of critical state is not an option for an OS and neither is a full reboot, which introduces unacceptable downtime and loss of all the running processes. Luckily, similar restrictions also apply to an adversary determined to attack the system. Unlike application-level attacks, an exploit needs to explicitly recover any critical memory object corrupted during the attack or the system will immediately crash after successful exploitation.

3.3.4 Attack model

Kernel-level exploitation allows for a powerful attack model. Both remote and local attacks are possible, although local attacks mounted from a compromised or attacker-controlled application are more common. In addition, many known attack strategies become significantly more effective inside the OS. For example, noncontrol data attacks are more appealing given the amount of sensitive data available. In addition, ROP-based control-flow attacks can benefit from the large codebase and easily find all the necessary gadgets to perform arbitrary computations, as demonstrated in [152]. This means that disclosing information on the locations of “useful” text fragments can drastically increase the odds of successful ROP-based attacks. Finally, the particular context opens up more attack opportunities than those detailed in §3.2. First, unchecked pointer dereferences with user-provided data—a common vulnerability in kernel development [67]—can become a vector of arbitrary kernel memory reads/writes with no assumption on the location of the original pointer. Second, the combined user/kernel address space design used in most operating systems may allow an attacker controlling a user program to directly leverage known application code or data for the attack. The conclusion is that making both the relative positioning between *any* two memory objects and the location of individual objects unpredictable becomes much more critical inside the OS.

3.3.5 Information leakage

Prior work on ASR has often dismissed information leakage attacks—in which the attacker is able to acquire information about the internal memory layout and carry out an exploit in spite of ASR—as relatively rare for user applications [50; 262; 288]. Unfortunately, the situation is completely different inside the OS. First, there are several possible entry points and a larger leakage surface than user applications. For instance, a recent study has shown that uninitialized data leading to information leakage is the most common vulnerability in the Linux kernel [67]. In addition, the common combined user/kernel address space design allows arbitrary memory writes to easily become a vector of information leakage for attacker-controlled applications. To make things worse, modern operating systems often disclose sensitive information to unprivileged applications voluntarily, in an attempt to simplify deployment and debugging. An example is the `/proc` file system, which has already been used in several attacks that exploit the exposed information in conventional [236] and nonconventional [297] ways. For instance, the `/proc` implementation on Linux discloses details on kernel symbols (i.e., `/proc/kallsyms`) and slab-level memory information (i.e., `/proc/slabinfo`). To compensate for the greater chances of information leakage, ASR at the finest level of granularity possible and continuous rerandomization become both crucial to minimize the knowledge acquired by an attacker while probing the system.

3.3.6 Brute forcing

Prior work has shown that many existing application-level ASR solutions are vulnerable to simple brute-force attacks due to the low randomization entropy of shared libraries [262]. The attack presented in [262] exploits the crash recovery capabilities of the Apache web server and simply reissues the same return-into-libc attack with a newly guessed address after every crash. Unlike many long-running user applications, crash recovery cannot be normally taken for granted inside the OS. An OS crash is normally fatal and immediately hinders the attack while prompting the attention of the system administrator. Even assuming some crash recovery mechanism inside the OS [182; 107], brute-force attacks need to be far less aggressive to remain unnoticed. In addition, compared to remote clients hiding their identity and mounting a brute-force attack against a server application, the source of an OS crash can be usually tracked down. In this context, blacklisting the offensive endpoint/request becomes a realistic option.

3.4 A design for OS-level ASR

Our fine-grained ASR design requires confining different OS subsystems into isolated event-driven components. This strategy is advantageous for a number of reasons. First, this enables selective randomization and rerandomization for individual subsystems. This is important to fully control the randomization and rerandomization process with per-component ASR policies. For example, it should be possible to retune the rerandomization frequency of *only* the virtual filesystem after noticing a performance impact under particular workloads. Second, the event-driven nature of the OS components greatly simplifies synchronization and state management at rerandomization time. Finally, direct intercomponent control transfer can be more easily prevented, thus limiting the freedom of a control-flow attack and reducing the number of potential ROP gadgets by design.

Our ASR design is currently implemented by a microkernel-based OS architecture running on top of the MINIX 3 microkernel [140]. The OS components are confined in independent hardware-isolated processes. Hardware isolation is beneficial to overcome the problems of a combined user/kernel address space design introduced earlier and limit the options of an attacker. In addition, the MMU-based protection can be used to completely sandbox the execution of the untrusted rerandomization code. Our ASR design, however, is not bound to its current implementation and has more general applicability.

For example, our ASR design can be directly applied to other component-based OS architectures, including microkernel-based architectures used in common embedded OSes—such as L4 [178], Green Hills Integrity [8], and QNX [143]—and research operating systems using software-based component isolation schemes—such as Singularity [153]. Commodity operating systems, in contrast, are traditionally based on monolithic architectures and lack well-defined component boundaries. While this does not prevent adoption of our randomization technique, it does eliminate the ability to selectively rerandomize specific parts of the OS, yielding poorer flexibility and longer rerandomization times to perform whole-OS state migration. Encouragingly, there is an emerging trend towards allowing important commodity OS subsystems to run as isolated user-space processes, including filesystems [7] and user-mode drivers in Windows [202] or Linux [60]. Our end-to-end design can be used to protect all these subsystems as well as other operating system services from

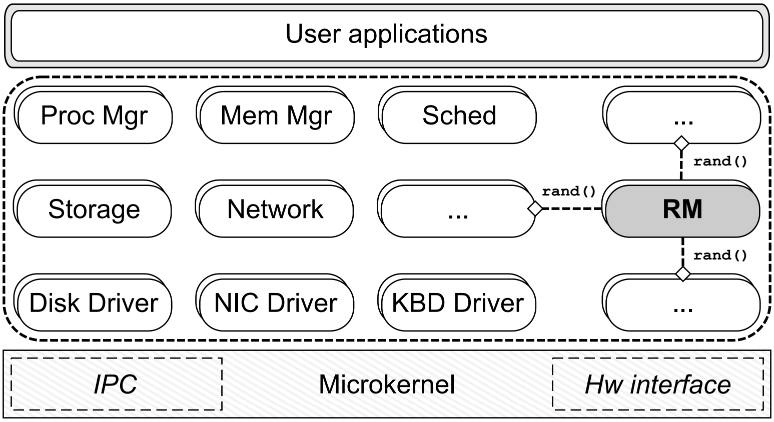


Figure 3.1: The OS architecture for our ASR design.

several classes of attacks. Note that, while running in user space, operating system services are typically trusted by the kernel and allowed to perform a variety of critical system operations. An example is `udev`, the device manager for the Linux kernel, which has already been target of several different exploits [61]. Finally, given the appropriate run-time support, our design could also be used to improve existing application-level ASR techniques and offer better protection against memory error exploits for generic user-space programs.

Figure 3.1 shows the OS architecture implementing our ASR design. At the heart lies the microkernel, providing only IPC functionalities and low-level resource management. All the other core subsystems are confined into isolated OS processes, including drivers, memory management, process management, scheduling, storage and network stack. In our design, all the OS processes (and the microkernel) are randomized using a link-time transformation implemented with the LLVM compiler framework [179]. The transformation operates on prelinked LLVM bitcode to avoid any lengthy recompilation process at runtime. Our link-time strategy avoids the need for fine-grained load-time ASR, eliminating delays in the boot process and the run-time overhead introduced by the indirection mechanisms adopted [50]. In addition, this strategy reduces the instrumentation complexity to the bare minimum, with negligible amount of untrusted code exposed to the runtime. The vast majority of our ASR transformations are statically verified by LLVM at the bitcode level. As a result, our approach is also safer than prior ASR solutions relying on binary rewriting [167].

As pointed out in [50], load-time ASR has a clear advantage over alternative strategies: the ability to create self-randomizing binaries distributed to every user in identical copies, thus preserving today's software distribution model. Fortunately, our novel live rerandomization strategy can fully address this concern. In our model, every user receives the same (unrandomized) binary version of the OS, as well as the prelinked LLVM bitcode of each OS component. The bitcode files are stored in a protected disk partition inaccessible to regular user programs, where a background process periodically creates new randomized variants of the OS components using our link-time ASR transformation (and any valid LLVM backend to generate the final binary). The generated variants are consumed by the *randomization manager* (RM), a special component that periodically rerandomizes every OS process (including itself). Unlike all the existing solutions, rerandomization is applied transparently online, with no system reboot or downtime required. The conclusion is that we can directly leverage our live rerandomization technique to randomize the original OS binary distributed to the user. This strategy retains the advantages of link-time ASR without affecting the software distribution model.

When the OS boots up for the first time, a full rerandomization round is performed to relinquish any unrandomized code and data present in the original binary. To avoid slowing down the first boot process, an option is to perform the rerandomization lazily, for example replacing one OS process at the time at regular time intervals. After the first round, we continuously perform live rerandomization of individual OS components in the background. Currently, the microkernel is the only piece of the OS that does not support live rerandomization. Rerandomization can only be performed after a full reboot, with a different variant loaded every time. While it is possible to extend our current implementation to support live rerandomization for the microkernel, we believe this should be hardly a concern. Microkernel implementations are typically in the order of 10kLOC, a vastly smaller TCB than most hypervisors used for security enforcement, as well as a candidate for formal verification, as demonstrated in prior work [168].

Our live rerandomization strategy for an OS process, in turn, is based on run-time state migration, with the entire execution state transparently transferred to the new randomized process variant. The untrusted rerandomization code runs completely sandboxed in the new variant and, in case of run-time errors, the old variant immediately resumes execution with no disruption of service or state loss. To support live migration, we rely on another LLVM link-time transformation to embed relocation and type information into the final process binary. This information is exposed to the runtime to accurately introspect the state of the two variants and migrate all the randomized memory objects in a layout-independent way.

3.5 ASR transformations

The goal of our link-time ASR transformation is to randomize all the code and data for every OS component. Our link-time strategy minimizes the time to produce new randomized OS variants on the deployment platform and automatically provides randomization for the program and all the statically linked libraries. Our transformation design is based on five key principles: (i) minimal performance impact; (ii) minimal amount of untrusted code exposed to the runtime; (iii) architecture-independence; (iv) no restriction on compiler optimizations; (v) maximum randomization granularity possible. The first two principles are particularly critical for the OS, as discussed earlier. Architecture-independence enhances portability and eliminates the need for complex binary rewriting techniques. The fourth principle dictates compiler-friendly strategies, for example avoiding indirection mechanisms used in prior solutions [49], which inhibit a number of standard optimizations (e.g., inlining). Eliminating the need for indirection mechanisms is also important for debuggability reasons. Our transformations are all debug-friendly, as they do not significantly change the code representation—only allocation sites are transformed to support live rerandomization, as detailed later—and preserve the consistency of symbol table and stack information. Finally, the last principle is crucial to provide lower predictability and better security than existing techniques.

Traditional ASR techniques [2; 274; 49] focus on randomizing the base address of code and data regions. This strategy is ineffective against all the attacks that make assumptions only about relative distances/alignments between memory objects, is prone to brute forcing [262], and is extremely vulnerable to information leakage. For instance, many examples of application-level information leakage have emerged on Linux over the years, and experience shows that, even by acquiring minimal knowledge on the memory layout, an attacker can completely bypass these basic ASR techniques [96].

To overcome these limitations, second-generation ASR techniques [50; 167; 288] propose fine-grained strategies to permute individual memory objects and randomize their relative distances/alignments. While certainly an improvement over prior techniques, these strategies are still vulnerable to information leakage, raising serious concerns on their applicability at the OS level. Unlike traditional ASR techniques, these strategies make it normally impossible for an attacker to make strong assumptions on the locations of arbitrary memory objects after learning the location of a single object. They are completely ineffective, however, in inhibiting precise assumptions on the layout of the leaked object itself. This is a serious concern inside the OS, where information leakage is the norm rather than the exception.

To address all the challenges presented, our ASR transformation is implemented by an LLVM link-time pass which supports fine-grained randomization of both the relative distance (or alignment) between *any* two memory objects and the *internal layout* of individual objects. We now present our transformations in detail and draw comparisons with prior techniques.

3.5.1 Code randomization

The code-transformation pass performs three primary tasks. First, it enforces a random permutation of all the program functions. In LLVM, this is possible by shuffling the symbol table in the intended order and setting the appropriate linkage to preserve the permutation at code generation time. Second, it introduces (configurable) random-sized padding before the first function and between any two functions in the bitcode, making the layout even more unpredictable. To generate the padding, we create dummy functions with a random number of instructions and add them to the symbol table in the intended position. Thanks to demand paging, even very large padding sizes do not significantly increase the run-time physical memory usage. Finally, unlike existing ASR solutions, we randomize the internal layout of every function.

To randomize the function layout, an option is to permute the basic blocks and the instructions in the function. This strategy, however, would hinder important compiler optimizations like branch alignment [294] and optimal instruction scheduling [195]. Nonoptimal placements can result in poor instruction cache utilization and inadequate instruction pipelining, potentially introducing significant run-time overhead. To address this challenge, our pass performs *basic block shifting*, injecting a dummy basic block with a random number of instructions at the top of every function. The block is never executed at runtime and simply skipped over, at the cost of only one additional jump instruction. Note that the order of the original instructions and basic blocks is left untouched, with no noticeable impact on run-time performance. The offset of every instruction with respect to the address of the function entry point is, however, no longer predictable.

This strategy is crucial to limit the power of an attacker in face of information leakage. Suppose the attacker acquires knowledge on the absolute location of a number of kernel functions (e.g., using `/proc/kallsyms`). While return-into-kernel-text attacks for these functions are still conceivable (assuming the attacker can subvert control flow), arbitrary ROP-based computations are structurally prevented, since the location of individual gadgets is no longer predictable. While the dummy basic block is in a predictable location, it is sufficient to cherrypick its instructions to avoid giving rise to any new useful gadget. It is easy to show that a sequence of `nop` instructions does not yield any useful gadget on the x86 [225], but other strategies may be necessary on other architectures.

3.5.2 Static data randomization

The data-transformation pass randomly permutes all the static variables and read-only data on the symbol table, as done before for functions. We also employ the same padding strategy, except random-sized dummy variables are used for the padding. Buffer variables are also separated from other variables to limit the power of buffer overflows. In addition, unlike existing ASR solutions, we randomize the internal layout of static data, when possible.

All the aggregate types in the C programming language are potential candidates for layout randomization. In practice, there are a number of restrictions. First, the order of the elements in an array cannot be easily randomized without changing large portions of the code and resorting to complex program analysis techniques that would still fail in the general case. Even when possible, the transformation would require indirection tables that translate many sequential accesses into random array accesses, sensibly changing the run-time cache behavior and introducing overhead.

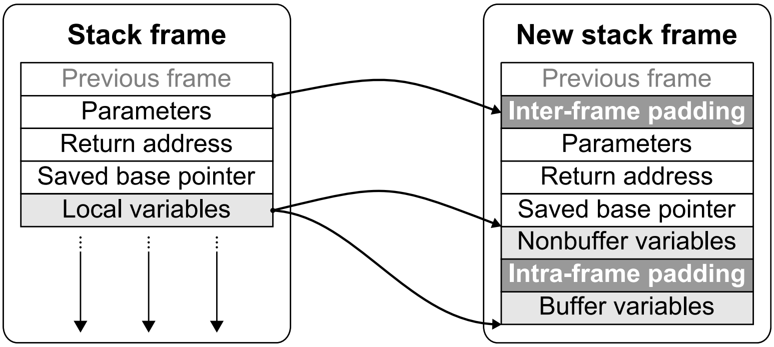


Figure 3.2: The transformed stack layout.

Second, `unions` are currently not supported natively by LLVM and randomizing their layout would introduce unnecessary complications, given their rare occurrence in critical system data structures and their inherent ambiguity that already weakens the assumptions made by an attacker. Finally, `packed structs` cannot be randomized, since the code makes explicit assumptions on their internal layout.

In light of these observations, our transformation focuses on randomizing the layout of regular `struct` types, which are pervasively used in critical system data structures. The layout randomization permutes the order of the `struct` members and adds random-sized padding between them. To support all the low-level programming idioms allowed by C, the type transformations are operated uniformly for all the static and dynamic objects of the same `struct` type. To deal with code which treats nonpacked `structs` as implicit `unions` through pointer casting, our transformation pass can be instructed to detect unsafe pointer accesses and refrain from randomizing the corresponding `struct` types.

Layout randomization of system data structures is important for two reasons. First, it makes the relative distance/alignment between two `struct` members unpredictable. For example, an overflow in a buffer allocated inside a `struct` cannot make precise assumptions about which other members will be corrupted by the overflow. Second, this strategy is crucial to limit the assumptions of an attacker in face of information leakage. Suppose an attacker is armed with a reliable arbitrary kernel memory write generated by a missing pointer check. If the attacker acquires knowledge on the location of the data structure holding user credentials (e.g., `struct cred` on Linux) for an attacker-controlled unprivileged process, the offset of the `uid` member is normally sufficient to surgically override the user ID and escalate privileges. All the existing ASR solutions fail to thwart this attack. In contrast, our layout randomization hinders any precise assumptions on the final location of the `uid`. While brute forcing is still possible, this strategy will likely compromise other data structures and trigger a system crash.

3.5.3 Stack randomization

The stack randomization pass performs two primary tasks. First, it randomizes the base address of the stack to make the absolute location of any stack object unpredictable. In LLVM, this can be accomplished by creating a dummy `alloca` instruction—which allocates memory on the stack frame of the currently executing function—at the beginning of the program, which is later expanded by the code generator. This strategy provides a portable and efficient mechanism to introduce random-sized padding for the initial stack placement. Second, the pass randomizes the relative distance/alignment between any two objects allocated on the stack. Prior ASR solutions have either ignored this issue [167; 288] or relied on a shadow stack and dynamically generated random padding [50], which introduces high run-time overhead (10% in the worst case in their experiments for user applications).

To overcome these limitations, our approach is completely static, resulting in good performance and code which is statically verified by LLVM. In addition, this strategy makes it realistic to use cryptographically random number generators (e.g., `/dev/random`) instead of pseudo-random generators to generate the padding. While care should be taken not to exhaust the randomness pool used by other user programs, this approach yields much stronger security guarantees than pseudo-random generators, like recent attacks on ASR demonstrate [96]. Our transformations can be configured to use cryptographically random number generators for code, data, and stack instrumentation, while, similar to prior approaches [50], we always resort to pseudo-random generation in the other cases for efficiency reasons.

When adopting a static stack padding strategy, great care should be taken not to degrade the quality of the randomization and the resulting security guarantees. To randomize the relative distances between the objects in a stack frame, we permute all the `alloca` instructions used to allocate local variables (and function parameters). The layout of every stack-allocated `struct` is also randomized as described earlier. Nonbuffer variables are all grouped and pushed to the top of the frame, close to the base pointer and the return address. Buffer variables, in turn, are pushed to the bottom, with random-sized padding (i.e., dummy `alloca` instructions) added before and between them. This strategy matches our requirements while allowing the code generator to emit a maximally efficient function prologue.

To randomize the relative alignment between any two stack frame allocations of the same function (and thus the relative alignment between their objects), we create random-sized padding before every function call. Albeit static, this strategy faithfully emulates dynamically generated padding, given the level of unpredictability introduced across different function calls. Function calls inside loops are an exception and need to be handled separately. Loop unrolling is a possible solution, but enforcing this optimization in the general case may be expensive. Our approach is instead to precompute N random numbers for each loop, and cycle through them before each function call. Figure 3.2 shows the randomized stack layout generated by our transformation.

3.5.4 Dynamic data randomization

Our operating system provides `malloc/mmap`-like abstractions to every OS process. Ideally, we would like to create memory allocation wrappers to accomplish the following tasks for both heap and memory-mapped regions: (i) add random-sized padding before the first allocated object; (ii) add random-sized padding between objects; (iii) permute the order of the objects. For memory-mapped regions, all these strategies are possible and can be implemented efficiently [167]. We simply need to intercept all the new allocations and randomly place them in any available location in the address space. The only restriction is for fixed OS component-specific virtual memory mappings, which cannot be randomized and need to be explicitly reserved at initialization time.

For heap allocations, we instrument the code to randomize the heap base address and introduce random-sized padding at allocation time. Permuting heap objects, however, is normally impractical in standard allocation schemes. While other schemes are possible—for example, the slab allocator in our memory manager randomizes block allocations within a slab page—state-of-the-art allocators that enforce a fully and globally randomized heap organization incur high overhead (117% worst-case performance penalty) [222]. This limitation is particularly unfortunate for kernel *Heap Feng Shui* attacks [98], which aim to carefully drive the allocator into a deterministic exploitation-friendly state. While random interobject padding makes these attacks more difficult, it is possible for an attacker to rely on more aggressive exploitation strategies (i.e., heap spraying [244]) in this context. Suppose an attacker can drive the allocator into a state with a very large unallocated gap followed by only two allocated buffers, with the latter vulnerable to underflow. Despite the padding, the attacker can induce a large underflow to override all the traversed memory locations with the same target value. Unlike stack-based overflows, this strategy could lead to successful exploitation without the attacker worrying about corrupting other critical data structures and crashing the system. Unlike prior ASR solutions, however, our design can mitigate these attacks by periodically rerandomizing every OS process and enforcing a new unpredictable heap permutation. We also rerandomize the layout of all the dynamically allocated `structs`, as discussed earlier.

3.5.5 Kernel modules randomization

Traditional loadable kernel module designs share many similarities—and drawbacks, from a security standpoint—with user-level shared libraries. The attack presented in [250] shows that the data structures used for dynamic linking are a major source of information leakage and can be easily exploited to bypass any form of randomization for shared libraries. Prior work on ASR [262; 50] discusses the difficulties of reconciling sharing with fine-grained randomization. Unfortunately, the inability to perform fine-grained randomization on shared libraries opens up opportunities for attacks, including probing, brute forcing [262], and partial pointer overwrites [95].

To overcome these limitations, our design allows only statically linked libraries for OS components and inhibits any form of dynamic linking inside the operating system. Note that this requirement does by no means limit the use of loadable modules, which our design simply isolates in independent OS processes following the same distribution and deployment model of the core operating system. This approach enables sharing and lazy loading/unloading of individual modules with no restriction, while allowing our rerandomization strategy to randomize (and rerandomize) every module in a fine-grained manner. In addition, the process-based isolation prevents direct control-flow and data-flow transfer between a particular module and the rest of the OS (i.e., the access is always IPC- or capability-mediated). Finally, this strategy can be used to limit the power of untrusted loadable kernel modules, an idea also explored in prior work on commodity operating systems [60].

3.6 Live rerandomization

Our live rerandomization design is based on novel automated run-time migration of the execution state between two OS process variants. The variants share the same operational semantics but have arbitrarily different memory layouts. To migrate the state from one variant to the other at runtime, we need a way to remap all the corresponding global state objects. Our approach is to transform the bitcode with another LLVM link-time pass, which embeds metadata information into the binary and makes run-time state introspection and automated migration possible.

3.6.1 Metadata transformation

The goal of our pass is to record metadata describing all the static state objects in the program and instrument the code to create metadata for dynamic state objects at runtime. Access to these objects at the bitcode level is granted by the LLVM API. In particular, the pass creates static metadata nodes for all the static variables, read-only data, and functions whose address is taken. Each metadata node contains three key pieces of information: node ID, relocation information, and type. The node ID provides a layout-independent mechanism to map corresponding metadata nodes across different variants. This is necessary because we randomize the order and the location of the metadata nodes (and write-protect them) to hinder new opportunities for attacks. The relocation information, in turn, is used by our run-time migration component to locate every state object in a particular variant correctly. Finally, the type is used to introspect any given state object and migrate the contained elements (e.g., pointers) correctly at runtime.

To create a metadata node for every dynamic state object, our pass instruments all the memory allocation and deallocation function calls. The node is stored before the allocated data, with canaries to protect the in-band metadata against buffer overflows. All the dynamic metadata nodes are stored in a singly-linked list, with

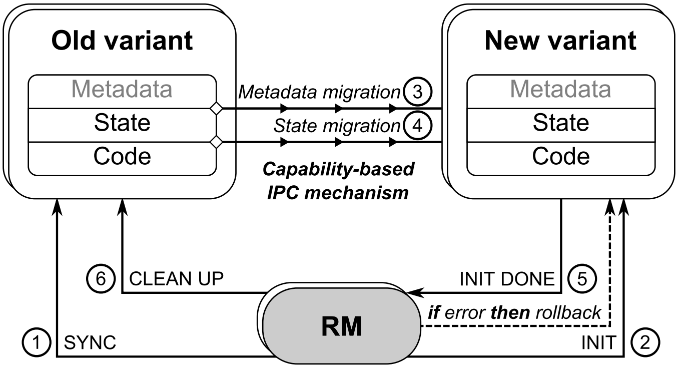


Figure 3.3: The rerandomization process.

each node containing relocation information, allocation flags, and a pointer to an allocation descriptor. Allocation flags define the nature of a particular allocation (e.g., heap) to reallocate memory in the new variant correctly at migration time. The allocation descriptors, in turn, are statically created by the pass for all the allocation sites in the program. A descriptor contains a site ID and a type. Similar to the node ID, the site ID provides a layout-independent mechanism to map corresponding allocation descriptors (also randomized and write-protected) across different variants. The type, in contrast, is determined via static analysis and used to correctly identify the run-time type of the allocated object (e.g., a `char` type with an allocation of 7 bytes results in a `[7 x char]` run-time type). Our static analysis can automatically identify the type for all the standard memory allocators and custom allocators that use simple allocation wrappers. More advanced custom allocation schemes, e.g., region-based memory allocators [46], require instructing the pass to locate the proper allocation wrappers correctly.

3.6.2 The rerandomization process

Our OS processes follow a typical event-driven model based on message passing. At startup, each process initializes its state and immediately jumps to the top of a long-running event-processing loop, waiting for IPC messages to serve. Each message can be processed in cooperation with other OS processes or the microkernel. The message dispatcher, isolated in a static library linked to every OS process, can transparently intercept two special system messages sent by the *randomization manager* (RM): *sync* and *init*. These messages cannot be spoofed by other processes because the IPC is mediated by the microkernel.

The rerandomization process starts with RM loading a new variant in memory, in cooperation with the microkernel. Subsequently, it sends a sync message to the designated OS process, which causes the current variant to immediately block in a well-defined execution point. A carefully selected synchronization point (e.g., in `main`) eliminates the need to instrument transient stack regions to migrate additional state, thus reducing the run-time overhead and simplifying the rerandomization strategy. The new variant is then allowed to run and delivered an init message with detailed instructions. The purpose of the init message is to discriminate between fresh start and rerandomization init. In the latter scenario, the message contains a capability created by the microkernel, allowing the new variant to read arbitrary data and metadata from the old variant. The capability is attached to the IPC endpoint of the designated OS process and can thus only be consumed by the new variant, which by design inherits the old variant's endpoint. This is crucial to transparently rerandomize individual operating system processes without exposing the change to the rest of the system.

When the rerandomization init message is intercepted, the message dispatcher requests the run-time migration component to initialize the new variant properly and then jumps to the top of the event-processing loop to resume execution. This preserves the original control flow semantics and transparently restores the correct execution state. The migration component is isolated in a library and runs completely sandboxed in the new variant. RM monitors the execution for run-time errors (i.e., panics, crashes, timeouts). When an error is detected, the new variant is immediately cleaned up, while the old variant is given back control to resume execution normally. When the migration completes correctly, in contrast, the old variant is cleaned up, while the new variant resumes execution with a rerandomized memory layout. We have also implemented rerandomization for RM itself, which only required some extra microkernel changes to detect run-time errors and arbitrate control transfer between the two variants. Our run-time error detection mechanism allows for safe rerandomization without trusting the (complex) migration code. Moreover, the reversibility of the rerandomization process makes detecting semantic errors in the migration code a viable option. For example, one could migrate the state from one variant to another, migrate it again to another instance of the original variant, and then compare the results. Figure 3.3 depicts the proposed rerandomization process.

3.6.3 State migration

The migration starts by transferring all the metadata from the old variant to a local cache in the new variant. Our capability-based design allows the migration code to locate a root metadata descriptor in the old variant and recursively copy all the metadata nodes and allocation descriptors to the new variant. To automate the metadata transfer, all the data structures copied use a fixed and predetermined layout. At the end, both the old and the new metadata are available locally, allowing the code to arbitrarily introspect the state of the two variants correctly. To automate the data transfer, we map every old metadata node in the local cache with its counterpart in the new variant. This is done by pairing nodes by ID and carefully reallocating every old dynamic state object in the new variant. Reallocations are performed in random order, thus enforcing a new unpredictable permutation of heap and memory-mapped regions. An interesting side effect of the reallocation process is the compaction of all the live heap objects, an operation that reduces heap fragmentation over time. Our strategy is indeed inspired by the way a compacting garbage collector operates [283].

The mapping phase generates all the perfect pairs of state objects in the two variants, ready for data migration. Note that paired state objects may not reflect the same type or size, due to the internal layout rerandomization. To transfer the data, the migration code introspects every state object in the old variant by walking its type recursively and examining each inner state element found. Nonpointer elements are simply transferred by value, while pointer elements require a more careful transfer strategy. To deal with layout randomization, each recursive step requires mapping the current state element to its counterpart (and location) in the new variant. This can be easily accomplished because the old type and the new type have isomorphic structures and only differ in terms of member offsets for randomized `struct` types. For example, to transfer a `struct` variable with 3 primitive members, the migration code walks the original `struct` type to locate all the members, computes their offsets in the two variants, and recursively transfers the corresponding data.

3.6.4 Pointer migration

The C programming language allows several programming idioms that make pointer migration particularly challenging in the general case. Our approach is to fully automate migration of all the common cases and only delegate the undecidable cases to the programmer. The first case to consider is a pointer to a valid static or dynamic state object. When the pointer points to the beginning of the object, we simply reinitialize the pointer with the address of the pointed object in the new variant. Interior pointers (i.e., pointers into the middle of an object) in face of internal layout rerandomization require a more sophisticated strategy. Similar to our introspection strategy, we walk the type of the pointed object and recursively remap the offset of the target element to its counterpart. This strategy is resilient to layout rerandomization and makes it easy to reinitialize the pointer in the new variant correctly.

Another scenario of interest is a pointer which is assigned a special integer value (e.g., `NULL` or `MAP_FAILED` (-1)). Our migration code can explicitly recognize special ranges and transfer the corresponding pointers by value. Currently, all the addresses in reserved memory ranges (e.g., zero pages) are marked as special values.

In another direction, memory addresses or other layout-specific information may be occasionally stored in integer variables. This is, unfortunately, a case of unsolvable ambiguity which cannot be automatically settled without programmer assistance. To this end, we support annotations to mark “hidden” pointers in the code.

Pointers stored in `unions` are another case of unsolvable ambiguity. Since C does not support tagged `unions`, it is impossible to resolve these cases automatically. In our experiments with OS code, `unions` with pointers were the only case of ambiguity that required manual intervention. Other cases are, however, possible. For example, any form of pointer encoding or obfuscation [48] would require knowledge on the particular encoding to migrate pointers correctly. Other classes of pointers—

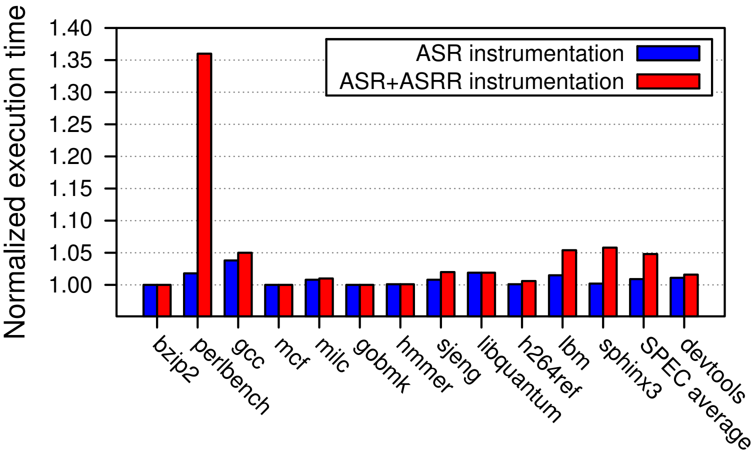


Figure 3.4: Execution time of the SPEC CPU2600 benchmarks and our *devtools* benchmark normalized against the baseline (no OS/benchmark instrumentation).

guard pointers, uninitialized pointers, dangling pointers—are instead automatically handled in our implementation. In the last two cases, the general strategy is to try to transfer the pointer as a regular pointer, and simply reinitialize it to `NULL` in the new variant whenever our dynamic pointer analysis reports an error.

3.7 Evaluation

We have implemented our ASR design on the MINIX 3 microkernel-based OS [140], which already guarantees process-based isolation for all the core operating system components. The OS is x86-based and exposes a complete POSIX interface to user applications. We have heavily modified and redesigned the original OS to implement support for our ASR techniques for all the possible OS processes. The resulting operating system comprises a total of 20 OS processes (7 drivers and 13 servers), including process management, memory management, storage and network stack. Subsequently, we have applied our ASR transformations to the system and evaluated the resulting solution.

3.7.1 Performance

To evaluate the performance of our ASR technique, we ported the C programs in the SPEC CPU 2006 benchmark suite to our prototype system. We also put together a *devtools* macrobenchmark, which emulates a typical syscall-intensive workload with the following operations performed on the OS source tree: compilation, **`find`**, **`grep`**, copying, and deleting. We performed repeated experiments on a workstation equipped with a 12-core 1.9Ghz AMD Opteron "Magny-Cours" processor and 4GB

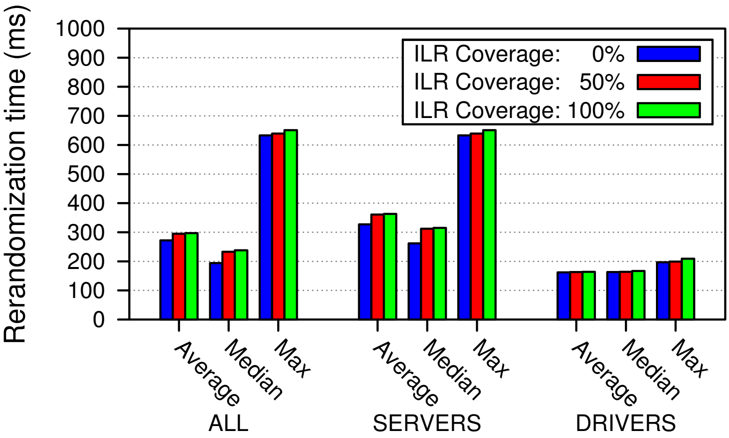


Figure 3.5: Rerandomization time against coverage of internal layout rerandomization.

of RAM, and averaged the results. All the OS code and our benchmarks were compiled using `Clang/LLVM` 2.8 with `-O2` optimization level. To thoroughly stress the system and identify all the possible bottlenecks, we instrumented *both* the OS and the benchmarks using the same transformation in each run. The default padding strategy used in the experiments extends the memory occupancy of every state object or `struct` member by 0-30%, similar to the default values suggested in [50]. Figure 3.4 depicts the resulting execution times.

The ASR instrumentation alone introduces 0.9% run-time overhead on average on SPEC benchmarks and 1.1% on *devtools*. The average run-time overhead increases to 4.8% and 1.6% respectively with ASRR instrumentation. The maximum overhead reported across all the benchmarks was found for `perlbench` (36% ASRR overhead). Profiling revealed this was caused by a massive amount of memory allocations. This test case pinpoints a potential source of overhead introduced by our technique, which, similar to prior approaches, relies on memory allocation wrappers to instrument dynamically allocated objects. Unlike prior comprehensive solutions, however, our run-time overhead is barely noticeable on average (1.9% for ASRR without `perlbench`). The most comprehensive second-generation technique presented in [50]—which, compared to other techniques, also provides fine-grained stack randomization—introduces a run-time overhead of 11% on average and 23% in the worst case, even by instrumenting *only* the test programs. The main reasons for the much higher overheads are the use of heavyweight stack instrumentation and indirection mechanisms that inhibit compiler optimizations and introduce additional pointer dereferences for every access to code and data objects. Their stack instrumentation, however, includes a shadow stack implementation that could complement our techniques to offer stronger protection against stack spraying attacks.

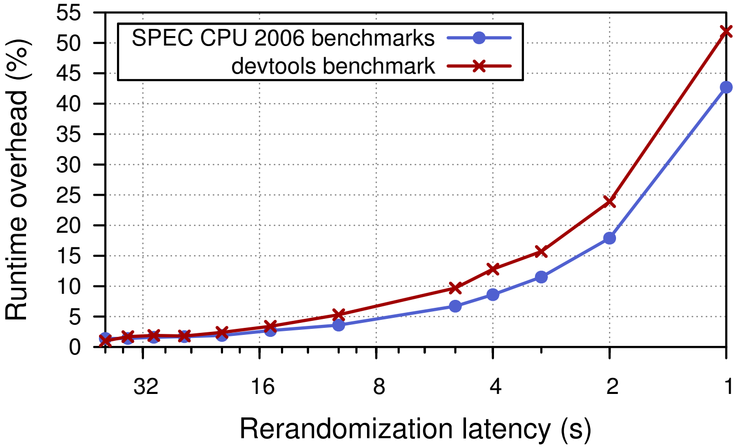


Figure 3.6: Run-time overhead against periodic rerandomization latency.

Although we have not observed strong variations in our macrobenchmark performance across different runs, our randomization technique can potentially affect the original spatial locality and yield nonoptimal cache usage at runtime. The possible performance impact introduced—inherent in all the fine-grained ASR techniques—is subject to the particular compiler and system adopted and should be carefully evaluated in each particular deployment scenario.

Figure 3.5 shows the rerandomization time (average, median, max) measured across all the OS components. With no internal layout rerandomization (ILR), a generic component completes the rerandomization process in 272ms on average. A higher ILR coverage increases the average rerandomization time only slightly (297ms at 100% coverage). The impact is more noticeable for OS servers than drivers, due to the higher concentration of complex rerandomized `structs` (and pointers to them) that need to be remapped during migration. Albeit satisfactory, we believe these times can be further reduced, for example using indexes to speed up our pointer analysis. Unfortunately, we cannot compare our current results against existing solutions, given that no other live rerandomization strategy exists to date.

Finally, Figure 3.6 shows the impact of periodic rerandomization on the execution time of SPEC and *devtools*. The experiment was performed by rerandomizing a single OS component at the end of every predetermined time interval. To ensure uniform coverage, the OS components were all rerandomized in a round-robin fashion. Figure 3.6 reports a barely noticeable overhead for rerandomization latencies higher than 20s. For lower latencies, the overhead increases steadily, reaching the value of 42.7% for SPEC and 51.9% for *devtools* at 1s. The rerandomization latency defines a clear tradeoff between performance and unobservability of the system. Reasonable choices of the rerandomization latencies introduce no performance impact and leave a small window with a stable view of the system to the attacker. In some cases,

Type	Overhead
ASRR state	16.1%
ASRR overall	14.6%
ASR padding$_a$	$((8a_s + 2a_h + 4a_f)\cdot 10^{-4} + c_{base})\%$
ASR padding$_r$	$((2r_s + 0.6r_h + 3r_f)\cdot 10^{-1} + c_{base})\%$

Table 3.1: Average run-time virtual memory overhead for our benchmarks.

a performance penalty may also be affordable to offer extra protection in face of particularly untrusted components.

3.7.2 Memory usage

Table 3.1 shows the average run-time virtual memory overhead introduced by our technique inside the OS during the execution of our benchmarks. The overhead measured is comparable to the space overhead we observed for the OS binaries on the disk. In the table, we report the virtual memory overhead to also account for dynamic state object overhead at runtime. For the average OS component, support for rerandomization introduces 16.1% state overhead (the extra memory necessary to store state metadata with respect to the original memory occupancy of all the static and dynamic static objects) and 14.6% overall memory overhead (the extra memory necessary to store state metadata and migration code with respect to the original memory footprint) on average. The virtual memory overhead (not directly translated to physical memory overhead, as noted earlier) introduced by our randomization strategy is only due to padding. Table 3.1 reports the overhead for two padding schemes using byte granularity (but others are possible): (i) padding$_a$, generating an inter-object padding of a bytes, with a uniformly distributed in $[0; a_{s,h,f}]$ for static, heap, and function objects, respectively; (ii) padding$_r$, generating an inter-object padding of $r \cdot s$ bytes, with a preceding object of size s, and r uniformly distributed in $[0; r_{s,h,f}]$ for static, heap, and function objects, respectively. The coefficient c_{base} is the overhead introduced by the one-time padding used to randomize the base addresses. The formulations presented here omit stack frame padding, which does not introduce persistent memory overhead.

3.7.3 Effectiveness

As pointed out in [50], an analytical analysis is more general and effective than empirical evaluation in measuring the effectiveness of ASR. Bhaktar et al. [50] present an excellent analysis on the probability of exploitation for different vulnerability classes. Their entropy analysis applies also to other second-generation ASR techniques, and, similarly, to our technique, which, however, provides additional entropy thanks to internal layout randomization and live rerandomization. Their analysis, however, is mostly useful in evaluating the effectiveness of ASR techniques against

	ASR₁	**ASR₂**	**ASR₃**
Vulnerability			
Buffer overflows	A_r	R_o	R_e
Format string bugs	A_r	R_o	R_e
Use-after-free	A_r	R_o	R_e
Uninitialized reads	A_r	R_o	R_e
Effect			
Arbitrary memory R|W	A_r	A_o	A_e
Controlled code injection	A_r	A_o	A_e
Return-into-libc/text	A_r	$N \cdot A_o$	$N \cdot A_o$
Return-oriented programming	A_r	$N \cdot A_o$	-

$A_r =$ Known region address
$A_o =$ Known object address
$A_e =$ Known element address
$R_o =$ Known relative distance/alignment between objects
$R_e =$ Known relative distance/alignment between elements

Table 3.2: Comparison of ASR techniques.

guessing and brute-force attacks. As discussed earlier, these attacks are far less attractive inside the OS. In contrast, information leakage dominates the scene.

For this reason, we explore another direction in our analysis, answering the question: "*How much information does the attacker need to acquire for successful exploitation?*". In this respect, Table 3.2 compares our ASR technique (ASR_3) with first-generation techniques like PaX [274] and second-generation techniques like the one presented in [50]. Most attacks require at least some knowledge of a memory area to corrupt and of another target area to achieve the intended effect (missing kernel pointer checks and non control data attacks are examples of exceptions in the two cases). Table 3.2 shows that first-generation techniques only require the attacker to learn the address of a memory region (e.g., stack) to locate the target correctly. Second-generation techniques, in turn, allow the attacker to corrupt the right location by learning the relative distance/alignment between two memory objects.

In this respect, our internal layout randomization provides better protection, forcing the attacker to learn the relative distance/alignment between two memory elements in the general case. For example, if the attacker learns the relative alignment between two heap-allocated data structures S_1 and S_2 and wants to exploit a vulnerable dangling pointer to hijack a write intended for a member of S_1 to a member of S_2, he still needs to acquire information on the relative positioning of the members.

Similarly, our technique constraints attacks based on arbitrary memory reads/writes to learn the address of the target element. In contrast, second-generation techniques only require knowledge of the target memory object. This is easier to acquire, because individual objects can be exposed by symbol information (e.g.,

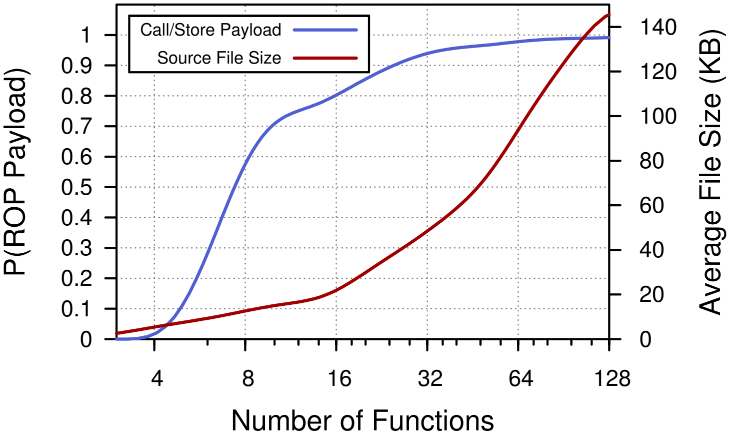


Figure 3.7: The probability that state-of-the-art techniques [256] can successfully generate valid ROP payloads to call linked functions or perform attacker-controlled arbitrary memory writes. The (fitted) distribution is plotted against the number of known functions in the program.

`/proc/kallsyms`) and are generally more likely to have their address taken (and leaked) than interior elements. Controlled code injection shows similar differences—spraying attacks are normally "A_r", in contrast. Return-into-libc/text, in turn, requires the attacker to learn the location of N chosen functions in both cases, because our function layout randomization has no effect.

Things are different in more general ROP-based attacks. Our strategy completely hinders these attacks by making the location of the gadgets inside a function unpredictable. Given that individual gadgets cannot have their address taken and function pointer arithmetic is generally disallowed in a program, the location of a gadget cannot be explicitly leaked. This makes information leakage attacks ineffective in acquiring any useful knowledge for ROP-based exploitation. In contrast, prior techniques only require the attacker to learn the address of *any* N functions with useful gadgets to mount a successful ROP-based attack. To estimate N, we made an analysis on GNU coreutils (v7.4), building on the results presented in [256]. Figure 3.7 correlates the number of program functions with the probability of locating all the necessary ROP gadgets, and shows, for example, that learning 16 function addresses is sufficient to carry out an attack in more than 80% of the cases.

Another interesting question is: "*How fast can the attacker acquire the required information*?". Our live rerandomization technique can periodically invalidate the knowledge acquired by an attacker probing the system (e.g., using an arbitrary kernel memory read). Shacham et al. [262] have shown that rerandomization slows down single-target probing attacks by only a factor of 2. As shown in Table 3.2, however, many attacks require knowledge of multiple targets when fine-grained ASR is in place. In addition, other attacks (e.g., Heap Feng Shui) may require multiple probing rounds to assert intermediate system states. When multiple rounds are required, the attacker is greatly limited by our rerandomization strategy because *any* knowledge acquired is only useful in the current rerandomization window. In particular, let us assume the duration of every round to be distributed according to some probability distribution $p(t)$ (e.g., computed from the probabilities given in [50]). Hence, the time to complete an n-round probing phase is distributed according to the convolution of the individual $p_i(t)$. Assuming the same $p_i(t)$ in every round for simplicity, it can be shown that the expected time before the attacker can complete the probing phase in a single rerandomization window (and thus the attack) is:

$$T_{attack} = T \cdot \left(\int_0^T p^{*n}(\tau) d\tau \right)^{-1},$$

where T is the size (ms) of the rerandomization window, n is the number of probing rounds, and $p^{*n}(t)$ is the n-fold convolution power of $p(t)$. Since the convolution power decreases rapidly with the number of targets n, the attack can quickly become impractical. Given a vulnerability and an attack model characterized by some $p(t)$, this formula gives a practical way to evaluate the impact of a given rerandomization frequency on attack prevention. When a new vulnerability is discovered, this formula can also be used to retune the rerandomization frequency (perhaps accepting a performance penalty) and make the attack immediately impractical, even before an appropriate patch is developed and made available. This property suggests that our ASR design can also be used as the first "live-workaround" system for security vulnerabilities, similar, in spirit, to other systems that provide immediate workarounds to bypass races at runtime [286].

3.8 Related work

3.8.1 Randomization

Prior work on ASR focuses on randomizing the memory layout of user programs, with solutions based on kernel support [167; 2; 274], linker support [289], compiler-based techniques [49; 50; 288], and binary rewriting [167; 57]. A number of studies have investigated attacks against poorly-randomized programs, including brute forcing [262], partial pointer overwrites [95], and return-oriented programming [256; 250]. Our ASR design is more fine-grained than existing techniques and robust against these attacks and information leakage. In addition, none of the existing approaches can support stateful live rerandomization. The general idea of randomization has also been applied to instruction sets (to thwart code injection attacks) [166; 238; 150], data representation (to protect noncontrol data) [48], data structures (to mitigate rootkits) [187], memory allocators (to protect against heap exploits) [222] Our struct layout randomization is similar to the one presented in [187], but our ASR design generalizes this strategy to the internal layout of any memory object (including code) and also allows live layout rerandomization. Finally, randomization as a general form of diversification [101] has been proposed to execute multiple program variants in parallel and detect attacks from divergent behavior [78; 253; 254].

3.8.2 Operating system defenses

Prior work on OS defenses against memory exploits focuses on control-flow attacks. SecVisor [260] is a hypervisor-based solution which uses memory virtualization to enforce W$\oplus$X protection and prevent code injection attacks. Similarly, NICKLE [246] is a VMM-based solution which stores authenticated kernel code in guest-isolated shadow memory regions and transparently redirects execution to these regions at runtime. Unlike SecVisor, NICKLE can support unmodified OSes and seamlessly handle mixed kernel pages with code and data. hvmHarvard [116] is a hypervisor-based solution similar to NICKLE, but improves its performance with a more efficient instruction fetch redirection strategy at the page level. The idea of memory shadowing is also explored in HookSafe [281], a hypervisor-based solution which relocates kernel hooks to dedicated memory pages and employs a hook indirection layer to disallow unauthorized overrides. Other techniques to defend against kernel hook hijacking have suggested dynamic monitoring strategies [291; 237] and compiler-based indirection mechanisms [183] Finally, Dalton et al. [81] present a buffer overflow detection technique based on data flow tracking and demonstrate its practical applicability to the Linux kernel. None of the techniques discussed here provides a comprehensive solution to OS-level attacks. Remarkably, none of them protects noncontrol data, a common target of attacks in local exploitation scenarios.

3.8.3 Live rerandomization

Unlike our solution, none of the existing ASR techniques can support live rerandomization with no state loss. Prior work that comes closest to our live rerandomization technique is in the general area of dynamic software updating. Many solutions have been proposed to apply run-time updates to user programs [214; 193; 32; 69] and operating systems [194; 43; 36] Our rerandomization technique shares with these solutions the ability to modify code and data of a running system without service interruption. The fundamental difference is that these solutions apply run-time changes in place, essentially assuming a fixed memory layout where any state transformation is completely delegated to the programmer. Our solution, in contrast, is generic and automated, and can seamlessly support arbitrary memory layout transformations between variants at runtime. Other solutions have proposed process-level run-time updates to release some of the assumptions on the memory layout [124; 131], but they still delegate the state transfer process completely to the programmer. This completely hinders their applicability in live rerandomization scenarios where arbitrary layout transformations are allowed.

3.9 Conclusion

In this paper, we introduced the first ASR design for operating systems. To fully explore the design space, we presented an analysis of the different constraints and attack models inside the OS, while highlighting the challenges of OS-level ASR. Our analysis reveals a fundamental gap with long-standing assumptions in existing application-level solutions. For example, we show that information leakage, traditionally dismissed as a relatively rare event, becomes a major concern inside the OS. Building on these observations, our design takes the first step towards truly fine-grained ASR for OSes. While our prototype system is targeted towards component-based OS architectures, the principles and the techniques presented are of much more general applicability. Our technique can also be applied to generic user programs, improving existing application-level techniques in terms of both performance and security, and opening up opportunities for third-generation ASR systems. The key to good performance (and no impact on the distribution model) is our link-time ASR strategy used in combination with live rerandomization. In addition, this strategy is more portable and much safer than existing techniques, which either rely on complex binary rewriting or require a substantial amount of untrusted code exposed to the runtime. In our technique, the complex rerandomization code runs completely sandboxed and any unexpected run-time error has no impact on normal execution. The key to good security is the better randomization granularity combined with periodic live rerandomization. Unlike existing techniques, we can (re)randomize the internal layout of memory objects and periodically rerandomize the system with no service interruption or state loss. These properties are critical to counter information leakage attacks and truly maximize the unobservability of the system.

3.10 Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments. This work has been supported by European Research Council under grant ERC Advanced Grant 227874.

4

Practical Automated Vulnerability Monitoring Using Program State Invariants

Abstract

Despite the growing attention to security concerns and advances in code verification tools, many memory errors still escape testing and plague production applications with security vulnerabilities. We present RCORE, an efficient dynamic program monitoring infrastructure to perform automated security vulnerability monitoring. Our approach is to perform extensive static analysis at compile time to automatically index *program state invariants* (PSIs). At runtime, our novel dynamic analysis continuously inspects the program state and produces a report when PSI violations are found. Our technique retrofits existing applications and is designed for both offline and production runs. To avoid slowing down production applications, we can perform our dynamic analysis on idle cores to detect suspicious behavior in the background. The alerts raised by our analysis are symptoms of memory corruption or other—potentially exploitable—dangerous behavior. Our experimental evaluation confirms that RCORE can report on several classes of vulnerabilities with very low overhead.

4.1 Introduction

Memory errors represent a major source of security vulnerabilities for widely deployed programs written in type-unsafe languages like C. According to the NIST's National Vulnerability Database [221], 662 memory error vulnerabilities were published in 2011 and 724 in 2012. While software engineers strive to identify memory errors and other vulnerabilities as part of the development process, dynamic vulnerability monitoring and identification in production runs is a compelling option for two important reasons.

First, many security vulnerabilities escape software testing and are only later discovered in production applications at a steady rate every year [154]. This is due to the limited power of code analysis tools and the inability to test all the possible execution scenarios effectively in offline runs. Given the large-scale deployment of today's production applications, it is no wonder that the testing surface can increase drastically in production runs, with different installations subject to very different environments and workloads. This gives a much better chance for zero-day vulnerabilities to emerge.

In addition, experience suggests that the number of unpatched vulnerabilities is substantial every year. A recent study [154] has shown that only 53% of the vulnerabilities disclosed in 2012 were patched by the end of the year. When prioritizing the known security vulnerabilities to go after becomes a necessity, a dynamic vulnerability monitoring infrastructure can provide a useful feedback to analyze the impact of those vulnerabilities in production.

Unfortunately, state-of-the-art solutions designed to detect and protect against different classes of memory errors [29; 21; 50; 293; 65; 73; 88; 251; 146] are not well-suited to be used for comprehensive vulnerability monitoring in production runs. Despite significant effort, they still incur substantial overhead and often fail to provide any meaningful feedback.

This paper presents RCORE, a new dynamic program monitoring infrastructure that continuously inspects the state of a running program and provides informative feedback about generic memory errors and potential security vulnerabilities. Our solution barely impacts running applications and retrofits existing programs and already deployed shared libraries. Our low-overhead design can completely decouple the execution of a target program and the execution of the monitor, isolating the monitoring thread on a separate core.

To detect many classes of memory errors, our approach builds on a combination of static and dynamic analysis. Static analysis is performed at compile time to embed in the final binary all the *program state invariants*, which specify inviolable safety constraints for the different components of the program state (i.e., objects, types, values). The key idea is to detect memory errors and suspicious behavior from runtime violations of the prerecorded invariants maintained in memory. To accomplish this task, our run-time monitor continuously inspects the program state in the background and reports every violation found, along with all the necessary information to track down the original problem. Our analysis is concerned with security and not space overhead, given that RAM is hardly a scarce resource nowadays.

To achieve the lowest possible overhead—at the cost of reduced precision— in production runs, our default invariants analysis strategy is fully asynchronous. This approach results in a probabilistic detection model specifically designed to detect forms of global state corruption. This category covers a significant fraction of emerging vulnerabilities, which produce latent errors or silent data corruption and may normally go undetected [100]. This trend is reflected in a growing number of exploits moving from stack-based attacks to data or heap-based attacks [222]. Our ultimate goal is to build an automated security vulnerability reporting service, similar, in spirit, to widely used remote crash reporting tools.

The contribution of this paper is threefold. First, we introduce a novel program state invariants analysis which is used as basis for our detection technique. Second, we show that our invariants analysis can be effectively used to infer both suspicious behavior that can cause memory errors and memory corruption caused by a memory error, even when the root cause is unknown. Our analysis covers the entire global program state (i.e., data, heap, and memory-mapped regions) and can detect a broad class of memory errors, including buffer overflows, dangling pointers, double or invalid frees, and uninitialized reads. Compared to existing techniques, we support detection of memory corruption caused by the libraries, application-specific memory management, and memory errors that do not spread across data structure boundaries (e.g., buffer overflow *inside* a struct). Third, we have developed a system, termed RCORE, which can reuse dedicated spare cores to perform our invariants analysis on a running program in real time. Our prototype shows that our analysis can be efficiently parallelized and used in practical vulnerability monitoring scenarios. To the best of our knowledge, we are the first to support such a fine-grained vulnerability analysis and show that it can be performed in real time with very low overhead.

4.2 Program State Invariants

Program state invariants (PSIs) represent global safety constraints that restrain the run-time behavior of every state element in the program in an *execution-agnostic* fashion. We use the term state element (or *s-element*) to refer to typed memory objects (i.e., variables or dynamically allocated objects) and their recursively defined members (e.g., `struct` fields or array elements) indiscriminately. We consider PSIs for both pointer and nonpointer *s-elements* for programs written in type-unsafe languages like C.

Our current system supports three types of PSIs: *value-based* PSIs, *target-based* PSIs, and *type-based* PSIs. Value-based PSIs restrict the set of legal values for both pointer and nonpointer *s-elements*. Target-based PSIs specify the set of valid targets a pointer *s-element* can point to (i.e., a pointer must point to a valid *s-element* in memory). Finally, type-based PSIs restrict the set of legal types a pointer *s-element* can point to (e.g., a function pointer must point to a valid function *s-element* with a matching type).

Unlike other invariant-based techniques [97; 127; 300; 89; 75; 22; 233] or more general learning-based techniques [247; 206] that aim to automatically detect anomalous behavior, our invariants are execution-agnostic and solely determined from static analysis. While this strategy might miss some valid invariants that can only be determined by fine-grained dynamic monitoring, our approach eliminates the coverage problems that arise when learning invariants at runtime and results in a more conservative invariants analysis, ruling out false alarms at detection time—other techniques incorrectly raise an alert whenever a program element reports a legitimate value never observed in the training phase. In particular, when not using proactive detection of suspicious behavior like long-lived dangling or off-by-N pointers, RCORE's conservative analysis squarely meets the goal of zero false positives.

Another advantage of using static compile-time information to learn properties of the program behavior is the ability to derive restrictive and fine-grained PSIs. This immediately suggests that run-time PSI violations can be used as an accurate predictor for memory errors. The key intuition is that, when some form of arbitrary state corruption occurs, the probability of no PSI being violated is low. This is true independently of the particular memory error that caused the corruption. For example, when a global data pointer is corrupted with arbitrary data, the chance that the pointer is still pointing to an object of a valid type (as determined by static analysis) is negligible—equivalent to the probability of randomly guessing the address of a valid memory object. Similarly, a global function pointer corrupted with arbitrary data by a memory error is unlikely to still point to a valid function with a valid type (as determined by static analysis). As a result, both scenarios will allow RCORE to detect a target- or type-based PSI violation with high probability for the corrupted pointer *s-elements* and immediately generate a report on the memory errors found.

Finally, the violation patterns reported can be used to classify the memory errors detected (e.g., contiguous *s-elements* with PSI violations in a buffer overflow).

4.3 Architecture

Our architecture comprises 5 main components: compiler driver, static instrumentation component, metadata framework, dynamic instrumentation component, and run-time analyzer. The compiler driver is designed to support static analysis and instrumentation of existing applications and libraries, integrating seamlessly with existing build systems.

The static instrumentation component—implemented on top of the LLVM compiler framework [179]—inspects the program to identify all the *s-elements* and PSIs at compile time and instruments the final binary to store the corresponding metadata. The latter are used to validate the program state against all the prerecorded invariants at runtime. For this purpose, the metadata framework provides an API to

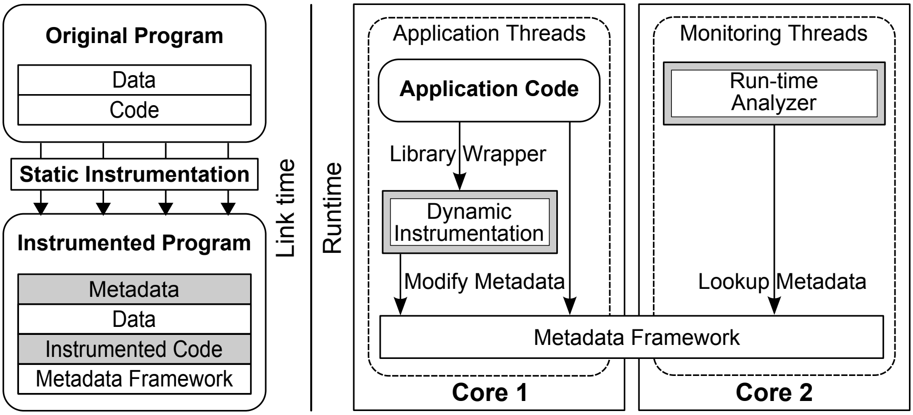


Figure 4.1: The RCORE architecture.

query and manage statically and dynamically created metadata. The framework is transparently linked to the program by the compiler driver during the build process.

The dynamic instrumentation component creates and destroys dynamic metadata to support uninstrumented shared libraries. This step is necessary to perform a conservative target-based analysis and avoid spurious alerts. In contrast, using only the dynamic instrumentation component without any static instrumentation as done in state-of-the-art memory allocators [27; 222], would degrade the effectiveness of our analysis, as explained later. The run-time analyzer is responsible for monitoring the behavior of the program in real time and detecting PSI violations. At each monitoring cycle, the analyzer uses the metadata framework to introspect the program state and check annotated PSIs for each *s-element* found. Figure 4.1 depicts RCORE's architecture and the interactions between the components at runtime.

4.3.1 Static Instrumentation

The static instrumentation component is an LLVM link-time pass that analyzes and instruments the program and statically linked libraries to embed state metadata into the resulting LLVM bitcode. The latter is then processed by a valid LLVM back end to produce the final binary.

Our static analysis starts by extracting relocation and type information on variables and functions used in the program. These objects will become part of the embedded metadata that index all the *s-elements* and the corresponding PSIs.

To index types, we extract all the relevant type information available using the LLVM API. The language- and architecture-independent LLVM type hierarchy found in the original bitcode is extracted and then stored directly into the program using a convenient format. Our data structures use a compact tree-like representation with leaf nodes representing primitive types. For example, similar to the original LLVM type system, an array of 10 `float*` pointers is represented by 3 distinct type nodes linked together: array (`[10 x float*]`), pointer (`float*`), and primitive (`float`), respectively. Only types referring to state entities are made part of the type hierarchy.

Global variables extracted from the program represent the first important state entity. Metadata information about global variables is stored in a number of state entries (or *s-entries*) made available at runtime. Each *s-entry* contains the name, type, and address of the variable. In addition, flags are used to mark constants and variables that have their address taken. Size and padding information for aggregate variables are computed at runtime (for portability reasons) and stored directly in the type hierarchy. This is important to efficiently support target-based and type-based PSIs.

A similar approach is adopted for functions, but only functions that have their address taken are made part of the embedded metadata information. This is possible since functions are only used to enforce target-based and type-based PSIs and not to introspect the state of the program.

4.3.2 Indexing pointer casts

To enforce type-based PSIs for pointer *s-elements* we need knowledge of pointer types that are allowed for each *s-element* at runtime. Unfortunately, type-unsafe languages like C allow for arbitrary casts between different types and recording metadata on static pointer types is not sufficient for a conservative analysis. To address this problem, our static analysis extracts all the pointer casts from the original program and enriches the type hierarchy with *casts-to* links between compatible type nodes. Luckily, LLVM explicitly represents both implicit and explicit pointer casts in the intermediate representation. The `bitcast` instruction is used to represent pointer-to-pointer casts, while the `inttoptr` and the `ptrtoint` instructions are used to handle integer-to-pointer casts and vice versa.

4.3.3 Indexing value sets

To enforce value-based PSIs it is necessary to store metadata for the set of legal values allowed for a given *s-element* at runtime. When possible, our static instrumentation annotates each type of the type hierarchy with the set of the legal values allowed. The value set is directly stored in a type node to simplify sharing and metadata management. Our current value analysis module supports basic value-set analysis (VSA) and can annotate both pointer and nonpointer *s-elements*. For pointer *s-elements*, we analyze `inttoptr` instructions and attempt to determine the set of all the legal integer values. When our conservative analysis fails, no value set is recorded but the pointer is marked as an *integer candidate*. Our simple analysis is, however, very often successful in real-world scenarios, where integer values usually refer to some special constants (e.g., `SIG_IGN` defined for the POSIX `sigaction` system call). For nonpointer *s-elements*, our current module records metadata for enums, constants, and variables assigned to constant expressions, but it would be straightforward to incorporate more sophisticated value analyses (e.g. range analysis).

4.3.4 Memory management instrumentation

To index dynamically allocated memory objects, we replace all the memory management functions in the program with our own wrappers to create and destroy metadata at runtime. Our current implementation supports all the POSIX `malloc`-like and `free`-like functions, including the `mmap` family, the `mem_align` family, and shared memory functions. Each memory allocation wrapper takes, along with the original arguments, an additional parameter that describes the run-time type of the to-be-created dynamic state entry (or *ds-entry*). Each *ds-entry* provides metadata for a dynamically allocated memory object represented as a typed array and contains similar information to the one included in a *s-entry*.

To determine the run-time type of a *ds-entry* correctly, we devised a conservative type inference algorithm for our static instrumentation component. Our algorithm recursively walks through all the possible uses of the value returned by each memory allocation function and considers all the global and local variables in the caller to which the value can be possibly assigned. When the run-time type can be determined unambiguously, the type node and all the relevant casts encountered are added to the type hierarchy. This approach allows introspecting dynamically allocated objects at the finest level of granularity possible at runtime, while dealing with the ambiguous cases in a conservative way by indexing all the pointer casts encountered.

Our type inference algorithm can also recognize and handle application-specific memory allocation wrappers (e.g., `my_malloc`) with arbitrary levels of nesting by analyzing the interprocedural propagation of weak pointer types (i.e., `void*`). In these cases, the algorithm is repeated recursively and the final run-time type propagated throughout all the wrappers encountered. When the type cannot be correctly determined, the instrumentation component resorts to the special `void` type used to describe a block of untyped memory, which, however, hampers the ability to introspect the memory object at runtime. In our current prototype, this scenario can occur in practice with programs relying on region-based memory management implementations [46]. To deal with such schemes and improve the coverage of our run-time analysis, our static instrumentation can be instructed to locate and automatically construct typed wrappers for region-based memory allocation/deallocation functions.

4.3.5 Metadata Framework

The metadata framework provides the data structures for all the metadata entities (i.e., *types*, *functions*, *s-entries*, and *ds-entries*) and the API to manage them at runtime. The metadata API provided by the framework offers 3 primary functionalities. First, API functions are available to introspect the entire program state. For example, a callback-based mechanism is used to process *s-entries* and *ds-entries* and conveniently operate on all the *s-elements* found therein. Second, a lookup API is available to locate any metadata entity given an appropriate search key. For example, the points-to lookup API—used to check target-based PSIs—locates a target *s-entry* given a valid pointer *s-element*. Finally, the framework provides an API to create and destroy metadata at runtime. This is used in the predefined memory management wrappers included in the framework.

To achieve good performance, our wrappers store the *ds-entries* using in-band metadata. Similar to prior approaches, we use canaries [248] to detect metadata corruption due to overflows, and optionally flip the top bit of every metadata word before and after use as suggested in [27] to mitigate dangling pointers. If necessary, strategies adopted by out-of-band memory allocators [27; 222] can be incorporated in our implementation to offer additional protection at the cost of more overhead. Note that the metadata canaries are continuously checked for consistency by the run-time analyzer. This eliminates the need to determine application points to check the canaries, which hampered the effectiveness of this approach in prior work, as evidenced in [222].

To achieve loose synchronization between memory wrappers executed in the application context and run-time analyzers executed on a separate monitoring thread, our design provides a lock-free interface for metadata management operations. This avoids any lock contention overhead and guarantees a scalable implementation. Our design arranges the *ds-entries* in a singly-linked list with newly-created *ds-entries* always added to the top. With the top of the list maintained stable, this strategy offers a lock-free stack interface to the memory allocation wrappers. A push-only lock-free stack can be implemented very efficiently using compare-and-swap (CAS) primitives, avoiding extra implementation complexities to ensure scalability in face of significant push and pop contention for the top of the stack or to deal with the "ABA problem" [139]. To maintain the top of the *ds-entry* list stable, the application threads are never allowed to perform a pop operation on the stack. This is accomplished by marking the state of the corresponding *ds-entry* as "*dead*" in the memory deallocation wrappers (e.g. `free`) without actually removing the *ds-entry* from the list. This allows metadata destroy operations to be completely lock-free, using lockless *ds-entry* state updates. The monitoring thread, in turn, periodically deallocates all the dead *ds-entries* and the corresponding data. To maintain the top of the list stable, a removal operation on the top is always deferred until the next *ds-entry* is pushed onto the stack. Multiple monitoring threads can concurrently operate and check PSIs on the same *ds-entry* list using lock-based synchronization (not exposed to the application threads).Note that this does not interfere with our memory wrappers and introduces no additional lock-contention overhead for the application threads.

4.3.6 Dynamic Instrumentation

The dynamic instrumentation component is an interposition library responsible for indexing deployed uninstrumented libraries at runtime to avoid spurious alerts in our invariants analysis. The component provides three key functionalities. First, the component creates *ds-entries* for dynamically linked libraries at program initialization time. This step is necessary to enforce target-based invariants in case of application pointers pointing to text or data regions created by the dynamic linker. To address this problem, we rely on the same data structures used by the dynamic linker to introspect the address space of the program and locate all the dynamic objects that belong to uninstrumented libraries.

Our current implementation is based on the ELF binary format. To introspect dynamic objects, it is sufficient to parse the ELF header to locate the GOT and the `link_map` data structure used by the linker. For each text region found, we extract all the symbols and create a single untyped *ds-entry* (i.e., a *ds-entry* with the special type `void`) for each library function. Note that we need to index all the uninstrumented library functions, since the knowledge of whether a function can have its address taken is irretrievably lost, and so are the original symbol types. For each data region found, in turn, we create a single untyped *ds-entry* describing the region.

The second important responsibility of the dynamic instrumentation component is to keep track of dynamically loaded shared libraries at runtime and create and destroy the corresponding *ds-entries* when necessary. For this purpose, the interposition library includes wrappers for the programming interface to the dynamic linking loader provided by POSIX. We use a `dlopen` wrapper to create or update *ds-entries*—POSIX defines a reference counter to reuse existing library mappings—and a `dlclose` wrapper to destroy existing *ds-entries* when the reference counter drops to zero.

For dynamically loaded libraries, our wrappers follow the same approach adopted to index dynamically linked libraries. New *ds-entries* are similarly created for the new memory regions, and each *ds-entry* is marked as text or data depending on the particular region type considered. Finally, the dynamic instrumentation component needs to handle all the memory management functions invoked at runtime from uninstrumented libraries. For this purpose, the interposition library includes dynamic wrappers for all the memory management functions supported by the metadata framework and redirects execution to the original static wrappers accordingly. Since the type information is lost for uninstrumented libraries, the run-time type provided to the original wrappers is always `void`. This explains why it is crucial to combine static and dynamic instrumentation to handle memory management functions. Dynamic instrumentation is necessary to create metadata for all the possible dynamic memory objects and avoid proliferation of spurious alerts. At the same time, dynamic instrumentation alone would produce only untyped *ds-entries*, making it harder to reason about dynamically allocated memory blocks, a common problem in prior work [27]. In our approach, untyped memory objects hamper state introspection—as for example expected for uninstrumented libraries—and decrease the accuracy of our target-based and type-based PSIs.

4.3.7 Run-time Analyzer

The run-time analyzer is responsible for sampling the program state periodically and checking PSIs to detect any violation. The initialization code prepares all the data structures used in the analysis and transparently allocates the monitoring thread on a predefined core. Depending on the configuration given, it is possible to allocate multiple monitoring threads on the same or different cores to increase the frequency PSIs are checked. In the other direction, it is also possible to reduce the frequency of monitoring cycles to reduce CPU utilization. This allows us to trade off security and CPU utilization when power consumption is of concern. If strict backward compatibility is not required, the application could also be slightly modified to start the analysis only in face of particular events, saving monitoring cycles when the application is idle. The analyzer runs the monitoring thread in an endless loop, although the analysis can be interrupted and restarted on demand, if necessary. At each cycle, the program state is sampled to check PSIs. The analyzer cycle comprises 5 (not necessarily sequential) phases: state introspection, invariants analysis, recording, reporting, and feedback generation.

4.3.8 State introspection

The analyzer locates all the indexed *s-entries* (and *ds-entries*) and recursively walks through all the *s-elements* found using the functions provided by the metadata framework. All the *s-elements* that have candidate PSIs are considered for analysis. Our default strategy analyzes all the relevant *s-elements* sequentially but, depending on the threat model considered, additional policies can be used to prioritize particular state regions (e.g., heap) and check corresponding PSIs at a higher frequency.

4.3.9 Invariants analysis

The analysis is performed for all the PSIs supported for any given *s-element*. First, the value of the *s-element* is atomically checked for value-based PSIs whenever a value set is available. If the value is not part of the value set, a violation is flagged.

We also analyze pointer *s-elements* that have been marked as *integer candidates* with no value set provided. In particular, if the pointer points anywhere in the set of reserved pages at the beginning or at the end of the address space, our analysis marks the pointer as safe. This strategy reflects the knowledge that pointers marked as *integer candidates* are typically assigned to special constants that do not reflect valid memory addresses. Although some corrupted pointer in this category may go undetected, when the pointer is dereferenced a fault will immediately be triggered. If an *integer candidate* points to an address outside the reserved range, the pointer is promoted to a regular pointer and further PSIs are normally checked for violations.

For all the pointer *s-elements* considered for further analysis, target-based PSIs are checked next. The metadata API is used to look up the *s-entry* or *ds-entry* each *s-element* points to. If no valid entry can be found or the entry refers to an object that does not have its address taken, a violation is flagged. Upon successful lookup, the target entry is recursively analyzed to determine the run-time type or types the pointer is pointing to. If the analysis fails, for example the pointer is illegitimately pointing to padding data of a `struct`, a violation is flagged. When valid target types are found, type-based PSIs are considered.

For type-based PSIs, we first check the static pointer type and determine whether it matches any of the run-time types the pointer is pointing to. When no match with the static pointer type is found, the analyzer examines the set of compatible pointer types retrieved from our *linked* type hierarchy. If no match is found, a PSI violation is flagged. When the target *s-entry* is untyped, the nature of the target is considered. If the original pointer is a data pointer and points to a *s-entry* referring to a text memory region (or vice versa), a violation is flagged.

4.3.10 Recording

The results of our analysis are recorded to collect fine-grained statistics on each *s-element* with annotated PSIs. For each *s-element*, we record the PSI violations found. For pointers, we also record the distribution of target types found (with the number of occurrences sampled for each type) and the corresponding memory regions (i.e., data, heap, mmap, shared memory, text).

4.3.11 Reporting

To be effective in different scenarios, RCORE supports policy-based detection mechanisms. Policies decide what events indicate suspicious behavior and need to be reported. RCORE supports two default detection mechanisms: *synchronous* detection and *window-based* detection. The synchronous detection mechanism simply logs all the PSI violations found. While useful in development mode, this mode of operation is not always desirable in production. Some short-lived PSI violations may be sometimes acceptable and expected in the normal execution of the program. For example, consider a pointer that is freed and then immediately set to `NULL`. The asynchronous analysis performed by the monitoring thread might sample the pointer value right after the `free` call. In this case, a dangling pointer would be immediately reported as dangerous although the pointer is dangling only for a very short period of time and never used. To address this issue and reduce the number of alerts in production, RCORE defaults to another (window-based) detection mechanism for dangling pointers. In this mode of operation a sliding window is used to collect a number of state samples over a time interval. The resulting distribution is used to enforce detection policies and log suspicious events on a per *s-element* basis. The size of the window is configurable, and so are the policies supported for each particular event. The default policy is to report only the PSI violations that occur for all the samples in a single detection window, but more sophisticated policies are possible. This simple policy is effective in real-world scenarios, allowing one to tune the number of alerts logged by simply varying the window size. Reasonably short detection windows avoid logging common dangling pointer violations and provide accurate detection for the suspicious cases.

4.3.12 Feedback generation

For each logged event, we generate accurate information on the PSIs violated and report all the statistics gathered on a per *s-element* basis. The detailed information provided in the feedback can help developers track and reproduce the original problem for debugging purposes. In addition, the feedback can be used to automatically classify the violations basing on the patterns observed. For example, a common pattern we have observed for the distribution of target types of a dangling pointer is `NULL`, `type x`, `target-based PSI violation`.

4.3.13 Debugging

RCORE includes a flexible debugging interface to support offline analysis of all the PSI violations found. Our asynchronous detection strategy provides a debug-friendly environment for offline runs, with the run-time analyzer imposing minimal disruption on the application threads and the static instrumentation preserving symbol table and stack information. To quickly locate and fix the original problem, developers can use the debugging interface to set arbitrary breakpoints and interrupt the program execution upon specific PSI violations (e.g., break on any PSI violation found for the pointer `my_ptr`). Debugging support reflects our goal of simplifying the entire vulnerability discovery and patch development-deployment lifecycle.

4.4 Memory Errors Detected

4.4.1 Dangling pointers

RCORE supports proactive detection of memory errors derived from dangling pointers, which are explicitly recognized using PSI violations and knowledge of known heap regions. Dangling pointers can be detected immediately, even before they are actually dereferenced. This is crucial for a dynamic vulnerability monitoring infrastructure. Common are cases where even extensive dynamic analysis fails to trigger corruption caused by a vulnerable dangling pointer. For instance, the pointer may be dereferenced only in code paths that are rarely triggered during normal execution. For this reason, RCORE uses window-based detection to report all the suspicious long-lived dangling pointers, which can then be further inspected offline. In addition, arbitrary corruption caused by incorrect use of dangling pointers can be detected by PSI violations on the corrupted target region.

4.4.2 Off-by-one pointers

RCORE supports proactive detection of off-by-one pointers, which are often legitimately used to mark buffer boundaries. If incorrectly used, however, they can introduce overflows or indirectly cause other memory errors. RCORE's target-based analysis can explicitly recognize generic off-by-N pointers even before they can do any harm. A policy determines whether our analysis should report (immediately or using window-based detection) or ignore these cases. Similar to dangling pointers, memory corruption caused by incorrect use of these pointers is still always reported by our invariants analysis.

4.4.3 Overflows/underflows

RCORE supports detection of buffer overflows (and underflows) using invariants analysis to detect the resulting memory corruption occurred. Note that, in contrast to existing source-level approaches, our fine-grained analysis allows RCORE to detect arbitrary overflows, even those for buffers inside a `struct` or buffers allocated using application-specific dynamic memory allocation. In most cases, it is very easy to classify a suspicious event as a buffer overflow or underflow, depending on the patterns observed. This is reflected by a number of PSI violations reported for contiguously allocated *s-elements*.

4.4.4 Double and invalid frees

Like other common memory allocators, RCORE detects most double and invalid frees directly in the memory management wrappers, using in-band metadata canaries. In the remaining cases, arbitrary memory corruption caused by the illegal operation can still be detected by PSI violations on the corrupted region.

4.4.5 Uninitialized reads

RCORE supports probabilistic detection of uninitialized reads. Our default strategy is to start checking PSIs for *s-elements* described by a given *ds-entry* as soon as the *ds-entry* is created. Dynamically allocated *s-elements*, however, may not have been initialized yet when the analysis starts and the random garbage contained therein would likely trigger PSI violations. To address this issue, we allow a configurable grace period before introspecting new *ds-entries* in the analysis. This strategy follows the intuition that new *s-elements* that are left uninitialized for too long increase the probability of uninitialized reads and should therefore be considered for offline inspection. This strategy is effective even for reasonably short grace periods. As in other cases, memory corruption indirectly caused by uninitialized reads—for example dereferencing an uninitialized pointer and write data to an arbitrary memory region—can be detected by PSI violations on the corrupted region.

4.5 Evaluation

The current RCORE implementation runs on Linux, but most components can be easily ported to other operating systems and binary formats other than ELF. Our compiler driver is implemented in python in 1200 lines of code (LOC). The static

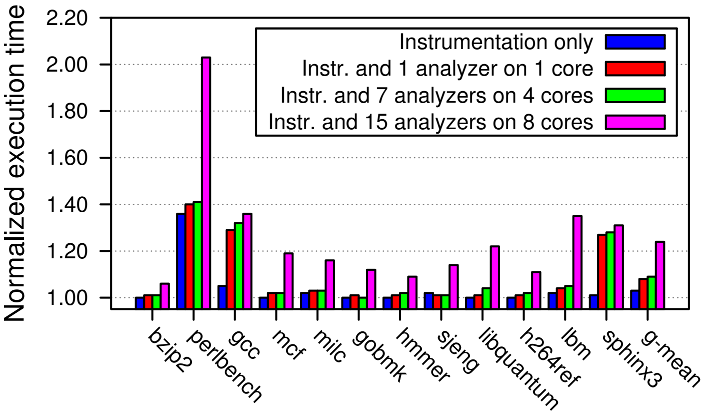


Figure 4.2: Run-time overhead introduced by RCORE for the SPEC CPU2006 benchmarks.

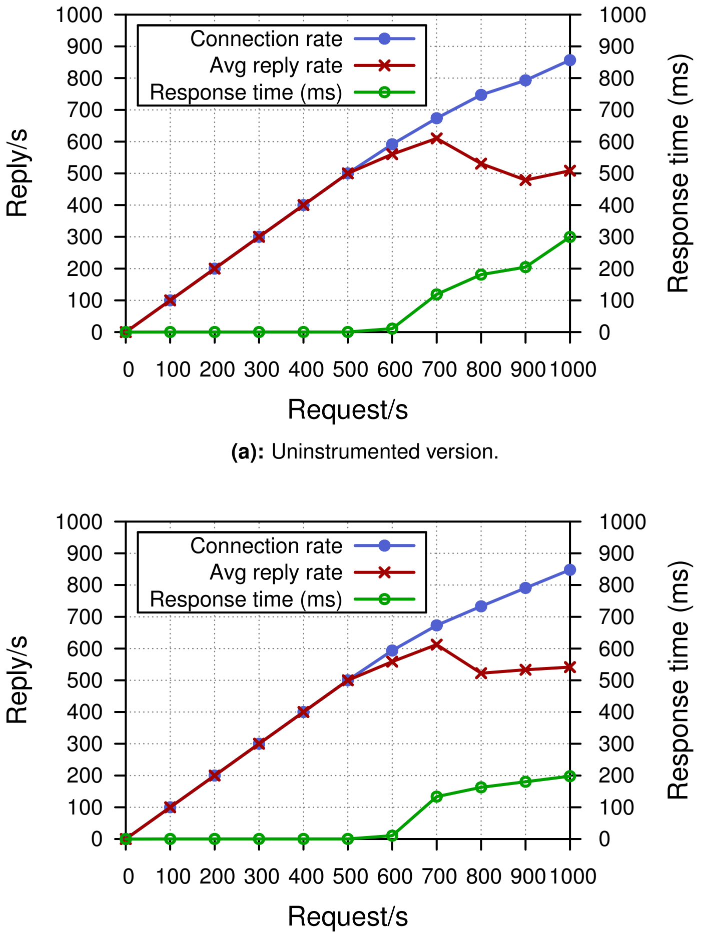
instrumentation component is implemented as an LLVM pass in 6000 LOC, and supports all the standard LLVM optimizations. The metadata framework is implemented as a static library written in C in 3700 LOC. The dynamic instrumentation component and the run-time analyzer are implemented as shared libraries written in C in 800 LOC and 2300 LOC, respectively. The libraries are preloaded using platform-specific support offered by the dynamic linker (e.g., the `LD_PRELOAD` UNIX environment variable) to override the default run-time program behavior.

4.5.1 Performance

We evaluated the overhead of our solution using the C programs in the SPEC CPU2006 benchmarks. We ran our experiments on a Dell Precision workstation with two 2.27GHz Intel Xeon E5520 quad-core processors and 4GB of RAM running a 2.6.35 Linux kernel. Each core has two hyper-threads sharing the L1 and L2 cache, whereas the four cores on the same die share an 8-MB L3 cache.

We executed each experiment 11 times and reported the median. We evaluated both the overhead introduced by our static and dynamic instrumentation and the one introduced by these components and the RCORE run-time analyzer allocated on dedicated cores. Figure 4.2 shows the execution time of the RCORE version of our benchmarks normalized against the baseline.

The static and dynamic instrumentation components of RCORE introduce 3% run-time overhead on average (geometric mean). The whole framework in its default configuration (one run-time analyzer) introduces 8% overhead on average. The average, however, is heavily influenced by benchmarks like `perlbench` due to the massive use of dynamic memory allocations, which inevitably results in high memory management instrumentation overhead and significantly increased contention



(a): Uninstrumented version.

(b): RCORE version.

Figure 4.3: Performance results for the `nginx` benchmark.

for memory bandwidth between application and monitoring threads. Encouragingly, for the majority of the benchmarks RCORE introduces a negligible overhead with a median value of only 1.5%.

The last two bars in each benchmark show the overhead imposed by RCORE when configured with 7 and 15 run-time analyzers assigned to 4 and 8 independent cores with hyper-threading (9% and 24% on average, respectively). The significantly higher overhead introduced in the latter scenario acknowledges the impact of the increased contention for memory bandwidth caused by multiple monitoring threads scheduled on different dies with no cache shared. Our results confirm the importance of a shared cache to achieve good performance in concurrent dynamic monitoring applications, as also recognized in prior work [126].

We now compare our SPEC results with WIT [28] and Cruiser [296], two recent low-overhead solutions to detect memory errors. RCORE reports lower overheads than WIT on average, which shows an average overhead of 10% on SPEC CPU2000 benchmarks. WIT's object-level analysis for memory writes is also more coarse-grained than ours, although WIT's run-time checks follow the main application flow and are thus less probabilistic than RCORE's asynchronous detection model. RCORE reports lower average overhead than Cruiser, which shows an average overhead of 12.5% on SPEC CPU2006 Integer benchmarks in its lazy version. The overhead drops to 5% for Eager Cruiser, which, however, requires recovery techniques and may incur false positives. Cruiser's detection model is asynchronous like ours, but focuses only on heap-based buffer overflows.

Our second set of experiments evaluated the throughput and latency degradations introduced by RCORE on `nginx` [19] (version 0.8.54) and `lighttpd` [17] (version 1.4.28), two popular web servers. The web servers were independently deployed on the same Dell Precision workstation used above. A number of clients were deployed on different Dell workstations with a 3.33GHz Intel Core 2 Duo CPU and 4GB of RAM, each running a 32-bit 2.6.35 Linux kernel and connected to the servers through a Gbit link.

Figure 4.3 shows the average throughput and latency of `nginx` under different workloads (i.e., requests per second) while retrieving a 50KB file. In particular, we started with a rate of 100 req/s up to 1000 req/s, increasing the request rate by 100 req/s on each subsequent run. Each request opened a connection to download the requested file. Each run lasted for at least 75 seconds and issued as many connections as needed to match the request rate for the duration of the whole run considered. This allowed `httperf` [16], our web benchmarking tool, to collect enough evidence (i.e., samples) needed to produce statistically sound results. Figure 4.3(a) refers to tests performed on an unmodified version of `nginx` and represents our baseline. Figure 4.3(b), in contrast, refers to tests performed on the RCORE (1 run-time analyzer on a dedicated core) version of `nginx`. Figure 4.3(a) shows that the baseline achieved the maximum throughput at around 500 requests per second, at which time `httperf` issued 37500 connections in total. As we further increased the request rate, the server became saturated, as shown by the gradual throughput and latency degradation. We have checked that the server was the bottleneck by performing a number of additional experiments to verify that all the clients could keep up with that maximum request rate. Similarly, Figure 4.3(b) shows that the RCORE version of `nginx` was able to match the same maximum rate and response time of the baseline, introducing only a negligible overhead. We have performed the same set of experiments on `lighttpd` and under different workload scenarios and obtained similar results with negligible overhead. We omit the figures of such experiments due to lack of space. The overall results here outlined are encouraging and show that our approach introduces negligible overhead on the end performance of the RCORE version of the program operating at full capacity. This enables a practical and realistic deployment of our solution in production systems.

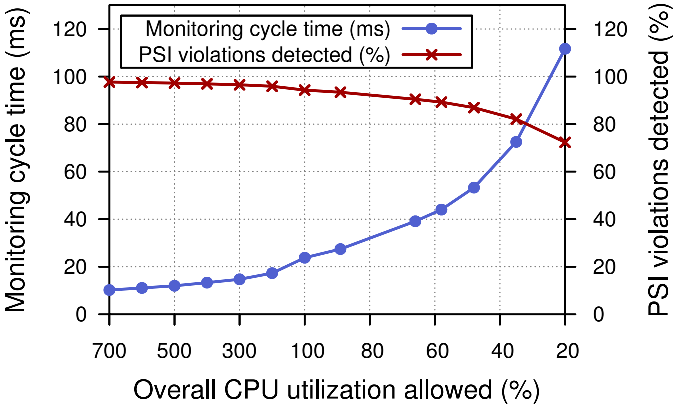


Figure 4.4: Monitoring cycle time and PSI violations in `nginx` for decreasing CPU utilization.

4.5.2 Detection Accuracy

Decoupling security checks from the main application flow guarantees low overhead but inevitably introduces a latency in the detection of PSI violations. The latency depends on the monitoring frequency and the number of run-time analyzers used. Lower detection latencies are desirable for better accuracy. Higher detection latencies, on the other hand, reduce power consumption. The appropriate tradeoff can be tuned for each particular scenario considered.

Figure 4.4 depicts the monitoring cycle time achieved by RCORE and the resulting number of PSI violations detected when injecting into the program 100 memory corruptions with a lifetime uniformly distributed in [1, 200]ms. The values are plotted as a function of the overall CPU utilization allowed to the run-time analyzers, with one or more dedicated hyper-threads hosting a single analyzer each. At 100% CPU utilization, the default configuration (1 run-time analyzer on 1 thread) completes a monitoring cycle and checks the PSIs of a given *s-element* every 24ms. Conversely, the time elapsed between any two checks with 1 run-time analyzer at 20% CPU utilization is around 110ms. The configuration with 7 analyzers allocated on 7 threads sharing the L3 cache (700% CPU utilization) with the application achieves the lowest detection latency of 10ms. The resulting percentage of PSI violations detected in these 3 configurations is 94%, 72%, and 98%, respectively. These results have been obtained for `nginx`, but similar behavior can be observed for other programs, with the detection latency dependent on the number and the complexity of the data structures used. When compared to the simpler canary-based detection strategy used in Cruiser [296], our analysis incurs higher detection latencies, but encompasses many more memory errors than only heap-based buffer overflows. To further improve RCORE detection latency, we can increase the number of monitoring threads or instruct the run-time analyzers to focus on particular portions of the program state, for example, on those that are observed to change more often.

4.5.3 Effectiveness

To evaluate the effectiveness of RCORE in detecting memory errors using our invariants analysis, we performed two complementary experiments: (1) a feedback evaluation, which measured the accuracy achieved by RCORE during testing runs of `proftpd` version 1.3.3e and `exim` version 4.69, two well-known FTP and SMTP servers; (2) a CVE (Common Vulnerabilities and Exposures) evaluation, which assessed the ability of RCORE to detect representative memory error vulnerabilities related to `nginx` and `openssl`. Our ultimate goal is to evaluate RCORE's effectiveness at detecting real-world memory errors as PSI violations and providing useful feedback to pinpoint the original problem.

The feedback evaluation performed on `exim` allowed us to find a previously unknown, potentially exploitable, vulnerability. In particular, the vulnerability is represented by an out-of-bounds pointer, `mainlog_datestamp` (`log.c`), that is dereferenced for reading. This may potentially lead to a denial of service situation, such as, process crash or file resource exhaustion. Conversely, `proftpd`'s feedback evaluation reported an interesting long-lived dangling pointer, `capabilities` (`mod_cap.c`). Even a careful code inspection was insufficient to assess whether this dangling pointer could lead to a vulnerability, but the code should probably be better restructured to avoid problems.

We then proceeded to assess whether RCORE is able to detect real-world vulnerabilities. To this end, we selected the CVE advisory 2009-2629, which describes a buffer underflow vulnerability that affects several versions of `nginx`, including those from 0.6.x before 0.6.39, among others (our analysis was performed on `nginx` version 0.6.38), and the CVE advisory 2010-2939, which describes a double-free vulnerability affecting `openssl` version 1.0.0a.

The CVE 2009-2629 advisory states that a specially crafted HTTP request may produce memory corruption enabling the execution of arbitrary code. We selected this CVE because it was a particularly representative case of global state corruption introduced by a typical memory underflow vulnerability. In addition, `nginx` relies heavily on application-specific memory management and uses many `struct` types with buffer variables; all elements that make the detection of memory corruption hard in the general case. The execution flow that triggers the vulnerability starts with `nginx` invoking `ngx_http_init_request()` when processing network input. This function allocates a 1024-byte pool using application-specific memory allocation functions and fills the pool with a number of data structures containing several nested pointers and the parsed input $\mathcal{I}$ at the end (e.g., `ngx_table_elt_t`). The underflow causes a temporary pointer, initially pointing to $\mathcal{I}$, to traverse back up to the next '/' character encountered (`ngx_http_parse_complex_uri`). Depending on the particular memory layout at the moment of the underflow, the temporary pointer may land within the same `struct` that included the original input buffer (e.g., `ngx_str_t`), within the same pool-dedicated block, or on a different memory block. Depending on the input provided, the pointer is then used to write garbage that overrides a number of consecutive *s-elements*. In our multiple experiments with different input distributions, the observed memory corruption repeatedly triggered several PSI violations, given the significant number of pointers corrupted. As a result, RCORE was able to detect the corruption in all our tests, no matter where the temporary pointer initially landed. Existing approaches would have failed to detect the corruption in the general case. We were also positively impressed by the accuracy of our invariants analysis. In particular, our type-based invariants were extremely accurate in detecting type violations for all the corrupted function pointers, given the small fraction of *s-elements* referring to the same given function type.

Conversely, the CVE 2010-2939 advisory states that a specially crafted private key may allow context-dependent attackers to execute arbitrary code due to a double-free vulnerability in the function `ssl3_get_key_exchange` of `openssl` version 1.0.0a. In our experiments, RCORE repeatedly detected the vulnerability in the memory management wrappers using in-band metadata canaries. To simulate the scenario of a new valid memory block overriding the memory location of the original canary with a legal canary value, we disabled all our checks in the memory management wrappers and only checked for PSI violations instead. When the allocator happened to allocate a new memory block in the same memory region as the old block's metadata, the unchecked double free corrupted arbitrary data in the new block. Similarly to what was observed for `nginx`, our experiments in this scenario promptly reported type- and target-based PSI violations on the corrupted data.

4.6 Limitations

RCORE is primarily targeted at reporting on known or unknown vulnerabilities during normal in-the-field execution. Due to the probabilistic nature of our asynchronous detection model (crucial to achieve low overhead), however, we can make no claim that RCORE can identify *all* the short-lived vulnerabilities or attacks that affect the global program state. Attacks, in particular, can only be detected under the following conditions. First, memory corruption induced by the attacker must trigger some PSI violations. This is often the case when the attacker cannot make strong assumptions on the layout of the corrupted region. In this scenario, RCORE is very likely to identify PSI violations, especially for pointers corrupted with arbitrary data. On the other hand, even if the attacker can reliably craft a request that produces no PSI violation, his exploitation power is clearly reduced. For example, our PSI analysis would only allow an attacker to corrupt a function pointer with the address of a function of the same type.

Second, the lifetime of the corruption introduced should be no shorter than a monitoring cycle. To evade detection, the attacker may be able to execute arbitrary code shortly after corrupting critical data and quickly perform recovery actions. In our experience, while generally practical for stack smashing and other short-lived attacks, this strategy cannot be taken for granted for attacks that exploit global state corruption. Our claim is also supported by other similar (but less generic) asynchronous detection models which have been successfully applied to heap-spraying [244] and heap-based buffer overflow [296] attacks.

In our future work, we intend to improve our attack detection accuracy by reducing RCORE's monitoring cycle and further investigate the different security-performance tradeoffs, e.g., by switching to a more deterministic "stop-the-world" detection model under particular conditions.

4.7 Related Work

Memory errors represent a major category of vulnerabilities and have received much attention in recent years. Bounds checking is a largely explored solution devised to address common memory errors in C programs, but traditional approaches [251; 88; 73] suffer from significant overhead. More recent bounds checkers have used efficient checks and static analysis [29; 293] to achieve better performance but are still unsuitable for large-scale adoption. More widespread adoption has been gained by StackGuard [77], which uses "canaries" before the return address of a function to detect buffer-overflow errors. Other techniques [70; 292] have used a shadow stack to separate the return address and other sensitive data from buffer variables that are subject to overflow. More recent approaches, in contrast, are specific to heap-based memory errors. Some suggest a particular memory allocator design [27; 222; 45], others use canaries to detect heap-based overflows [248; 278]. ValueGuard [278] instruments the original code to add canaries for both global and local variables. LibsafePlus [38] detects overflows that occur in particular unsafe C library functions (e.g., `strcpy`). This is done by analyzing debugging information and instrumenting the code to describe ranges for local, global, and dynamically allocated buffers. The metadata collected, however, is coarse grained and only used to perform range checking. In a similar direction, MEDS [146] uses a basic low-level type system to perform run-time detection of memory errors, but requires software dynamic translation incurring extremely high overhead. Like ours, other approaches have used the general idea of enforcing static analysis results at runtime. Control-flow integrity [21] computes the program control-flow graph and prevents deviations from it at runtime. Similarly, Castro et al. [65] present an approach to enforce data-flow integrity at runtime using a precomputed data-flow graph.

WIT [28] is a low-overhead solution that uses static analysis to determine the set of objects that can be written by each instruction in the program and instruments the code to enforce write integrity at runtime. Albeit static, their points-to analysis presents similarities with our target-based PSI analysis. Their checks, however, are always performed at the object level and subject to the precision of static analysis to identify accurate object sets. In contrast, our invariants analysis is fine-grained and generalizes their approach with generic program invariants. The approach we propose is more radical. Our static analysis extracts as much information as possible from the program and enforces all the PSIs found at runtime. In addition, WIT cannot support out-of-bound reads without incurring additional overhead.

None of the approaches examined is general and fine-grained enough to support several classes of memory errors, with low overhead. Our PSI analysis can be used to detect arbitrary memory corruption, even when the source of the corruption is unknown. For example, RCORE can also detect hardware memory errors when the resulting corruption leads to PSI violations. Moreover, we support fine-grained analysis of both static and dynamically allocated objects, including introspection of `structs` and objects managed by custom memory allocators. None of the approaches considered can support either. Finally, while RCORE was designed to operate in a fully asynchronous fashion, we believe our invariants analysis can be used in different contexts as a generic state checking mechanism, as also demonstrated by our prior work in the context of live update [113].

We conclude by briefly surveying a number of relevant multicore security applications. He et al. [137] describe dynamic multicore-based program monitoring and compare the performance of their compiler optimizations with instrumentation-based monitoring. Ruwase et al. [252] show how to efficiently parallelize dynamic information flow tracking with several threads running on different cores. Aftandilian et al. [23] propose asynchronous assertions to inexpensively evaluate heavyweight programmer-provided checks concurrently to the execution of the program.

Other approaches [254; 253] explore parallel execution of program variants to detect attacks from divergent behavior. Finally, Cruiser [296] is a low-overhead solution for concurrent heap buffer overflow monitoring. Their work is similar in spirit to ours, but they focus only on a particular class of memory errors using a detection mechanism based on canaries. Our invariants analysis, in contrast, is much more general and targeted toward several classes of memory errors. Their implementation, however, includes a very efficient lock-free dynamic memory allocator that maintains out-of-band metadata, which could also be incorporated in RCORE if using out-of-band metadata is required.

4.8 Conclusion

Current approaches that aim to detect memory error vulnerabilities are either specific to particular categories of memory errors or incur significant overhead, which hinders their widespread adoption in vulnerability monitoring scenarios in production. Despite claiming backward compatibility, prior solutions make also strong assumptions on the nature of the program under analysis, for example that no application-specific memory management is used.

In this paper, we presented RCORE, a low-overhead dynamic program monitoring infrastructure that can leverage available cores to continuously inspect running programs and report on a broad class of memory errors. RCORE uses extensive static analysis to extract as many PSIs as possible from a given program and make them available at runtime for fine-grained invariants analysis. RCORE covers all the standard C features and explicitly supports application-specific memory management not to lower the accuracy of the results at invariants checking time. Our investigation demonstrates that common memory errors can all be mapped to PSI violations and classified to provide an informative feedback to the developers. Our invariants analysis can be used to detect both dangerous behavior (e.g., long-lived dangling pointers) and memory corruption. In the latter case, our dynamic analysis concentrates on the effect of the corruption rather than on the cause, enabling probabilistic detection of arbitrary memory errors, even when the cause is unknown or not directly controlled. As a result, RCORE can seamlessly detect memory corruptions in the program state caused by the libraries or by arbitrary hardware memory errors.

4.9 Acknowledgments

We would like to thank the anonymous reviewers for their comments. This work has been supported by European Research Council under ERC Advanced Grant 227874.

5

EDFI: A Dependable Fault Injection Tool for Dependability Benchmarking Experiments

Abstract

Fault injection is a pivotal technique in dependability benchmarking. Unfortunately, existing general-purpose fault injection tools either inject faults in predetermined memory locations or resort to random injection, approaches that generally result in poor fault coverage and controllability guarantees. This makes it difficult to reproduce or compare experiments across different systems or workloads.

This paper presents EDFI, a new tool for *dependable* general-purpose fault injection experiments. EDFI combines static and dynamic program instrumentation to perform *execution-driven fault injection*, a technique which allows realistic software faults to be injected in a controlled way as the target system executes. Our instrumentation strategy guarantees a predetermined faultload distribution during the entirety of the experiment, independently of the particular system or workload considered. Our evaluation confirms that EDFI significantly improves the precision and controllability of prior tools, at the cost of only modest memory and performance overhead during fault-free execution.

5.1 Introduction

As we enter the pervasive computing era, complex software systems play an increasingly important role in our everyday life. In this emerging landscape, assessing the dependability properties of a software system becomes a growing concern. For dependability benchmarking purposes, researchers have traditionally relied on software-implemented fault injection (SWIFI) tools, which provide a relatively inexpensive strategy to mimic real-world faults in a synthetic experimental setting.

In the past decades, fault injection campaigns have been applied to several classes of software, including distributed systems [165], user applications [40; 198; 196], operating systems [119; 172], device drivers [272; 299; 141], and file caches [219]. These experiments have served a number of different purposes, including: (i) analyzing the behavior of different systems under a given faultload [119; 172]; (ii) evaluating the effectiveness of fault-tolerance mechanisms [272; 299; 141]; (iii) performing high-coverage testing of error detection and recovery mechanisms [40; 163; 120].

Recent research efforts to build practical fault injection tools are largely focused on the latter scenario [198; 196; 163]. In this context, experiments are designed to surgically inject targeted faults into the system and trigger rarely executed code paths, rather than mimicking real-world software faults. The ultimate goal is typically to increase the code coverage explored during automated testing. Ironically, earlier efforts, which are instead focused on designing general-purpose fault injection tools, are, in turn, heavily affected by limited program code coverage achieved during the experiment.

In detail, the dominant approach followed by existing general-purpose tools is to inject faults into predetermined (or random) memory locations [164; 41; 299; 219; 94; 275; 64; 165], a *location-based* strategy which cannot guarantee that faults are actually "covered"—with a predetermined *faultload distribution*, i.e., characterization of fault locations and types (§5.5)—at runtime. Not surprisingly, prior studies have reported fault injection campaigns with no faults activated in as many as 40% of the experiments [219; 165]. A way to address this problem is to substantially increase the number of faults injected, but at the cost of more experiments invalidated by spurious faults activated before even starting the test workload. An alternative is to surgically inject faults into hot spots stressed by the test workload [276; 161; 160], a strategy which, however, requires a deterministic workload and does not account for code covered only during faulty execution (e.g., error handling code).

Other approaches, in turn, periodically interrupt the system at random execution points (i.e., typically using a timer) and inject faults into the current runtime context [275; 164; 64]. This *time-based* strategy, however, biases the experiment toward hot code paths and severely limits the nature of the faults that can actually be injected at runtime, ultimately producing poorly representative software faults [155; 191; 74]. We believe all these shortcomings have significantly affected the "*dependability*" of existing tools, often even prompting researchers to question the validity of fault injection as a dependability benchmarking technique [171].

This paper presents EDFI, a new tool for *dependable* general-purpose fault injection experiments. Unlike all the prior tools, EDFI implements *execution-driven fault injection*, a novel technique which allows injecting a controlled and predetermined faultload distribution as the system executes at runtime. To address this challenge, EDFI relies on a combination of static and dynamic instrumentation, which transforms the original code into *multiple heterogeneous versions* at compile time and provides the ability to interleave them in a controlled way during the experiment. This strategy allows EDFI to (i) statically inject multiple *simultaneous* [284] faults over the entire code—with a predetermined faultload distribution—to avoid coverage problems during the experiment and (ii) seamlessly switch between fault-free and faulty execution to allow fault activation only in a user-controlled fault injection window at runtime.

EDFI's hybrid instrumentation strategy delivers *precise* (i.e., how well the tool follows the original fault model), *controllable* (i.e., how well the user can mark the beginning/end of an experiment with no spurious faults activated before/after then), and *observable* (i.e., how well the user can observe the output of an experiment) fault injection experiments with negligible system impact during normal execution. Unlike all the existing tools, EDFI offers strong guarantees that a predetermined fault characterization given in input (i.e., *input faultload distribution*) will be precisely reflected in the observed output of the experiment (i.e., *output faultload distribution*) without introducing spurious faults that may compromise the validity of the results.

5.2 Background

Software-implemented fault injection (*SWIFI*) is a well-established technique in dependability benchmarking experiments. Its merit lies in emulating real-world faults in a synthetic setting with relatively simple tools.

SWIFI tools are typically designed to either inject generic faults into the original program or emulate error conditions at the library interfaces. The latter scenario is popular in robustness testing campaigns, which aim to analyze the behavior of the system in face of error codes returned by the libraries [198; 196] or invalid arguments supplied to library (or system) calls [172]. While important in robustness testing applications, these strategies are orthogonal to general-purpose fault injection techniques in terms of both goals and representativeness, as also demonstrated in prior work [205].

General-purpose SWIFI tools, in turn, are traditionally classified into two main categories, depending on whether fault injection is performed at *preruntime* or at *runtime*. The former approach injects faults by statically mutating the original program via compiler-based techniques [151; 299] or binary rewriting [164; 41; 219; 94]. Mutations can affect code or data and follow a predetermined *location-based* strategy. Locations are either user-defined or selected at random. Early approaches, such as [164; 41], corrupt the program image with hardware-like faults (e.g., bit flips). More recent approaches, such as [219; 299; 191; 94; 211], in contrast, explicitly aim at emulating realistic software faults. G-SWFIT [94], for example, injects only fault types obtained from the analysis of 668 real-world bugs found in the field.

In both cases, preruntime location-based approaches have a number of important shortcomings. First, fault activation cannot be easily guaranteed, as it is subject to code coverage induced by the test workload. Even when faults are activated, limited coverage immediately translates to very weak guarantees on the dynamic faultload distribution actually injected at runtime. Second, it is infeasible to prevent faults from being activated outside the user-controlled fault injection window, which should, however, clearly mark the boundaries of the experiment. This greatly limits the controllability of the approach. For example, the system may inadvertently crash at initialization time before even starting the test workload considered in the experiment.

Runtime location-based SWIFI strategies, such as those explored in [275; 164; 64; 165], seek to address the controllability issues of preruntime techniques. These strategies rely on hardware or software traps to interrupt the execution at predetermined (or random) memory locations and inject faults. This approach, however, is still inherently prone to the coverage problems discussed earlier. In addition, prior studies have shown that the low-level nature of these (and other) runtime techniques offers poor representativeness guarantees when emulating realistic software faults [191].

Other runtime SWIFI strategies, such as those explored in [275; 164; 64], have resorted to *time-based* fault triggers to periodically interrupt the execution (e.g., every 2 seconds) and inject faults into the current runtime context. While a potential solution to the coverage problems that plague all the other fault injection approaches, this strategy is hardly free from important shortcomings. First, the injection is heavily influenced by the workload and biased toward code paths that are executed more often during the experiment. Second, given that interruptions occur at random execution points, the nature of the faults that can effectively be injected is significantly constrained. This typically results in a weak and poorly predictable faultload distribution injected into the program at runtime. Not surprisingly, prior studies have found time-based approaches to be the least representative fault injection strategies [155]. Finally, the unpredictability of the injection events makes it really difficult to reproduce and compare the results across different experiments.

To conclude, a number of approaches have been devised to mitigate the coverage problems incurred by location-based techniques. The general idea is to profile the behavior of the system under the test workload and inject faults into hot spots with high probability of fault activation [276; 161; 160]. The main problem with these approaches is the inability to account for code paths only covered during faulty execution and the high sensitiveness to the workload. The latter, in particular, results in weak fault activation guarantees and also limits the reproducibility and comparability of the experiments. These issues have often emerged in prior studies. For example, the analysis presented in [161] assumes a deterministic test workload to

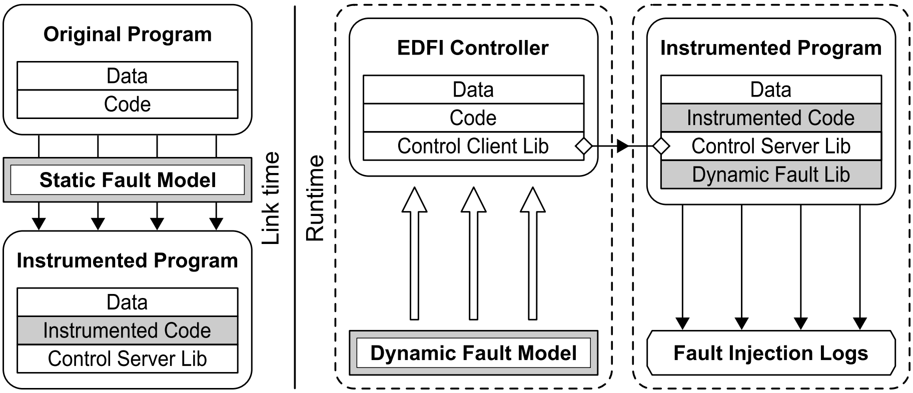


Figure 5.1: Architecture of the EDFI fault injector.

obtain stable experimental results. DEFINE [165] reports no fault activation in as many as 40% of the experiments even with faults explicitly designed to match the test workload. Finally, the analysis presented in [155] reports a high-variance faultload distribution observed across repeated fault injection experiments with only slight variations in the workload.

In contrast to all the prior SWIFI strategies, EDFI's hybrid instrumentation approach provides a *dependable* fault injection environment, combining and outperforming existing preruntime and runtime approaches. Unlike traditional preruntime strategies, EDFI provides full controllability of the experiment, with faults only activated (and observed) within a user-controlled fault injection window. Unlike all the location-based strategies, EDFI is robust to limited coverage induced by the test workload. Faults are injected (and activated) directly into the currently executed code paths, independently of the particular system or workload considered. Unlike time-based strategies, EDFI imposes no restrictions on fault representativeness, nor does it yield a biased or poorly predictable fault injection experiment. Overall, EDFI's execution-driven strategy offers much stronger guarantees on the precision of the dynamic faultload, naturally yielding more reproducible and comparable fault injection results. Its LLVM-based architecture, in turn, provides a powerful, extensible, and portable framework suitable for several fault injection scenarios.

5.3 System Overview

The goal of EDFI is to provide a generic and extensible fault injection framework which dependability researchers and practitioners can easily adapt to their needs in many different contexts and usage scenarios. This vision is reflected in EDFI's modular architecture (Figure 5.1).

To use EDFI, users need to statically instrument the target program with a link-time transformation pass, implemented using the LLVM compiler framework [179].

The pass accepts several command-line arguments to allow the user to specify the *static fault model* (§5.5), which describes the input faultload distribution to inject into the program, for example, a distribution mimicking fault types found in the field [94] and locations that are representative of residual faults [210; 211]. The pass translates the static fault model requested into targeted code mutations and prepares the program for *execution-driven fault injection* (§5.4). The transformations are all performed at the LLVM IR (*intermediate representation*) level before optimizations are applied. This strategy preserves the fundamental source-level abstractions required to inject realistic and representative faults [74]. In addition, the LLVM IR-level strategy seamlessly provides fault injection capabilities for all the architectures supported by LLVM.

If the source code is not available, our fault injection strategy could also be applied starting from legacy binaries, for example using recently proposed techniques to translate generic binaries into LLVM IR [33]. Great care should, however, be taken when using this strategy, given that the resulting LLVM IR would no longer reflect the structure and the abstractions of the original source code. This issue has been also recognized in prior studies, which demonstrated the representativeness problems of binary-level fault injection [74]. In particular, the analysis in [74] found inlining and C-style preprocessor macro expansion to be the most disrupting factors for fault injection representativeness.

To avoid the representativeness problems introduced by inlined functions, our instrumentation strategy ensures that program mutations are always applied before inlining (or any other optimizations). Preprocessor macro expansion, however, is always performed in the language front end, with the original macro information irremediably lost in the LLVM IR. In its current implementation, EDFI opts for a pure LLVM IR-based strategy, losing the ability to identify the original preprocessor macros, but at the benefit of a uniform instrumentation strategy across all the source languages supported by LLVM. If macro-level representativeness is an issue in particular scenarios, a simple source-to-source transformation could be used to automatically transform function-like macros into inline functions. Recent work on source code rejuvenation demonstrates how to implement this strategy in C++11 using *perfect forwarding* [176].

A similar warning is in order for shared libraries. Our link-time transformation pass can automatically instrument the program and all the statically linked libraries. Shared libraries, however, must be separately instrumented. Nevertheless, EDFI can automatically corrupt the arguments supplied to library calls or emulate error codes returned by shared libraries, similar to the library-level strategy adopted by LFI [198].

Once instrumented, the binaries can run without deviating from their original runtime behavior. The instrumentation, however, allows the user to initiate and terminate a fault injection experiment at any point during the execution of the target system. To control the experiment at runtime, EDFI relies on two *control libraries*, which together coordinate the communication between the system and an external *controller*. The *control server library*—which listens for external fault injection events in the background—is transparently linked against the program binary as part of our instrumentation process. The *control client library*—which delivers fault injection events to the server—provides a generic client-side interface to initialize, start, and stop a fault injection experiment on demand. EDFI already includes a simple stock *controller* (i.e., `edfi-ctl`), which relies on the control client library to expose a convenient command-line interface to the user. The client library, however, can be also as easily embedded in other complex systems to build more sophisticated controllers. Note that the control libraries are the only platform-specific components in our architecture, also designed to be easily extended and support new fault injection settings. Our current implementation includes support for UNIX applications, using UNIX domain sockets to establish the client-server communication. A portable `sysctl`-based implementation to perform fault injection into Linux and BSD OS kernels is underway.

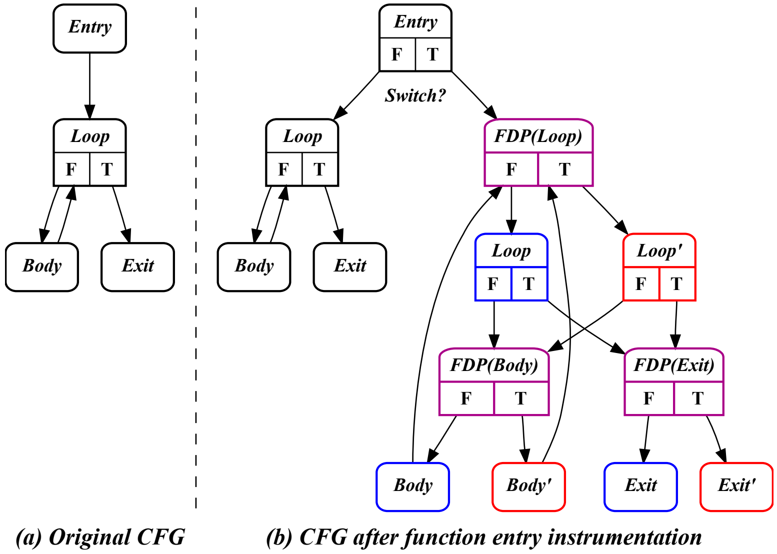
To initiate an experiment, the user typically starts the target program, activates a test workload, and finally instructs the controller to start (and later stop) the fault injection experiment in well-known system states. To configure the experiment, the user can specify a number of parameters and a custom *dynamic fault library* (or use the stock library included in our framework), dynamically loaded into the program immediately before starting the experiment. The input to the controller defines the *dynamic fault model* (§5.6) adopted, which gives the user fine-grained control over the experiment and the ability to emulate special dynamic conditions at runtime.

The user can also specify the logging mechanism to use among those supported by the control libraries. At the end of the entire process, the user can inspect the logs to determine the number, locations, types, and faultload distribution of all the faults injected during the fault injection experiment.

5.4 Execution-driven Fault Injection

Execution-driven fault injection is a new fault injection technique which ensures predetermined faults to be systematically injected, activated, and observed as the system executes at runtime. This strategy entails several challenges. First, the faults injected during the experiment should accurately follow the faultload distribution defined by the static fault model. Second, it should be possible to seamlessly switch between faulty and fault-free execution during an experiment, as dictated by the dynamic fault model. Finally, the switching strategy should guarantee fine-grained control over the execution during the experiment, but also minimize the impact on the system during normal execution. This property is particularly important to avoid perturbing the system before initiating the experiment.

To address these challenges, our instrumentation uses the *basic block cloning* idea, which replicates and transforms the original code into multiple heterogeneous and interchangeable code versions. The general idea has been explored in prior work



(a) Original CFG *(b) CFG after function entry instrumentation*

Figure 5.2: Basic block cloning example.

in different forms, using either static [286] or dynamic [239] program instrumentation strategies. EDFI embraces a new static approach to implement an efficient and flexible cloning strategy. Our transformation pass translates a generic basic block in the original control flow graph (CFG) of the program into the following basic blocks:

- **The *pristine* basic block**. This is the original basic block found in the input CFG. This block is *always* executed during normal execution at runtime when no fault injection experiment is in progress.
- **The *fault-free* basic block**. This is a semantically-equivalent copy of the *pristine* basic block, but with different predecessor and successor blocks. This basic block emulates fault-free execution within the fault injection window and can *only* be actively executed when a fault injection experiment is in progress.
- **The *faulty* basic block**. This is a transformed version of the original basic block found in the input CFG, instrumented with the faultload distribution defined by the static fault model. This block emulates faulty execution within the fault injection window and can *only* be executed when an experiment is in progress.
- **The *FDP* basic block**. This is a newly generated basic block which implements the *fault decision point* (*FDP*) for the benefit of the dynamic fault model. This block determines the basic block to run next within the fault injection window, allowing the experiment to seamlessly switch between *fault-free* and *faulty* execution.

Figure 5.2 shows a simplified example of EDFI's basic block cloning strategy. The original CFG in the example was generated from a simple function with a single loop summing all the elements of an array. As exemplified in the figure, the transformation preserves the basic structure of the original CFG, but a number of *pristine* basic blocks are modified to check the value of a special *switch* flag and redirect execution to a different code version when necessary. To minimize the runtime overhead, the flag is only checked at function entries and loop back edges (not shown in the figure, for simplicity), similar to [286]. While efficient, this approach provides only coarse-grained control over the execution with no ability to switch to a different code version at every basic block. To address this challenge, EDFI only relies on the *switch* flag to interrupt (and restore) normal execution at the beginning (and at the end) of the fault injection window, but introduces *FDP* blocks to support basic block-level switching granularity during the experiment. Note that supporting instruction-level switching granularity is also an option, but we found this strategy to drastically increase the complexity of the CFG—hindering optimizations and encouraging memory overhead—without significantly improving the expressiveness of the dynamic fault model. If more expressiveness is necessary, our basic block cloning strategy could also be configured to generate multiple *faulty* basic block versions rather than just one for each *pristine* basic block, also providing the ability to switch between different static fault models at runtime.

At the beginning of the fault injection experiment, the control library sets the *switch* flag to allow the execution to switch to a different code version at the next function entry or loop back edge—the latter is necessary to support execution-driven fault injection in face of long-running loops. From that moment, the execution percolates through a network of *FDP* blocks, which reflects the original CFG structure but can selectively redirect the execution flow to *faulty* or *fault-free* basic block versions according to the dynamic fault model. When the *switch* flag is unset to terminate the experiment—as dictated by the dynamic fault model or by the controller—the *FDP* blocks channel the execution exclusively into *fault-free* basic blocks, while allowing the system to restore normal execution at the next function entry or loop back edge.

5.5 Static Fault Model

The goal of the static fault model is to shape the faultload distribution adopted at runtime for the fault injection experiments. In particular, the model should give the user the ability to accurately specify *what* faults to inject and with what *distribution*, according to the particular fault scenario considered. For this purpose, EDFI relies on generic *static fault handlers* (*SFHs*). A single *SFH* implements a particular fault injection strategy, characterized by a *static fault trigger* (*SFT*) (i.e., conditions that designate particular code locations for fault injection) and *fault type* (i.e., actions to inject the fault into the program). *SFHs* are implemented by pluggable objects that adhere to a well-defined C++ programming interface, shown in Listing 5.1.

The abstract C++ class `StaticFaultHandler` defines a number of virtual methods for the benefit of the subclasses. The optional `init()` method can be used to

```
class StaticFaultHandler {
    virtual void init(Module &M, string &params) {}
    virtual bool canInject(Value *faultLocation,
                  double faultProb) = 0;
    virtual void inject(Value *faultLocation) = 0;
};

class StaticFaultAnalyzer {
    static double getMaxSFIF(void);
    virtual void init(Module &M, string &params) {}
    virtual double getSFIF(Value *faultLocation) = 0;
};
```

Listing 5.1: Static fault C++ programming interface.

perform one-time initialization operations. The `inject()` method is used to implement the fault injection strategy. Finally, the `canInject()` method is used to implement the static fault trigger. Our static fault triggers are similar, in spirit, to generic fault triggers proposed in prior work [196; 66]. Our *SFTs*, however, are completely static—dynamic triggers are, in contrast, used in our dynamic fault model (§5.6).

At the end of the basic block cloning process, our transformation pass locates all the `StaticFaultHandler` implementations (built-in or user-defined) scheduled for injection according to the static fault model specified by the user. Next, the pass scans the entire LLVM IR program to identify all the candidate fault locations (e.g., store instruction). Our current implementation supports fault locations at three different levels of granularity, reflected in the `Function`, `BasicBlock`, and `Instruction` LLVM IR objects. For each candidate fault location, the pass invokes the `canInject()` method on all the designated *SFH* objects. Our current `StaticFaultHandler` implementations consider only instruction-level fault locations, but it is straightforward to implement more complex fault injection strategies that operate at the function or basic block level.

The `canInject()` method accepts two arguments: the current candidate fault location and the fault probability. The latter determines the probability that a particular fault type will be injected in a candidate fault location. The fault types (i.e., *SFHs*) to consider, their corresponding probabilities, and any other optional parameters (i.e., `params`) are specified by the user via command-line arguments to our pass. These arguments reflect our definition of *faultload distribution*, which is fundamentally different from prior characterizations adopted in the literature [94].

Traditional faultload characterizations describe the set of fault types in terms of the fraction of the total faults each fault type represents [94]. While convenient for location-based approaches and single-fault injection strategies, this definition often translates to a weak faultload characterization, which ignores the structure and size of the program. Our probability-based characterization, in contrast, is inherently code size-agnostic and enables *simultaneous* fault injection [284]. The former property is particularly important to compare fault injection results across different programs, while also giving strong guarantees that the given fault probabilities will be reflected in the output at runtime—precision problems should only be expected in cases of very limited coverage (§5.6).

As acknowledged in the analysis presented in [228], however, there are many factors that may nontrivially increase the fault density in particular code locations. For example, prior studies have shown that the fault density is statistically correlated with code complexity measures [208]. Other studies have presented empirical evidence that imports and function calls correlate with security vulnerabilities [217]. To alter the original faultload distribution and express more sophisticated fault models that consider these (and other) conditions, EDFI relies on generic *static fault impact factors* (*SFIFs*). These factors can be used to amplify or (reduce) the fault probabilities in particular code locations, orthogonally to the original fault types considered. The *SFIFs* are computed on a per-fault location basis by pluggable *static fault analyzer* (*SFAs*) objects, which also adhere to a well-defined C++ programming interface (Listing 5.1).

In addition to the conventional initialization method (i.e., `init()`), the abstract C++ class `StaticFaultAnalyzer` exposes two methods to the transformation pass. The virtual method `getSFIF()` returns the fault impact factor of the current candidate fault location. The static method `getMaxSFIF()` returns the maximum fault impact factor possible across all the user-specified *SFAs*. For each candidate fault location in the program, the pass invokes the `getSFIF()` method on each designated *SFA* and stops when the first valid *SFIF* for the current location is found (if any). The priority of application of a particular *SFA* is determined by the original order specified by the user. The final fault probability given to the `canInject()` method of each `StaticFaultHandler` object is the normalized version of the original fault probability, which is simply computed as:

```
faultProb *= SFA.getSFIF()/SFA.getMaxSFIF()
```

Unlike *SFIFs*, static fault triggers are never evaluated in a priority-based fashion. After calling the `canInject()` method on all the designated *SFHs*, the pass selects only those that have returned a positive answer and performs random selection to resolve eventual collisions. This strategy is necessary to avoid perturbing the faultload distribution specified by the user and also eliminate duplicate faults that can introduce representativeness problems. The selected *SFH* (if any) is finally requested to inject the fault into the program (i.e., with a call to the `inject()` method). At the end of the process, the pass reports accurate statistics on the faultload distribution injected. This is important to give the user a feedback on the quality of the final static fault model applied (e.g., a high fault collision rate may introduce discrepancies in the original faultload distribution).

EDFI includes a number of built-in *SFHs* and *SFAs* that users can combine (and extend) to express several different static fault models. In particular, EDFI implements *SFHs* for all the standard software fault types described in the literature and commonly found in the field [94; 72; 270]. In addition, EDFI can specifically emulate several memory errors, including buffer overflows, off-by-N errors, uninitialized reads, memory leaks, invalid `free()` errors, and use-after-free errors. Finally, EDFI can emulate interface errors similar to those described in [198; 172] (although we have not yet implemented LFI's return code analysis [198]), while also generalizing these strategies to generic function interfaces.

The built-in *SFAs* implemented in EDFI, in turn, can be used and combined to emulate a number of sophisticated fault scenarios. The most basic *SFA* (i.e., `RandomFaultAnalyzer`) allows the user to override the default fault impact factor for a predetermined number of basic blocks selected at random in the program. This strategy can be used to mimic the behavior of existing location-based fault injection strategies, as also done in our evaluation. The `FunctionFaultAnalyzer` and `ModuleFaultAnalyzer` *SFAs*, in turn, allow the user to override the fault impact factors of a set of predetermined functions or modules, respectively. This is useful, for example, to emulate and analyze the impact of particularly faulty components. Finally, the `CallerFaultAnalyzer` *SFA* allows the user to override the fault impact factors of all the instructions (or basic blocks) which call a particular set of functions. This is useful, for example, to emulate interface-level fault injection at the library interfaces [198].

Other than using the built-in *SFHs* and *SFAs*, users can easily implement their own. Using the programming interface introduced earlier, users can add new *SFHs* and *SFAs* directly to the existing framework or include them in separate LLVM plugins. The LLVM API provides several opportunities to implement complex extensions with minimal effort. For example, implementing a *SFA* that amplifies the *SFIF* according to the number of lines of code in a module or the *McCabe*'s cyclomatic complexity computed over the current function (one of the best fault predictors, according to [208]) is straightforward.

5.6 Dynamic Fault Model

The static fault model describes a systematic faultload distribution for the fault injection experiment, but cannot alone express more sophisticated dynamic conditions that affect the runtime system behavior. This is the main goal of the dynamic fault model. The users specify a dynamic fault model for fault scenarios that need to alter or control the faultload distribution during the experiment at runtime. In particular, the model can be used to specify *when* to switch to faulty execution and *what* to do when faults are injected into the execution. In addition, the model defines all the actions to perform at the beginning and at the end of the fault injection experiment. To meet these goals, EDFI supports a convenient event-driven interface to customize and control the runtime behavior of the experiment.

In detail, EDFI's dynamic instrumentation model defines four primary events: *start event* (triggered at the beginning of the experiment, as dictated by the controller), *fdp event* (triggered at every fault decision point encountered), *fault event*

```
void edfi_onstart(edfi_context_t *context);
int  edfi_onfdp(edfi_context_t *context,
                const char *file, int line);
void edfi_onfault(edfi_context_t *context,
                  const char *file, int line,
                  int num_fault_types, ...);
void edfi_onstop(edfi_context_t *context);
```

Listing 5.2: Dynamic fault C programming interface.

(triggered when switching to faulty execution), *stop event* (triggered at the end of the experiment). For each of these events, EDFI defines a corresponding event handler in the C programming interface exported by the dynamic fault library. The four event handlers are shown in Listing 5.2.

Every event handler receives as an argument a pointer to the *fault injection context* (i.e., **edfi_context_t** object). The context includes all the fault injection variables that are normally used to initialize, track, and influence the state of the experiment. For example, the context holds the counters to provide statistics on the faultload distribution observed at runtime, as well as the policies to control the behavior of our stock dynamic fault library. To prevent corruption of the fault injection variables during the experiment, the control libraries guarantee that the context is always mapped high in memory and protected with guard pages.

The **edfi_onstart()** handler, automatically called when the controller signals the beginning of the experiment, initializes the fault injection context and other implementation-specific data structures. The default implementation in the stock dynamic fault library initializes the context with default values, while allowing the user to override these values through the control interface. The **edfi_onstop()** handler, automatically called at the end of the experiment, performs implementation-specific cleanup operations and outputs statistics. Our default implementation logs the termination event along with all the statistics on the faultload distribution observed. The end of the experiment can be triggered by any of the other event handlers or determined by the control libraries—in response to a user event or when a termination event is detected. To detect termination events, the current implementation of the control server library (tailored to UNIX applications) can register **atexit()** functions and abnormal termination signal handlers (e.g., **SIGSEGV**, **SIGABRT**, etc.).

The **edfi_onfdp()** handler, automatically called by our instrumentation at fault decision points, implements EDFI's *dynamic fault trigger* (*DFT*). The *DFT* returns a nonzero value to request switching to faulty execution in the next basic block. The **edfi_onfault()** handler, automatically called by our instrumentation at the beginning of a faulty basic block, implements EDFI's *dynamic fault logger* (*DFL*). The *DFL* receives as arguments the static callsite information and a variable number of arguments that indicate the types and the number of the faults injected in the current basic block. Our default implementation simply updates faultload distribution statistics in the fault injection context.

EDFI includes three main built-in *DFT* implementations:

- **Time-based *DFT***. This *DFT* can be configured to ensure a minimum predetermined time interval between faulty execution blocks. The time interval is initialized in the fault injection context and can be dynamically adjusted to specify more complex time distributions. Albeit not necessarily useful to represent realistic fault scenarios, this *DFT* can be used to analyze the behavior of existing time-based fault injection approaches.
- ***FDP*-based *DFT***. This *DFT* can be configured to ensure a minimum predetermined *FDP* interval between faulty execution blocks. This is similar to the time-based *DFT* above, but the time is measured in terms of number of *FDPs* executed instead of microseconds. This *DFT* can be used to accurately specify the timing of runtime faulty behavior in an execution-driven fashion. Unlike time-based *DFTs*, this strategy translates to reproducible and unbiased fault injection experiments.
- **Probability-based *DFT***. This *DFT* can be configured to express a predetermined dynamic probability of switching to faulty execution. As for the other *DFTs*, the probability is initialized in the context and can be dynamically adjusted to specify complex distributions. This *DFT* can be used to accurately specify the likelihood of runtime faulty execution and emulate particular fault scenarios (e.g., bug clustering effects). In addition, dynamic probabilities can be used to adjust (or replace) the static probabilities given for the static fault model in case of limited or highly polarized code coverage—which may affect the precision of the resulting output faultload distribution. For example, a program executing only a few in-loop basic blocks may result in poor precision and fault activation guarantees with particular static fault models. A possible solution is to instruct the static fault model to inject faults in every fault location candidate and rely exclusively on dynamic probabilities to shape the resulting faultload distribution.

The default *DFT* implementation in our stock dynamic fault library evaluates all the built-in *DFTs* which have been parametrized by the user (if any). Further, our default implementation allows the user to specify conditions that can automatically terminate the experiment. Termination can be triggered basing on *time*, *FDPs*, and *faults* observed from the beginning of the experiment. To parametrize the experiment, users can, for example, rely on our stock controller:

```
edfi-ctl <start|stop> [options]
```

The optional `[options]` argument allows the user to configure the fault injection context for the experiment and the dynamic fault library to use. When no option is given, EDFI resorts to the stock library implementation and systematically switches to faulty execution with no restriction during the experiment.

Other than configuring and combining the built-in *DFTs* and *DFLs*, users can easily implement their own dynamic fault library. Our C programming interface provides convenient access to the fault injection context and the entire program state. For example, it would be straightforward to implement a *DFT* that switches to faulty execution only when the program reaches a particular state, similar to [66; 196]. Further, EDFI exposes *static fault IDs* (derived by callsite information) and *dynamic fault IDs* (derived by calltrace information) directly to the *DFTs* and *DFLs*, generalizing *failure IDs* in [120]. Other than supporting simple call stack-based or call count-based triggers as in [196], this interface can be used to implement more complex dynamic fault models, including:

- **Emulate transient (or intermittent) faults**. In this scenario, the *DFL* implementation logs all the *fault IDs* in memory, allowing the *DFT* to identify duplicate *fault IDs*. To emulate transient (or intermittent) faults, the *DFT* implementation discards (or selectively enables/disables) duplicate *fault IDs* in a single run.
- **Record/replay a fault injection experiment**. In this scenario, the *DFL* implementation logs all the *fault IDs* to persistent storage. In subsequent fault injection runs, the *DFT* implementation systematically replays a previously logged run. If necessary, deterministic replay can be enforced using third-party frameworks [122].
- **Implement high-coverage fault exploration**. In this scenario, the *DFL* implementation logs all the *fault IDs* to persistent storage. The *DFT* implementation, in turn, discards duplicate *fault IDs* across different runs. More advanced fault exploration strategies, such as those in [40; 163; 120] are also possible.

5.7 Evaluation

Our current EDFI implementation runs on standard UNIX systems, being specifically designed to support fault injection for user-space UNIX programs. Its portability, however, is only subject to the platform-specific control libraries, which are easy to retarget given their small size (234 LOC [1]). The static instrumentation, in turn, is implemented as an LLVM pass in 1150 LOC. The stock dynamic fault library and the command-line controller, finally, are implemented in C in 259 and 55 LOC, respectively.

We evaluated EDFI on a workstation running Linux v2.6.32 and equipped with a 12-core 1.3Ghz AMD Opteron processor and 4GB of RAM. For our evaluation, we considered MySQL (v5.1.65) and Apache httpd (v2.2.23), a popular open-source DBMS and web server, respectively. To directly compare our results with recent fault injection techniques [198], we performed our tests using the SysBench OLTP

[1] Source lines of code reported by David Wheeler's SLOCCount.

Test scenario	Static HTML	PHP
Normal execution	1.024	1.007
FDPs only	1.052	1.018
Default *DFL* only	2.091	1.138
Default *DFT* (nonparametrized)	2.416	1.185
FDP-based *DFT*	4.190	1.468
Time-based *DFT*	4.206	1.472
Probability-based *DFT*	4.464	1.521

Table 5.1: Time to complete the Apache benchmark (AB) normalized against the baseline.

benchmark [15] (MySQL) and the AB benchmark [1] (Apache httpd). We configured our programs and benchmarks using the default settings. To obtain unbiased results toward particular fault types, we allowed EDFI to use the same static fault probability $P = \Phi$ (with $\Phi = 0.5$, unless otherwise noted) in all our tests. We repeated all our experiments 21 times and report the median.

Our evaluation answers four questions: (i) *Performance*: Does EDFI yield low runtime overhead during normal execution and reasonable slowdown during the experiment? (ii) *Memory usage*: How much memory does EDFI use? (iii) *Precision*: Does EDFI yield more precise faultload distributions than prior tools? (iv) *Controllability*: Does EDFI yield more controllable experiments than prior tools?

5.7.1 Performance

We evaluated the runtime overhead imposed by the fault injection mechanisms used in EDFI. To this end, we evaluated our application benchmarks in a number of test scenarios. In the first scenario, we instrumented the applications and measured the overhead imposed on our benchmarks during normal execution. The question we wish to address is whether our instrumentation introduces minimal impact on normal execution and the overhead of checking the *switch* flag is effectively amortized by hardware caches and branch prediction mechanisms. In the second scenario, we measured the overhead imposed on our benchmarks during a fault injection experiment with no *DFTs* and *DFLs* used. This scenario isolates the overhead of basic block-level switching introduced by the *FDPs*. The third and fourth scenarios, in turn, add the default *DFL* and the default nonparametrized *DFT* (respectively) to the previous configuration, isolating their overheads for comparison. Finally, the last three scenarios measure the overhead of *FDP*-based, time-based, and probability-based *DFTs*, respectively. In all the experiments, EDFI's instrumentation is configured to skip (*only*) the code mutations that inject the actual faults. This is necessary to allow our benchmarks to complete and obtain representative performance results.

We first evaluated our test scenarios with Apache httpd, measuring the time to complete the AB benchmark compared to the baseline. Similar to [198], we

Test scenario	Read-only	Read-write
Normal execution	1.053	1.054
FDPs only	1.095	1.060
Default *DFL* only	1.161	1.070
Default *DFT* (nonparametrized)	1.213	1.116
FDP-based *DFT*	1.509	1.408
Time-based *DFT*	5.201	3.920
Probability-based *DFT*	7.448	5.638

Table 5.2: MySQL throughput normalized against the baseline.

ran 1,000 requests—designed to retrieve a 1 KB file—and two different workloads (static HTML and PHP) with AB in each test scenario. For the PHP workload, we did not instrument `mod_php` to evaluate the impact of uninstrumented shared libraries. Table 5.1 presents our results. As shown in the table, the overhead introduced by our instrumentation during normal execution is negligible for the two workloads (2.4% and 0.7%), directly comparable, for example, to LFI executing with 4 triggers [196]. The other test scenarios, in turn, highlight the overhead of our *FDPs*, *DFLs*, and *DFTs* during the fault injection experiment. Compared to LFI, the overhead grows considerably when evaluating additional triggers (and event handlers in general), reaching maximum values of 346.4% and 52.1% with the stock *DFL* and probability-based *DFT* enabled. This behavior is, however, to be expected, given that LFI solely operates at the library interfaces. EDFI's execution-driven fault injection strategy, in contrast, aims at full execution coverage. While nontrivial, this overhead is strictly confined in the fault injection window and always conditioned by the complexity of the dynamic fault model adopted by the user. For example, the basic EDFI configuration with no *DFTs* and no *DFLs* reported an overhead of only 5.2% and 1.8%.

In our second run of experiments, we evaluated our test scenarios with MySQL, measuring the throughput during the execution of the SysBench OLTP benchmark compared to the baseline. Similar to [198], we ran two different workloads (read-only queries and read-write queries) in each test scenario. Table 5.2 presents our findings. As shown in the table, the results match the behavior of our earlier experiments performed on Apache httpd, with negligible performance overhead reported during normal execution (5.3% and 5.4%) and maximum performance overhead (644.8% and 463.8%) with the stock *DFL* and probability-based *DFT* enabled.

5.7.2 Memory usage

Our hybrid instrumentation leads to larger binary sizes and larger runtime memory footprints. This stems from our basic block cloning strategy and the libraries required to support fault injection capabilities. To evaluate their impact, we measured

Type	Apache httpd	MySQL
Static	1.919	1.418
Runtime (idle)	1.015	1.445
Experiment initialization	1.015	1.445
Experiment in progress	1.015	1.329

Table 5.3: Memory usage normalized against the baseline.

the memory overhead incurred by Apache httpd and MySQL when instrumented with our stock EDFI components in their default configuration. Table 5.3 presents our results. The static memory overhead (91.9% and 41.8%, respectively) measures the impact of our basic block cloning strategy and our stock control server library on the binary size. The runtime (idle) overhead (1.5% and 44.5%, respectively) measures the impact of the same instrumentation on the virtual memory size observed at runtime, right after initialization. The next row in the table is similar, but shows the virtual memory overhead at the beginning of the experiment, with our stock dynamic fault library already loaded in memory and only marginally impacting Apache httpd and MySQL's memory footprint. The last row, finally, shows the average virtual memory overhead observed within the fault injection window during the execution of our benchmarks (1.5% and 32.9%, respectively). As expected, the memory overhead introduced by EDFI is heavily influenced by the structure of the application. For example, Apache httpd reports a very low virtual memory overhead due to its large memory footprint—234MB after initialization, compared to only 42MB for MySQL. Overall, EDFI's memory overhead is modest, confirming the realistic and practical deployment of our techniques.

5.7.3 Precision

To assess the effectiveness of EDFI's execution-driven fault injection strategy, we evaluated the precision of the faultload distribution observed in the output of a fault injection experiment. For this purpose, we performed repeated experiments with increasing values of the fault probability Φ. In each experiment, we synchronized the fault injection window with the execution of our benchmarks, while collecting statistics on the faultload distribution observed in output. From the statistics, we directly computed the output fault probabilities for each fault type and compared their values with the input fault probabilities statically applied by our instrumentation. From the input and output probabilities collected, we computed—in each test scenario—the *faultload degradation*, which we define as the median relative error (*MRE*) across all the output fault type probabilities observed in the experiment. The faultload degradation gives an indication of the error the tool introduces when repre-

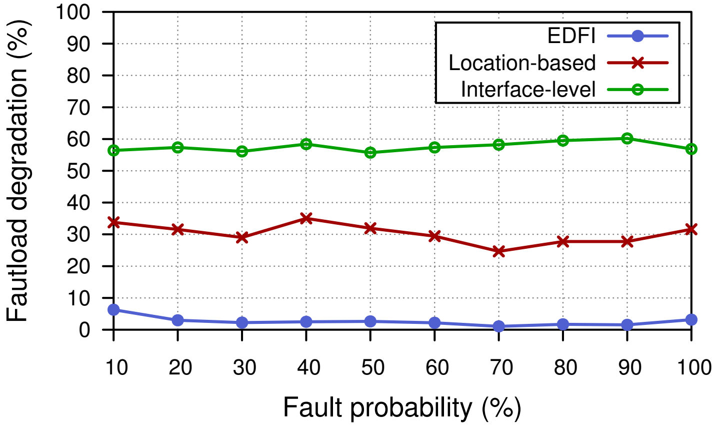


Figure 5.3: Comparative faultload degradation for Apache httpd (static HTML).

senting the output faultload distribution starting from the original input distribution specified by the user. We selected *faultload degradation* as a measure of precision, since it captures both (i) the ability of a tool to actually activate the faults specified by the user without being affected by code coverage problems and (ii) its ability to preserve the original distribution of fault types considered.

To allow our benchmarks to complete correctly, we again configured EDFI's instrumentation to skip (*only*) the code mutations that inject the actual faults. To compare EDFI's fault injection strategy with prior approaches, we also simulated location-based strategies and interface-level strategies using our built-in *SFAs*. We did not consider runtime time-based strategies in our evaluation, given that prior studies have already demonstrated their serious representativeness problems [155]. Using the **`CallerFaultAnalyzer`** *SFA*, we simulated interface-level strategies by instructing EDFI to inject faults only into basic blocks that contained library calls into *libc*. We specifically selected *libc* as a reference library for our experiments to obtain general and unbiased results. Using the **`RandomFaultAnalyzer`** *SFA*, we simulated location-based strategies by instructing EDFI to inject faults only into β basic blocks selected at random at every run (averaging the results over 201 runs). For comparability purposes, we selected the value of β according to the number of basic blocks that contained at least one library call into *libc* (1689 and 1808 for Apache httpd and MySQL, respectively). Figure 5.3 presents our results for the Apache benchmark (AB) (static HTML). We omit the results obtained for the PHP workload and for MySQL (read-only queries and read-write queries), since they matched the behavior observed for Apache httpd with no significant difference.

As shown in the figure, EDFI generated a very precise faultload distribution in output, with almost no faultload degradation for any choice of the fault probability

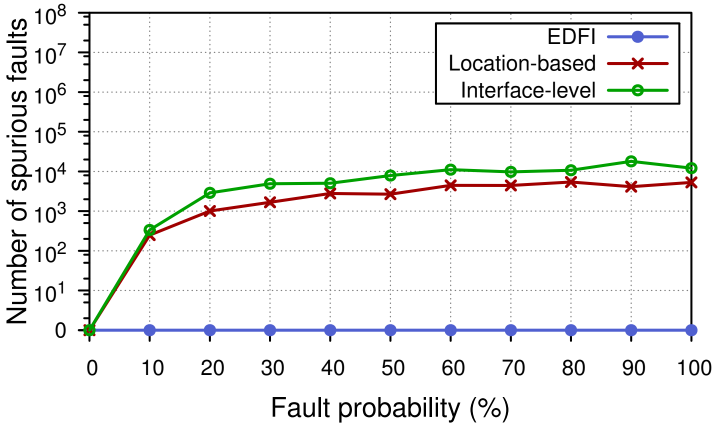


Figure 5.4: Comparative number of spurious faults activated during Apache httpd initialization.

Φ. This demonstrates the benefits of injecting faults over the entirety of the program code. The other fault injection strategies, in contrast, generated imprecise faultload distributions in output, with much higher faultload degradation across all the experiments. This behavior stems from the limited coverage achieved by existing strategies. Interestingly, the location-based strategy reported lower faultload degradation (30% on average) compared to the interface-level strategy (57% on average). We interpret this behavior as the ability of random injection to achieve better coverage (on average) than injection into predetermined interface-level locations.

5.7.4 Controllability

We also evaluated the controllability properties of EDFI when compared to prior approaches. In particular, the question we wish to address is how well EDFI improves prior strategies in terms of user control over the fault injection experiment. For this purpose, we evaluated the number of spurious faults (i.e., faults activated before starting the experiment) introduced by the different strategies during the initialization of Apache httpd. The rationale is that every reasonable fault injection strategy should allow the target program to complete initialization before starting the fault injection experiment under a user-specified test workload. If spurious initialization-time faults are activated, however, the target program may prematurely crash (or reach a tainted and nonrepresentative state), thus compromising the validity of the entire fault injection experiment.

As done earlier, we performed repeated experiments with increasing values of the fault probability Φ. We also simulated location-based and interface-level strategies using our built-in *SFAs* and the same configuration adopted earlier. A word of warning is in order for the interpretation of the results in this particular test sce-

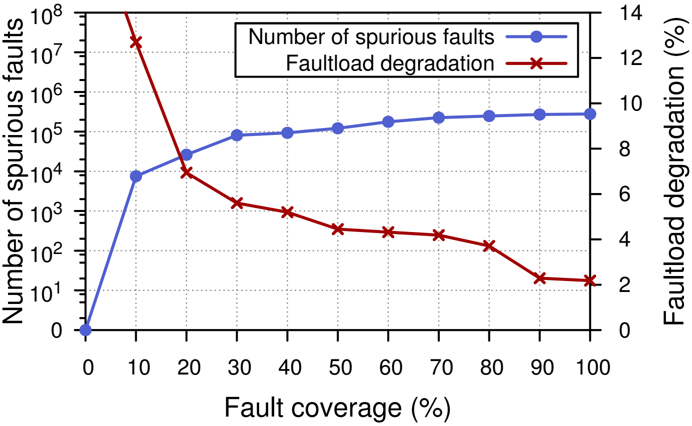


Figure 5.5: Impact of fault coverage on location-based strategies (Apache httpd).

nario. Our controllability analysis is only applicable to static (location-based and interface-level) fault injection strategies. Dynamic strategies—such as LFI [198]—are not affected by controllability issues, given that faults are always injected on demand and under direct control of the user. Figure 5.4 presents our results.

As expected, EDFI reported no spurious faults during the initialization of Apache httpd. For the other strategies, in contrast, the number of spurious faults increases with the value of the fault probability Φ. This behavior is expected, given the higher chances of fault activation in various (and arbitrary) parts of the program. As the figure shows, the interface-level strategy reported a consistently higher number of spurious faults compared to the location-based strategy, with 12120 and 5309 faults (respectively), for $\Phi = 100\%$. We interpret this behavior as a result of the particularly high density of *libc* calls during initialization.

This test scenario also highlights the precision-controllability tradeoff for existing static fault injection strategies. A larger number of faults injected results in better precision but, at the same time, lower controllability. To better investigate this tradeoff, we evaluated the impact of varying the value of the number of faulty basic blocks β in location-based strategies. This experiment was useful to understand the impact of fault coverage on static fault injection strategies, location-based approaches in particular. Figure 5.5 presents our findings. For low fault coverage values (e.g., around 10%), the number of spurious faults is more limited (around 7550), but faultload degradation is high (around 13%), thus resulting in poor precision. Conversely, for high fault coverage values (e.g., around 90%), faultload degradation is much lower (around 2%), but the number of spurious faults observed is substantial (around 270850), thus resulting in poor controllability. This experiment efficaciously pinpoints important limitations in existing strategies, while highlighting the better controllability and precision properties of EDFI's execution-driven fault injection strategy.

5.8 Conclusion

Fault injection experiments have been long proposed as an answer to a key question in the dependability community: "*How can we thoroughly assess the dependability of a software system?*" Undoubtedly, another equally important question is: "*How can we thoroughly assess the dependability of fault injection experiments?*" We believe the answer lies in building a new generation of general-purpose fault injection tools that can support truly precise, controllable, and observable fault injection experiments in a controlled setting. EDFI represents an important step in this direction.

EDFI injects faults in a controlled way during the execution to ensure a predetermined faultload distribution at runtime. Its hybrid instrumentation strategy provides fine-grained control over the experiment, while avoiding unnecessary perturbations to the system—or its performance—during fault-free execution. Its portable and extensible LLVM-based architecture can support several possible static and dynamic fault models, generalizing existing general-purpose fault injection tools while providing the ability to adapt to different execution contexts.

Our ultimate goal is to foster the development of a common fault injection framework for dependability researchers and practitioners, in order to support dependable, reproducible, and comparable experiments in fault injection campaigns.

5.9 Acknowledgments

We would like to thank the anonymous reviewers for their comments. This work has been supported by European Research Council under ERC Advanced Grant 227874.

6

Back to the Future: Fault-tolerant Live Update with Time-traveling State Transfer

Abstract

Live update is a promising solution to bridge the need to frequently update a software system with the pressing demand for high availability in mission-critical environments. While many research solutions have been proposed over the years, systems that allow software to be updated on the fly are still far from reaching widespread adoption in the system administration community. We believe this trend is largely motivated by the lack of tools to automate and validate the live update process. A major obstacle, in particular, is represented by state transfer, which existing live update tools largely delegate to the programmer despite the great effort involved.

This paper presents *time-traveling state transfer*, a new automated and fault-tolerant live update technique. Our approach isolates different program versions into independent processes and uses a semantics-preserving state transfer transaction—across multiple *past*, *future*, and *reversed* versions—to validate the program state of the updated version. To automate the process, we complement our live update technique with a generic state transfer framework explicitly designed to minimize the overall programming effort. Our time-traveling technique can seamlessly integrate with existing live update tools and automatically recover from arbitrary run-time and memory errors in any part of the state transfer code, regardless of the particular implementation used. Our evaluation confirms that our update techniques can withstand arbitrary failures within our fault model, at the cost of only modest performance and memory overhead.

6.1 Introduction

In the era of pervasive and cloud computing, we are witnessing a major paradigm shift in the way software is developed and released. The growing demand for new features, performance enhancements, and security fixes translates to more and more frequent software updates made available to the end users. In less than a decade, we quickly transitioned from Microsoft's "*Patch Tuesday*" [115] to Google's "*perpetual beta*" development model [226] and Facebook's tight release cycle [212], with an update interval ranging from days to a few hours.

With more frequent software updates, the standard halt-update-restart cycle is irremediably coming to an impasse with our growing reliance on nonstop software operations. To reduce downtime, system administrators often rely on "*rolling upgrades*" [91], which typically update one node at a time in heavily replicated software systems. While in widespread use, rolling upgrades have a number of important shortcomings: (i) they require redundant hardware, which may not be available in particular environments (e.g., small businesses); (ii) they cannot normally preserve program state across versions, limiting their applicability to stateless systems or systems that can tolerate state loss; (iii) in heavily replicated software systems, they lead to significant update latency and high exposure to "*mixed-version races*" [93] that can cause insidious update failures. A real-world example of the latter has been reported as "*one of the biggest computer errors in banking history*", with a single-line software update mistakenly deducting about $15 million from over 100,000 customers' accounts [128].

Live update—the ability to update software on the fly while it is running with no service interruption—is a promising solution to the update-without-downtime problem which does not suffer from the limitations of rolling upgrades. A key challenge with this approach is to build trustworthy update systems which come as close to the usability and reliability of regular updates as possible. A significant gap is unlikely to encourage adoption, given that experience shows that administrators are often reluctant to install even regular software updates [245].

Surprisingly, there has been limited focus on automating and validating generic live updates in the literature. For instance, traditional live update tools for C programs seek to automate only basic type transformations [214; 213], while more recent solutions [132] make little effort to spare the programmer from complex tasks like *pointer transfer* (§6.5). Existing live update validation tools [133; 136; 134], in turn, are only suitable for *offline* testing, add no *fault-tolerant* capabilities to the update process, require *manual* effort, and are inherently *update timing*-centric. The typical strategy is to verify that a given test suite completes correctly—according to some manually selected [133; 136] or provided [134] specification—regardless of the particular time when the update is applied. This testing method stems from the extensive focus on live update timing in the literature [135].

Much less effort has been dedicated to automating and validating *state transfer* (*ST*), that is, initializing the state of a new version from the old one (§6.2). This is somewhat surprising, given that ST has been repeatedly recognized as a challenging and error-prone task by many researchers [42; 68; 69; 193] and still represents a major obstacle to the widespread adoption of live update systems. This is also confirmed by the commercial success of Ksplice [36]—already deployed on over 100,000 production servers [18]—explicitly tailored to small security patches that hardly require any state changes at all (§6.2).

In this paper, we present *time-traveling state transfer* (TTST), a new live update technique to automate and validate generic live updates. Unlike prior live update testing tools, our validation strategy is *automated* (manual effort is never strictly required), *fault-tolerant* (detects and immediately recovers from any faults in our fault model with no service disruption), *state-centric* (validates the ST code and the full integrity of the final state), and *blackbox* (ignores ST internals and seamlessly integrates with existing live update tools). Further, unlike prior solutions, our fault-tolerant strategy can be used for *online* live update validation in the field, which is crucial to automatically recover from unforeseen update failures often originating from differences between the testing and the deployment environment [79]. Unlike commercial tools like Ksplice [36], our techniques can also handle complex updates, where the new version has significantly different code and data than the old one.

To address these challenges, our live update techniques use two key ideas. First, we confine different program versions into distinct processes and perform *process-level* live update [113]. This strategy simplifies state management and allows for automated state reasoning and validation. Note that this is in stark contrast with traditional *in-place* live update strategies proposed in the literature [214; 69; 32; 43; 194; 68; 36; 213], which "glue" changes directly into the running version, thus mixing code and data from different versions in memory. This mixed execution environment complicates debugging and testing, other than introducing address space fragmentation (and thus run-time performance overhead) over time [113].

Second, we allow two process-level ST runs using the time-traveling idea. With time travel, we refer to the ability to navigate backward and forward across program state versions using ST. In particular, we first allow a *forward* ST run to initialize the state of the new version from the old one. This is already sufficient to implement live update. Next, we allow a second *backward* run which implements the reverse state transformation from the new version back to a copy of the old version. This is done to validate—and safely rollback when necessary—the ST process, in particular to detect specific classes of programming errors (i.e., memory errors) which would otherwise leave the new version in a corrupted state. To this end, we compare the program state of the original version against the final state produced by our overall transformation. Since the latter is semantics-preserving by construction, we expect differences in the two states *only* in presence of memory errors in the ST code.

Our contribution is threefold. First, we analyze the state transfer problem (§6.2) and introduce *time-traveling state transfer* (§6.3, §6.4), an automated and fault-tolerant live update technique suitable for online (or offline) validation. Our TTST strategy can be easily integrated into existing live update tools described in the liter-

```
--- a/drivers/md/dm-crypt.c
+++ b/drivers/md/dm-crypt.c
@@ -690,6 +690,8 @@ bad3:
 bad2:
    crypto_free_tfm(tfm);
 bad1:
+    /* Must zero key material before freeing */
+    memset(cc, 0, sizeof(*cc) + cc->key_size * sizeof(u8));
    kfree(cc);
    return -EINVAL;
 }
@@ -706,6 +708,9 @@ static void crypt_dtr(...)
       cc->iv_gen_ops->dtr(cc);
    crypto_free_tfm(cc->tfm);
    dm_put_device(ti, cc->dev);
+
+    /* Must zero key material before freeing */
+    memset(cc, 0, sizeof(*cc) + cc->key_size * sizeof(u8));
    kfree(cc);
 }
```

Listing 6.1: A patch to fix an information disclosure vulnerability (CVE-2006-0095) in Linux.

ature, allowing system administrators to seamlessly transition to our techniques with no extra effort. We present a TTST implementation for user-space C programs, but the principles outlined here are also applicable to operating systems, with the process abstraction implemented using lightweight protection domains [271], software-isolated processes [153], or hardware-isolated processes and microkernels [149; 140]. Second, we complement our technique with a TTST-enabled state transfer framework (§6.5), explicitly designed to allow arbitrary state transformations and high validation surface with minimal programming effort. Third, we have implemented and evaluated the resulting solution (§6.6), conducting fault injection experiments to assess the fault tolerance of TTST.

6.2 The State Transfer Problem

The state transfer problem, rigorously defined by Gupta for the first time [125], finds two main formulations in the literature. The traditional formulation refers to the live initialization of the data structures of the new version from those of the old version, potentially operating structural or semantic data transformations on the fly [42]. Another formulation also considers the execution state, with the additional concern of remapping the call stack and the instruction pointer [124; 193]. We here adopt the former definition and decouple *state transfer* (*ST*) from *control-flow transfer* (*CFT*), solely concerned with the execution state and subordinate to the particular update mechanisms adopted by the live update tool considered—examples documented in the literature include manual control migration [124; 132], adaptive function cloning [194], and stack reconstruction [193].

We illustrate the state transfer problem with two update examples. Listing 6.1 presents a real-world security patch which fixes an information disclosure vulner-

```
--- a/example.c
+++ b/example.c
@@ -1,13 +1,12 @@
 struct s {
     int count;
-    char str[3];
-    short id;
+    int id;
+    char str[2];
     union u u;
-    void *ptr;
     int addr;
-    short *inner_ptr;
+    int *inner_ptr;
 } var;

 void example_init(char *str) {
-    snprintf(var.str, 3, "%s", str);
+    snprintf(var.str, 2, "%s", str);
 }
```

Listing 6.2: A sample patch introducing code and data changes that require state transfer.

ability (detailed in CVE-2006-0095 [5]) in the *md* (Multiple Device) driver of the Linux kernel. We sampled this patch from the dataset [9] originally used to evaluate Ksplice [36]. Similar to many other common security fixes, the patch considered introduces simple code changes that have no direct impact on the program state. The only tangible effect is the secure deallocation [71] of sensitive information on cryptographic keys. As a result, no state transformations are required at live update time. For this reason, Ksplice [36]—and other similar in-place live update tools—can deploy this update online with no state transfer necessary, allowing the new version to reuse the existing program state as is. Redirecting function invocations to the updated functions and resuming execution is sufficient to deploy the live update.

Listing 6.2 presents a sample patch providing a reduced test case for common code and data changes found in real-world updates. The patch introduces a number of type changes affecting a global `struct` variable (i.e., `var`)—with fields changed, removed, and reordered—and the necessary code changes to initialize the new data structure. Since the update significantly changes the in-memory representation of the global variable `var`, state transfer—using either automatically generated mapping functions or programmer-provided code—is necessary to transform the existing program state into a state compatible with the new version at live update time. Failure to do so would leave the new version in an invalid state after resuming execution. In §6.5, we show how our state transfer strategy can effectively automate this particular update, while traditional live update tools would largely delegate this major effort to the programmer.

State transfer has already been recognized as a hard problem in the literature. Qualitatively, many researchers have described it as "*tedious implementation of the transfer code*" [42], "*tedious engineering efforts*" [68], "*tedious work*" [69]. Others have discussed speculative [54; 44; 106; 105] and practical [215] ST scenarios

	#Upd	LU LOC	ST LOC	Norm ST IF
Ginseng	30	140	336	$8.0x$
STUMP	13	186	173	$7.1x$
Kitsune	40	523	554	$2.6x$

Table 6.1: ST impact (normalized after 100 updates) for prior user-level solutions for C programs.

which are particularly challenging (or unsolvable) even with programmer intervention. Quantitatively, a number of user-level live update tools for C programs (Ginseng [214], STUMP [213], and Kitsune [132]) have evaluated the ST manual effort in terms of lines of code (LOC). Table 6.1 presents a comparative analysis, with the number of updates analyzed, initial source changes to implement their live update mechanisms (LU LOC), and extra LOC to apply all the updates considered (ST LOC). In the last column, we report a normalized ST impact factor (Norm ST IF), measured as the expected ST LOC necessary after 100 updates normalized against the initial LU LOC.

As the table shows, the measured impacts are comparable (the lower impact in Kitsune stems from the greater initial annotation effort required by program-level updates) and demonstrate that ST increasingly (and heavily) dominates the manual effort in long-term deployment. Worse yet, any LOC-based metric underestimates the real ST effort, ignoring the atypical and error-prone programming model with nonstandard entry points, unconventional data access, and reduced testability and debuggability. Our investigation motivates our focus on automating and validating the state transfer process.

6.3 System Overview

We have designed our TTST live update technique with portability, extensibility, and interoperability in mind. This vision is reflected in our modular architecture, which enforces a strict separation of concerns and can support several possible live update tools and state transfer implementations. To use TTST, users need to statically instrument the target program in preparation for state transfer. In our current prototype, this is accomplished by a link-time transformation pass implemented using the LLVM compiler framework [179], which guarantees pain-free integration with existing GNU build systems using standard `configure` flags. We envision developers of the original program (i.e., users of our TTST technique) to gradually integrate support for our instrumentation into their development model, thus releasing live update-enabled software versions that can be easily managed by system administrators using simple tools. For this purpose, our TTST prototype includes `ttst-ctl`, a simple command-line tool that interacts with the running program and

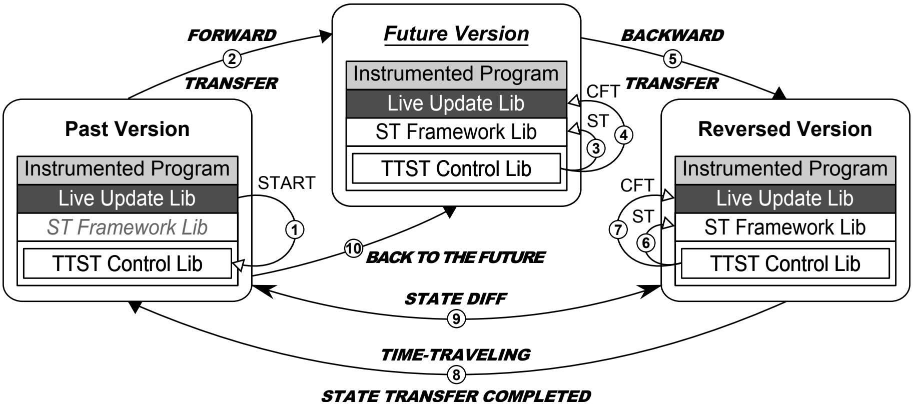


Figure 6.1: Time-traveling state transfer overview. The arrows indicate the order of operations.

allows administrators to deploy live updates using our TTST technique with minimal effort. This can be simply done by using the following command-line syntax:

```
ttst-ctl `pidof program` ./new.bin
```

Runtime update functionalities, in turn, are implemented by three distinct libraries, transparently linked with the target program as part of our instrumentation process. The *live update library* implements the update mechanisms specific to the particular live update tool considered. In detail, the library is responsible to provide the necessary update timing mechanisms [133] (e.g., start the live update when the program is *quiescent* [133] and all the external events are blocked) and CFT implementation. The *ST framework library*, in turn, implements the logic needed to automate state transfer and accommodate user-provided ST code. The *TTST control library*, finally, implements the resulting *time-traveling state transfer* process, with all the necessary mechanisms to coordinate the different process versions involved.

Our TTST technique operates across three process instances. The first is the original instance running the old software version (*past version*, from now on). This instance initiates, controls, and monitors the live update process, in particular running the only trusted library code in our architecture with respect to our fault model (§6.4). The second is a newly created instance running the new software version (*future version*, from now on). This instance is instructed to reinitialize its state from the past version. The third process instance is a clone of the past version created at live update time (*reversed version*, from now on). This instance is instructed to reinitialize its state from the future version. Figure 6.1 depicts the resulting architecture and live update process.

As shown in the figure, the update process is started by the live update library in the past version. This happens when the library detects that an update is available and all the necessary update timing restrictions (e.g., *quiescence* [133]) are met. The *start* event is delivered to the past version's TTST control library, which sets out to initiate the *time-traveling state transfer* transaction. First, the library locates the new program version on the file system and creates the process instances for the future and reversed versions. Next, control is given to the future version's TTST control library, requesting to complete a *forward* state transfer run from the past version. In response, the library instructs the live update and ST framework libraries to perform ST and CFT, respectively. At the end of the process, control is given to the reversed version, where the TTST control library repeats the same steps to complete a *backward* state transfer run from the future version. Finally, the library notifies back the past version, where the TTST control library is waiting for TTST events. In response, the library performs *state differencing* between the past and reversed version to validate the TTST transaction and detect state corruption errors violating the semantics-preserving nature of the transformation. In our fault model, the past version is always immutable and adopted as a oracle when comparing the states. If the state is successfully validated (i.e., the past and reversed versions are identical), control moves *back to the future* version to resume execution. The other processes are automatically cleaned up.

When state corruption or run-time errors (e.g., crashes) are detected during the TTST transaction, the update is immediately aborted with the past version cleaning up the other instances and immediately resuming execution. The immutability of the past version's state allows the execution to resume exactly in the same state as it was right before the live update process started. This property ensures instant and transparent recovery in case of arbitrary TTST errors. Our recovery strategy enables fast and automated offline validation and, more importantly, a fault-tolerant live update process that can immediately and automatically rollback failed update attempts with no consequences for the running program.

6.4 Time-traveling State Transfer

The goal of TTST is to support a truly fault-tolerant live update process, which can automatically detect and recover from as many programming errors as possible, seamlessly support several live update tools and state transfer implementations, and rely on a minimal amount of trusted code at update time. To address these challenges, our TTST technique follows a number of key principles: a well-defined *fault model*, a large *state validation surface*, a *blackbox validation* strategy, and a generic *state transfer interface*.

6.4.1 Fault model

TTST assumes a general fault model with the ability to detect and recover from arbitrary *run-time* errors and *memory* errors introducing state corruption. In particular, run-time errors in the future and reversed versions are automatically detected by the TTST control library in the past version. The process abstraction allows the library to intercept abnormal termination errors in the other instances (e.g., crashes, panics) using simple tracing. Synchronization errors and infinite loops that prevent the TTST transaction from making progress, in turn, are detected with a configurable update timeout (5s by default). Memory errors, finally, are detected by state differencing at the end of the TTST process.

Our focus on memory errors is motivated by three key observations. First, these represent an important class of nonsemantic state transfer errors, the only errors we can hope to detect in a fully automated fashion. Gupta's formal framework has already dismissed the possibility to automatically detect semantic state transfer errors in the general case [125]. Unlike memory errors, semantic errors are consistently introduced across forward and backward state transfer runs and thus cannot automatically be detected by our technique. As an example, consider an update that operates a simple semantic change: renumbering all the global error codes to use different value ranges. If the user does not explicitly provide additional ST code to perform the conversion, the default ST strategy will preserve the same (wrong) error codes across the future and the reversed version, with state differencing unable to detect any errors in the process.

Second, memory errors can lead to insidious latent bugs [100]—which can cause silent data corruption and manifest themselves potentially much later—or even introduce security vulnerabilities. These errors are particularly hard to detect and can easily escape the specification-based validation strategies adopted by all the existing live update testing tools [136; 133; 134].

Third, memory errors are painfully common in pathologically type-unsafe contexts like state transfer, where the program state is treated as an opaque object which must be potentially reconstructed from the ground up, all relying on the sole knowledge available to the particular state transfer implementation adopted.

Finally, note that, while other semantic ST errors cannot be detected in the general case, this does not preclude individual ST implementations from using additional knowledge to automatically detect some classes of errors in this category. For example, our state transfer framework can detect all the semantic errors that violate automatically derived *program state invariants* [109] (§6.5).

6.4.2 State validation surface

TTST seeks to validate the largest possible portion of the state, including state objects (e.g., global variables) that may only be accessed much later after the live update. To meet this goal, our state differencing strategy requires valid forward and backward transfer functions for each state object to validate. Clearly, the existence and the properties of such functions for every particular state object are subject to the nature of the update. For example, an update dropping a global variable in the new version has no defined backward transfer function for that variable. In other cases, forward and backward transfer functions exist but cannot be automatically

State	Diff	Fwd ST	Bwd ST	Detected
Unchanged	✓	STF	STF	Auto
Structural chg	✓	STF	STF	Auto
Semantic chg	✓	User	User [1]	Auto [1]
Dropped	✓	-	-	Auto
Added	✗	Auto/User	-	STF

[1] Optional

Table 6.2: State validation and error detection surface.

generated. Consider the error code renumbering update exemplified earlier. Both the forward and backward transfer functions for all the global variables affected would have to be manually provided by the user. Since we wish to support fully automated validation by default (mandating extra manual effort is likely to discourage adoption), we allow TTST to gracefully reduce the state validation surface when backward transfer functions are missing—without hampering the effectiveness of our strategy on other fully transferable state objects. Enforcing this behavior in our design is straightforward: the reversed version is originally cloned from the past version and all the state objects that do not take part in the backward state transfer run will trivially match their original counterparts in the state differencing process (unless state corruption occurs).

Table 6.2 analyzes TTST's state validation and error detection surface for the possible state changes introduced by a given update. The first column refers to the nature of the transformation of a particular state object. The second column refers to the ability to validate the state object using state differencing. The third and fourth column characterize the implementation of the resulting forward and backward transfer functions. Finally, the fifth column analyzes the effectiveness in detecting state corruption. For unchanged state objects, state differencing can automatically detect state corruption and transfer functions are automatically provided by the state transfer framework (STF). Note that unchanged state objects do not necessarily have the same representation in the different versions. The memory layout of an updated version does not generally reflect the memory layout of the old version and the presence of pointers can introduce representation differences for some unchanged state objects between the past and future version. State objects with structural changes exhibit similar behavior, with a fully automated transfer and validation strategy. With structural changes, we refer to state changes that affect only the type representation and can be entirely arbitrated from the STF with no user intervention (§6.5). This is in contrast with semantic changes, which require user-provided transfer code and can only be partially automated by the STF (§6.5). Semantic state changes highlight the tradeoff between state validation coverage and the manual effort required by the user. In a traditional live update scenario, the user would normally only provide a forward transfer function. This behavior is seamlessly supported by TTST, but the transferred state object will not be considered

```
function STATE_DIFF($pid1, pid2$)
    $a \leftarrow addr\_start$
    while $a < shadow\_start$ do
        $m1 \leftarrow$ IS_MAPPED_WRITABLE($a, pid1$)
        $m2 \leftarrow$ IS_MAPPED_WRITABLE($a, pid2$)
        if $m1$ or $m2$ then
            if $m1 \neq m2$ then
                return $true$
            if MEMPAGECMP($a, pid1, pid2$) $\neq 0$ then
                return $true$
        $a \leftarrow a + page\_size$
    return $false$
```

Figure 6.2: State differencing pseudocode.

for validation. If the user provides code for the reverse transformation, however, the transfer can be normally validated with no restriction. In addition, the backward transfer function provided can be used to perform a cold rollback from the future version to the past version (i.e., live updating the new version into the old version at a later time, for example when the administrator experiences an unacceptable performance slowdown in the updated version). Dropped state objects, in turn, do not require any explicit transfer functions and are automatically validated by state differencing as discussed earlier. State objects that are added in the update (e.g., a new global variable), finally, cannot be automatically validated by state differencing and their validation and transfer is delegated to the STF (§6.5) or to the user.

6.4.3 Blackbox validation

TTST follows a blackbox validation model, which completely ignores ST internals. This is important for two reasons. First, this provides the ability to support many possible updates and ST implementations. This also allows one to evaluate and compare different STFs. Second, this is crucial to decouple the validation logic from the ST implementation, minimizing the amount of trusted code required by our strategy. In particular, our design goals dictate the minimization of the *reliable computing base* (*RCB*), defined as the core software components that are necessary to ensure correct implementation behavior [83]. Our fault model requires four primary components in the RCB: the update timing mechanisms, the TTST arbitration logic, the run-time error detection mechanisms, and the state differencing logic. All the other software components which run in the future and reversed versions (e.g., ST code and CFT code) are fully untrusted thanks to our design.

The implementation of the update timing mechanisms is entirely delegated to the live update library and its size subject to the particular live update tool considered. We trust that every reasonable update timing implementation will have a small RCB impact. For the other TTST components, we seek to reduce the code size (and complexity) to the minimum. Luckily, our TTST arbitration logic and run-time error detection mechanisms (described earlier) are straightforward and only marginally contribute to the RCB. In addition, TTST's semantics-preserving ST transaction and structural equivalence between the final (reversed) state and the original (past) state ensure that the memory images of the two versions are always identical in error-free ST runs. This drastically simplifies our state differencing strategy, which can be implemented using trivial word-by-word memory comparison, with no other knowledge on the ST code and marginal RCB impact. Our comparison strategy examines all the writable regions of the address space excluding only private shadow stack-/heap regions (mapped at the end of the address space) in use by the TTST control library. Figure 6.2 shows the pseudocode for this simple strategy.

6.4.4 State transfer interface

TTST's state transfer interface seeks to minimize the requirements and the effort to implement the STF. In terms of requirements, TTST demands only a *layout-aware* and *user-aware* STF semantic. By layout-aware, we refer to the ability of the STF to preserve the original state layout when requested (i.e., in the reversed version), as well as to automatically identify the state changes described in Table 6.2. By user-aware, we refer to the ability to allow the user to selectively specify new forward and backward transfer functions and candidate state objects for validation. To reduce the effort, TTST offers a convenient STF programming model, with an error handling-friendly environment—our fault-tolerant design encourages undiscriminated use of assertions—and a generic interprocess communication (IPC) interface. In particular, TTST implements an IPC *control* interface to coordinate the TTST transaction and an IPC *data* interface to grant read-only access to the state of a given process version to the others. These interfaces are currently implemented by UNIX domain sockets and POSIX shared memory (respectively), but other IPC mechanisms can be easily supported. The current implementation combines fast data transfer with a secure design that prevents impersonation attacks (access is granted only to the predetermined process instances).

6.5 State Transfer Framework

Our state transfer framework seeks to automate all the possible ST steps, leaving only the undecidable cases (e.g., semantic state changes) to the user. The implementation described here optimizes and extends our prior work [104; 108; 113; 109] to the TTST model. We propose a STF design that resembles a *moving*, *mutating*, and *interprocess* garbage collection model. By moving, we refer to the ability to relocate (and possibly reallocate) static and dynamic state objects in the next version. This is to allow arbitrary changes in the memory layout between versions. By mutating, we refer to the ability to perform on-the-fly type transformations when transferring every given state object from the previous to the next version. Interprocess, finally, refers to our process-level ST strategy. Our goals raise 3 major challenges for a low-level language like C. First, our moving requirement requires precise object and pointer analysis at runtime. Second, on-the-fly type transformations require the ability to dynamically identify, inspect, and match generic data types. Finally, our interprocess strategy requires a mechanism to identify and map state objects across process versions.

6.5.1 Overview

To meet our goals, our STF uses a combination of static and dynamic ST instrumentation. Our static instrumentation, implemented by a LLVM link-time pass [179], transforms each program version to generate *metadata* information that surgically describes the entirety of the program state. In particular, static metadata, which provides *relocation* and *type* information for all the static state objects (e.g., global variables, strings, functions with address taken), is embedded directly into the final binary. Dynamic metadata, which provides the same information for all the dynamic state objects (e.g., heap-allocated objects), is, in turn, dynamically generated/destroyed at runtime by our allocation/deallocation site instrumentation—we currently support `malloc/mmap-like` allocators automatically and standard region-based allocators [46] using user-annotated allocator functions. Further, our pass can dynamically generate/destroy local variable metadata for a predetermined number of functions (e.g., `main`), as dictated by the particular update model considered. Finally, to automatically identify and map objects across process versions, our instrumentation relies on *version-agnostic* state IDs derived from unambiguous *naming* and *contextual* information. In detail, every static object is assigned a static ID derived by its source name (e.g., function name) and scope (e.g., static variable module). Every dynamic object, in turn, is assigned a static ID derived by allocation site information (e.g., caller function name and target pointer name) and an incremental dynamic ID to unambiguously identify allocations at runtime.

Our ID-based naming scheme fulfills TTST's layout-awareness goal: static IDs are used to identify state changes and to automatically reallocate dynamic objects in the future version; dynamic IDs are used to map dynamic objects in the future version with their existing counterparts in the reversed version. The mapping policy to use is specified as part of generic *ST policies*, also implementing other TTST-aware extensions: (i) *randomization* (enabled in the future version): perform fine-grained address space randomization [108] for all the static/dynamically reallocated objects, used to amplify the difference introduced by memory errors in the overall TTST transaction; (ii) *validation* (enabled in the reversed version): zero out the local copy of all the mapped state objects scheduled for automated transfer to detect missing write errors at validation time.

Our dynamic instrumentation, included in a preloaded shared library (ST framework library), complements the static pass to address the necessary run-time tasks:

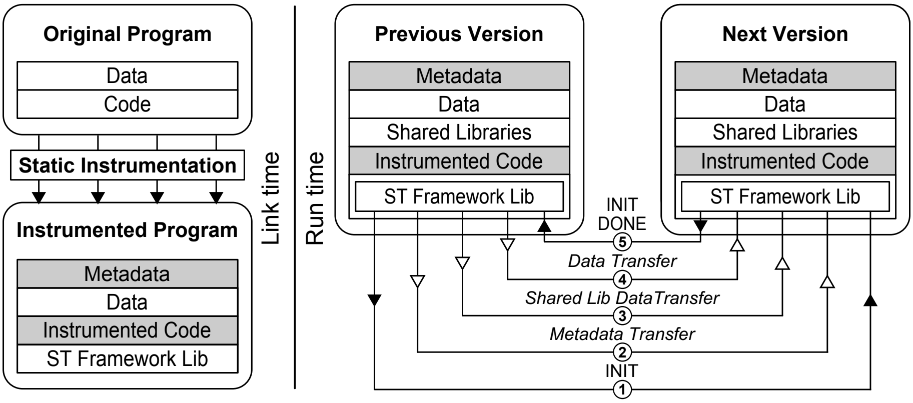
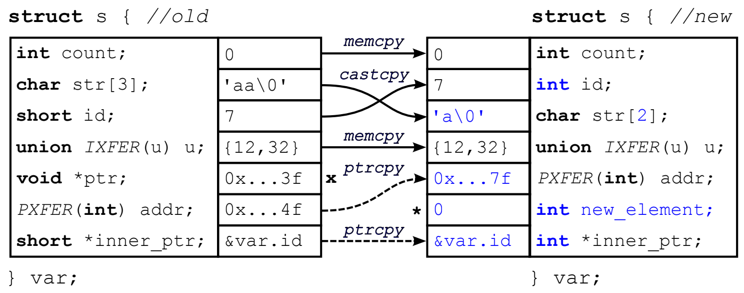


Figure 6.3: State transfer framework overview.

type and pointer analysis, metadata management for shared libraries, error detection. In addition, the ST framework library implements all the steps of the ST process, as depicted in Figure 6.3. The process begins with an initialization request from the TTST control library, which specifies the ST policies and provides access to the TTST's IPC interface. The next *metadata transfer* step transfers all the metadata from the previous version to a cache in the next version (local address space). At the end, the local state objects (and their metadata) are mapped into the external objects described by the metadata cache and scheduled for transfer according to their state IDs and the given ST policies. The next two *data transfer* steps complete the ST process, transferring all the data to reinitialize shared library and program state to the next version. State objects scheduled for transfer are processed one at a time, using metadata information to locate the objects and their internal representations in the two process versions and apply pointer and type transformations on the fly. The last step performs cleanup tasks and returns control to the caller.

6.5.2 State transfer strategy

Our STF follows a well-defined automated ST strategy for all the mapped state objects scheduled for transfer, exemplified in Figure 6.4. As shown in the figure—which reprises the update example given earlier (§6.2)—our type analysis automatically and recursively matches individual type elements between object versions by *name* and *representation*, identifying added/dropped/changed/identical elements on the fly. This strategy automates ST for common structural changes, including: primitive type changes, array expansion/truncation, and addition/deletion/reordering of `struct` members. Our pointer analysis, in turn, implements a generic pointer transfer strategy, automatically identifying (base and interior) pointer targets in the previous version and reinitializing the pointer values correctly in the next version, in spite of type and memory layout changes. To perform efficient pointer lookups, our analysis organizes all the state objects with address taken in a splay tree, an idea pre-



```
struct s { //old
  int count;
  char str[3];
  short id;
  union IXFER(u) u;
  void *ptr;
  PXFER(int) addr;
  short *inner_ptr;
} var;
```

```
struct s { //new
  int count;
  int id;
  char str[2];
  union IXFER(u) u;
  PXFER(int) addr;
  int new_element;
  int *inner_ptr;
} var;
```

Figure 6.4: Automated state transfer example for the data structure presented in Listing 6.2.

viously explored by bounds checkers [251; 88; 29]. We also support all the special pointer idioms allowed by C (e.g., guard pointers) automatically, with the exception of cases of "*pointer ambiguity*" [104].

To deal with ambiguous pointer scenarios (e.g., `unions` with inner pointers and pointers stored as integers) as well as more complex state changes (e.g., semantic changes), our STF supports user extensions in the form of preprocessor annotations and callbacks. Figure 6.4 shows an example of two ST annotations: `IXFER` (force memory copying with no pointer transfer) and `PXFER` (force pointer transfer instead of memory copying). Callbacks, in turn, are evaluated whenever the STF maps or traverses a given object or type element, allowing the user to override the default mapping behavior (e.g., for renamed variables) or express sophisticated state transformations at the object or element level. Callbacks can be also used to: (i) override the default validation policies, (ii) initialize new state objects; (iii) instruct the STF to checksum new state objects after initialization to detect memory errors at the end of the ST process.

6.5.3 Shared libraries

Uninstrumented shared libraries (*SLs*) pose a major challenge to our pointer transfer strategy. In particular, failure to reinitialize SL-related pointers correctly in the future version would introduce errors after live update. To address this challenge, our STF distinguishes 3 scenarios: (i) program/SL pointers into static SL state; (ii) program/SL pointers into dynamic SL state; (iii) SL pointers into static or dynamic program state. To deal with the first scenario, our STF instructs the dynamic linker to remap all the SLs in the future version at the same addresses as in the past version, allowing SL data transfer (pointer transfer in particular) to be implemented via simple memory copying. SL relocation is currently accomplished by prelinking the SLs on demand when starting the future version, a strategy similar to "*retouching*" for mobile applications [57]. To address the second scenario, our dynamic instrumentation intercepts all the memory management calls performed by SLs and generates dedicated metadata to reallocate the resulting objects at the same address in the future version. This is done by restoring the original heap layout (and content) as part of the SL data transfer phase. To perform heap randomization and type transformations correctly for all the program allocations in the future version, in turn, we allow the STF to deallocate (and reallocate later) all the non-SL heap allocations right after SL data transfer. To deal with the last scenario, we need to accurately identify all the SL pointers into the program state and update their values correctly to reflect the memory layout of the future version. Luckily, these cases are rare and we can envision library developers exporting a public API that clearly marks long-lived pointers into the program state once our live update technique is deployed. A similar API is desirable to mark all the process-specific state (e.g., *libc*'s cached pids) that should be restored after ST—note that shareable resources like file descriptors are, in contrast, automatically transferred by the `fork/exec` paradigm. To automate the identification of these cases in our current prototype, we used conservative pointer analysis techniques [55; 56] under stress testing to locate long-lived SL pointers into the program state and state differencing at `fork` points to locate process-specific state objects.

6.5.4 Error detection

To detect certain classes of semantic errors that escape TTST's detection strategy, our STF enforces *program state invariants* [109] derived from all the metadata available at runtime. Unlike existing *likely* invariant-based error detection techniques [97; 127; 89; 22; 233], our invariants are conservatively computed from static analysis and allow for no false positives. The majority of our invariants are enforced by our dynamic pointer analysis to detect semantic errors during pointer transfer. For example, our STF reports invariant violation (and aborts ST by default) whenever a pointer target no longer exists or has its address taken (according to our static analysis) in the new version. Another example is a transferred pointer that points to an illegal target type according to our static pointer cast analysis.

6.6 Evaluation

We have implemented TTST on Linux (x86), with support for generic user-space C programs using the ELF binary format. All the platform-specific components, however, are well isolated in the TTST control library and easily portable to other operating systems, architectures, and binary formats other than ELF. We have integrated address space randomization techniques developed in prior work [108] into our ST instrumentation and configured them to randomize the location of all the static and dynamically reallocated objects in the future version. To evaluate TTST, we have also developed a live update library mimicking the behavior of state-of-the-art live update tools [132], which required implementing preannotated per-thread update points to control update timing, manual control migration to perform CFT, and a UNIX domain sockets-based interface to receive live update commands from our `ttst-ctl` tool.

We evaluated the resulting solution on a workstation running Linux v3.5.0 (x86), with a 4-core 3.0Ghz AMD Phenom II X4 B95 processor and 8GB of RAM. For our evaluation, we first selected Apache httpd (v.2.2.23) and nginx (v0.8.54), the two most popular open-source web servers. For comparison purposes, we also considered vsftpd (v1.1.0) and the OpenSSH daemon (v3.5p1), a popular open-source ftp and ssh server, respectively. The former [214; 69; 215; 193; 131; 132; 136] and the latter [214; 69; 136] are by far the most used server programs (and versions) in prior work in the field. We annotated all the programs considered to match the implemented live update library as described in prior work [132; 136]. For Apache httpd and nginx, we redirected all the calls to custom allocation routines to the standard allocator interface (i.e., `malloc/free` calls), given that our current instrumentation does not yet support custom allocation schemes based on nested regions [46] (Apache httpd) and slab-like allocations [58] (nginx). To evaluate our programs, we performed tests using the Apache benchmark (AB) [1] (Apache httpd and nginx), dkftpbench [6] (vsftpd), and the provided regression test suite (OpenSSH). We configured our programs and benchmarks using the default settings. We repeated all our experiments 21 times and reported the median—with negligible standard deviation measured across multiple test runs.

Our evaluation answers five key questions: (i) *Performance*: Does TTST yield low run-time overhead and reasonable update times? (ii) *Memory usage*: How much memory do our instrumentation techniques use? (iii) *RCB size*: How much code is (and is not) in the RCB? (iv) *Fault tolerance*: Can TTST withstand arbitrary failures in our fault model? (v) *Engineering effort*: How much effort does TTST require?

6.6.1 Performance

To evaluate the overhead imposed by our update mechanisms, we first ran our benchmarks to compare our base programs with their instrumented and annotated versions. Our experiments showed no appreciable performance degradation. This is expected, since update points only require checking a flag at the top of long-running loops and metadata is efficiently managed by our ST instrumentation. In detail, our static metadata—used only at update time—is confined in a separate ELF section so as not to disrupt locality. Dynamic metadata management, in turn, relies on in-band descriptors to minimize the overhead on allocator operations. To evaluate the latter, we instrumented all the C programs in the SPEC CPU2006 benchmark suite. The results evidenced a 4% average run-time overhead across all the benchmarks. We also measured the cost of our instrumentation on 10,000 `malloc/free` and `mmap/munmap` repeated `glibc` allocator operations—which provide worst-case results, given that common allocation patterns generally yield poorer locality. Experiments with multiple allocation sizes (0-16MB) reported a maximum overhead of 41% for `malloc`,

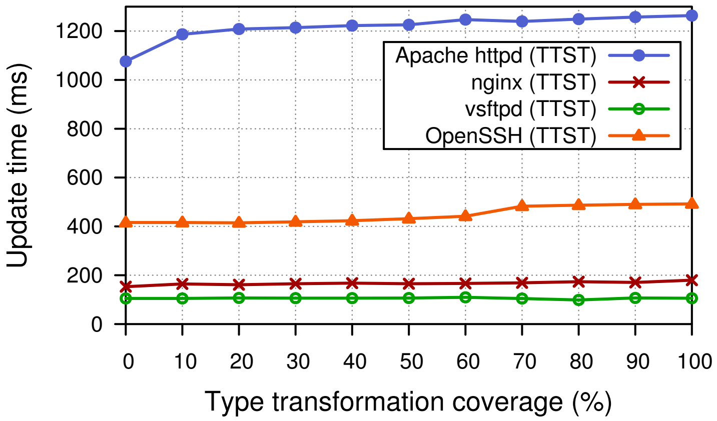


Figure 6.5: Update time vs. type transformation coverage.

9% for `free`, 77% for `mmap`, and 42% for `munmap`. While these microbenchmark results are useful to evaluate the impact of our instrumentation on allocator operations, we expect any overhead to be hardly visible in real-world server programs, which already strive to avoid expensive allocations on the critical path [46].

When compared to prior user-level solutions, our performance overhead is much lower than more intrusive instrumentation strategies—with worst-case macrobenchmark overhead of 6% [214], 6.71%[213], and 96.4% [193]—and generally higher than simple binary rewriting strategies [32; 69]—with worst-case function invocation overhead estimated around 8% [194]. Unlike prior solutions, however, our overhead is strictly isolated in allocator operations and never increases with the number of live updates deployed over time. Recent program-level solutions that use minimal instrumentation [132]—no allocator instrumentation, in particular—in turn, report even lower overheads than ours, but at the daunting cost of annotating all the pointers into heap objects.

We also analyzed the impact of process-level TTST on the update time—the time from the moment the update is signaled to the moment the future version resumes execution. Figure 6.5 depicts the update time—when updating the master process of each program—as a function of the number of type transformations operated by our ST framework. For this experiment, we implemented a source-to-source transformation able to automatically change 0-1,327 type definitions (adding/reordering `struct` fields and expanding arrays/primitive types) for Apache httpd, 0-818 type definitions for nginx, 0-142 type definitions for vsftpd, and 0-455 type definitions for OpenSSH between versions. This forced our ST framework to operate an average of 1,143,981, 111,707, 1,372, and 206,259 type transformations (respectively) at 100% coverage. As the figure shows, the number of type transformations has a steady but low impact on the update time, confirming that the latter is heavily dominated by memory copying and pointer analysis—albeit optimized with splay trees. The data

Type	httpd	nginx	vsftpd	OpenSSH
Static	2.187	2.358	3.352	2.480
Run-time	3.100	3.786	4.362	2.662
Forward ST	3.134	5.563	6.196	4.126
TTST	3.167	7.340	8.031	5.590

Table 6.3: Memory usage (measured statically or at runtime) normalized against the baseline.

points at 100% coverage, however, are a useful indication of the upper bound for the update time, resulting in 1263 ms, 180 ms, 112 ms, and 465 ms (respectively) with our TTST update strategy. Apache httpd reported the longest update times in all the configurations, given the greater amount of state transferred at update time. Further, TTST update times are, on average, 1.76x higher than regular ST updates (not shown in figure for clarity), acknowledging the impact of backward ST and state differencing on the update time. While our update times are generally higher than prior solutions, the impact is bearable for most programs and the benefit is stateful *fault-tolerant* version updates.

6.6.2 Memory usage

Our state transfer instrumentation leads to larger binary sizes and run-time memory footprints. This stems from our metadata generation strategy and the libraries required to support live update. Table 6.3 evaluates the impact on our test programs. The static memory overhead (235.2% worst-case overhead for vsftpd) measures the impact of our ST instrumentation on the binary size. The run-time overhead (336.2% worst-case overhead for vsftpd), in turn, measures the impact of instrumentation and support libraries on the virtual memory size observed at runtime, right after server initialization. These measurements have been obtained starting from a baseline virtual memory size of 234 MB for Apache httpd and less than 6 MB for all the other programs. The third and the fourth rows, finally, show the maximum virtual memory overhead we observed at live update time for both regular (forward ST only) and TTST updates, also accounting for all the transient process instances created (703.1% worst-case overhead for vsftpd and TTST updates). While clearly program-dependent and generally higher than prior live update solutions, our measured memory overheads are modest and, we believe, realistic for most systems, also given the increasingly low cost of RAM in these days.

6.6.3 RCB size

Our TTST update technique is carefully designed to minimize the RCB size. Table 6.4 lists the LOC required to implement every component in our design and the

Component	**RCB**	**Other**
ST instrumentation	1,119	8,211
Live update library	235	147
TTST control library	412	2,797
ST framework	0	13,311
`ttst-ctl` tool	0	381
Total	1,766	24,847

Table 6.4: Contribution to the RCB size (LOC) for every component in our architecture.

contributions to the RCB. Our ST instrumentation requires 1,119 RCB LOC to perform dynamic metadata management at runtime. Our live update library requires 235 RCB LOC to implement the update timing mechanisms and interactions with client tools. Our TTST control library requires 412 RCB LOC to arbitrate the TTST process, implement run-time error detection, and perform state differencing—all from the past version. Our ST framework and `ttst-ctl` tool, in contrast, make no contribution to the RCB. Overall, our design is effective in producing a small RCB, with only 1,766 LOC compared to the other 26,613 non-RCB LOC. Encouragingly, our RCB is even substantially smaller than that of other systems that have already been shown to be amenable to formal verification [168]. This is in stark contrast with all the prior solutions, which make no effort to remove *any* code from the RCB.

6.6.4 Fault tolerance

We evaluated the fault tolerance of TTST using software-implemented fault injection (SWIFI) experiments. To this end, we implemented another LLVM pass which injects specific classes of software faults into predetermined program code regions. Our pass accepts a list of target program functions/modules, the fault types to inject, and a fault probability ϕ—which specifies how many fault locations should be randomly selected for injection out of all the possible candidates found in the code. We configured our pass to randomly inject faults in the ST code, selecting $\phi = 1\%$—although we observed similar results for other ϕ values—and fault types that matched common programming errors in our fault model. In detail, similar to prior SWIFI strategies that evaluated the effectiveness of fault-tolerance mechanisms against state corruption [218], we considered generic *branch* errors (branch/loop condition flip or stuck-at errors) as well as common memory errors, such as *uninitialized reads* (emulated by missing initializers), *pointer corruption* (emulated by corrupting pointers with random or off-by-1 values), *buffer overflows* (emulated by extending the size passed to data copy functions, e.g., `memcpy`, by 1-100%), and *memory leakage* (emulated by missing deallocation calls). We repeated our experiments 500 times for each of the 5 fault types considered, with each run starting a live update between randomized program versions and reporting the outcome of our

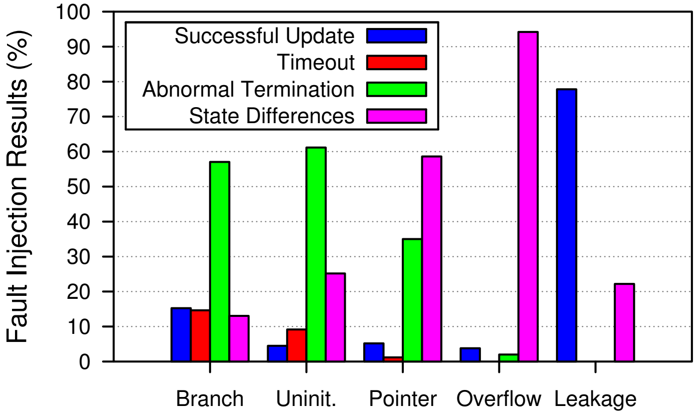


Figure 6.6: TTST behavior in our automated fault injection experiments for varying fault types.

TTST strategy. We report results only for vsftpd—although we observed similar results for the other programs—which allowed us to collect the highest number of fault injection samples per time unit and thus obtain the most statistically sound results.

Figure 6.6 presents our results breaking down the data by fault type and distribution of the observed outcomes—that is, update succeeded or automatically rolled back after *timeout*, *abnormal termination* (e.g., crash), or past-reversed *state differences* detected. As expected, the distribution varies across the different fault types considered. For instance, branch and initialization errors produced the highest number of updates aborted after a timeout (14.6% and 9.2%), given the higher probability of infinite loops. The first three classes of errors considered, in turn, resulted in a high number of crashes (51.1%, on average), mostly due to invalid pointer dereferences and invariants violations detected by our ST framework. In many cases, however, the state corruption introduced did not prevent the ST process from running to completion, but was nonetheless detected by our state differencing technique. We were particularly impressed by the effectiveness of our validation strategy in a number of scenarios. For instance, state differencing was able to automatically recover from as many as 471 otherwise-unrecoverable buffer overflow errors. Similar is the case of memory leakages—actually activated in 22.2% of the runs—with any extra memory region mapped by our metadata cache and never deallocated immediately detected at state diffing time. We also verified that the future (or past) version resumed execution correctly after every successful (or aborted) update attempt. When sampling the 533 successful cases, we noted the introduction of irrelevant faults (e.g., missing initializer for an unused variable) or no faults actually activated at runtime. Overall, our TTST technique was remarkably effective in detecting and recovering from a significant number of observed failures (1,967 overall), with no consequences for the running program. This is in stark contrast with all the prior solutions, which make *no* effort in this regard.

	Updates		Changes			Engineering effort		
	#	**LOC**	**Fun**	**Var**	**Ty**	**ST Ann LOC**	**Fwd ST LOC**	**Bwd ST LOC**
Apache httpd	5	10,844	829	28	48	79	302	151
nginx	25	9,681	711	51	54	24	335	0
vsftpd	5	5,830	305	121	35	0	21	21
OpenSSH	5	14,370	894	84	33	0	135	127
Total	40	40,725	2,739	284	170	103	793	299

Table 6.5: Engineering effort for all the updates analyzed in our evaluation.

6.6.5 Engineering effort

To evaluate the engineering effort required to deploy TTST, we analyzed a number of official incremental releases following our original program versions and prepared the resulting patches for live update. We considered 5 updates for Apache httpd (v2.2.23-v2.3.8), vsftpd (v1.1.0-v2.0.2), and OpenSSH (v3.5-v3.8), and 25 updates for nginx (v0.8.54-v1.0.15), given that nginx's tight release cycle generally produces patches that are much smaller than those of the other programs considered. Table 6.5 presents our findings. The first two grouped columns provide an overview of our analysis, with the number of updates considered for each program and the number of lines of code (LOC) added, deleted, or modified in total by the updates. As shown in the table, we manually processed more than 40,000 LOC across the 40 updates considered. The second group shows the number of functions, variables, and types changed (i.e., added, deleted, or modified) by the updates, with a total of 2,739, 284, and 170 changes (respectively). The third group, finally, shows the engineering effort (LOC) required to prepare our test programs and our patches for live update. The first column shows the one-time annotation effort required to integrate our test programs with our ST framework. Apache httpd and nginx required 79 and 2 LOC to annotate 12 and 2 `unions` with inner pointers, respectively. In addition, nginx required 22 LOC to annotate a number of global pointers using special data encoding—storing metadata information in the 2 least significant bits. The latter is necessary to ensure precise pointer analysis at ST time. The second and the third column, in turn, show the number of lines of state transfer code we had to manually write to complete forward ST and backward ST (respectively) across all the updates considered. Such ST extensions were necessary to implement complex state changes that could not be automatically handled by our ST framework.

A total of 793 forward ST LOC were strictly necessary to prepare our patches for live update. An extra 299 LOC, in turn, were required to implement backward ST. While optional, the latter is important to guarantee full validation surface for our TTST technique. The much lower LOC required for backward ST (37.7%) is easily explained by the additive nature of typical state changes, which frequently entail only adding new data structures (or fields) and thus rarely require extra LOC in our backward ST transformation. The case of nginx is particularly emblematic. Its disciplined update strategy, which limits the number of nonadditive state changes to the minimum, translated to no manual ST LOC required to implement backward ST. We believe this is particularly encouraging and can motivate developers to deploy our TTST techniques with full validation surface in practice.

6.7 Related Work

6.7.1 Live update systems

We focus on *local* live update solutions for generic and widely deployed C programs, referring the reader to [26; 279; 173; 91; 25] for distributed live update systems. LUCOS [68], DynaMOS [194], and Ksplice [36] have applied live updates to the Linux kernel, loading new code and data directly into the running version. Code changes are handled using binary rewriting (i.e., trampolines). Data changes are handled using shadow [194; 36] or parallel [68] data structures. OPUS [32], POLUS [69], Ginseng [214], STUMP [213], and Upstare [193] are similar live update solutions for user-space C programs. Code changes are handled using binary rewriting [32; 69], compiler-based instrumentation [214; 213], or stack reconstruction [193]. Data changes are handled using parallel data structures [69], type wrapping [214; 213], or object replacement [193]. Most solutions delegate ST entirely to the programmer [68; 194; 36; 32; 69], others generate only basic type transformers [214; 213; 193]. Unlike TTST, none of these solutions attempt to fully automate ST—pointer transfer, in particular—and state validation. Further, their in-place update model hampers isolation and recovery from ST errors, while also introducing address space fragmentation over time. To address these issues, alternative update models have been proposed. Prior work on process-level live updates [124; 131], however, delegates the ST burden entirely to the programmer. In another direction, Kitsune [132] encapsulates every program in a hot swappable shared library. Their state transfer framework, however, does not attempt to automate pointer transfer without user effort and no support is given to validate the state or perform safe rollback in case of ST errors. Finally, our prior work [108; 113] demonstrated the benefits of process-level live updates in component-based OS architectures, with support to recover from run-time ST errors but no ability to detect a corrupted state in the updated version.

6.7.2 Live update safety

Prior work on live update safety is mainly concerned with safe update timing mechanisms, neglecting important system properties like fault tolerance and RCB minimization. Some solutions rely on *quiescence* [32; 42; 43; 36] (i.e., no updates to active code), others enforce *representation consistency* [214; 213; 267] (i.e., no updated code accessing old data). Other researchers have proposed using transactions in local [215] or distributed [173; 279] contexts to enforce stronger timing constraints. Recent work [135], in contrast, suggests that many researchers may have been overly concerned with update timing and that a few predetermined update points [214; 213; 132; 131; 108; 113] are typically sufficient to determine safe and timely update states. Unlike TTST, none of the existing solutions have explicitly addressed ST-specific update safety properties. Static analysis proposed in OPUS [32]—to detect unsafe data updates—and Ginseng [214]—to detect unsafe pointers into updated objects—is somewhat related, but it is only useful to *disallow* particular classes of (unsupported) live updates.

6.7.3 Update testing

Prior work on live update testing [133; 136; 134] is mainly concerned with validating the correctness of an update in all the possible update timings. Correct execution is established from manually written specifications [134] or manually selected program output [133; 136]. Unlike TTST, these techniques require nontrivial manual effort, are only suitable for offline testing, and fail to validate the entirety of the program state. In detail, their state validation surface is subject to the coverage of the test programs or specifications used. Their testing strategy, however, is useful to compare different update timing mechanisms, as also demonstrated in [136]. Other related work includes online patch validation, which seeks to efficiently compare the behavior of two (original and patched) versions at runtime. This is accomplished by running two separate versions in parallel [62; 199; 148] or a single hybrid version using a split-and-merge strategy [277]. These efforts are complementary to our work, given that their goal is to test for errors in the patch itself rather than validating the state transfer code required to prepare the patch for live update. Complementary to our work are also efforts on upgrade testing in large-scale installations, which aim at creating sandboxed deployment-like environments for testing purposes [298] or efficiently testing upgrades in diverse environments using staged deployment [79]. Finally, fault injection has been previously used in the context of update testing [209; 224; 91], but only to emulate upgrade-time operator errors. Our evaluation, in contrast, presents the first fault injection campaign that emulates realistic programming errors in the ST code.

6.8 Conclusion

While long recognized as a hard problem, state transfer has received limited attention in the live update literature. Most efforts focus on automating and validating update timing, rather than simplifying and shielding the state transfer process from programming errors. We believe this is a key factor that has discouraged the system administration community from adopting live update tools, which are often deemed impractical and untrustworthy.

This paper presented *time-traveling state transfer*, the first fault-tolerant live update technique which allows generic live update tools for C programs to automate and validate the state transfer process. Our technique combines the conventional forward state transfer transformation with a backward (and logically redundant) transformation, resulting in a semantics-preserving manipulation of the original program state. Observed deviations in the reversed state are used to automatically identify state corruption caused by common classes of programming errors (i.e., memory errors) in the state transfer (library or user) code. Our process-level update strategy, in turn, guarantees detection of other run-time errors (e.g., crashes), simplifies state management, and prevents state transfer errors to propagate back to the original version. The latter property allows our framework to safely recover from errors and automatically resume execution in the original version. Further, our modular and blackbox validation design yields a minimal-RCB live update system, offering a high fault-tolerance surface in both online and offline validation runs. Finally, we complemented our techniques with a generic state transfer framework, which automates state transformations with minimal programming effort and can detect additional semantic errors using statically computed invariants. We see our work as the first important step toward truly practical and trustworthy live update tools for system administrators.

6.9 Acknowledgments

We would like to thank our shepherd, Mike Ciavarella, and the anonymous reviewers for their comments. This work has been supported by European Research Council under grant ERC Advanced Grant 227874.

8

Conclusion

Despite nearly 40 years of research in the area, with Robert Fabry authoring the first paper on the subject in 1976 [99], existing live update solutions still suffer from important dependability and usability limitations that hinder their widespread adoption. Fabry describes two fundamental subproblems in the design of live update systems. The first subproblem is concerned with the mechanisms to implement live update, that is, in Fabry's own words: "*the mechanics of constructing a system in such a way that the programs and the data structures which they manage can be changed without stopping the system.*" The second subproblem, in turn, is concerned with live update safety, that is, in Fabry's own words: "*how one convinces oneself that a change will operate correctly if it is installed.*" This dissertation makes important contributions for both subproblems, investigating general techniques for truly *safe* and *automatic* live update both at the OS and application level. In the following, we summarize the key results of this dissertation.

1. ***Process-level updates***. We presented *process-level updates*, a new live update technique which confines different program versions in distinct processes and automatically performs state transfer and control migration between them. Process-level updates provide three major improvements over prior techniques. First, they allow version updates to be installed in the most natural way with no need for complex—and potentially unsafe—patch analysis and preparation tools. Second, they preserve the original internal representation of the program, without inhibiting compiler optimizations or introducing address space fragmentation over time. Finally, they allow updates between program versions with arbitrarily different code and data layout, posing no restrictions on the nature of the updates supported. We presented process-level update implementations at the OS level (in PROTEOS) and at the application level (on Linux), demonstrating their performance and updatability benefits.

2. ***Automated state transfer***. We presented a new *state transfer framework*, complementing our process-level update implementation with a generic mechanism to remap the program state between versions. Ours is the first state transfer framework that supports updates between arbitrarily different program versions, operating type and pointer transformations on the fly with little user intervention. The framework fully automates state transfer for many common structural changes and supports a convenient programming model for more complex user extensions. Unlike prior solutions, our framework can automatically support all the standard programming idioms allowed by C, including interior pointers, `void*` pointers, and custom memory allocation schemes. In addition, our *mutable GC-style tracing* strategy can automatically transfer "*hidden pointers*" with no extra annotations required. We integrated the proposed framework in our process-level update implementation, evaluating the update time and the engineering effort required to deploy our techniques.

3. ***Automated control migration***. We presented *state-driven mutable record-replay*, a new technique to automatically perform control migration between program versions. Our technique provides two key improvements over prior solutions. First, it drastically reduces the engineering effort, reusing existing initialization-time code paths and only resorting to manual annotations for nonpersistent quiescent points. Second, it provides support for generic multiprocess multithreaded programs, automatically reconstructing the process/thread hierarchy while tolerating changes in the thread behavior. We implemented state-driven mutable record-replay support for generic C programs on Linux, demonstrating its effectiveness on real-world server applications. In PROTEOS, in contrast, we relied on a well-defined event-driven model to automatically migrate control between program versions, confirming that conscious system design can significantly simplify the live update problem.

4. ***Fault-tolerant live update***. We presented three new techniques which substantially improve the fault tolerance guarantees of prior live update solutions. *Hot rollback* relies on our process-level abstraction to implement fault-tolerant live update transactions, with the ability to detect run-time errors and safely resume execution in the original version when needed. *Program state invariants* rely on data invariants derived from static analysis to extend our error detection guarantees to several classes of logical and memory errors. *Time-traveling state transfer*, finally, implements multi-version and semantics-preserving state transfer transactions to detect arbitrary memory errors with a minimal amount of trusted code. We evaluated our techniques both at the OS and at the application level, conducting a number of fault and error injection experiments to assess their fault-tolerance properties. Chapter 6, in particular, presented the first fault injection campaign on live update code.

5. ***System support for safe update state detection***. We presented two new techniques to detect safe update states. PROTEOS's event-driven design provides support for *state quiescence*, a generic quiescence mechanism that gives users fine-grained control over update states. In particular, users can specify *state filters* to express the conditions under which an update is possible and *interface filters* to defer delivery of particular events that may otherwise weaken convergence guarantees. We demonstrated the benefits of our techniques in complex update scenarios involving a number of interconnected components, where adequate system support is crucial to simplify reasoning on update safety. *Profile-guided quiescence detection*, in turn, provides a general quiescence detection technique suitable for legacy C programs. Unlike prior solutions, our technique relies on profile-guided program instrumentation to provide strong convergence guarantees with little annotation effort. We implemented profile-guided quiescence detection support for generic C programs on Linux, demonstrating its effectiveness and scalability properties on real-world server applications.

6. ***Memory leakage reclaiming***. We presented *memory leakage reclaiming*, a new self-healing application of live update. Memory leakage reclaiming relies on our live update techniques to implement a poor man's garbage collector. The idea is to periodically allow same-version live updates to automatically reclaim memory leakage using the heap traversal strategy implemented by our state transfer framework. Unlike traditional garbage collectors, our implementation relies on process-level updates to guarantee a fully untrusted design. We implemented our ideas in PROTEOS and evaluated the resulting accuracy-performance tradeoffs.

7. ***Live rerandomization***. We presented *live rerandomization*, a new systems security application of live update. Live rerandomization combines our live update techniques with fine-grained address space randomization to implement an online diversification strategy. The idea is to periodically allow live updates between semantically-equivalent program variants with arbitrarily different memory layouts. This strategy improves the security of prior address space randomization solutions in face of information disclosure vulnerabilities, minimizing the knowledge acquired by an attacker probing the system. We implemented our ideas in PROTEOS and evaluated the resulting frequency-performance tradeoffs.

Future Directions

With the live update problem proven to be undecidable in the general case [125], the techniques presented in this dissertation can be hardly considered conclusive for the design of truly *safe* and *automatic* live update systems. The job will be finished when the average user has never experienced a "*restart*" update in his lifetime and update alerts have passed into history. In the following, we highlight a number of opportunities for future research directions.

1. ***Complex state transformations***. Our state transfer framework can automatically operate common structural state transformations, but requires user intervention to handle more complex semantic state changes. While fully automating the process for arbitrarily complex updates is unrealistic, it may be valuable to investigate techniques to handle nontrivial state changes for many common cases of interest. A promising direction is to rely on patch analysis strategies to automatically generate state transformation stubs, for example by drawing inspiration from live patch testing [277; 199] or impact analysis [223] techniques. Recent work on generic Java programs has presented encouraging results in this direction [192].

2. ***Complex control transformations***. A limitation of our current state-driven mutable record-replay implementation is the inability to support nonpersistent quiescent points without user intervention. To overcome this limitation, it may be valuable to investigate techniques to efficiently automate control migration for arbitrary quiescent points, for example, by extending our state-driven mutable record-replay strategy to specific code paths and operations determined via program analysis. Other challenges are to automatically (or semiautomatically) support control migration for programs with a nondeterministic process model and for software updates that induce changes in the original quiescent behavior of the program. For the latter, a promising direction is to record program runs leading to valid quiescent configurations and rely on general-purpose mutable record-replay techniques [280] to replay the recorded executions and generate all the relevant thread configuration mappings across versions.

3. ***Safe quiescence***. The quiescence mechanisms presented in this dissertation provide system support for stable and predictable update states, but cannot alone identify errors in user-driven policies. For example, if the user specifies incorrect state filters or quiescent points, the system may never reach quiescence in bounded time. To improve the convergence guarantees, it may be valuable to investigate techniques to validate or reduce user input. For validation purposes, a promising direction is to rely on live update testing tools [133; 136] to stress and verify all the legal quiescence configurations encountered at runtime, possibly leveraging symbolic execution tools [63] to improve the coverage of the analysis. To reduce user input for state quiescence, an option is to suggest likely state filters to the user, for example, by relying on patch impact analysis tools [223] to determine systems states that are minimally affected by an update. To reduce user input for profile-guided quiescence, in turn, an option is to adopt more sophisticate strategies to detect blocking calls listening for internal events, for example, by relying on system-wide dynamic taint analysis [240; 80; 290; 81; 295] to track event propagation in a cross-process fashion.

4. ***Automatic allocator instrumentation***. Our state transfer framework relies on accurate allocator instrumentation to implement precise pointer analysis and increase the state transformation coverage, ultimately reducing the effort to support live update over time. The current implementation provides support for the standard allocator and for user-annotated region-based allocation functions [46]. To reduce the engineering effort, it may be valuable to investigate techniques to efficiently and automatically instrument generic allocator implementations. A first step is to implement metadata management for currently unsupported custom allocation schemes—e.g., slabs [58]. A second step is to investigate techniques to automatically identify custom allocation functions and completely eliminate the need for user annotations. Prior work on specification inference [175] and reverse engineering [287] has presented encouraging results in this direction.

5. ***Advanced error detection***. Our state management techniques provide strong fault-tolerance guarantees at live update time, but cannot alone detect and recover from arbitrary errors outside our fault model. We envision three main directions to improve the error detection and recovery coverage of our techniques. A first direction is to investigate techniques for tainted state management, important for bug fix updates that may be deployed with the old version already in a tainted state. Promising solutions are self-healing techniques specific to particular classes of errors—for example, infinite loop escape [169]—or more generic automated program repair techniques—a subject which has been the focus of much recent research [159; 282; 255; 235; 180; 188; 220]. A second direction is to investigate techniques to detect complex semantic errors at live update time, for example, by relying on more sophisticated static analysis techniques—for example, range analysis [249]—to improve the accuracy of our program state invariants. A third direction is to investigate techniques to efficiently detect errors in the update itself, for example by relying on live patch testing [277; 199], symbolic patch testing [197], or multi-version execution [148].

6. ***Efficient state transfer***. Our process-level state transfer strategy may yield substantial update times for programs with a very large and continuously modified state, a scenario in which our dirty state tracking optimization, in particular, may prove less effective. While not a concern for update safety, a lengthy state transfer process may appreciably weaken availability or real-time guarantees for particular systems. We envision three main directions to further reduce the state transfer time. A first direction is to investigate techniques to automatically identify state filters that yield a minimal amount of in-flight state to transfer. A second direction is to investigate techniques to speed up our current implementation, for example, by drawing inspiration from parallel garbage collection techniques [37] to parallelize our mutable GC-style tracing strategy. A third direction is to investigate techniques to lazily transform the relevant portions of the state, an idea explored in prior solutions in an in-place live update context [214].

7. ***New applications***. To conclude, another future direction is to investigate the applicability of our live update techniques to other contexts and applications. An option is to rely on our techniques to implement rebootless update support for commodity operating systems, for example, by combining our process-level update strategy with shadow kernel techniques described in prior work [85; 264]. Other directions include applications of live update to fast prototyping or runtime adaptation, for example, to dynamically adapt the behavior of the system to the monitored workload [265]. Another promising direction is to rely on our techniques to implement new live workaround systems [286], similar to the applications proposed in previous chapters to mitigate memory leakage bugs (Chapter 2) and security vulnerabilities (Chapter 3). Prior work has already investigated live workaround techniques for concurrency bugs [286], but, as demonstrated in this dissertation, the general idea has much broader applicability.

References

[1] Apache benchmark (AB). http://httpd.apache.org/docs/2.0/programs/ab.html.

[2] ASLR: Leopard versus Vista. http://blog.laconicsecurity.com/2008/01/aslr-leopard-versus-vista.html.

[3] CRIU. http://criu.org.

[4] Cryopid2. http://sourceforge.net/projects/cryopid2.

[5] Vulnerability summary for CVE-2006-0095. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2006-0095.

[6] dkftpbench. http://www.kegel.com/dkftpbench.

[7] FUSE: Filesystem in userspace. http://fuse.sourceforge.net.

[8] Green hills integrity. http://www.ghs.com/products/rtos/integrity.html.

[9] Ksplice performance on security patches. http://www.ksplice.com/cve-evaluation.

[10] Linux vmsplice vulnerabilities. http://isec.pl/vulnerabilities/isec-0026-vmsplice_to_kernel.txt.

[11] Microsoft Windows TCP/IP IGMP MLD remote buffer overflow vulnerability. http://www.securityfocus.com/bid/27100.

[12] OpenVZ. http://wiki.openvz.org.

[13] pyftpdlib. https://code.google.com/p/pyftpdlib.

[14] The story of a simple and dangerous kernel bug. `http://butnotyet.tumblr.com/post/175132533/the-story-of-a-simple-and-dangerous-kernel-bug`.

[15] SysBench. `http://sysbench.sourceforge.net`.

[16] httperf. `http://www.hpl.hp.com/research/linux/httperf`.

[17] lighttpd. `http://www.lighttpd.net`.

[18] Ksplice Uptrack. `http://www.ksplice.com`.

[19] nginx. `http://nginx.org`.

[20] OpenBSD's IPv6 mbufs remote kernel buffer overflow. `http://www.securityfocus.com/archive/1/462728/30/0/threaded`.

[21] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. and Syst. Secur.*, 13(1):1–40, 2009.

[22] Sarita V Adve, Vikram S Adve, and Yuanyuan Zhou. Using likely program invariants to detect hardware errors. In *Proc. of the Int'l Conf. on Dependable Systems and Networks*, pages 70–79, 2008.

[23] Edward E. Aftandilian, Samuel Z. Guyer, Martin Vechev, and Eran Yahav. Asynchronous assertions. In *Proc. of the 26th ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 275–288, 2011.

[24] Sameer Ajmani. A review of software upgrade techniques for distributed systems.

[25] Sameer Ajmani, Barbara Liskov, and Liuba Shrira. Scheduling and simulation: How to upgrade distributed systems. In *Proc. of the Ninth Workshop on Hot Topics in Operating Systems*, volume 9, pages 43–48, 2003.

[26] Sameer Ajmani, Barbara Liskov, Liuba Shrira, and Dave Thomas. Modular software upgrades for distributed systems. In *Proc. of the 20th European Conf. on Object-Oriented Programming*, pages 452–476, 2006.

[27] Periklis Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *Proc. of the 19th USENIX Security Symp.*, page 12, 2010.

[28] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with WIT. In *Proc. of the IEEE Symp. on Security and Privacy*, pages 263–277, 2008.

[29] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proc. of the 18th USENIX Security Symp.*, pages 51–66, 2009.

[30] João Paulo A. Almeida, Marten van Sinderen, and Lambert Nieuwenhuis. Transparent dynamic reconfiguration for CORBA. In *Proc. of the Third Int'l Symp. on Distributed Objects and Applications*, pages 197–207, 2001.

[31] Gautam Altekar and Ion Stoica. ODR: Output-deterministic replay for multicore debugging. In *Proc. of the 22nd ACM Symp. on Oper. Systems Prin.*, pages 193–206, 2009.

[32] Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz. OPUS: Online patches and updates for security. In *Proc. of the 14th USENIX Security Symp.*, pages 19–19, 2005.

[33] Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. A compiler-level intermediate representation based binary analysis and rewriting system. In *Proc. of the Eighth ACM European Conf. on Computer Systems*, pages 295–308, 2013.

[34] Jakob R. Andersen, Lars Bak, Steffen Grarup, Kasper V. Lund, Toke Eskildsen, Klaus Marius Hansen, and Mads Torgersen. Design, implementation, and evaluation of the resilient Smalltalk embedded platform. *Comput. Lang. Syst. Struct.*, 31(3-4):127–141, 2005.

[35] Jason Ansel, Kapil Arya, and Gene Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *Proc. of the Int'l Symp. on Parallel and Distributed Processing*, pages 1–12, 2009.

[36] Jeff Arnold and M. Frans Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Proc. of the Fourth ACM European Conf. on Computer Systems*, pages 187–198, 2009.

[37] Clement R. Attanasio, David F. Bacon, Anthony Cocchi, and Stephen Smith. A comparative evaluation of parallel garbage collector implementations. In *Proc. of the 14th Int'l Conf. on Languages and Compilers for Parallel Computing*, pages 177–192, 2003.

[38] Kumar Avijit, Prateek Gupta, and Deepak Gupta. TIED, LibsafePlus: tools for runtime buffer overflow protection. In *Proc. of the 13th USENIX Security Symp.*, page 4, 2004.

[39] J. Baker, A. Cunei, T. Kalibera, F. Pizlo, and J. Vitek. Accurate garbage collection in uncooperative environments revisited. *Concurr. Comput.: Pract. Exper.*, 21(12):1572–1606, 2009.

[40] Radu Banabic and George Candea. Fast black-box testing of system recovery code. In *Proc. of the Seventh ACM European Conf. on Computer Systems*, pages 281–294, 2012.

[41] J. H. Barton, E. W. Czeck, Z. Z. Segall, and D. P. Siewiorek. Fault injection

experiments using FIAT. *IEEE Trans. Comput.*, 39(4):575–582, 1990.

[42] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W. Wisniewski, and Jeremy Kerr. Providing dynamic update in an operating system. In *Proc. of the USENIX Annual Tech. Conf.*, page 32, 2005.

[43] Andrew Baumann, Jonathan Appavoo, Robert W. Wisniewski, Dilma Da Silva, Orran Krieger, and Gernot Heiser. Reboots are for hardware: Challenges and solutions to updating an operating system on the fly. In *Proc. of the USENIX Annual Tech. Conf.*, pages 1–14, 2007.

[44] Rida A. Bazzi, Kristis Makris, Peyman Nayeri, and Jun Shen. Dynamic software updates: The state mapping problem. In *Proc. of the Second Int'l Workshop on Hot Topics in Software Upgrades*, page 2, 2009.

[45] Emery D Berger and Benjamin Zorn. DieHard: probabilistic memory safety for unsafe languages. In *Proc. of the 27th ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 158–168, 2006.

[46] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Reconsidering custom memory allocation. In *Proc. of the 17th ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–12, 2002.

[47] B. N. Bershad, S. Savage, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proc. of the 15th ACM Symp. on Oper. Systems Prin.*, pages 267–284, 1995.

[48] Sandeep Bhatkar and R. Sekar. Data space randomization. In *Proc. of the Fifth Int'l Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 1–22, 2008.

[49] Sandeep Bhatkar, Daniel C DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a board range of memory error exploits. In *Proc. of the 12th USENIX Security Symp.*, page 8, 2003.

[50] Sandeep Bhatkar, R. Sekar, and Daniel C DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Proc. of the 14th USENIX Security Symp.*, page 17, 2005.

[51] E. W. Biederman. Multiple instances of the global Linux namespaces. In *Proc. of the Linux Symp.*, 2006.

[52] M. Blair, S. Obenski, and P. Bridickas. Patriot missile defense: Software problem led to system failure at Dhahran. Technical Report GAO/IMTEC-92-26, United States-General Accounting Office-Information Management and

Technology Division, 1992.

[53] T. Bloom. *Dynamic module replacement in a distributed programming system*. PhD thesis, MIT, 1983.

[54] T. Bloom and M. Day. Reconfiguration and module replacement in Argus: Theory and practice. *Software Eng. J.*, 8(2):102–108, 1993.

[55] Hans-J. Boehm. Bounding space usage of conservative garbage collectors. In *Proc. of the 29th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 93–100, 2002.

[56] Hans-Juergen Boehm. Space efficient conservative garbage collection. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 197–206, 1993.

[57] Hristo Bojinov, Dan Boneh, Rich Cannings, and Iliyan Malchev. Address space randomization for mobile devices. In *Proc. of the Fourth ACM Conf. on Wireless Network Security*, pages 127–138, 2011.

[58] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *Proc. of the USENIX Summer Tech. Conf.*, page 6, 1994.

[59] Chandrasekhar Boyapati, Barbara Liskov, Liuba Shrira, Chuang-Hue Moh, and Steven Richman. Lazy modular upgrades in persistent object stores. In *Proc. of the 18th ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 403–417, 2003.

[60] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating malicious device drivers in Linux. In *Proc. of the USENIX Annual Tech. Conf.*, page 9, 2010.

[61] C-skills. Linux udev trickery. `http://c-skills.blogspot.com/2009/04/udev-trickery-cve-2009-1185-and-cve.html`.

[62] C. Cadar and P. Hosek. Multi-version software updates. In *Proc. of the Fourth Int'l Workshop on Hot Topics in Software Upgrades*, pages 36–40, 2012.

[63] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. of the 8th USENIX Symp. on Operating Systems Design and Implementation*, pages 209–224, 2008.

[64] João Carreira, Henrique Madeira, and João Gabriel Silva. Xception: A technique for the experimental evaluation of dependability in modern computers. *IEEE Trans. Softw. Eng.*, 24(2):125–136, 1998.

[65] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proc. of the Seventh USENIX Symp. on Operating Systems Design and Implementation*, pages 147–160, 2006.

[66] Ramesh Chandra, R.M. Lefever, K.R. Joshi, M. Cukier, and W.H. Sanders. A global-state-triggered fault injector for distributed system evaluation. *IEEE Trans. Parallel Distrib. Syst.*, 15(7):593–605, 2004.

[67] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M.F. Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proc. of the Second Asia-Pacific Workshop on Systems*, 2011.

[68] Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang, and Pen-Chung Yew. Live updating operating systems using virtualization. In *Proc. of the Second Int'l Conf. on Virtual Execution Environments*, pages 35–44, 2006.

[69] Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. POLUS: A POwerful live updating system. In *Proc. of the 29th Int'l Conf. on Software Eng.*, pages 271–281, 2007.

[70] Tzicker Chiueh and FuHau Hsu. RAD: A compile-time solution to buffer overflow attacks. In *Proc. of the 21st Int'l Conf. on Distr. Computing Systems*, pages 409–417, 2001.

[71] Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation. In *Proc. of the 14th USENIX Security Symp.*, pages 22–22, 2005.

[72] J. Christmansson and R. Chillarege. Generation of an error set that emulates software faults based on field data. In *Proc. of the 26th Int'l Symp. on Fault-Tolerant Computing*, page 304, 1996.

[73] James Clause, Ioannis Doudalis, Alessandro Orso, and Milos Prvulovic. Effective memory protection using dynamic tainting. In *Proc. of the 22nd IEEE/ACM Int'l Conf. on Automated Software Eng.*, pages 284–292, 2007.

[74] D. Cotroneo, A. Lanzaro, R. Natella, and R. Barbosa. Experimental analysis of binary-level software fault injection in complex software. In *Proc. of the Ninth European Dependable Computing Conf.*, pages 162–172, 2012.

[75] Marco Cova, Davide Balzarotti, Viktoria Felmetsger, and Giovanni Vigna. Swaddler: An approach for the anomaly-based detection of state violations in web applications. In *Proc. of the 10th Int'l Conf. on Recent Advances in Intrusion Detection*, pages 63–86, 2007.

[76] C. Cowan, T. Autrey, C. Krasic, C. Pu, and J. Walpole. Fast concurrent dynamic linking for an adaptive operating system. In *Proc. of the Third Int'l Conf. on Configurable Distributed Systems*, pages 108–115, 1996.

[77] Crispin Cowan, Coltan Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattle, Aaron Grier, Perry Wagle, and Qian Zhang. Stack-Guard: Automatic adaptive detection and prevention of buffer-overflow at-

tacks. In *Proc. of the Seventh USENIX Security Symp.*, page 5, 1998.

[78] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-variant systems: A secretless framework for security through diversity. In *Proc. of the 15th USENIX Security Symp.*, pages 105–120, 2006.

[79] Olivier Crameri, Nikola Knezevic, Dejan Kostic, Ricardo Bianchini, and Willy Zwaenepoel. Staged deployment in Mirage, an integrated software upgrade testing and distribution system. In *Proc. of the 21st ACM Symp. on Oper. Systems Prin.*, pages 221–236, 2007.

[80] Jedidiah R. Crandall and Frederic T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proc. of the 37th Int'l Symp. on Microarchitecture*, pages 221–232, 2004.

[81] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Real-world buffer overflow protection for userspace & kernelspace. In *Proc. of the 17th USENIX Security Symp.*, pages 395–410, 2008.

[82] Francis M David, Ellick M Chan, Jeffrey C Carlyle, and Roy H Campbell. CuriOS: Improving reliability through operating system structure. In *Proc. of the Eighth USENIX Symp. on Operating Systems Design and Implementation*, pages 59–72, 2008.

[83] Björn Döbel, Hermann Härtig, and Michael Engel. Operating system support for redundant multithreading. In *Proc. of the 10th Int'l Conf. on Embedded software*, pages 83–92, 2012.

[84] Ronald F. DeMara, Yili Tseng, and Abdel Ejnioui. Tiered algorithm for distributed process quiescence and termination detection. *IEEE Trans. Parallel Distrib. Syst.*, 18(11):1529–1538, 2007.

[85] Alex Depoutovitch and Michael Stumm. Otherworld: Giving applications a chance to survive OS kernel crashes. In *Proc. of the Fifth ACM European Conf. on Computer systems*, pages 181–194, 2010.

[86] Solar Designer. Getting around non-executable stack (and fix). `http://seclists.org/bugtraq/1997/Aug/63`.

[87] Mathieu Desnoyers, Paul E. McKenney, Alan S. Stern, Michel R. Dagenais, and Jonathan Walpole. User-level implementations of read-copy update. *IEEE Trans. Parallel Distrib. Syst.*, 23(2):375–382, 2012.

[88] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for C with very low overhead. In *Proc. of the 28th Int'l Conf. on Software Eng.*, pages 162–171, 2006.

[89] Martin Dimitrov and Huiyang Zhou. Unified architectural support for soft-

error protection or software bug detection. In *Proc. of the 16th Int'l Conf. on Parallel Architecture and Compilation Techniques*, pages 73–82, 2007.

[90] Dominic Duggan. Type-based hot swapping of running modules. In *Proc. of the Sixth ACM SIGPLAN Int'l Conf. on Functional programming*, pages 62–73, 2001.

[91] Tudor Dumitras and Priya Narasimhan. Why do upgrades fail and what can we do about it?: Toward dependable, online upgrades in enterprise system. In *Proc. of the 10th Int'l Conf. on Middleware*, pages 1–20, 2009.

[92] Tudor Dumitras, Jiaqi Tan, Zhengheng Gho, and Priya Narasimhan. No more HotDependencies: Toward dependency-agnostic online upgrades in distributed systems. In *Proc. of the Third Workshop on Hot Topics in System Dependability*, page 14, 2007.

[93] Tudor Dumitras, Priya Narasimhan, and Eli Tilevich. To upgrade or not to upgrade: Impact of online upgrades across multiple administrative domains. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Appilcations*, pages 865–876, 2010.

[94] Joao A. Duraes and Henrique S. Madeira. Emulation of software faults: A field data study and a practical approach. *IEEE Trans. Softw. Eng.*, 32(11): 849–867, 2006.

[95] Tyler Durden. Bypassing PaX ASLR protection. *Phrack Magazine*, 9(59), 2002.

[96] Jake Edge. Linux ASLR vulnerabilities. `http://lwn.net/Articles/330866/`.

[97] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proc. of the 21st Int'l Conf. on Software Eng.*, pages 213–224, 1999.

[98] Stefan Esser. Exploiting the iOS kernel. In *Black Hat USA*, 2011.

[99] R. S. Fabry. How to design a system in which modules can be changed on the fly. In *Proc. of the Second Int'l Conf. on Software Eng.*, pages 470–476, 1976.

[100] Pedro Fonseca, Cheng Li, and Rodrigo Rodrigues. Finding complex concurrency bugs in large multi-threaded applications. In *Proc. of the Sixth ACM European Conf. on Computer Systems*, pages 215–228, 2011.

[101] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *Proc. of the Sixth Workshop on Hot Topics in Operating Systems*, page 67, 1997.

[102] Armando Fox. When does fast recovery trump high reliability? In *Proc. of*

the Second Workshop on Evaluating and Architecting System Dependability, 2002.

[103] Ophir Frieder and Mark E. Segal. On dynamically updating a computer program: From concept to prototype. *J. Syst. Softw.*, 14(2):111–128, 1991.

[104] C. Giuffrida and A.S. Tanenbaum. Safe and automated state transfer for secure and reliable live update. In *Proc. of the Fourth Int'l Workshop on Hot Topics in Software Upgrades*, pages 16–20, 2012.

[105] Cristiano Giuffrida and Andrew S. Tanenbaum. Cooperative update: A new model for dependable live update. In *Proc. of the Second Int'l Workshop on Hot Topics in Software Upgrades*, pages 1–6, 2009.

[106] Cristiano Giuffrida and Andrew S. Tanenbaum. A taxonomy of live updates. In *Proc. of the 16th ASCI Conf.*, 2010.

[107] Cristiano Giuffrida, Lorenzo Cavallaro, and Andrew S Tanenbaum. We crashed, now what? In *Proc. of the Sixth Workshop on Hot Topics in System Dependability*, pages 1–8, 2010.

[108] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *Proc. of the 21st USENIX Security Symp.*, page 40, 2012.

[109] Cristiano Giuffrida, Lorenzo Cavallaro, and Andrew S. Tanenbaum. Practical automated vulnerability monitoring using program state invariants. In *Proc. of the Int'l Conf. on Dependable Systems and Networks*, pages 1–12, 2013.

[110] Cristiano Giuffrida, Calin Iorgulescu, Anton Kuijsten, and Andrew S. Tanenbaum. Back to the future: Fault-tolerant live update with time-traveling state transfer. In *Proc. of the 27th USENIX Systems Administration Conf.*, 2013.

[111] Cristiano Giuffrida, Calin Iorgulescu, Anton Kuijsten, and Andrew S. Tanenbaum. Mutable checkpoint-restart: Automating live update for generic long-running c programs. In *Under Review*, 2013.

[112] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. EDFI: A dependable fault injection tool for dependability benchmarking experiments. In *Proc. of the Pacific Rim Int'l Symp. on Dependable Computing*, 2013.

[113] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Safe and automatic live update for operating systems. In *Proc. of the 18th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 279–292, 2013.

[114] Wolfram Gloger. ptmalloc. `http://www.malloc.de/en`.

[115] Brent Goodfellow. Patch tuesday. `http://www.thetechgap.com/2005/01/strongpatch_tue.html`.

[116] Michael Grace, Zhi Wang, Deepa Srinivasan, Jinku Li, Xuxian Jiang, Zhenkai Liang, and Siarhei Liakh. Transparent protection of commodity OS kernels using hardware virtualization. In *Proc. of the Sixth Conf. on Security and Privacy in Communication Networks*, pages 162–180, 2010.

[117] Jim Gray and Daniel P Siewiorek. High-availability computer systems. *IEEE Computer*, 24:39–48, 1991.

[118] Penny Grubb and Armstrong A. Takang. *Software maintenance: Concepts and practice*. World Scientific, 2nd edition, 2003.

[119] Weining Gu, Z. Kalbarczyk, Ravishankar, K. Iyer, and Zhenyu Yang. Characterization of Linux kernel behavior under errors. In *Proc. of the Int'l Conf. on Dependable Systems and Networks*, pages 459–468, 2003.

[120] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. FATE and DESTINI: A framework for cloud recovery testing. In *Proc. of the Eighth USENIX Conf. on Networked Systems Design and Implementation*, pages 18–18, 2011.

[121] P. J. Guo and D. Engler. Linux kernel developer responses to static analysis bug reports. In *Proc. of the USENIX Annual Tech. Conf.*, pages 285–292, 2009.

[122] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. R2: An application-level kernel for record and replay. In *Proc. of the Eighth USENIX Symp. on Operating Systems Design and Implementation*, pages 193–208, 2008.

[123] Deepak Gupta. *On-line software version change*. PhD thesis, Indian Institute of Technology Kanpur, 1994.

[124] Deepak Gupta and Pankaj Jalote. On-line software version change using state transfer between processes. *Softw. Pract. and Exper.*, 23(9):949–964, 1993.

[125] Deepak Gupta, Pankaj Jalote, and Gautam Barua. A formal framework for on-line software version change. *IEEE Trans. Softw. Eng.*, 22(2):120–131, 1996.

[126] Jungwoo Ha, Matthew Arnold, Stephen M. Blackburn, and Kathryn S. McKinley. A concurrent dynamic analysis framework for multicore hardware. In *Proc. of the 24th ACM Conf. on Object-Oriented Programming, Systems, Languages, and Appilcations*, pages 155–174, 2009.

[127] Sudheendra Hangal and Monica S Lam. Tracking down software bugs using automatic anomaly detection. In *Proc. of the 24th Int'l Conf. on Software Eng.*, pages 291–301, 2002.

[128] Saul Hansell. Glitch makes teller machines take twice what they give. *The New York Times*, 1994.

[129] Paul H. Hargrove and Jason C. Duell. Berkeley lab checkpoint/restart (BLCR) for Linux clusters. *J. Physics: Conference Series*, 46(1):494, 2006.

[130] Daniel Hartmeier. Design and performance of the OpenBSD stateful packet filter (pf). In *Proc. of the USENIX Annual Tech. Conf.*, pages 171–180, 2002.

[131] C. M Hayden, E. K Smith, M. Hicks, and J. S Foster. State transfer for clear and efficient runtime updates. In *Proc. of the Third Int'l Workshop on Hot Topics in Software Upgrades*, pages 179–184, 2011.

[132] C. M Hayden, E. K Smith, M. Denchev, M. Hicks, and J. S Foster. Kitsune: Efficient, general-purpose dynamic software updating for C. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Apilcations*, 2012.

[133] Christopher M. Hayden, Eric A. Hardisty, Michael Hicks, and Jeffrey S. Foster. Efficient systematic testing for dynamically updatable software. In *Proc. of the Second Int'l Workshop on Hot Topics in Software Upgrades*, pages 1–5, 2009.

[134] Christopher M. Hayden, Stephen Magill, Michael Hicks, Nate Foster, and Jeffrey S. Foster. Specifying and verifying the correctness of dynamic software updates. In *Proc. of the Fourth Int'l Conf. on Verified Software: Theories, Tools, Experiments*, pages 278–293, 2012.

[135] C.M. Hayden, K. Saur, M. Hicks, and J.S. Foster. A study of dynamic software update quiescence for multithreaded programs. In *Proc. of the Fourth Int'l Workshop on Hot Topics in Software Upgrades*, pages 6–10, 2012.

[136] C.M. Hayden, E.K. Smith, E.A. Hardisty, M. Hicks, and J.S. Foster. Evaluating dynamic software update safety using systematic testing. *IEEE Trans. Softw. Eng.*, 38(6):1340–1354, 2012.

[137] Guojin He and Antonia Zhai. Efficient dynamic program monitoring on multicore systems. *J. Syst. Architecture*, 57:121–133, 2011.

[138] Fergus Henderson. Accurate garbage collection in an uncooperative environment. In *Proc. of the Third Int'l Symp. on Memory management*, pages 150–156, 2002.

[139] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *Proc. of the 16th Symp. on Parallelism in Algorithms and Architectures*, pages 206–215, 2004.

[140] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Reorganizing UNIX for reliability. In *Proc. of the 11th Asia-*

Pacific Conf. on Advances in Computer Systems Architecture, pages 81–94, 2006.

[141] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Failure resilience for device drivers. In *Proc. of the Int'l Conf. on Dependable Systems and Networks*, pages 41–50, 2007.

[142] M. Hicks. *Dynamic software updating*. PhD thesis, Univ. of Pennsylvania, 2001.

[143] Dan Hildebrand. An architectural overview of QNX. In *Proc. of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 113–126, 1992.

[144] Martin Hirzel and Amer Diwan. On the type accuracy of garbage collection. In *Proc. of the Second Int'l Symp. on Memory Management*, pages 1–11, 2000.

[145] Martin Hirzel, Amer Diwan, and Johannes Henkel. On the usefulness of type and liveness accuracy for garbage collection and leak detection. *ACM Trans. Program. Lang. Syst.*, 24(6):593–624, 2002.

[146] Jason D Hiser, Clark L Coleman, Michele Co, and Jack W Davidson. MEDS: The memory error detection system. In *Proc. of the First Int'l Symp. on Engineering Secure Software and Systems*, pages 164–179, 2009.

[147] Gísli Hjálmtýsson and Robert Gray. Dynamic C++ classes: A lightweight mechanism to update code in a running program. In *Proc. of the USENIX Annual Tech. Conf.*, page 6, 1998.

[148] Petr Hosek and Cristian Cadar. Safe software updates via multi-version execution. In *Proc. of the 35th Int'l Conf. on Software Eng.*, pages 612–621, 2013.

[149] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Jean Wolter, and Sebastian Schönberg. The performance of microkernel-based systems. In *Proc. of the 16th ACM Symp. on Oper. Systems Prin.*, pages 66–77, 1997.

[150] Wei Hu, Jason Hiser, Dan Williams, Adrian Filipi, Jack W. Davidson, David Evans, John C. Knight, Anh Nguyen-Tuong, and Jonathan Rowanhill. Secure and practical defense against code-injection attacks using software dynamic translation. In *Proc. of the Second Int'l Conf. on Virtual Execution Environments*, pages 2–12, 2006.

[151] J.J. Hudak, B.-H. Suh, D.P. Siewiorek, and Z. Segall. Evaluation and comparison of fault-tolerant software techniques. *IEEE Trans. Rel.*, 42(2):190–204, 1993.

[152] Ralf Hund, Thorsten Holz, and Felix C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Proc. of the 18th*

USENIX Security Symp., pages 383–398, 2009.

[153] Galen C. Hunt and James R. Larus. Singularity: Rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, 2007.

[154] IBM Security X-Force. Mid-Year trend and risk report. `http://www-935.ibm.com/services/us/iss/xforce/trendreports`.

[155] M. Hiller J. Christmansson and M. Rimén. An experimental comparison of fault and error injection. In *Proc. of the Ninth Int'l Symp. on Software Reliability Eng.*, page 369, 1998.

[156] Kelly Jackson Higgins. The SCADA patch problem. `http://www.darkreading.com/vulnerability/the-scada-patch-problem/240146355`.

[157] Karl Janmar. FreeBSD 802.11 remote integer overflow. In *Black Hat Europe*, 2007.

[158] Jakub Jelinek. Prelink. `http://people.redhat.com/jakub/prelink.pdf`.

[159] Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu. Automated concurrency-bug fixing. In *Proc. of the 10th USENIX Symp. on Operating Systems Design and Implementation*, pages 221–236, 2012.

[160] A Johansson, Neeraj Suri, and Brendan Murphy. On the selection of error model(s) for OS robustness evaluation. In *Proc. of the Int'l Conf. on Dependable Systems and Networks*, pages 502–511, 2007.

[161] Andréas Johansson, Neeraj Suri, and Brendan Murphy. On the impact of injection triggers for OS robustness evaluation. In *Proc. of the 18th Int'l Symp. on Software Reliability Eng.*, pages 127–126, 2007.

[162] Paul Johnson and Neeraj Mittal. A distributed termination detection algorithm for dynamic asynchronous systems. In *Proc. of the 29th Int'l Conf. on Distr. Computing Systems*, pages 343–351, 2009.

[163] Pallavi Joshi, Haryadi S. Gunawi, and Koushik Sen. PREFAIL: A programmable tool for multiple-failure injection. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, volume 46, pages 171–188, 2011.

[164] Ghani A. Kanawati, Nasser A. Kanawati, and Jacob A. Abraham. FERRARI: A flexible software-based fault and error injection system. *IEEE Trans. Comput.*, 44(2):248–260, 1995.

[165] Wei-Lun Kao and R.K. Iyer. DEFINE: A distributed fault injection and monitoring environment. In *Proc. of the IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pages 252–259, 1994.

[166] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proc. of the 10th ACM Conf. on Computer and Commun. Security*, pages 272–280, 2003.

[167] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address space layout permutation (ASLP): Towards Fine-Grained randomization of commodity software. In *Proc. of the 22nd Annual Computer Security Appl. Conf.*, pages 339–348, 2006.

[168] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proc. of the 22nd ACM Symp. on Oper. Systems Prin.*, pages 207–220, 2009.

[169] Michael Kling, Sasa Misailovic, Michael Carbin, and Martin Rinard. Bolt: On-demand infinite loop escape in unmodified binaries. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Appilcations*, pages 431–450, 2012.

[170] Clemens Kolbitsch, Engin Kirda, and Christopher Kruegel. The power of procrastination: Detection and mitigation of execution-stalling malicious code. In *Proc. of the 18th ACM Conf. on Computer and Commun. Security*, pages 285–296, 2011.

[171] P. Koopman. What's wrong with fault injection as a benchmarking tool? In *Proc. of the Workshop on Dependability Benchmarking*, page 31, 2002.

[172] Philip Koopman, John Sung, Christopher Dingman, Daniel Siewiorek, and Ted Marz. Comparing operating systems using robustness benchmarks. In *Proc. of the 16th Int'l Symp. on Reliable Distributed Systems*, page 72, 1997.

[173] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *IEEE Trans. Softw. Eng.*, 16(11):1293–1306, 1990.

[174] Ilia Kravets and Dan Tsafrir. Feasibility of mutable replay for automated regression testing of security updates. In *Proc. of the Workshop on Runtime Environments, Systems, Layering and Virtualized Environments*, 2012.

[175] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. From uncertainty to belief: Inferring the specification within. In *Proc. of the 7th Symp. on Operating Systems Design and Implementation*, pages 161–176, 2006.

[176] A. Kumar, A. Sutton, and B. Stroustrup. Rejuvenating C++ programs through demacrofication. In *Proc. of the 28th IEEE Int'l Conf. on Software Maintenance*, 2012.

[177] Oren Laadan, Nicolas Viennot, and Jason Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proc. of the Int'l Conf. on Measurement and Modeling of Computer Systems*, pages 155–166, 2010.

[178] Open Kernel Labs. OKL4 community site. `http://wiki.ok-labs.com/`.

[179] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. of the Int'l Symp. on Code Generation and Optimization*, page 75, 2004.

[180] C. Le Goues, ThanhVu Nguyen, S. Forrest, and W. Weimer. GenProg: A generic method for automatic software repair. *IEEE Trans. Softw. Eng.*, 38 (1):54–72, 2012.

[181] Insup Lee. *Dymos: A dynamic modification system.* PhD thesis, Univ. of Wisconsin-Madison, 1983.

[182] Andrew Lenharth, Vikram S. Adve, and Samuel T. King. Recovery domains: An organizing principle for recoverable operating systems. In *Proc. of the 14th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 49–60, 2009.

[183] Jinku Li, Zhi Wang, T. Bletsch, D. Srinivasan, M. Grace, and Xuxian Jiang. Comprehensive and efficient protection of kernel control data. *IEEE Trans. Inf. Forensics and Security*, 6(4):1404–1417, 2011.

[184] Siarhei Liakh, Michael Grace, and Xuxian Jiang. Analyzing and improving Linux kernel memory protection: A model checking approach. In *Proc. of the 26th Annual Computer Security Appl. Conf.*, pages 271–280, 2010.

[185] J. Liedtke. On micro-kernel construction. In *Proc. of the 15th ACM Symp. on Oper. Systems Prin.*, pages 237–250, 1995.

[186] Jochen Liedtke. Improving IPC by kernel design. In *Proc. of the 14th ACM Symp. on Oper. Systems Prin.*, pages 175–188, 1993.

[187] Zhiqiang Lin, Ryan D. Riley, and Dongyan Xu. Polymorphing software by randomizing data structure layout. In *Proc. of the Sixth Int'l Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 107–126, 2009.

[188] Francesco Logozzo and Thomas Ball. Modular and verified automatic program repair. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages, and Appilcations*, pages 133–146, 2012.

[189] David E. Lowell, Yasushi Saito, and Eileen J. Samberg. Devirtualizable virtual machines enabling general, single-node, online maintenance. In *Proc. of the 11th Int'l Conf. on Architectural Support for Programming Languages*

and Operating Systems, pages 211–223, 2004.

[190] Neil MacDonald. Devops needs to become devopssec. `http://blogs.gartner.com/neil_macdonald/2012/01/17/devops-needs-to-become-devopssec`.

[191] Henrique Madeira, Diamantino Costa, and Marco Vieira. On the emulation of software faults by software fault injection. In *Proc. of the Int'l Conf. on Dependable Systems and Networks*, pages 417–426, 2000.

[192] Stephen Magill, Michael Hicks, Suriya Subramanian, and Kathryn S. McKinley. Automating object transformations for dynamic software updating. In *Proc. of the 27th ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 265–280, 2012.

[193] K. Makris and R. Bazzi. Immediate multi-threaded dynamic software updates using stack reconstruction. In *Proc. of the USENIX Annual Tech. Conf.*, pages 397–410, 2009.

[194] Kristis Makris and Kyung Dong Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *Proc. of the Second ACM European Conf. on Computer Systems*, pages 327–340, 2007.

[195] Abid M. Malik, Jim McInnes, and Peter van Beek. Optimal basic block instruction scheduling for Multiple-Issue processors using constraint programming. In *Proc. of the 18th Int'l Conf. on Tools with Artificial Intelligence*, pages 279–287, 2006.

[196] Paul D. Marinescu, Radu Banabic, and George Candea. An extensible technique for high-precision testing of recovery code. In *Proc. of the USENIX Annual Tech. Conf.*, pages 23–23, 2010.

[197] Paul Dan Marinescu and Cristian Cadar. KATCH: High-coverage testing of software patches. In *Proc. of the 9th Joint Meeting of the European Software Eng. Conf. and the ACM SIGSOFT Symp. on the Foundations of Software Eng.*, pages 235–245, 2013.

[198] P.D. Marinescu and G. Candea. LFI: A practical and general library-level fault injector. In *Proc. of the Int'l Conf. on Dependable Systems and Networks*, pages 379–388, 2009.

[199] Matthew Maurer and David Brumley. TACHYON: Tandem execution for efficient live patch testing. In *Proc. of the 21st USENIX Security Symp.*, page 43, 2012.

[200] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Proc. of the 10th Int'l Conf. on Parallel and Distributed Computing and Systems*, pages 509–518, 1998.

[201] Paul E. McKenney and Jonathan Walpole. What is RCU, fundamentally? `http://lwn.net/Articles/262464`.

[202] Microsoft. Windows User-Mode driver framework. `http://msdn.microsoft.com/en-us/windows/hardware/gg463294`.

[203] R. G Minnich. A dynamic kernel modifier for Linux. In *Proc. of the LACSI Symposium*, 2002.

[204] Neeraj Mittal, S. Venkatesan, and Sathya Peri. A family of optimal termination detection algorithms. *Distributed Computing*, 20(2):141–162, 2007.

[205] R. Moraes, R. Barbosa, J. Duraes, N. Mendes, E. Martins, and H. Madeira. Injection of faults at component interfaces and inside the component code: Are they equivalent? In *Proc. of the Sixth European Dependable Computing Conf.*, pages 53–64, 2006.

[206] D. Mutz, W. Robertson, G. Vigna, and R. Kemmerer. Exploiting execution context for the detection of anomalous system calls. In *Proc. of the 10th Int'l Conf. on Recent Advances in Intrusion Detection*, pages 1–20, 2007.

[207] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proc. of the 14th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 265–276, 2009.

[208] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proc. of the 28th Int'l Conf. on Software Eng.*, pages 452–461, 2006.

[209] Kiran Nagaraja, Fábio Oliveira, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. Understanding and dealing with operator mistakes in internet services. In *Proc. of the Sixth USENIX Symp. on Operating Systems Design and Implementation*, pages 5–5, 2004.

[210] R. Natella, D. Cotroneo, J. Duraes, and H. Madeira. Representativeness analysis of injected software faults in complex software. In *Proc. of the Int'l Conf. on Dependable Systems and Networks*, pages 437–446, 2010.

[211] R. Natella, D. Cotroneo, J. Duraes, and H. Madeira. On fault representativeness of software fault injection. *IEEE Trans. Softw. Eng.*, PP(99):1, 2012.

[212] I. Neamtiu and T. Dumitras. Cloud software upgrades: Challenges and opportunities. In *Proc. of the Int'l Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems*, pages 1–10, 2011.

[213] Iulian Neamtiu and Michael Hicks. Safe and timely updates to multi-threaded programs. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 13–24, 2009.

[214] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for C. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 72–83, 2006.

[215] Iulian Neamtiu, Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 37–49, 2008.

[216] Nergal. The advanced return-into-lib(c) exploits. *Phrack Magazine*, 4(58), 2001.

[217] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proc. of the 14th ACM Conf. on Computer and Commun. Security*, pages 529–540, 2007.

[218] Wee Teck Ng and Peter M. Chen. The systematic improvement of fault tolerance in the Rio file cache. In *Proc. of the 29th Int'll Symp. on Fault-Tolerant Computing*, page 76, 1999.

[219] Wee Teck Ng and Peter M. Chen. The design and verification of the Rio file cache. *IEEE Trans. Comput.*, 50(4):322–337, 2001.

[220] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. SemFix: Program repair via semantic analysis. In *Proc. of the Int'l Conf. on Software Eng.*, pages 772–781, 2013.

[221] NIST. National vulnerability database. `http://nvd.nist.gov`.

[222] Gene Novark and Emery D Berger. DieHarder: Securing the heap. In *Proc. of the 17th ACM Conf. on Computer and Commun. Security*, pages 573–584, 2010.

[223] Jon Oberheide, Evan Cooke, and Farnam Jahanian. If it ain't broke, don't fix it: Challenges and new directions for inferring the impact of software patches. In *Proc. of the 12th Workshop on Hot Topics in Operating Systems*, page 17, 2009.

[224] Fábio Oliveira, Kiran Nagaraja, Rekha Bachwani, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. Understanding and validating database system administration. In *Proc. of the USENIX Annual Tech. Conf.*, pages 213–228, 2006.

[225] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-Free: Defeating return-oriented programming through gadget-less binaries. In *Proc. of the 26th Annual Computer Security Appl. Conf.*, pages 49–58, 2010.

[226] Tim O'Reilly. What is Web 2.0. `http://oreilly.com/pub/a/web2/`

`archive/what-is-web-20.html`.

[227] Thomas J. Ostrand and Elaine J. Weyuker. The distribution of faults in a large industrial software system. *ACM SIGSOFT Softw. Eng. Notes*, 27(4):55–64, 2002.

[228] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. Where the bugs are. In *Proc. of the Int'l Symp. on Software Testing and Analysis*, pages 86–96, 2004.

[229] Yoann Padioleau, Julia L. Lawall, and Gilles Muller. Understanding collateral evolution in Linux device drivers. In *Proc. of the First ACM European Conf. on Computer Systems*, pages 59–71, 2006.

[230] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in Linux device drivers. In *Proc. of the Third ACM European Conf. on Computer Systems*, pages 247–260, 2008.

[231] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in Linux: Ten years later. In *Proc. of the 16th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 305–318, 2011.

[232] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. PRES: Probabilistic replay with execution sketching on multiprocessors. In *Proc. of the 22nd ACM Symp. on Oper. Systems Prin.*, pages 177–192, 2009.

[233] Karthik Pattabiraman, Giacinto Paolo Saggese, Daniel Chen, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. Automated derivation of application-specific error detectors using dynamic analysis. *IEEE Trans. Dep. Secure Comput.*, 8(5):640–655, 2011.

[234] David A Patterson. A simple way to estimate the cost of downtime. In *Proc. of the 16th USENIX Systems Administration Conf.*, pages 185–188, 2002.

[235] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *Proc. of the 22nd ACM Symp. on Oper. Systems Prin.*, pages 87–102, 2009.

[236] E. Perla and M. Oldani. *A guide to kernel exploitation: Attacking the core*. Syngress Publishing, 2010.

[237] Nick L. Petroni,Jr. and Michael Hicks. Automated detection of persistent kernel control-flow attacks. In *Proc. of the 14th ACM Conf. on Computer and*

Commun. Security, pages 103–115, 2007.

[238] Georgios Portokalidis and Angelos D Keromytis. Fast and practical instruction-set randomization for commodity systems. In *Proc. of the 26th Annual Computer Security Appl. Conf.*, pages 41–48, 2010.

[239] Georgios Portokalidis and Angelos D. Keromytis. REASSURE: A self-contained mechanism for healing software using rescue points. In *Proc. of the Sixth Int'l Conf. on Advances in Information and Computer Security*, pages 16–32, 2011.

[240] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: An emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. In *Proc. of the First ACM European Conf. on Computer Systems*, pages 15–27, 2006.

[241] Shaya Potter and Jason Nieh. Reducing downtime due to system maintenance and upgrades. In *Proc. of the 19th USENIX Systems Administration Conf.*, pages 6–6, 2005.

[242] K. Poulsen. Software bug contributed to blackout. *Security Focus*, 2004.

[243] Jon Rafkind, Adam Wick, John Regehr, and Matthew Flatt. Precise garbage collection for C. In *Proc. of the Eighth Int'l Symp. on Memory management*, pages 39–48, 2009.

[244] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin Zorn. NOZZLE: A defense against heap-spraying code injection attacks. In *Proc. of the 18th USENIX Security Symp.*, pages 169–186, 2009.

[245] Eric Rescorla. Security holes... who cares? In *Proc. of the 12th USENIX Security Symp.*, pages 6–6, 2003.

[246] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing. In *Proc. of the 11th Int'l Conf. on Recent Advances in Intrusion Detection*, pages 1–20, 2008.

[247] W. Robertson, G. Vigna, C. Kruegel, and R. Kemmerer. Using generalization and characterization techniques in the anomaly-based detection of web attacks. In *Proc. of the 13th Network and Distributed System Security Symp.*, 2006.

[248] William Robertson, Christopher Kruegel, Darren Mutz, and Fredrik Valeur. Run-time detection of heap-based overflows. In *Proc. of the 17th USENIX Systems Admin. Conf.*, pages 51–60, 2003.

[249] R.E. Rodrigues, V.H. Sperle Campos, and F. Magno Quintao Pereira. A fast and low-overhead technique to secure programs against integer overflows. In *Proc. of the Int'l Symp. on Code Generation and Optimization*, pages 1–11,

2013.

[250] Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. Surgically returning to randomized lib(c). In *Proc. of the 2009 Annual Computer Security Appl. Conf.*, pages 60–69, 2009.

[251] Olatunji Rowase and Monica S. Lam. A practical dynamic buffer overflow detector. In *Proc. of the 11th Network and Distributed System Security Symp.*, pages 159–169, 2004.

[252] Olatunji Ruwase, Phillip B Gibbons, Todd C Mowry, Vijaya Ramachandran, Shimin Chen, Michael Kozuch, and Michael Ryan. Parallelizing dynamic information flow tracking. In *Proc. of the 20th Symp. on Parallelism in Algorithms and Architectures*, pages 35–45, 2008.

[253] Babak Salamat, Andreas Gal, Todd Jackson, Karthikeyan Manivannan, Gregor Wagner, and Michael Franz. Multi-variant program execution: Using multi-core systems to defuse buffer-overflow vulnerabilities. In *Proc. of the 2008 Int'l Conf. on Complex, Intelligent and Software Intensive Systems*, pages 843–848, 2008.

[254] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. Orchestra: Intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proc. of the ACM Fourth European Conf. on Computer Systems*, pages 33–46, 2009.

[255] Eric Schulte, Jonathan DiLorenzo, Westley Weimer, and Stephanie Forrest. Automated repair of binary and assembly programs for cooperating embedded devices. In *Proc. of the 18th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 317–328, 2013.

[256] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *Proc. of the 20th USENIX Security Symp.*, page 25, 2011.

[257] Mark E. Segal and Ophir Frieder. On-the-fly program modification: Systems for dynamic updating. *IEEE Softw.*, 10(2):53–65, 1993.

[258] M. Seltzer and C. Small. Self-monitoring and self-adapting operating systems. In *Proc. of the Sixth Workshop on Hot Topics in Operating Systems*, pages 124–129, 1997.

[259] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proc. of the Second USENIX Symp. on Operating Systems Design and Implementation*, pages 213–227, 1996.

[260] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A tiny

hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proc. of the 21st ACM Symp. on Oper. Systems Prin.*, pages 335–350, 2007.

[261] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proc. of the 14th ACM Conf. on Computer and Commun. Security*, pages 552–561, 2007.

[262] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proc. of the 11th ACM Conf. on Computer and Commun. Security*, pages 298–307, 2004.

[263] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. In *Proc. of the 17th ACM Symp. on Oper. Systems Prin.*, pages 170–185, 1999.

[264] Maxim Siniavine and Ashvin Goel. Seamless kernel updates. In *Proc. of the Int'l Conf. on Dependable Systems and Networks*, 2013.

[265] Craig A. N Soules, Dilma Da Silva, Marc Auslander, Gregory R Ganger, and Michal Ostrowski. System support for online reconfiguration. In *Proc. of the USENIX Annual Tech. Conf.*, pages 141–154, 2003.

[266] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proc. of the USENIX Annual Tech. Conf.*, page 3, 2004.

[267] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. Mutatis mutandis: Safe and predictable dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 29(4), 2007.

[268] Dinesh Subhraveti and Jason Nieh. Record and transplay: Partial checkpointing for replay debugging across heterogeneous systems. In *Proc. of the Int'l Conf. on Measurement and Modeling of Computer Systems*, pages 109–120, 2011.

[269] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic software updates: A VM-centric approach. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 1–12, 2009.

[270] M. Sullivan and R. Chillarege. A comparison of software defects in database management systems and operating systems. In *Proc. of the 22nd Int'll Symp. on Fault-Tolerant Computing*, pages 475–484, 1992.

[271] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. *ACM Trans. Comput. Syst.*, 23

(1):77–110, 2005.

[272] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. *ACM Trans. Comput. Syst.*, 24 (4):333–360, 2006.

[273] Ariel Tamches and Barton P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proc. of the Third ACM Symp. on Oper. Systems Prin.*, pages 117–130, 1999.

[274] PaX Team. Overall description of the PaX project. `http://pax.grsecurity.net/docs/pax.txt`.

[275] Timothy K. Tsai and Ravishankar K. Iyer. Measuring fault tolerance with the FTAPE fault injection tool. In *Proc. of the Eighth Int'l Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 26–40, 1995.

[276] Timothy K. Tsai, Mei-Chen Hsueh, Hong Zhao, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Stress-based and path-based fault injection. *IEEE Trans. Comput.*, 48(11):1183–1201, 1999.

[277] Joseph Tucek, Weiwei Xiong, and Yuanyuan Zhou. Efficient online validation with delta execution. In *Proc. of the 14th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 193–204, 2009.

[278] Steven Van Acker, Nick Nikiforakis, Pieter Philippaerts, Yves Younan, and Frank Piessens. ValueGuard: Protection of native applications against data-only buffer overflows. In *Proc. of the Sixth Int'l Conf. on Inf. Systems Security*, pages 156–170, 2010.

[279] Yves Vandewoude, Peter Ebraert, Yolande Berbers, and Theo D'Hondt. Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE Trans. Softw. Eng.*, 33(12):856–868, 2007.

[280] Nicolas Viennot, Siddharth Nair, and Jason Nieh. Transparent mutable replay for multicore debugging and patch validation. In *Proc. of the 18th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 127–138, 2013.

[281] Zhi Wang, Xuxian Jiang, Weidong Cui, and Peng Ning. Countering kernel rootkits with lightweight hook protection. In *Proc. of the 16th ACM Conf. on Computer and Commun. Security*, pages 545–554, 2009.

[282] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. In *Proc. of the 19th Int'l Symp. on Software Testing and Analysis*, pages 61–72, 2010.

[283] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proc. of the Int'l Workshop on Memory Management*, pages 1–42, 1992.

[284] Stefan Winter, Michael Tretter, Benjamin Sattler, and Neeraj Suri. simFI: From single to simultaneous software fault injections. In *Proc. of the Int'l Conf. on Dependable Systems and Networks*, pages 1–12, 2013.

[285] Thomas Würthinger, Christian Wimmer, and Lukas Stadler. Dynamic code evolution for java. In *Proc. of the 8th Int'l Conf. on the Principles and Practice of Programming in Java*, pages 10–19, 2010.

[286] Jingyue Wu, Heming Cui, and Junfeng Yang. Bypassing races in live applications with execution filters. In *Proc. of the Ninth USENIX Symp. on Operating Systems Design and Implementation*, pages 1–13, 2010.

[287] Chen Xi, Asia Slowinska, and Herbert Bos. Who allocated my memory? detecting custom memory allocators in c binaries. In *Proc. of the 20th Working Conf. on Reverse Eng.*, 2013.

[288] Haizhi Xu and Steve J. Chapin. Improving address space randomization with a dynamic offset randomization technique. In *Proc. of the 2006 ACM Symp. on Applied Computing*, pages 384–391, 2006.

[289] Jun Xu, Z. Kalbarczyk, and R. K Iyer. Transparent runtime randomization for security. In *Proc. of the 22nd Int'l Symp. on Reliable Distributed Systems*, pages 260– 269, 2003.

[290] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proc. of the 14th ACM Conf. on Computer and Commun. Security*, pages 116–127, 2007.

[291] Heng Yin, Pongsin Poosankam, Steve Hanna, and Dawn Song. HookScout: Proactive binary-centric hook detection. In *Proc. of the Seventh Int'l Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 1–20, 2010.

[292] Yves Younan, Davide Pozza, Frank Piessens, and Wouter Joosen. Extended protection against stack smashing attacks without performance loss. In *Proc. of the 22nd Annual Computer Security Appl. Conf.*, pages 429–438, 2006.

[293] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R. Sekar, Frank Piessens, and Wouter Joosen. PAriCheck: An efficient pointer arithmetic checker for C programs. In *Proc. of the Fifth ACM Symp. on Inf., Computer and Commun. Security*, pages 145–156, 2010.

[294] Cliff Young, David S. Johnson, Michael D. Smith, and David R. Karger. Near-optimal intraprocedural branch alignment. In *Proc. of the ACM SIGPLAN*

Conf. on Programming Language Design and Implementation, pages 183–193, 1997.

[295] Angeliki Zavou, Georgios Portokalidis, and Angelos D. Keromytis. Taint-exchange: A generic system for cross-process and cross-host taint tracking. In *Proc. of the 6th Int'l Conf. on Advances in Information and Computer Security*, pages 113–128, 2011.

[296] Q. Zeng, D. Wu, and P. Liu. Cruiser: Concurrent heap buffer overflow monitoring using lock-free data structures. In *Proc. of the 32nd ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 367–377, 2011.

[297] Kehuan Zhang and XiaoFeng Wang. Peeping tom in the neighborhood: Keystroke eavesdropping on multi-user systems. In *Proc. of the 18th USENIX Security Symp.*, pages 17–32, 2009.

[298] Wei Zheng, Ricardo Bianchini, G. John Janakiraman, Jose Renato Santos, and Yoshio Turner. JustRunIt: Experiment-based management of virtualized data centers. In *Proc. of the USENIX Annual Tech. Conf.*, page 18, 2009.

[299] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *Proc. of the Seventh USENIX Symp. on Operating Systems Design and Implementation*, pages 45–60, 2006.

[300] Pin Zhou, Wei Liu, Long Fei, Shan Lu, Feng Qin, Yuanyuan Zhou, Samuel Midkiff, and Josep Torrellas. AccMon: Automatically detecting memory-related bugs via program counter-based invariants. In *Proc. of the 37th Int'l Symp. on Microarchitecture*, pages 269–280, 2004.

Summary

Many real-world systems need to run 24/7 for years and never go down. Consider, for example, industrial control systems and e-banking servers. Unfortunately, software updates—a necessary evil to cope with the fast-paced evolution of modern software—are on a collision course with our growing need for nonstop operation, with traditional update practices resorting to a full system restart to deploy new versions or even small security patches. The update-without-downtime problem has recently fostered much research on *live update*—the ability to update running software on the fly. Prior live update solutions, however, offer no practical and general way to update operating systems (OSes) or long-running applications, requiring extensive manual effort for nontrivial updates and offering poor system support to detect and recover from update-time errors. These factors have significantly lowered the usability and dependability guarantees of prior techniques compared to those of regular software updates, ultimately discouraging widespread adoption of live update.

This dissertation introduces new techniques to implement *safe* and *automatic* live update for the entire software stack, with support for several possible classes of updates—ranging from security patches to complex version updates—and generic systems software written in C. At the heart of our live update mechanisms lies a novel *process-level update* strategy, which starts the new version as a new independent process and automatically transfers the state of the old process to the new one. When an update becomes available, our framework allows the new version to start up and connect to the old version to request all the information from the old execution state it needs (e.g., data structures, even if they have changed between versions). When all the necessary data have been transferred over and the old state completely remapped into the new version, our framework terminates the old version and allows the new version to atomically resume execution in a way completely transparent to the users.

Our automated state transfer strategy is empowered by a new instrumentation pass in the LLVM compiler framework that maintains metadata about all the program data structures in memory. This allows our framework to inspect the state of two different versions and seamlessly remap data structures between them, even in face of complex state and memory layout transformations. To ensure a safe update process, our framework can also detect common update-time errors using three different mechanisms: (i) *run-time error detection*—detecting crashes and other abnormal run-time events using hardware/software exceptions—(ii) *invariants-based detection*—detecting state corruption from violations of statically extracted program state invariants—(iii) *time-traveling state transfer-based detection*—detecting memory errors from state differences between distinct process versions known to be equivalent by construction. When an error is detected during the update process, our framework automatically rolls back the update, terminating the new version and allowing the old version to resume execution normally, similar to an aborted atomic transaction. This process-level update prevents update-time errors in the new version from propagating back to the old version, allowing for safe error recovery at update time. This is in stark contrast with prior solutions, which typically patch the running program in place, so if the update fails, there is no fallback to a working version.

At the OS level, we demonstrate the effectiveness of our techniques in PROTEOS, a new research OS designed with live update in mind. In PROTEOS, process-level updates are a first-class abstraction implemented on a multiserver OS architecture based on MINIX 3. PROTEOS combines our live update techniques with a rigorous event-driven programming model adopted in the individual OS components, allowing updates to happen only in predictable and controllable system states. At the application level, we demonstrate the effectiveness of our techniques in *Mutable Checkpoint-Restart* (*MCR*), a new live update framework for generic long-running C programs. MCR extends our techniques to allow legacy user programs to support safe and automatic live update with little manual effort.

In conclusion, this dissertation presents evidence that a major paradigm shift is necessary in the design of live update systems. Unlike existing approaches, our live update techniques allow OS components and long-running application programs to be updated without patching running programs in place and potentially endangering their execution should the update fail for any reason. We demonstrate that this new paradigm is amenable to effectively automating and safeguarding the live update process, while reducing the implementation burden to the bare minimum. Our experience with live updating major components of the entire software stack shows important limitations in prior solutions and confirms our hypothesis that safe and automatic live update is a realistic option if careful software design and adequate system support are available. We see our work as the first important step towards truly practical, general, and trustworthy live update systems for the real world.

Samenvatting

Voor veel in de praktijk gebruikte systemen is het noodzakelijk dat ze jarenlang 24 uur per dag beschikbaar zijn zonder downtime. Voorbeelden hiervan zijn industriële aansturingssystemen en servers voor internetbankieren. Helaas staan software updates—een noodzakelijk kwaad om om te kunnen gaan met de voortdurende ontwikkeling van moderne software—in de weg bij de groeiende noodzaak van non-stop beschikbaarheid, omdat de traditionele aanpak van updates een volledige herstart van het systeem vereist om nieuwe versies en zelfs kleine beveiligingsupdates te installeren. Het probleem van updaten-zonder-downtime heeft recent geleid tot veel onderzoek naar *live update*—de mogelijkheid om software bij te werken terwijl deze beschikbaar blijft. Eerdere live update oplossingen boden echter geen praktische en algemeen toepasbare manier om besturingssystemen en langdraaiende applicaties bij te werken, resulterend in aanzienlijke handmatige inspanningen voor niet-triviale updates en slechte ondersteuning voor het ontdekken en herstellen van fouten tijdens de update. Deze factoren verminderen de bruikbaarheid en betrouwbaarheid van eerdere technieken aanzienlijk ten opzichte van reguliere software updates, met als eindresultaat het verhinderen van grootschalig gebruik van live update.

Deze dissertatie introduceert nieuwe technieken om *veilige* en *automatische* live update te implementeren voor de gehele software stack, met ondersteuning voor meerdere klassen van updates—variërend van beveiligingsupdates tot complexe versie updates—en generieke systeemsoftware geschreven in C. De kern van onze live update mechanismen is een nieuwe *updatestrategie op processniveau*, waarbij de nieuwe versie gestart wordt als een onafhankelijk process en de staat automatisch wordt overgedragen van het oude process naar het nieuwe. Wanneer een update beschikbaar komt laat ons raamwerk de nieuwe versie opstarten en verbinden met de oude versie om alle informatie die het nodig heeft over de oude execution state op te vragen (dwz. datastructuren, zelfs als deze tussen versies gewijzigd zijn). Wanneer al de benodigde gegevens zijn overgedragen en de staat volledig is omgezet naar de nieuwe versie sluit ons raamwerk de oude versie af en laat de nieuwe versie atomisch verder gaan op een wijze die voor de gebruiker volledig transparant is.

Onze automatische state transfer strategie wordt ondersteund door een nieuwe instrumentatie pass in het LLVM compiler raamwerk dat metadata bijhoudt over alle datastructuren van het programma in het geheugen. Dit stelt ons raamwerk in staat om de staat te inspecteren van twee verschillende versies en naadloos de datastructuren tussen de twee om te zetten, zelfs in geval van complexe transformaties van de staat en de geheugenindeling. Om veilig uitvoeren van de update te garanderen kan ons raamwerk ook veel voorkomende fouten tijdens updates ontdekken met behulp van drie verschillende mechanismen: (i) *run-time foutdetectie*—het ontdekken van crashes en andere abnormale run-time gebeurtenissen door middel van hardware/-software exceptions—(ii) *detectie gebaseerd op invariants*—het ontdekken van een gecorrumpeerde staat via afwijkingen van statisch bepaalde eigenschappen van de staat van het programma—(iii) *detectie gebaseerd op time-traveling state transfer*—het ontdekken van geheugenfouten uit verschillen tussen de staat van versies van het process die per constructie equivalent zouden moeten zijn.

Als een fout wordt ontdekt tijdens het bijwerken, maakt ons raamwerk de update automatisch ongedaan. Hierbij wordt de nieuwe versie afgesloten en de oude versie normaal voortgezet, vergelijkbaar met een atomische transactie. Deze updatestrategie op process-niveau voorkomt dat fouten in de nieuwe versie ten tijde van de update zich kunnen verspreiden naar de oude versie, zodat veilig herstel van fouten tijdens de update mogelijk is. Dit staat in sterk contrast met eerdere oplossingen, welke het programma bijwerken op de oorspronkelijke plaats, zodat als de update faalt er geen mogelijkheid is om op de werkende versie terug te vallen.

Op het niveau van het besturingssysteem tonen we de effectiviteit van onze technieken in PROTEOS, een nieuw onderzoeksbesturingssysteem ontworpen met live update in het achterhoofd. In PROTEOS zijn updates op het niveau van het process een eersteklas abstractie, geïmplementeerd op een multiserver besturingssysteemarchitectuur gebaseerd op MINIX 3. PROTEOS combineert onze live update technieken met een grondig gebeurtenis-gedreven programmeermodel gebruikt in de individuele componenten van het besturingssysteem, om mogelijk te maken dat updates alleen worden uitgevoerd wanneer het systeem in een voorspelbare en beheersbare staat is. Op het niveau van de applicatie tonen we de effectiviteit aan met *Mutable Checkpoint-Restart* (*MCR*), een nieuw live update raamwerk voor generieke langdraaiende programma's geschreven in C. MCR breidt onze technieken uit om legacy-gebruikersprogramma's in staat te stellen veilige en automatische live update te ondersteunen met minimaal handmatig werk.

Ter afsluiting presenteert deze dissertatie bewijs dat een grote paradigmaverschuiving noodzakelijk is in het ontwerp van live update systemen. In tegenstelling tot bestaande methodes maken onze live update technieken het mogelijk om componenten van het besturingssysteem en langdraaiende applicatieprogramma's bij te werken zonder dat draaiende programma's ter plekke aangepast hoeven te worden.

Hiermee wordt voorkomen dat ze mogelijk beëindigd moeten worden als het bijwerken om welke reden dan ook zou mislukken. We tomen aan dat deze nieuw aanpak geschikt is om effectief het live update proces te automatiseren en te beschermen, terwijl het werk voor implementatie tot een minimum beperkt wordt. Onze ervaring met het live updaten van belangrijke onderdelen van de gehele software stack toont belangrijke beperkingen van eerdere oplossingen aan en bevestigt onze hypothese dat veilige en automatische live updates een realistische optie zijn als zorgvuldig softwareontwerp en toereikende ondersteuning vanuit het systeem beschikbaar zijn. We zien ons werk als de eerste belangrijke stap richring een werkelijk praktisch, algemeen en betrouwbaar live update systeem dat geschikt is voor gebruik in de buitenwereld.