Execution Synthesis: A Technique for Automating the Debugging of Software

THÈSE N^{O} 5914 (2013)

PRÉSENTÉE LE 18 NOVEMBRE 2013

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DES SYSTEMES FIABLES
PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Cristian ZAMFIR

acceptée sur proposition du jury:

Prof. M. Grossglauser, président du jury
Prof. G. Candea, directeur de thèse
Prof. I. Stoica, rapporteur
Prof. Y. Zhou, rapporteur
Prof. W. Zwaenepoel, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2013

Abstract (German)

Fehlerbeseitigung in Computersystemen ist schwierig, zeitraubend, und erfordert detaillierte Kenntnisse des Programmcodes dieser Systeme. Fehlerberichte enthalten selten genügend Informationen zur Beseitigung des Fehlers. Entwickler müssen in anstrengender Detektivarbeit herausfinden, wie das Programm zu der beschriebenen Fehlersituation gekommen ist.

Diese Doktorarbeit beschreibt eine Technik zur Synthese von Programmausführungen, welche diese Detektivarbeit automatisiert: Ausgehend von einem Programm und einem Fehlerbericht, generiert die Programmausführungssynthese automatisch einen Programmablauf, welche zum beschriebenen Fehler führt. Mittels einer Kombination aus statischer Programmanalyse und symbolischer Ausführung *synthetisiert* sie sowohl eine Reihenfolge der Ausführung der Programmthreads, als auch benötigte Eingabedaten, welche den Fehler auslösen. Die synthetisierte Ausführung kann schrittweise in einem Debugger, zum Beispiel gdb, nachverfolgt werden. Besonders nützlich ist dies zur Beseitigung von Fehlern in nebenläufigen Programmen: Fehler, die sonst nur sporadisch auftauchen, können nun deterministisch in einem Debugger analysiert werden.

Weil die Programmausführungssynthese weder Aufzeichnungen zur Laufzeit, noch Änderungen an Programm oder Hardware benötigt, entsteht keinerlei Beeinträchtigung der Programmleistung. Dadurch ist Programmausführungssynthese auch für Programme im laufenden Betrieb möglich. Diese Doktorarbeit enthält sowohl eine theoretische Analyse der Programmausführungssynthese, als auch experimentelle Belege, die zeigen, dass sie erfolgreich in der Praxis angewendet werden kann. Innert Minuten generiert sie, von einem Fehlerbericht ausgehend, Programmausführungen für verschiedene Speicherzugriffsfehler und Nebenläufigkeitsfehler in echten Systemen.

Diese Doktorarbeit präsentiert ausserdem die Rückwärtssynthese von Programmausführungen. Diese Technik nimmt einen Speicherauszug (einen sogenannten coredump, der beim Auftreten des Fehlers erstellt wird) und berechnet das Ausführungssuffix, welches zu diesem Speicherinhalt führt. Die Rückwärtssynthese generiert alle nötigen Informationen, um dieses Suffix deterministisch in einem Debugger zu analysieren, bis die Fehlerquelle gefunden ist. Da nur der letzte Teil einer Programmausführung generiert wird, eignet sich die Rückwärtssynthese besonders zur Analyse von beliebig lange laufenden Programmen, in denen die Fehlerquelle und das Auftreten des Fehlers zeitlich nahe beieinander liegen.

Diese Dissertation beschreibt ebenfalls, wie Programmausführungssynthese mit Techniken zur Aufzeichnung von Programmausführungen kombiniert werden kann. Dies dient der Klassifizierung von Data Races und dem Beheben von Synchronisationsfehlern wie zum Beispiel Deadlocks.

Stichworte: Automatisierte Debugging, Programmausführungssynthese, Aufnahme, Wiedergabe, symbolische Programmausführung.

Abstract

Debugging real systems is hard, requires deep knowledge of the target code, and is time-consuming. Bug reports rarely provide sufficient information for debugging, thus forcing developers to turn into detectives searching for an explanation of how the program could have arrived at the reported failure state.

This thesis introduces execution synthesis, a technique for automating this detective work: given a program and a bug report, execution synthesis automatically produces an execution of the program that leads to the reported bug symptoms. Using a combination of static analysis and symbolic execution, the technique "synthesizes" a thread schedule and various required program inputs that cause the bug to manifest. The synthesized execution can be played back deterministically in a regular debugger, like gdb. This is particularly useful in debugging concurrency bugs, because it transforms otherwise non-deterministic bugs into bugs that can be deterministically observed in a debugger.

Execution synthesis requires no runtime recording, and no program or hardware modifications, thus incurring no runtime overhead. This makes it practical for use in production systems. This thesis includes a theoretical analysis of execution synthesis as well as empirical evidence that execution synthesis is successful in starting from mere bug reports and reproducing on its own concurrency and memory safety bugs in real systems, taking on the order of minutes.

This thesis also introduces reverse execution synthesis, an automated debugging technique that takes a coredump obtained after a failure and automatically computes the suffix of an execution that leads to that coredump. Reverse execution synthesis generates the necessary information to then play back this suffix in a debugger deterministically as many times as needed to complete the debugging process. Since it synthesizes an execution suffix instead of the entire execution, reverse execution is particularly well suited for arbitrarily long executions in which the failure and its root cause occur within a short time span, so developers can use a short execution suffix to debug the problem.

The thesis also shows how execution synthesis can be combined with recording techniques in order to automatically classify data races and to efficiently debug deadlock bugs.

Keywords: Automated debugging, execution synthesis, record-replay, symbolic execution.

Acknowledgments

I am very grateful for all the help I received during my doctoral studies, all of which lead to this thesis.

I thank my advisor Professor George Candea for the vision and the passion with which he guided and helped me during my research. It was truly inspirational to be advised by him. While working together, I grew beyond any of my expectations, both as a researcher and as a person.

I would like to thank in advance Professor Yuanyuan Zhou, Professor Ion Stoica, and Professor Willy Zwaenepoel for their help and for accepting to be part of my thesis committee.

I thank Microsoft for their generosity in financially supporting a substantial part of my research through an ICES grant. I also thank EPFL for its unmatched support and for providing my second home. I thank Intel for their interest in my research and for their generosity in providing the Intel Student Honors Award.

I thank my paper co-authors: Baris Kasikci, Stefan Bucur, Gautam Altekar, Ion Stoica, Johannes Kinder, Edouard Bugion, Vitaly Chipounov, Horatiu Jula, Daniel Tralamazza, and Andreas Zeller. I am honored to have worked with such a talented team. I learned a lot during this process.

I was part of a great, close-knit, and technically-strong team. My lab mates Silviu Andrica, Baris Kasikci, Stefan Bucur, Radu Banabic, Vitaly Chipounov, Jonas Wagner, Johannes Kinder, Vova Kuznetzov, Horatiu Jula, and our nanny Nicoleta Isaac provided inspiration, a great deal of insights, and an unmatched team atmosphere. I thank you for all of that.

I thank Silviu Ganceanu, Vlad Ureche, and Martin Milata, for their contribution to execution synthesis, their help in building the systems described in this thesis was invaluable.

I thank my parents, Ecaterina and Marin Zamfir for their relentless support and guidance throughout my entire education, and most of all for the unbounded care with which they raised me. I am proud to be their son, and I am thankful for the values they instilled into me.

I thank Alexandra Covaci for her support and encouragement during my entire doctoral studies and for putting a large smile on my face every day.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Statement

This thesis aims to provide a solution for automatically debugging failures that occur in software running in production. Developers should be able to deterministically replay an entire execution or an execution suffix that is relevant for debugging the failed execution. The replayed execution should be useful for debugging, which means, at a minimum, that developers should be able to reproduce the same root cause and failure as the original execution. We consider that a practical solution should not require any program recording, any program modifications, and no changes to hardware. A practical solution should use as input only the program and the bug report (e.g., the coredump) that is generated after the failure and is sent to developers. This thesis is only concerned with failures that produce a coredump, such as program crashes or assert failures.

1.2 Motivation

Debugging software deployed in the real world is hard, frustrating, and typically requires deep knowledge of the code of the program. With increasing parallelism in both hardware and software, the classic problem of bugs in sequential execution is now being compounded by concurrency bugs and other hard-to-reproduce behavior. Bug reports rarely provide sufficient information about how the failure occurred, so developers must turn into detectives in search of an explanation of how the program could have reached the reported failure state. If developers had a better way to triage, analyze, and debug these failures, they would spend less time debugging and more time implementing useful features.

To fix a bug, developers traditionally try to reproduce it and observe its manifestation in a debugger. Alas, this approach is often challenging, especially for concurrency bugs—in a recent survey, almost 70% of respondents considered reproducing concurrency bugs to be hard or very hard [57]. Moreover, the large amount of guesswork involved in debugging leads to error-prone patches, with many concurrency bug fixes either introducing new bugs or, instead of fixing the underlying bug, merely decreasing its probability of occurrence [85]. Increasingly parallel hardware causes multi-threaded software to experience increasingly concurrent executions, making latent bugs more likely to manifest, yet no easier to fix.

There are multiple reasons why reproducing bugs is challenging: First, complex sequences of low-probability events (e.g., a particular thread schedule) are required for a bug to manifest, and programmers do not have the means of directly controlling such events. Second, the probe effect—unintended alteration of program behavior through the introduction of instrumentation and breakpoints [49]—can make bugs "vanish" when hunted with a debugger. Third, variations in the operating system and runtime environment (e.g., kernel or library version differences) may make it practically impossible to reproduce a bug exactly as it occurred at the end user's site.

This thesis sets out to address the question of how would one debug failures post-mortem with *no runtime recording* and *no execution control* in production—once the application fails, the ideal tool would use the information that can be collected "for free" after the failure (e.g., the coredump) to automatically infer how to make the program fail in the same way again. Such a tool would enable developers to home in on the root cause and fix it. This tool would essentially automate what developers do manually today.

1.3 Background

One way to do automated debugging is to record all key events during the real execution and, when a failure occurs, ship the log of these events along with the failure to the developers, who can then reproduce the execution that led to the failure. This is called deterministic record-replay [9, 42, 43, 77, 81].

Record-replay systems, however, are not an ideal solution, mainly because of performance and storage overheads. For example, making a multi-threaded execution on a multicore CPU reproducible requires logging a large number of memory operations, and this causes existing deterministic record-replay systems to have high performance overhead (e.g., 400% for SMP-ReVirt [42] and 60% for ODR [9]). Several systems choose to trade some of the reproducibility guarantees for lower runtime overhead [9, 17, 102, 35], but this trade-off hurts their utility for debugging [131]. In record-replay for datacenter applications [132], a big challenge is that such applications are data-intensive, and the large volume of data they process increases proportionally with the size of the system and the power of individual nodes. Recording all this data and storing it for debugging purposes is impractical; checkpointing can help trim the logs, but it increases recording overhead and still does not get rid of logs. Since recording must be always-on, to catch the occurrence of infrequent bugs (which are the hard ones to debug), such performance and storage overheads make record-replay impractical for debugging failures in most production systems.

Another option would be to use deterministic execution systems [13, 16, 38, 39], but they too are prohibitively heavyweight, especially for multi-CPU hardware. Deterministic execution systems also incur storage overhead, since they have to record program inputs in order to reproduce failures that occur in production.

A third option to do automated debugging is static analysis. Tools like PSE [87] and SherLog [126] use static analysis to improve error diagnosis and infer the likely cause of a failed execution. SherLog uses existing program logs to enhance the results of the static analysis. However, static analysis tools may produce imprecise results, because they do not produce an execution that can be deterministically replayed by developers in a traditional environment. Moreover, such tools focus on sequential programs, because static analysis for multi-threaded programs is more challenging due to the complexity of alias analysis [117]. The precision of these techniques can be improved by enhancing logs [129, 127], at the expense of adding runtime overhead.

1.4 Thesis Objectives

The key objectives of the automated debugging technique we develop in this thesis are: (1) the technique should reproduce an execution of the program that can help developers understand the failure and fix the root cause of the failure; (2) the technique should not require any runtime recording; (3) the technique should not require program modifications or specialized hardware.

Record replay techniques do not meet objective (2), because they must record information while software is running in production. Deterministic execution systems do not meet objective (3), because they require changes to the program, the hardware, or the production environment where the program is running. Static analysis tools have false positives and do not infer an execution that can be deterministically replayed to help debugging, therefore they do not meet objective (1).

1.5 Solution Overview

This thesis introduces *execution synthesis* (Chapter 4), a technique for automatically finding "explanations" for hard-to-reproduce bugs. Execution synthesis takes as input a program plus a bug report and produces an execution of that program that causes the reported bug to manifest deterministically. Execution synthesis requires no tracing or execution recording at the end user's site, making it well suited for debugging long-running, performance-sensitive software, like Web servers, database systems, application servers, game servers, etc.

Successful debugging with execution synthesis is premised on the observation that, in order to diagnose a given bug, a developer rarely needs to replay the *exact same* execution that evidenced the bug at the user's site. Instead, playing back *any* feasible execution that exhibits that same bug will typically be sufficient. The execution produced by execution synthesis provides an explanation of the bug, even if it is not precisely the execution experienced by the user reporting the bug. A synthesized execution provides the causality chain leading to the bug, thus eliminating the guessing and lengthy detective work involved in debugging. In addition to a bug report, developers now also have an execution they can play back in their debugger. This allows them to deterministically observe the buggy behavior and to use classic techniques for finding a suitable bug fix, such as step-by-step execution and data structure dumps.

Execution synthesis consists of two parts. *Sequential path synthesis* combines symbolic execution with context-sensitive inter- and intra-procedural static analysis to efficiently produce a guaranteed-feasible sequential execution path from the start of the program to any target basic block in each program thread. *Thread schedule synthesis* finds a schedule for interleaving thread-level sequential paths such that the program's execution exhibits the reported bug.

We prototyped the proposed technique in ESD, a tool that automatically analyzes common elements provided in bug reports (coredumps, stack traces, etc.), synthesizes an execution that leads to the reported misbehavior, and allows developers to play back this execution in a debugger. ESD is practical and scales to real systems. For example, it takes less than three minutes to synthesize an execution for a deadlock bug in SQLite, an embedded database engine with over 100,000 lines of C/C++ code [99] used in Firefox, Skype, Mac OS X, Symbian OS, and other popular software [108].

Based on the lessons learned from developing execution synthesis, we developed reverse execution synthesis (Chapter 5), a technique for automated debugging that targets arbitrarily long executions. The key difference between execution synthesis and reverse execution synthesis is that the latter focuses on reproducing just a suffix of the execution, while the former technique reproduces an entire execution (i.e., from the start of the program up to the failure point).

The insight behind reverse execution synthesis is that developers rarely need a *full execution* for debugging. A *suffix of the failure-bound execution* is sufficient for debugging as long as this suffix can be replayed in a debugger and contains the root cause of the failure. Reverse execution synthesis conceptually reverse-executes the program from the coredump and generates possible suffixes that generate the same coredump. The key technique we developed for reverse execution synthesis is the ability to execute suffixes of an execution starting from an unknown program memory state.

1.6 Thesis Roadmap

The rest of this thesis is structured in the following way: Chapter 2 describes prior work on automated debugging, Chapter 3 defines the debugging utility of an automated debugging system, Chapter 4 describes execution synthesis, and Chapter 5 describes reverse execution synthesis. Chapter 6 provides an empirical evaluation of execution synthesis and reverse execution synthesis and describes several uses cases of the two techniques. Chapter 7 describes how to improve the performance of execution synthesis by adding a lightweight recording layer. Finally, Chapter 8 presents future work ideas and Chapter 9 concludes the thesis.

Chapter 2

Related Work

In this chapter we review related work. Some of the described related work provided inspiration for execution synthesis and reverse execution synthesis, while other is related by virtue of addressing similar problems. We broadly divide the body of related work into record-replay systems (Section 2.1), deterministic execution systems (Section 2.2), bug finding tools (some of which focus on inferring inputs (Section 2.3.1), while others focus on finding schedules (Section 2.3.2)), debugging tools based on static analysis (Section 2.4), and bug triaging systems (Section 2.5).

2.1 Record-Replay Systems

A classic way to do automated debugging is to use a record-replay system [80]: record all relevant details of an execution (e.g., network and disk inputs, the order of asynchronous events, the thread schedule, etc.) into a log file, ship this log file to the developers' site and replay the recorded details of the execution from the log file. This section discusses several record-replay systems, focusing on their suitability for automated debugging of software that runs in production.

Record-replay systems are fundamentally different from execution synthesis primarily because execution synthesis does not record the execution. Record-replay systems have to record non-deterministic events, while execution synthesis must synthesize these events offline. Thus, on the one hand, the offline synthesis process is substantially more complex and expensive than simply replaying the recorded events from a log. On the other hand, the runtime recording process of record-replay systems has several disadvantages. First, the high runtime overhead makes record-replay systems impractical for use in production. Second, record-replay systems do not have predictable performance guarantees (their overhead varies depending on the recorded program, the workload, and the hardware platform). Third, record-replay systems are complex pieces of software running underneath the recorded program, therefore they may themselves be responsible for introducing bugs.

Whole-system replay records the execution with high fidelity: the program is run inside a specialized virtual machine, which records all sources of non-determinism for later replay [42, 43, 6]. Recording is done below the virtual machine. The main challenge is recording and replaying asynchronous events. ReVirt [42] uses a lightweight mechanism based on hardware performance counters to record the point in the program's execution when asynchronous events occur, and then replays events at the recorded execution point. ReVirt was shown to have less than 8% overhead [42], but only works for recording the execution of single-CPU virtual machines. SMP-ReVirt [43] extends ReVirt to multi-processor machines, using commodity hardware page protection to record and accurately replay memory sharing between multiple CPUs. Recording the order of shared memory accesses with low runtime overhead is the main challenge faced by record-replay systems: SMP-ReVirt is expensive (e.g., 400% runtime overhead), therefore it is mostly suitable for use during development. Scribe [77] makes several optimizations to improve the performance of multi-processor recording to 15% for a 2-CPU system. Execution synthesis transforms the problem of recording shared memory accesses into that of synthesizing a thread schedule offline.

Whole-system replay works well for bugs that occur relatively frequently. However, bugs in production are rare occurrences, so the performance and space overhead of always-on recording of the entire execution offers less payback. Reverse debugging [75] uses VMs to travel back and forth in an execution, which is useful in dealing with hard-to-reproduce bugs; this approach typically incurs prohibitive recording overhead for bugs that occur infrequently. In contrast, execution synthesis does not require recording, so it presents unique advantages in dealing with rare events, such as concurrency bugs.

Higher-level record-replay systems record just the target processes to reduce runtime overhead (e.g., Flashback [109], Chimera [81], ADDA [132]), or record library-level interactions and replay them (e.g., R2 [60]). These approaches typically incur lower overhead than whole-system replay. For instance, recording at the SQL interface in R2 can reduce the log size by up to 99%, compared to doing so at the Win32 API. R2 does not record all sources of non-determinism (e.g., data races, multi-processor non-deterministic events, etc.), therefore it may not always produce a faithful replay. Execution synthesis uses techniques similar to R2 to playback the synthesized execution and extends these techniques with the ability to play back asynchronous events (such as thread preemptions) that are crucial to reproducing concurrency bugs.

One way to reduce the overhead of recording shared memory accesses was introduced by Chimera [81]: use static analysis to identify code without data races, and only record the order of memory accesses that are potentially racing. Systems based on static analysis require source code as input (accurate static analysis is harder for program binaries [19]), therefore, they are not suitable for whole-system record-replay. Moreover, Chimera makes several assumptions about the program's source code (e.g., lack of pointer arithmetic) to guarantee that static data race analysis has no false negatives. If the static analysis it employs has false negatives, Chimera does not guarantee deterministic replay.

Recent work looked at replaying bugs while aiming to reduce recording overhead [9, 102, 35, 67]. ODR [9] and PRES [102] reproduce concurrency bugs for programs running on multiprocessors, while Oasis [35] and BugRedux [67] reproduce bugs in sequential programs. All four systems trade the fidelity of the recording (i.e., they do not record some details of the execution) for achieving lower runtime overhead, however, they rely on an offline inference process to reconstruct the information that was not recorded. For instance, both ODR and PRES have a mode in which they do not record the order of the racing memory accesses. Instead, these systems perform an offline search through the set of possible thread schedules to infer the order of the racing memory accesses offline. Oasis records a trace of the executed branch instructions and uses a training phase—before the program is deployed at the user site—to determine which parts of the branch trace can be easily inferred offline. BugRedux records a trace of the called functions to achieve a good tradeoff between recording overhead and inference time. While similar in spirit to execution synthesis, these tools do not fully eliminate the need for recording, adding overheads as high as 50%, making them less practical for production systems.

Long-running executions pose an important challenge for record-replay systems. One option to record-replaying long running executions is to use checkpointing [6] and to replay starting from a recent checkpoint instead of the beginning of the execution. Another approach is to focus on replaying an execution suffix. BBR [28] is a record-replay system targeted at long-running single-threaded applications. BBR records in a circular buffer pieces of information about the last seconds of the execution (i.e., a trace of the branches taken by the program, consisting of a single bit for each branch taken) and reconstructs the missing pieces offline. Compared to systems like ODR and PRES, BBR focuses on reproducing a short execution suffix, therefore the recorded trace is small and the runtime overhead to under 10%. Like reverse execution synthesis, BBR targets long running applications and reproduces an execution suffix. However, reverse execution synthesis does not entail any recording and works for multi-threaded instead of single-threaded programs.

Aftersight [30] is an efficient way to observe and analyze the behavior of running programs on production workloads. Aftersight decouples analysis from normal execution by logging non-deterministic virtual machine inputs and replaying them on a separate analysis platform or in real time on a different core of the system, thus enabling synchronous safety monitors that perform realtime analysis to detect safety violations. Execution synthesis does not monitor the running program. Instead, it requires a coredump to perform its analysis at the developer's site. However, since the analysis is done offline, execution synthesis is not appropriate for realtime analysis.

Compared to replaying a single node, record-replay for distributed applications poses additional challenges, due to the inherent recording overheads. First, applications are data intensive, and the volume of the data they process increases proportionally with the size of the system. Recording this volume of data to persistent storage is even a larger challenge than for a single node. Second, coordinating a set of distributed nodes to perform a faithful replay of a failed execution requires having captured all critical causal dependencies between control messages exchanged during execution. Knowing a priori which dependencies matter is undecidable. Existing work on debugging distributed systems does not fully address these challenges: Friday [50] and Liblog [51] address distributed replay, but have high overhead for data-intensive applications. ADDA [132] provides distributed replay for datacenters, but has non-negligible overhead for large datacenters due to both the storage and multi-core recording overheads. Execution synthesis and reverse execution synthesis can be applied to individual application processes, but do not address the debugging of an entire distributed system.

One way to reduce runtime overhead is to use specialized hardware ([124, 93]) to do the recording. For instance, FDR [124] piggybacks on the cache coherence hardware to record thread ordering information. While this approach can help debugging, it requires hardware features that are not available today and that are uncertain to exist in the future. Hardware mechanisms such as LBA [27] were used to enable efficient and flexible logging and extraction of run time execution events for debugging and security purposes. LBA was used to accelerate dynamic software-based analyses by offloading the dynamic analyses to idle cores. iWatcher [138] is a hardware mechanism that enables associating monitoring functions with specific memory location, thus accelerating memory safety tools like Valgrind [114]. Execution synthesis could leverage future specialized hardware recording systems to reduce synthesis time, assuming that the specialized will not introduce runtime overhead and will become part of commodity hardware.

Record-replay systems like ReVirt [75], FlashBack [109], UndoDB [5], and VMWare Workstation 7.0 [6] leverage both record-replay and checkpointing to provide reverse debugging capabilities [1] (e.g., execute *reverse-step* in a debugger). While the program appears to be executing in reverse in the debugger, it is in fact executing forward from a previously taken snapshot. Execution synthesis provides a similar functionality, however it does not use checkpointing to implement the *reverse-** commands in gdb. Instead, it replays the synthesized execution from the beginning of the execution.

Program sampling [83] has been proposed as a way to share the cost of dense code assertions among many users. This technique uses statistical methods to identify program behaviors that are strongly correlated with failure, therefore it can be applied to debug a wide range of bugs. Sampling does not extract from the program execution bug-specific information, therefore debugging non-deterministic bugs such as deadlocks and data races is still hard. In Section 7.1 we introduce a system that gathers bug-specific information: rather than identifying behaviors correlated to failures, this approach requires a predefined recording layer that is optimized for a specific class of bugs.

2.2 Deterministic Execution

Another approach to automated debugging is to make program execution deterministic using a deterministic execution system [13, 16, 38, 39, 100]. Deterministic execution systems use deterministic schedulers to execute a non-deterministic program using a particular thread schedule (i.e., a particular order of the synchronization operations and of the shared memory accesses): as long as the program is run using the same inputs, it is guaranteed to run the same thread schedule. The thread schedule is typically chosen arbitrarily among the possible thread schedules. Deterministic execution systems are prohibitively heavyweight, especially for multi-CPU systems and also incur storage overhead, since they have to record program inputs in order to reproduce failures that occur in production.

Deterministic execution systems could be used in conjunction with execution synthesis: the more deterministic programs are (e.g., if they are guaranteed to experience a single possible ordering of any racing memory accesses), the easier it will be for execution synthesis to synthesize the thread schedule. One challenge in using deterministic execution in conjunction with execution synthesis is that each program input leads to a different deterministic schedule, therefore one would still need to record all program inputs, which is not practical for production systems. Peregrine [36] partially alleviates this challenge by reusing the same deterministic thread schedule for multiple sets of inputs that drive the program down the same execution path.

2.3 Bug Finding Tools

2.3.1 Finding Inputs That Trigger Bugs

There is a rich body of work focused on discovering bugs in programs [105, 44, 56, 55, 23], with recent tools typically employing symbolic execution [74]. Execution synthesis builds upon techniques developed for these systems, most notably KLEE [23].

In combining static analysis with symbolic execution, we were inspired by a series of systems [34, 33, 24] which compute inputs that take a program to a specified undesired state, such as the location of a crash. Unlike execution synthesis, these systems are targeted at program states that can be deterministically reached when given the right program arguments. Execution synthesis was specifically motivated by the difficulty of reproducing elusive non-deterministic bugs, hence our emphasis on inferring not only program arguments, but also inputs from the program's environment and scheduling decisions. Moreover, these prior tools require recording of certain program inputs and/or events; in execution synthesis we go to the extreme of zero program tracing, in order to be practical for production systems.

Static analysis and symbolic execution were used to create vulnerability signatures [21] and to show that it is possible to automatically create exploits by analyzing program patches [20]. Execution synthesis is similar to this work in that it aims to create inputs that execute the program toward a certain vulnerability. However, execution synthesis addresses bugs more broadly than just input validation bugs and is able to handle multi-threaded programs. Moreover, automatic patch-based exploit generation works best when the constraint formula is partially generated from an existing sample execution. We did not yet explore this approach in execution synthesis because sample executions may add constraints on program inputs, that may prevent execution synthesis from reproducing a particular bug.

AEG [12] is an automated exploit-generation system. It first uses static analysis to find potential bug locations in a program, then uses a combination of static and dynamic analysis to find an execution path that reproduces the bug, and then generate an exploit automatically. AEG generates exploits, which provide evidence that the bugs it finds are critical security vulnerabilities. Instead, execution synthesis takes as input the manifestation of a known bug and works for both exploitable and non-exploitable bugs. Unlike execution synthesis, AEG does not start from an existing bug and does not leverage the rich source of information present in the coredump. Moreover, AEG is targeted at buffer overflow bugs in sequential programs, while execution synthesis also works for concurrency bugs. AEG [12] was developed after execution synthesis.

UC-KLEE [103, 45] introduced under-constrained execution as a way to test the equivalence of two arbitrary C functions. Under-constrained execution runs a function in isolation from the rest of the program by providing symbolic arguments to the function and then using symbolic execution to explore paths inside the function. This approach can generate paths that would be infeasible in a real execution. UC-KLEE partially mitigates this problem by checking the equivalence (in terms of program output) of two library functions that are meant to provide the same functionality. Thus, both feasible and infeasible paths are likely to generate the same output in both functions. UC-KLEE flags any output differences as potential bugs. The approach used by reverse execution synthesis to execute symbolic snapshots (Chapter 5) was inspired by under-constrained execution. Reverse execution synthesis extends under-constrained execution to concurrent programs.

Several bug-finding systems published after execution synthesis combine static and dynamic program analysis techniques to prioritize testing program patches [89, 110] or to find security vulnerabilities [14]. Similarly to how execution synthesis prunes execution paths that cannot reproduce the failure, eXpress [110] prunes execution paths which do not lead to the target patched code. Similarly to execution synthesis, several recent systems use distance-based heuristics to evaluate which execution paths are more likely to reach a particular program location (e.g., patched code [89] or security vulnerabilities [14]). These systems are also addressing a program reachability problem, but they are designed to search for bugs instead of trying to reproduce a particular bug. Thus, unlike execution synthesis and reverse execution synthesis, they do not leverage the information in the bug report and they do not handle concurrency bugs.

2.3.2 Finding Thread Schedules that Trigger Bugs

Even though program testing is different from debugging, we drew inspiration for schedule synthesis from tools that search for concurrency bugs, like Chess [94], DeadlockFuzzer [70], and CTrigger [101]. Still, there exist major differences. These tools exercise target programs in a special environment and, when a bug occurs, the tools are able to replay those bugs. In contrast, execution synthesis reproduces bugs discovered in the field by end users, in which case requiring the program to run in a special setting is not feasible. These tools also require the existence of a test case that provides all required program input, whereas execution synthesis automatically infers this input.

Another important difference appears in the use of heuristics. Chess, for example, employs a technique called iterative context bounding (ICB) [94]. ICB assumes that prioritizing executions with fewer context switches is an efficient way to find concurrency bugs, so Chess repeatedly runs an existing test case, each time with a different schedule, and limits the total number of possible context switches, as in ICB. When searching for a specific bug, we found execution synthesis to be much faster. Also, execution synthesis achieves scalability without having to bound the number of context switches. However, Chess's goals are different from those of execution synthesis, so direct performance comparisons must be done carefully.

Similarly to RaceFuzzer [106], execution synthesis dynamically detects potential data races and performs context switches before memory accesses suspected to be in a race. However, execution synthesis is more precise, because it is targeted at a specific bug and uses checkpoints to explore alternate thread interleavings, unlike RaceFuzzer's random scheduler. Moreover, by using symbolic execution, execution synthesis can achieve substantially higher coverage for data race detection.

CTrigger [101] finds atomicity violation bugs by identifying unserializable interleavings and then exercising small perturbations in the thread schedule to identify the atomicity violation. Execution synthesis can reproduce data race bugs, but it does not explicitly handle atomicity violation bugs. However, if reproducing the failure requires synthesizing a particular atomicity violation, one could leverage the techniques developed by CTrigger during the synthesis process.

Most other testing tools based on symbolic execution [55, 23] work only for single-threaded programs, while execution synthesis enables symbolic execution for multi-threaded programs.

Prior work [118] analyzes coredumps to reconstruct the thread schedule that caused a crash. Their technique runs the program with the initial inputs—which need to be recorded in production—and generates a coredump at the same program counter as the original coredump. If the coredumps do not match, this technique analyzes the differences between the two coredumps and leverages the execution index [122] to generate a failure-inducing thread schedule. This technique introduces runtime overhead because it assumes the program inputs are recorded and must also record various pieces of runtime information in order to reconstruct the execution index.

2.4 Static Analysis

PSE [87] and SherLog [126] use backward static analysis to improve error diagnosis. PSE performs a static backward dataflow analysis based on precise alias analysis for a value of interest from the failure point to where the value originated. SherLog leverages existing program logs to infer execution suffixes that explain the logs. SherLog uses a path-sensitive backward static analysis based on Saturn [41] and a constraint solver to identify must-paths (partial execution paths that were definitely executed), may-paths (partial execution paths that may have been executed), or must-not-have (infeasible execution paths), and then stitched them together to form possible execution suffixes that explain the failure. Based on a study of logging practices [128] in open source software, SherLog was extended to perform runtime logging pro-actively [127, 129] to gather—with low overhead—more runtime information that is relevant for debugging and that can improve the accuracy of its static analysis.

Tools based on static analysis [87, 37] do not infer a guaranteed-to-be-feasible path, since, unlike execution synthesis, they do not synthesize the inputs that were not recorded. These tools are efficient, but work at a higher level of abstraction, which is a source of false positives. Reverse execution synthesis also uses the coredump and dynamic analysis to obtain more accurate suffixes than techniques based only on static analysis. Moreover, (forward) execution synthesis provides fully accurate executions by reconstructing a full execution trace that can be played back in a debugger.

!exploitable [92] is a debugging tool based on static analysis that assigns exploitability ratings to crashes. *!exploitable* uses heuristics, and unfortunately this can lead to both false positives and false negatives. Execution synthesis can improve the accuracy of *!exploitable* if it succeeds in synthesizing a full execution path. Otherwise, reverse execution synthesis provides an execution suffix that explains the failure, which can further improve the accuracy of *!exploitable*'s classification.

In some sense, reverse execution synthesis is like computing weakest preconditions [40] for the coredump (i.e., the coredump can be seen as an extraordinarily large postcondition). Interprocedural weakest precondition computation is hard for imperative programs. The state-of-the art weakest precondition computation tools [21, 25] do not work for concurrent programs, do not leverage the coredump, and assume some level of recording [21]. The full use of the coredump, the accurate memory handling, and the support for concurrent programs are key differentiators of reverse execution synthesis from work on weakest precondition computation.

Execution synthesis drew inspiration from the large body of prior work on model checking [61, 15, 65, 98, 62, 130, 48]. Given a program safety property, a model checker either proves that the property holds on all possible execution paths of the program or finds an execution that violates the property. If the property is undecidable, a model checker may not terminate. Execution synthesis also aims to solve a program reachability problem, however, it is focused on a particular failure, while model checkers are typically used for finding bugs or proving program properties. Moreover, unlike execution synthesis, model checkers do not leverage the coredump and they require a model of the program environment.

Model checkers like SLAM [15] and BLAST [61] use CEGAR (counterexample-guided abstraction refinement): they start with a coarse abstraction of the program and gradually refine the abstraction if they encounter a violation of the property. The advantage of over-approximating the set of possible paths using program abstractions is that it becomes feasible to prove that a property is safe without enumerating all possible program paths and thread schedules. However, to find a path that violates the safety property, an abstraction-based model checker still needs to fully refine the program abstraction. BLAST [61] improves over SLAM [15] by using lazy predicate abstraction (i.e., it refines the program abstraction on demand and non-uniformly at each program location) and interpolation-based predicate discovery (i.e., a way to refine predicates in order to efficiently prove that an execution path of the abstracted program is infeasible). Execution synthesis uses real executions instead of program abstractions. Abstractions are useful for proving a safety property, while execution synthesis deals with finding execution paths that violate a property (the bug report already evidences that the property can be violated). On the one hand, a model checker would have to fully refine the abstraction in order to find a path that explains a given bug report, therefore predicate abstractions—as used by BLAST and SLAM— would not be useful for execution synthesis. On the other hand, predicate abstraction could be useful to trim the search space of execution synthesis: an abstraction could quickly determine if a given program state cannot reach the program location where the failure occurred, while execution synthesis might have to enumerate a potentially large set of execution paths that cannot be trimmed statically.

2.5 Bug Triaging

Bug triaging faces an important scalability concern when dealing with many users. Traditional approaches (e.g., users call technical support to report a problem) do not scale to many users. Bug triaging requires automatically obtaining bug reports and submitting them to the developers, where they are stored in a database that enables prioritization, complex queries on the data, spotting trends and testing hypothesis.

The state of the art in automated bug reporting and triaging systems are Windows Error Reporting [54] and Google Breakpad [58]. These systems collect bug reports from a large number of users (e.g., WER collected billions of error reports in ten years of operation). These bug reports reveal some information about the bug (e.g., the end state of the application), but not how the application got there. Typically, WER bug reports provide a minidump (a partial coredump) and a coarse-grained description of the hardware and software installed at the user machine. WER uses error statistics to isolate bugs at a large scale and also to prioritize bugs that impact multiple users. WER also enables empirical analyses of hardware failures [97] at a large scale, showing that hardware induced failures are recurrent and CPU fault rates are correlated with the number of executed cycles.

Triage [113] performs automatic debugging at the end-user site. It uses checkpointing to repeatedly replay the moment of failure, analyze the failed execution to infer the cause of the bug using delta debugging [136], and provide a detailed diagnostic report. Triage requires that the execution be checkpointed frequently, which introduces non-negligible overhead. Moreover, Triage records all program inputs, which makes it impractical for long running executions with large input streams. One important difference between Triage and execution synthesis is that execution synthesis does not pinpoint the root cause of a failure, while Triage identifies the root cause of the failure using delta debugging. F3 [68], a recent technique that postdates execution synthesis demonstrated that it is feasible to combine execution synthesis, delta debugging, and lightweight recording in order to identify the root cause of a failure.

The privacy leaked by a coredump sent to a bug triaging system can be mitigated by designing special APIs. For instance, .NET provides a way to declare private data and store it in encrypted containers, ensuring this data is no leaked in the coredump. A solution to anonymize bug reports [24] works by replacing concrete (potentially private) inputs with inputs that take the program down the same execution path, but leak less data (e.g., send out a bug report that replaces the credit card information with random numbers). If execution synthesis is performed at the user's site instead of the developer's site, it could extend the benefits provided by anonymized bug reports [24], since it does not synthesize the exact original execution. Therefore, execution synthesis, could be used to generate an anonymized coredump that evidences the same failure, and ship it to developers. We leave this for future work.

Drawing inspiration from execution synthesis, we developed Recore [82], a technique for reproducing bugs in Java programs. Recore starts from a Java memory dump and stack trace (the execution synthesis equivalent of a coredump generated by a C/C++ program) and uses genetic algorithms to create a program that generates a similar coredump. Recore's fitness function is based on the similarity of the stack and the heap of the original coredump and the state of a candidate execution state. To speed up the search, Recore uses the memory values in the coredump to determine which program arguments to pass to the functions in the constructed program. An important difference between Recore and execution synthesis is that Recore constructs a different program that generates a similar coredump, while execution synthesis uses the original program. Another approach that uses genetic algorithms for reachability analysis is Fitnex [121]. Fitnex extends Pex [112] with the ability to derive program inputs based on a fitness function. This function evaluates how close a particular execution path is to a particular program location. The reachability analysis in execution synthesis is more general, in that it combines static analysis, proximity-guided dynamic search, and algorithms to reproduce concurrency bugs.

Chapter 3

Debug Determinism: The Holy Grail of Automated Debugging

Deterministic replay tools offer a compelling approach to debugging hard-to-reproduce bugs. Recent work on relaxed-deterministic replay techniques shows that debugging using a record-replay system is possible with low in-production overhead [9, 102, 35, 133]. However, despite considerable progress, a record-replay system that offers not only low in-production runtime overhead but also high debugging utility remains out of reach. To this end, we argue for *debug determinism*—a new determinism model premised on the idea that effective debugging entails reproducing the *same failure* and the *same root cause* as the original execution. Debug determinism provides a reasoning framework for how to trade recording overhead for debugging utility. This chapter presents ideas on how to achieve and quantify debug determinism.

3.1 A Determinism Model Focused on High Debugging Utility

We argue that the ideal automated debugging system should provide *debug determinism*. Intuitively, a debug-deterministic system produces an execution that manifests the same *failure* and the same *root cause* (of the failure) as the original execution, hence making it possible to debug the application. The key challenge in understanding debug determinism is understanding exactly what is a failure and what is a root cause:

A **failure** occurs when a program produces incorrect output according to an I/O specification. The output includes all observable behavior, including performance characteristics. Along the execution that leads to failure, there are one or more points where the developer can fix the program so that it produces correct output. Assuming such a fix, let P be the predicate on the program state that constrains the execution—according to the fix—to produce correct output. The **root cause** is the negation of predicate P.

A *perfect implementation* fully satisfies the I/O specification, that is, for any input and execution it generates the correct output. A deviation from the perfect implementation may lead to a failure. So, more intuitively, this deviation represents the root cause.

In identifying the root cause, a key aspect is the boundary of the system: e.g., if the root cause is in an external library (i.e., the developer has no access to the code), a fix requires replacing the library. Else, if the library is part of the system, the fix is a direct code change.

Debug determinism is the property of a replay-debugging system that it consistently reproduces an execution that exhibits the same root cause and the same failure as the original execution.

For example, to fix a buffer overflow that crashes the program, a developer may add a check on the input size and prevent the program from copying the input into the buffer if it exceeds the buffer's length. This check is the predicate associated with the fix. Not performing this check before doing the copy represents a deviation from the ideal perfect implementation, therefore this is the root cause of the crash. A debug-deterministic system replays an execution that contains the crash and in which the crash is caused by the same root cause, instead of some other possible root cause for the same crash.

The definition of the root cause is based on the program fix, which is knowledge that is unlikely to be available before the root cause is fixed—it is akin to having access to a perfect implementation. In this chapter we discuss how to achieve debug determinism without access to this perfect implementation.

Replay-debugging techniques [9, 13, 17, 43, 102, 133] offer a compelling approach to dealing with non-deterministic failures. A replay debugger produces an execution that is similar to the original failed execution. The hope is that the developer can then employ traditional cyclic-debugging techniques or automated analyses on the generated execution to isolate the defect causing the failure. Many kinds of replay techniques have emerged over the years, differing primarily in how they deal with non-deterministic events (e.g., inputs, scheduling order, etc.). Record-replay techniques [9, 17, 43, 102], for example, record non-deterministic events at runtime. Deterministic execution techniques [13], eliminate non-determinism (e.g., by precomputing scheduling order) to ensure deterministic replay. Finally, inference-based techniques [9, 102, 35, 133] provide replay by computing unrecorded non-deterministic events after the original execution has finished.

Despite a plethora of replay techniques, a truly practical replay debugger remains out of reach. The traditional obstacle has been high runtime overhead that is unacceptable in production environments. Alas, this is exactly where most unexpected and hard-to-reproduce bugs often surface. It seems clear now, however, that in-production overhead is not an impenetrable barrier. In particular, recent work on relaxed-determinism models [9, 102, 35, 133] shows that, by making fewer guarantees about the execution properties that are reproduced, one can shift runtime overhead from production time to debugging time. The failure determinism model provided by execution synthesis [133], for example, guarantees only that the replayed execution exhibits the same final failure state as the original execution. In so doing, it avoids the need to record non-determinism, but has to infer it after the failure.

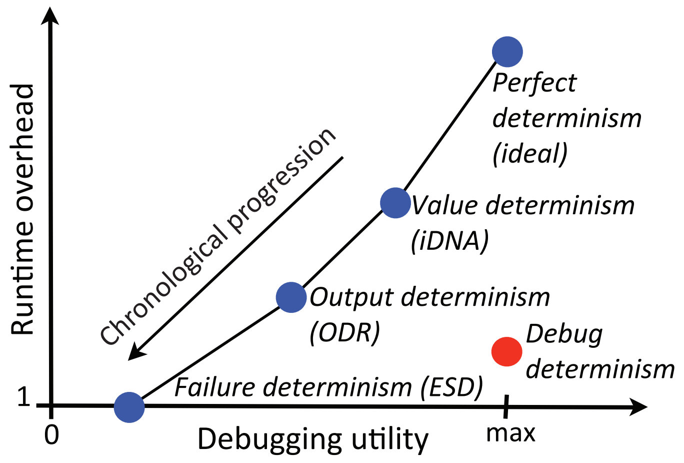


Figure 3.1: Trend in determinism relaxation: recent relaxed systems reduce runtime overhead, but forego debugging utility. The figure is not based on new measurements. It shows the current trend in relaxation based on published results.

While trying to satisfy the low runtime overhead requirement, designers of modern replay systems may have ignored another equally important one: effective debugging. Systems that provide relax determinism (plotted qualitatively in Fig. 3.1) have traded debugging utility for low runtime overhead.

We argue that a replay debugger should strive not only for low runtime overhead but also for high debugging utility. This introduces two questions: what is high debugging utility, and how do we get it?

Debug Determinism. To answer to the first question, this chapter describes a new determinism model called "debug determinism". The key observation behind debug determinism is that, to provide effective debugging, it suffices to reproduce *some* execution with the *same failure* and the *same root cause* as the original. A debug-deterministic replay system enables a developer to backtrack from the original failure to its root cause.

Root Cause-Driven Selectivity. One way to achieve debug determinism is to precisely record or precompute the portions of the execution containing only the failure and its root cause, while relaxing the recording everywhere else. Unfortunately, this approach is infeasible, as the root cause of a failure is not known *a priori*. To this end, we give several heuristics that approximate this ideal approach by *predicting* the portions of the execution containing the root causes.

The key challenge in achieving debug determinism is that the notion of root cause is subjective—

a detailed and correct developer-provided specification is needed to precisely identify it. In reality, such a specification is rarely available. Thus, we give several heuristics to approximate debug determinism in the absence of such a specification.

3.2 The Impact of Relaxed Determinism on Debugging Utility

This section describes several replay determinism models and the problems that arise when over-relaxing determinism.

Failure determinism, implemented by execution synthesis, ensures that the replay exhibits the same failure as the original run. Execution synthesis does not do any recording. Instead, it extracts the failure information from a bug report or coredump and uses post-factum program analysis to infer an execution that exhibits the same failure and can be replayed in a debugger.

Output determinism, implemented by ODR [9], ensures that the replay produces the same output as the original run. ODR uses several recording schemes. In the most lightweight scheme, ODR records just the outputs of the original run and infers all unrecorded non-determinism. Scaling this inference process is hard, therefore ODR provides another scheme that also records the program inputs, the execution path, and the scheduling order. However, ODR does not record the causal order of the racing instructions running on different CPUs. Instead, it uses symbolic execution to infer the values that were read by the racing instructions.

Value determinism, implemented by iDNA [17], ensures that a replay run reads and writes the same values to and from memory at the same execution points as the original run. Value determinism does not guarantee causal ordering of instructions running on different CPUs, thus requiring more effort from the developer to track causality across CPUs.

Relaxed determinism models (e.g., ODR [9], execution synthesis [133], PRES [102]) assume that debugging is possible regardless of the degree of relaxation. For some bugs, this is not true: relaxed models may not be able to reproduce the failure, hence making it hard to backtrack to and fix the underlying defect (i.e., root cause). For other bugs, these models help reproduce the failure, but may not reproduce the original root cause, hence potentially deceiving the developer into thinking that there isn't a problem at all. Finally, for some bugs, a significant amount of runtime information may need to be reconstructed, leading to prohibitively large post-factum analysis times.

To see how failures may not be reproduced under relaxed determinism models, consider a program that outputs the sum of two numbers. Suppose, however, that the program has a bug such that for inputs 2 and 2, it outputs 5. To replay this execution, an output deterministic replay system (which guarantees only that the replay run exhibits the same outputs [9]) may produce an execution in which the output is 5 (like the original), but the inputs are 1 and 4. 1 plus 4, however, *is* 5 and thus is not a failure at all, much less the original failure. Unfortunately, without an execution exhibiting the original failure, developers cannot determine the true root cause of the faulty arithmetic (e.g., an array indexing bug).

To see how root causes may not be reproduced under ultra-relaxed determinism models, and why that can trick the developer into thinking there isn't a problem at all, consider the case of a server application that drops messages at higher than expected rates. Unbeknownst to the developer, the true root cause of this failure is a race condition on the buffer holding incoming messages. However, an output- or failure-deterministic replay debugger may not reproduce the true root cause. Instead, it may produce an execution in which the packets were dropped due to network congestion. Network congestion is beyond the developer's control and thus she naturally, yet mistakenly, assumes nothing more can be done to improve the program's performance. In the end, the true root cause (a race condition) remains undiscovered.

3.2.1 Focusing on the Root Cause to Achieve Debug Determinism

The definition of debug determinism suggests a simple strategy for achieving it in a real replay system: record or precompute just the root cause events and then use inference to fill in the missing pieces. However, the key difficulty with this approach is in identifying the root cause events. One approach is to conservatively record or precompute all non-determinism (hence providing perfect determinism during replay), but this strategy results in high runtime overhead. Another approach is to leverage developer-provided hints as to where potential root causes may lie, but this is likely to be imprecise since it assumes a priori knowledge of all possible root causes.

To identify the root cause, we observe that, based on various program properties, one can often *guess* with high accuracy where the root cause is located. This motivates our approach of using heuristics to detect when a change in determinism is required without actually knowing where the root cause is. We call this heuristic-driven approach *root cause-driven selectivity (RCSE)*. The idea behind RCSE is that, if strong determinism guarantees are provided for the portion of the execution surrounding the root cause and the failure, then the resulting replay execution is likely to be debug-deterministic. Of course, RCSE is not perfect, but preliminary evidence (Section 3.4) suggests that it can provide a close approximation of debug determinism.

Next, we present several variants of RCSE.

Code-Based Selection

This heuristic is based on the assumption that, for some application types, the root cause is more likely to be contained in certain parts of the code. For example, in datacenter applications like Bigtable [26], a recent study [10] argues that the control-plane code—the application component responsible for managing data flow through the system—is responsible for most program failures.

This observation suggests an approach in which we identify control-plane code and reproduce its behavior precisely, while taking a more relaxed approach toward reproducing data-plane code. Since control-plane code executes less frequently and operates at substantially lower data rates than data-plane code, this heuristic can reduce the recording overhead of a replay-debugging system. The key challenge is in identifying control-plane code, as the answer is dependent on program semantics. One approach is suggested in [10] and we implemented it in ADDA [132]. ADDA deems code that processes inputs at a low data rate as control-plane, since data plane code often operates at high data rates. We provide more details on this approach in Section 3.4.

Data-Based Selection

Data-based selection can be used when a certain condition holds on program state. For instance, if the goal is to reproduce a bug that occurs when a server processes large requests, developers could make the selection based on when the request sizes are larger than a threshold. Thus, high determinism will be provided for debugging failures that occur when processing large requests.

A more general approach is to watch for a set of invariants on program state: the moment the execution violates these invariants, it is likely executing an error path. This is a good heuristic to increase the determinism guarantees for that particular segment of the execution. Ideally, assuming perfect invariants (or specification), the root cause and the events up to the failure will be recorded with the highest level of determinism guarantees. If such invariants are not available, one could use dynamic invariant inference [46] before the software is released. While the software is running in production, the replay-debugging system monitors the invariants. If the invariants do not hold, the system switches to high determinism recording, to ensure the root cause is recorded with high accuracy.

Combined Code/Data Selection

Another approach is to make the selection at runtime using dynamic triggers on both code and data. A trigger is a predicate on both code and data that is evaluated at runtime in order to specify when to increase recording granularity. An example trigger is a "potential-bug detector". Given a class of bugs, one can in many cases identify deviant execution behaviors that result in potential failures [134]. For instance, data corruption failures in multi-threaded code are often the result of data races. Low-overhead data race detection [69] could be used to dial up recording fidelity when a race is detected.

Therefore, triggers can be used to detect deviant behavior at runtime and to increase the determinism guarantees onward from the point of detection. The primary challenge with this approach is in characterizing and capturing deviant behavior for a wide class of root causes. For example, in addition to data races, data corruption may also arise due to forgetting to check system call arguments for errors, and increasing determinism for all such potential causes may increase overhead substantially. A compelling approach to create triggers is to use static analysis to identify potential root causes at compile time and synthesize triggers for them.

All heuristics described above determine when to dial up recording fidelity. However, if these heuristics misfire, dialing down recording fidelity is also important for achieving low-overhead recording. For code-based selection, we can dial down recording fidelity for data-plane code. For trigger-based selection, we can dial down recording fidelity if no failure is detected and no trigger fired for a certain period of time.

3.2.2 Assessing Debug Determinism

So far, work on replay-debugging has not employed metrics that evaluate debugging power. Instead, the comparison was mainly based on recording performance figures and ad-hoc evidence of usefulness in debugging. Instead, we propose a metric aimed at encouraging systematic progress toward improving debugging utility.

Debugging fidelity (*DF*) is the ability of a system to reproduce accurately the root cause and the failure. Assume that a system reports k executions, out of which, $k_{original}$ is the number of executions that reproduce the correct root cause and the failure, k_{fp} is the number of false positives (i.e., they report either the failure or the root cause incorrectly), and k_{other} is the number of executions that reproduce the failure, but reproduce a feasible root cause, yet this root cause is different from the original root cause. Thus, $k = k_{original} + k_{fp} + k_{other}$. If an execution produced by the debugging system does not reproduce the failure, debugging fidelity is 0, because developers cannot inspect how the program reaches failure. If the system reproduces the original root cause and the failure, debugging fidelity is 1. If an execution reproduces the failure, but a different root cause from the original, debugging fidelity is *1/n*, where n is the number of possible root causes for the failure observed in the original execution. This definition takes into account the fact that a replayed execution is still useful for debugging even if it reproduces the failure through a different root cause, yet the replay is useless for debugging if it does not reproduce the failure.

Thus, debugging fidelity is $DF = \frac{k_{original} + \frac{k_{other}}{n}}{k_{original} + k_{other} + k_{fp}}$, where n is the number of feasible root causes for the same failure. For instance, a system that reproduces the original root cause and failure through a single execution ($k = 1$) has $DF = 1$, a system that has a false positive has $DF = 0$, and a system that reproduces the original failure but a different root cause than the original has $DF = 0.5$.

This definition is more general than the one we proposed in [131]. Unlike the definition in [131], this definition allows assessing the debugging fidelity of systems that (1) replay or synthesize more than one path and (2) can have false positives.

It may be difficult to analytically determine a replay system's debugging fidelity. However, it is possible to determine it empirically. For instance, static analysis could be used to identify the location of all possible root causes for a certain failure, potentially including false positives. One can then manually weed out the false positives and check if the system can replay all of the true positives. Another approach is to empirically test if a replay-debugging system correctly replays when the given root causes are guaranteed to be present in the original execution through some other means (e.g., deterministic execution).

Debugging efficiency (*DE*) is the duration of the original execution divided by the time the tool takes to reproduce the failure, including any analysis time. Normally this metric has values less than 1, but it is possible for techniques such as execution synthesis [133] to synthesize a substantially shorter execution. If this shorter execution compensates for post-factum analysis time, debugging efficiency can have values greater than 1.

Debugging utility (*DU*) is the product of debugging fidelity and debugging efficiency: $DU = DF \times DE$.

We will use debug determinism to evaluate the debugging fidelity of execution synthesis (Section 4.7) and reverse execution synthesis (Section 5.7).

3.3 Challenges

Debug determinism assumes that the developer is interested solely in the original failure and root cause. It is possible, however, that a developer may want to find *all* potential root causes for a given failure. Thus, a system that records just the failure and finds all executions that share the same root cause and failure would be ideal. The challenge is scaling this approach to real programs.

Finally, while debug determinism may be the sweet spot in the problem domain of debugging, it is unclear what the sweet spot is for other replay-amenable problem domains. In particular, what are the ideal determinism models for replay-based forensic analysis and fault tolerance? Can the same principles behind debug determinism be applied to these problems?

3.4 Example

Inspired by RCSE, we built ADDA [132], a system for record replaying data center applications. ADDA has lower recording and storage overhead than existing systems, owing to two techniques: First, ADDA provides *control plane determinism*, leveraging our observation that many typical datacenter applications consist of a separate control plane and data plane, and most bugs reside in the former. Second, ADDA does not record “data plane” inputs, instead it synthesizes them during replay, starting from the application’s external inputs, which are typically persisted in append-only storage for reasons unrelated to debugging. We showed in [132] that ADDA deterministically replays real-world failures in Hypertable [4] and Memcached [47]. ADDA is a concrete example that shows it is possible to lower the overhead of recording data-intensive data center applications using RCSE based on control plane / data plane selectivity.

Although ADDA synthesizes missing data plane inputs, it is not based on execution synthesis (Section 4.1), therefore a detailed description of ADDA is beyond the scope of this thesis.

This chapter described debug determinism, a determinism model premised on the idea that effective debugging entails reproducing the *same failure* and the *same root cause* as the original execution and proposed a metric to evaluate the debugging utility of a an automated debugging system. The work described in this chapter appeared in [131] and [132]. The next chapter describes execution synthesis, an automated debugging technique.

Chapter 4

Execution Synthesis

Having seen in Chapter 2 the efforts other researchers have devoted to solving the debugging challenge, and in Chapter 3 how to judge the debugging utility of a solution in this space, we now describe execution synthesis, our technique for automated debugging. In further chapters we will describe interesting variants of this approach.

4.1 Definition

Definition 1 *Given a program P and an execution E that produces coredump Core as a result of a failure F, execution synthesis is a computation that yields an execution E' of the unmodified program P that, when executed, deterministically reproduces the same failure F as the one that appears in coredump Core. The inputs to the execution synthesis computation are solely the program P and its state at the time of the failure F.*

4.2 Overview

Execution synthesis is the first automated debugging technique we developed. Reverse execution synthesis (described in Section 5) is a followup technique on execution synthesis. Reverse execution synthesis trades the deterministic execution of E' for the ability to reproduce arbitrarily long executions faster than execution synthesis.

Execution synthesis is a "purist" approach to automated debugging: it does not require any recording of execution E. The only requirement is the coredump *Core*, which is generated when *P* encounters a failure. The *no-recording* requirement sets execution synthesis apart from record-replay techniques. Nevertheless, execution synthesis can be combined with record-replay techniques, and Chapter 7 describes how to trade synthesis time for the runtime overhead introduced by execution recording.

The input to execution synthesis (ESD) consists of the coredump associated with a bug report and the program the developer is trying to debug. ESD then outputs a trace that can be played back in a debugger using the ESD runtime environment. Given a class of bugs, ESD can extract from the coredump all information it needs to find a way to reproduce that class of bugs (e.g., for debugging deadlocks, it extracts the call stacks of the deadlocked threads).

At the end user site, the buggy program is run normally, i.e., without instrumentation or special environments, no annotations, and no debug symbols.

Execution synthesis shifts the burden of bug reproduction from the user side to the developer side, thus avoiding the performance and storage overhead of runtime tracing. This overhead can be substantial: a long-running server that handles many requests and fails after several weeks of execution can incur high cumulative recording overhead.

This design choice means that ESD must reproduce the behavior of a bug (i.e., an execution that fails due to that bug) without knowledge of some crucial runtime information, such as the inputs to the program or the schedule of its threads. Our premise is that, to remove a bug, one need not see the exact same execution that caused the bug to manifest at the end user, but merely some execution that triggers the bug. For this slightly more modest goal, runtime information is not strictly necessary—it can all be inferred with a combination of program analysis and symbolic execution.

Besides automating the laborious parts of debugging, execution synthesis may even generate a path to the bug that is shorter than (but still equivalent from the point of view of debugging fidelity) the one that occurred at the user's site, thus further saving debugging time.

We use the example in Listing 4.1 to illustrate how execution synthesis works. In this example, two threads executing *CriticalSection()* concurrently may deadlock if the condition on line 10 is true. An execution in which the threads deadlock is the following: one thread runs up to line 11 and is preempted right after the unlock call, then a second thread executes up to line 9 and blocks waiting for mutex M_2, then the first thread resumes execution and blocks waiting for M_1 on line 12. The program is now deadlocked.

The bug report for this deadlock would likely contain the final stack trace of each thread, but would be missing several important pieces of information needed for debugging, such as the return values of external calls—*getchar()* and *getenv()*—and the interleaving of threads. ESD "fills in the blanks" and infers two key aspects of the buggy execution: a program path in each thread from the beginning to where the bug occurs, and a schedule that makes this path feasible.

To synthesize the path through the program for each thread, ESD first statically analyzes the program and then performs a dynamic symbolic analysis. In the static analysis phase, ESD computes the control flow graph (CFG) and performs intra- and inter-procedural data flow analysis to

```
          ...
         idx=0;
 1:      if (getchar() == 'm')
 2:        idx++;
 3:      if (getenv(``mode'')[0] == 'Y')
 4:        mode=MOD_Y;
 5:      else
 6:        mode=MOD_Z;
         ...
 7:      CriticalSection() {
 8:        lock(M1);
 9:        lock(M2);
           ...
10:        if (mode==MOD_Y && idx==1) {
11:          unlock(M1);
             ...
12:          lock(M1);
           }
           ...
```

Listing 4.1: Example of a deadlock bug. Two threads executing this code may deadlock if the condition on line 10 is true and one thread is preempted right after executing statement 11.

identify the set of paths through the graph that reach the bug location. For the example in Listing 4.1, ESD's static analysis identifies two paths that could lead the first thread to statement 12: 1→2→3→4→7→...→12 and 1→3→4→7→...→12, both of which require *getenv("mode")* to return a string starting with *'Y'*. Since ESD cannot decide statically whether statement 2 is part of the path to statement 12 or not, both alternatives are considered possible. For the second thread, a similar analysis finds four possible paths to statement 9.

In the dynamic analysis phase, ESD symbolically executes [23] the program in search of a guaranteed-feasible path from the start of the program to the failure point. The search space is restricted to the paths identified during the static analysis phase. In our example, ESD determines that only path 1→2→3→4→7→...→12 can take the first thread to statement 12, since it is the only one that sets *idx* to value 1. This dynamic phase also identifies the need for *getchar()* to return *'m'*. For the second thread, all four paths appear feasible for the time being.

Symbolic execution suffers from the notorious "path explosion" problem [18]. Execution synthesis therefore incorporates a number of techniques to cope with the large number of paths that typically get explored during symbolic execution. The foremost of these techniques is the use of a *proximity heuristic* to guide symbolic execution on those paths most likely to reach the bug. ESD uses the CFG to estimate the distance (in basic blocks) from any given node in the CFG to the bug location. Using this estimate, the exploration of paths is steered toward choices that have a shorter distance to the bug, thus enabling ESD to find a suitable path considerably faster than mere symbolic execution.

For multi-threaded programs, synthesizing the execution path for each thread is not enough—ESD must also identify a thread interleaving that makes these paths possible. ESD does this thread schedule search within the dynamic analysis phase. To make it fast, ESD uses the stack traces from the bug report to attempt thread preemptions in strategic places—such as before calls to mutex lock operations—that have high probability of leading to the desired schedule. In our example, ESD identifies the required preemption points after statement 11 (first thread) and statement 9 (second thread). It also propagates the constraints on *getchar()* and *getenv()* in the first thread to the path choice for the second thread.

In the rest of this chapter, we describe sequential path synthesis (Section 4.3), thread schedule synthesis (Section 4.4), and execution playback (Section 4.5). We then discuss the complexity of execution synthesis (Section 4.6) and discuss execution synthesis (Section 4.7).

4.3 Synthesis of Sequential Executions

In this section we describe how ESD finds a sequential bug-bound execution path within each thread of a program: first it identifies a search goal (Section 4.3.1), then performs static analysis (Section 4.3.2), and finally a dynamic search (Section 4.3.3).

4.3.1 Identifying the End Target

For each thread present in the bug report, we define the *goal* as a tuple $<B,C>$ containing the basic block B in which the bug-induced failure was detected, and the condition C on program state that held true when the bug manifested.

ESD can automatically extract B and C from a coredump for most types of crashes, hangs, and wrong-output failures. The extraction process depends on the type of the bug. For example, in the case of a segmentation fault, B is determined by the instruction that triggered the access violation, and C indicates the value of the corresponding pointer (e.g, NULL), extracted from the coredump. For a deadlock, B contains the lock statement the thread was blocked on at the time the program hung, and C captures the fact that there was a circular wait between the deadlocked threads. As a final example, for a race condition, B is where the inconsistency was detected—not where the race itself occurred—such as a failed assert, and C is the observed inconsistency (e.g., a negation of the assert condition).

If the crash occurs inside an external library, B contains the call to the external library function and C indicates that the values of the arguments are the ones with which that library function was called when the crash occurred. The values of the arguments are extracted from the coredump and the call stack in the bug report.

4.3.2 Identifying Intermediate Steps with Static Analysis

Once the goal $<B,C>$ has been established, ESD does a static analysis pass to narrow down the search space of paths to the goal. This phase operates on the program's control flow graph (CFG) and data flow graph (DFG). First, ESD identifies the critical edges in the CFG, i.e., those that must be present on the path to the goal. Then, ESD identifies intermediate goals, i.e., basic blocks that, according to the DFG, must execute in order for the critical edges to be traversable. The intermediate goals are then passed to the dynamic analysis phase, described in the next section.

ESD first computes the full inter-procedural CFG of the program. It performs alias analysis [11] and resolves as many function pointers as possible, replacing them with the corresponding direct calls; this can substantially simplify the CFG. ESD can handle the case when not all function pointers are resolved, though it may lose precision. In this latter case, subsequent analyses will still be sound and complete, but may take longer to execute. ESD also eliminates all basic blocks that cannot be reached from the start of the program (i.e., dead code) and all basic blocks from which there is no path to B.

We define a *critical edge* as an edge that must be executed by any execution that reaches the goal. Conditional branch instructions generate two outgoing edges in the CFG, corresponding to the true and else branches, respectively. If, for a given branch instruction b, only one of the outgoing edges can be part of a path to the goal, then it is a critical edge. When branch instruction b is encountered during dynamic analysis, ESD will ensure the critical edge is followed; otherwise, the search would miss the goal.

ESD identifies the critical edges by starting from the goal block and working backward, in a manner similar to backward slicing [119]. Starting from B, the algorithm finds at each step a predecessor node in the CFG. For each such node, if only one of its outgoing edges can lead to B, then that edge is marked as critical. The current version of the ESD prototype can only explore one predecessor for each node, so as soon as a block with multiple predecessors is found, the marking of critical edges stops and ESD moves to the next step. A more effective, but potentially slower, algorithm would explore all predecessors and identify multiple sets of critical edges.

An *intermediate goal* is a basic block in the CFG that is guaranteed to be present on the path to the goal block B, i.e., it is a "must have." The knowledge that certain instructions must be executed helps the dynamic analysis break down the search for a path to the final goal into smaller searches for sub-paths from one intermediate goal to the next.

To determine intermediate goals, ESD relies on the critical edges. For each critical edge, the corresponding branch condition and its desired value (true or false) are retrieved. For each variable $x, y, \ldots$ in the branch condition, ESD finds the sets of instructions $\mathbb{D}_x, \mathbb{D}_y, \ldots$ that are reaching definitions [7] of the respective variable. It then looks for combinations of instructions from $\mathbb{D}_x, \mathbb{D}_y, \ldots$ that would give the branch condition the desired value, i.e., instructions for which there is a static guarantee that, if they were executed, the critical edge would be followed. When such a combination is found, the basic blocks that contain the reaching definitions in the combination are marked as intermediate goals. Should more than one combination exist, the corresponding sets of instructions are marked as disjunctive sets of intermediate goals.

While condition C in goal $<B,C>$ is not explicitly used in the above algorithms, ESD does use C in its analyses. To a first degree of approximation, basic block B is replaced in the program with a statement of the form *if (C) then BugStrikes*, and the static analysis phase runs on the transformed program, with *BugStrikes* as the goal basic block. By finding a path along which the program executes *BugStrikes*, ESD will have found a path that executes block B while condition C holds, i.e., a path that reaches the original goal $<B,C>$. Some conditions, however, cannot be readily expressed in this way. For example, a deadlock condition is a property that spans the sequential execution paths of multiple threads. For such cases, ESD has special-case handling to check condition C during the dynamic phase; this will be further described in Section 4.4.

4.3.3 Stitching Intermediate Steps Together with Dynamic Analysis

The previous section showed how ESD statically derives intermediate goals, producing an over-approximation of the path from program start to goal $<B,C>$. We now describe how ESD employs symbolic execution [23] to narrow down this over-approximation into one feasible path to the goal.

To perform the dynamic analysis, ESD runs program P with symbolic inputs that are initially unconstrained, i.e., which can take on any value, unlike regular “concrete” inputs. Correspondingly, program variables are assigned symbolic values. When the program encounters a branch that involves symbolic values—either program variables or inputs from the environment—program state is forked to produce two parallel executions, one following each outcome of the branch (we say that the symbolic branch results in two “execution states”). Program variables are constrained in the two execution states so as to make the branch condition evaluate to true or false, respectively. If, due to existing constraints, one of the branches is not feasible, then no forking occurs.

For example, the first *if* statement in Listing 4.1 depends on the return value of *getchar()*. ESD therefore forks off a separate execution in which *getchar()*=*‘m’*. The current execution continues with *getchar()*≠*‘m’*. Executions recursively split into sub-executions at each subsequent branch, creating an execution tree like the one in Figure 4.1. Constraints on program state accumulate in each independent execution. Once an execution finishes, the conjunction of all constraints along the path to that terminal leaf node can be solved to produce a set of program inputs that exercises that particular path. For example, the rightmost leaf execution (after the third fork) has constraints *mode*=*MOD_Y* and *idx*=1 and the first character of *getenv()*’s return must be *‘Y’* and the return of *getchar()* must be *‘m’*. Everything else is unconstrained in this particular execution.

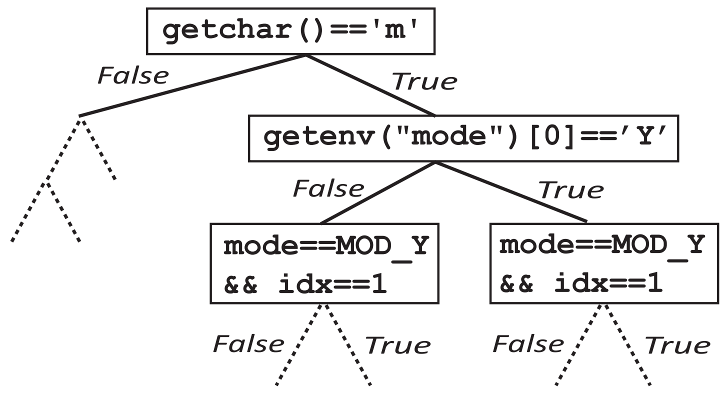


Figure 4.1: Execution tree for the example in Listing 4.1.

An execution state consists of a program counter, a stack, and an address space. Such states can be "executed," i.e., the instruction pointed to by the program counter is executed and may cause corresponding updates to the state's stack and address space. We chose this representation for compatibility with the KLEE symbolic execution engine [23], since the ESD prototype relies on (a modified version of) KLEE.

As new executions are forked, the corresponding execution states are added to a priority queue. At every step of the symbolic execution, a state is chosen from the priority queue and one instruction is executed in that state, after which a new choice is made, and so on. In this way, the entire space of execution paths can be explored, and the symbolic execution engine switches from one execution to the other, depending on the ordering of the states in the queue. When the goal $<B,C>$ is encountered in one of these executions, ESD knows it has found a feasible path from start to goal.

There are two key challenges, though: the execution tree grows very fast (the notorious path explosion problem [18]), and determining the satisfiability of constraints at every branch condition, in order to determine which of the branches are feasible, is CPU-intensive. These two properties make symbolic execution infeasible for large programs. For ESD to be practical, the search for a path to the goal must be very focused: the less of the tree is expanded and searched, the less CPU and memory are consumed.

ESD uses three key techniques to focus the search: First, it uses statically derived intermediate goals (Section 4.3.2) as anchor points in the search space, to divide a big search into several small searches. Second, ESD leverages the information about critical edges (Section 4.3.2) to promptly abandon during symbolic execution paths that are statically known to not lead to the goal. Third, ESD orders the priority queue of execution states based on each state's estimated proximity to the next intermediate goal. In this way, the search is consistently steered toward choosing and exploring executions that appear to be more likely to reach the intermediate goal soon.

We refer to this latter technique as proximity-guided search and describe it in the next section.

4.3.4 Faster Stitching with Proximity-Guided Path Search

ESD uses guided forward symbolic execution to search for a path that reaches the goal extracted from the bug report. In doing so, ESD uses a proximity heuristic to estimate how long it would take each execution state to reach the goal, and it then executes the one that is closest.

The proximity of an execution state to a goal equals the least number of instructions ESD estimates would need to be executed in order to reach that goal from the current program counter in the execution state (line 1 in Algorithm 1). This bound aims to be as tight as possible and can be computed with low overhead.

Algorithm 1: Heuristic Proximity to Goal

```
   Input: Execution state $S$, goal $G$ (potentially intermediate)
   Output: Estimate of $S$'s distance to $G$
1  $d_{min} \leftarrow distance(S.pc, G)$
2  if $d_{min} = \infty$ then
3      foreach procedure $\pi \in S.callStack$ do
4          $I_{ra} \leftarrow$ instruction to be executed after $\pi$ call returns
5          $d \leftarrow dist2ret(S.pc) + distance(I_{ra}, G) + 1$
6          $d_{min} \leftarrow \min(d_{min}, d)$
7  return $d_{min}$

8  function distance ( instruction $I$, instruction $G$ )
9    $d_{min} \leftarrow \infty$
10   if $I$ and $G$ are in the same procedure $\pi$ then
11       foreach acyclic path $\rho$ in $\pi$'s CFG from $I$ to $G$ do
12           $d \leftarrow$ number of instructions on path $\rho$
13           foreach call to procedure $\gamma$ along path $\rho$ do
14               $d \leftarrow d + dist2ret(\gamma.startInstruction)$
15           $d_{min} \leftarrow \min(d_{min}, d)$
16   return $d_{min}$

17 function dist2ret ( instruction $I$ )
18   $d_{min} \leftarrow \infty$
19   $\pi \leftarrow$ procedure to which $I$ belongs
20   foreach return instruction $R$ in $\pi$ do
21       $d_{min} \leftarrow \min(d_{min},\ distance(I, R)\ )$
22   return $d_{min}$
```

When the goal is inside the currently executing procedure, function *distance* computes the proximity. If there are no calls to other procedures, the distance is the length of the path to the goal with the fewest number of instructions (lines 9-12). If, however, any of the instructions along the path are calls to other procedures, then ESD factors in the costs of executing those procedures by adding to the path length the cost of the calls (lines 13-14).

The cost of calling a procedure corresponds to the number of instructions along the shortest path from the procedure's start instruction to the nearest return point. This is a special case of computing the distance of an arbitrary instruction to the nearest return (function *dist2ret*, lines 17–22).

When the goal is not in the currently executing procedure, it may be reached via a procedure that is in a frame higher up in the call stack. In other words, the currently executing procedure may return, and the caller of the procedure may be able to reach the goal, or the caller's caller may do so, etc. Thus, ESD computes a distance estimate for each function on the call stack of the current execution state (lines 3-4). It takes into account the instructions that have to be executed to return from the call plus the distance to the goal for the instruction that will be executed right after the call returns (line 5). The final distance to the goal is the minimum among the distances for each function on the call stack (line 6).

Each execution state S in ESD has n distances associated with it, corresponding to S's distance to the $G_1,...,G_{n-1}$ intermediate goals inferred through static analysis and to the final goal $G_n = B$. The closer an intermediate goal truly is, the more accurate the distance estimate. ESD maintains n "virtual" priority queues $Q_1,...,Q_n$, which provide an ordering of the state's distance to the respective goal: the state at the front of Q_i has the shortest estimated distance to goal G_i. We refer to these queues as "virtual" because the queue elements are just pointers to the execution states. Each state can be found on each of the virtual queues.

At each step of the dynamic analysis, ESD picks a state S from the front of one of the queues. The choice of which queue to consult is uniformly random across the queues. The front state is dequeued, and the instruction at $S.pc$ is symbolically executed, which updates the program counter, stack, and address space, and recomputes the distances from the new $S.pc$. The rationale of choosing states this way is to progressively advance states toward the nearest intermediate goal. Since the static analysis does not provide an ordering of the intermediate goals, ESD cannot choose which goal to try to reach first. It is possible, in principle, for the static phase to provide a partial order on the intermediate goals based on the inter-procedural CFG.

Once a state has reached the final goal (i.e., $S.pc = B$) the search completes: ESD has found a feasible path that explains the buggy behavior. ESD solves the constraints that accumulated along the path and computes all the inputs required for the program to execute that path, in a way similar to automated test generation [55, 23]. The ESD prototype relies on symbolic models of the filesystem [23] and the network stack to ensure all symbolic I/O stays consistent, although it could also work without models if implemented based on S^2E instead of KLEE.

Several programming constructs (such as recursion, system calls, and indirect calls) can pose challenges to the computation of a distance heuristic. We choose to increase the cost of a path that encounters recursion and system calls by a fixed amount—e.g., if a path leads to a recursive or multi-level recursive call, we assign a constant weight (e.g., 1000) instructions to that call. Indirect calls are resolved with alias analysis; if that is not possible, then ESD averages the cost of the call instruction across all possible targets. The distance estimate is just a heuristic, so a wrong choice would merely make the path search take longer, but not affect correctness.

Another concern in heuristic-driven searches are local minima. Fortunately, they are a danger mainly for search processes that cannot backtrack; in path search, ESD can backtrack to execution states that are higher up in the execution tree, thus avoiding getting stuck in local minima.

We found that the three techniques of focusing the search—proximity-based guidance, the use of intermediate goals, and path abandonment based on critical edges—can speed up the search by several orders of magnitude compared to other search strategies (Chapter 6).

Nevertheless, further techniques could be employed to improve the search strategy. For instance, if the initialization phase of the program can be reproduced by other means, such as from an existing test case (ESD does not require existing test cases), ESD could run concretely the initialization phase and automatically switch from concrete to symbolic execution later in the execution of the program [29, 56], thus reducing execution synthesis time. We leave this for future work.

4.4 Synthesis of Thread Schedules

In the case of multi-threaded programs, ESD must also synthesize a schedule for interleaving the execution paths of the individual threads. It seeks a single-processor, sequential execution that consists of contiguous segments from the individual threads' paths. In other words, ESD synthesizes a serialized execution of the multi-threaded program.

To do so, ESD employs symbolic execution, but instead of only treating inputs and variables as symbolic, it also treats the underlying scheduler's decisions as symbolic. It associates with each preemption point (i.e., each point where the scheduler could preempt a thread) a hypothetical branch instruction that is conditional on a single-bit predicate: if true, the currently running thread is preempted, otherwise not. These single-bit predicates[1] can be viewed as bits in the representation of a variable that represents the serial schedule. ESD treats this variable as symbolic, and the question becomes: What value of this schedule variable would cause the corresponding execution to exhibit the reported bug?

Preemption points of interest are before and after concurrency-sensitive operations: load instructions, store instructions, and calls to synchronization primitives. While conceptually the sequential path synthesis phase is separate from schedule synthesis, ESD overlaps them and syn-

[1]For programs with more than two threads, predicates have multiple bits, to indicate which thread is scheduled in place of the currently running one.

thesizes one “global” sequential path, by exploring the possible thread preemptions as part of the sequential path synthesis.

Just as for sequential path synthesis, ESD employs heuristics to make the search for a thread schedule efficient. It is substantially easier to choose the right heuristic if ESD knows the kind of concurrency bug it is trying to debug, and this can often be inferred from the coredump. Execution synthesis can synthesize schedules for deadlocks (Section 4.4.1) and data races (Section 4.4.2).

4.4.1 Synthesizing Thread Schedules for Deadlock Bugs

When looking for a path to a deadlock, the preemption points of interest are solely the calls to synchronization primitives, like mutex *lock* and *unlock*. In most programs, there are orders-of-magnitude fewer such calls than branches that depend (directly or indirectly) on symbolic inputs, so the magnitude of the deadlock schedule search problem can be substantially smaller than that of sequential path search.

Moreover, information about the deadlocked threads’ final call stacks provides strong clues as to how threads must interleave in order to deadlock. ESD leverages these clues to bias the search toward interleavings that are more likely to lead to the reported deadlock.

For the deadlock example in Listing 4.1, a coredump would indicate call stacks that (in stylized form) would look like T_1 : [... 12] and T_2 : [... 9], meaning that thread T_1 was blocked in a *lock* call made from line 12, while T_2 was blocked in a *lock* call made from line 9. The call stack shows the call sequence that led to the lock request that blocked the thread. This lock request appears in the last frame, and we refer to it as the thread’s *inner lock*. We call *outer locks* those that are already held by the deadlocked thread. This naming results from the fact that a deadlock typically arises from nested locks [85], where an inner lock is requested while holding an outer lock. At the time of deadlock, the acquisitions of the outer locks are not visible in the call stack anymore.

For the example bug, the search goal for each thread is $T_1 :<12, T2@9>$ and $T_2 :<9, T_1@12>$, meaning that T_1 blocks at line 12 while T_2 blocks at line 9. ESD now seeks an interleaved execution that leads to this goal, without any knowledge of where the outer locks were acquired.

Any time ESD encounters a *lock* or *unlock* operation, it forks off an execution state in which the current thread is preempted. The running execution state maintains a pointer to that forked state, in case ESD needs to return to it to explore alternate schedules. More generally, we augment each execution state S with a map $\mathbb{K}_S$: *mutex* $\rightarrow$ *execution state*. An element $<M, S'>\in \mathbb{K}_S$ indicates that S is exploring one schedule outcome connected with the acquisition of mutex M, while S' is the starting point for exploring alternative scheduling outcomes. A snapshot entry $<M, S'>$ is deleted as soon as M is unlocked. The size of $\mathbb{K}_S$ is therefore bounded by the program’s maximum depth of lock nesting. ESD leverages Klee’s copy-on-write mechanisms at the level of memory objects to maximize memory sharing between execution states. As a result, snapshots are cheap.

We augment execution state S with *S.scheduleDistance*, an estimate of how much context switching is required to reach the deadlock. For the case of two-thread deadlocks, this schedule distance can take one of two values: *far* or *near*. ESD computes a weighted average of the path distance estimate (Section 4.3.4) and the schedule distance estimate, with a heavy bias toward schedule distance. The virtual state priority queues are kept sorted by this weighted average. The bias ensures that low-schedule-distance execution states are selected preferentially over low-path-distance states.

The general strategy for schedule synthesis is to help each thread "find" its outer lock as quickly as possible.

If a thread T_1 requests a mutex M that is free, ESD forks state S' from S and allows the mutex acquisition to proceed in S, while in S' thread T_1 is preempted before acquiring M. In S, ESD must decide whether to let T_1 continue running after having acquired M, or to preempt it. If, by acquiring M, T_1 did not acquire its inner lock (i.e., the *S.pc* of the lock statement is different from that in the goal), then ESD lets T_1 run unimpeded. However, if T_1 just acquired its inner lock, then ESD preempts it and marks $S.scheduleDistance = near$. This keeps M locked and creates the conditions for some other thread T_2 to request M; when this happens, it is a signal that M could be T_2's outer lock. The updated schedule distance ensures state S is favored for execution over other states that have no indication of being close to the deadlock.

If thread T_1 requests mutex M, and M is currently held by another thread T_2, ESD must decide whether to "roll back" T_2 to make M available to T_1, or to let T_1 wait. If M is T_2's inner lock, then it means that M could be T_1's outer lock, so ESD tries to make M available, to give T_1 a chance to acquire it: ESD switches to state S_k (from the $<M, S_k>$ snapshot taken just prior to T_2 acquiring M), which moves execution back to the state in which T_2 got preempted prior to acquiring M.

ESD does this by setting, for each state in $\mathbb{K}_S$, the schedule distance to *near*. It then sets the current state's schedule distance to *far*, to deprioritize it. This creates the conditions for T_1 to acquire M, its potential outer lock. When T_2 later resumes in a state in which T_2 does not hold M, mutex M is likely to be held by T_1 and about to be requested by T_2, thus increasing the chances of arriving at the desired deadlock.

Whenever a mutex M is unlocked, the snapshot corresponding to M is deleted, i.e., $\mathbb{K}_S \leftarrow \mathbb{K}_S - <M, *>$. ESD deletes these snapshots because a mutex that is free (unlocked) cannot be among the mutexes that cause a deadlock.

We illustrate on the example from Listing 4.1, for which the search goals are $T_1 : <12, T2@9>$ and $T_2 : <9, T_1@12>$.

Thread T_1 needs to get to line 12. ESD takes T_1 rather uneventfully up to line 10, with snapshots having been saved prior to the *lock* operations at line 8 and 9. Once ESD encounters condition *mode* $= MOD\_Y \wedge idx = 1$ on line 10, it must follow the true-branch, because it is a critical edge. This brings T_1 eventually to line 12. By this time, due to the *unlock* on line 11, there is only one snapshot left in $\mathbb{K}_S = \{<M_2, S_9>\}$, from the *lock* on line 9. At line 12, a copy of the current state is forked and $\mathbb{K}_S = \{<M_2, S_9>, <M_1, S_{12}>\}$. T_1 acquires M_1 and then T_1 is preempted.

T_2 runs until it reaches line 8, where it blocks for M_1 (held by T_1). Since M_1 was acquired as T_1's inner lock, ESD switches to state S_{12}, in which T_1 is preempted immediately prior to acquiring M_1. This allows T_2 to run and acquire M_1, but it blocks again on line 9 when trying to get M_2 (held by T_1). T_1 is scheduled back, and the program is now in the situation that T_1 is holding M_2 while waiting for M_1 at line 12, and T_2 is holding M_1 while waiting for M_2 at line 9—the deadlock goal. ESD saves the required inputs for *getchar()* and *getenv()* along with the synthesized schedule (i.e., the one in which T_1 acquires M_1 and M_2, then releases M_1, then T_2 gets to run until it acquires M_1 and blocks on M_2, after which T_1 gets to run again and blocks on M_1).

This algorithm generalizes in a relatively obvious way to more than two threads. Our ESD prototype can synthesize thread schedules for deadlocks involving an arbitrary number of threads, even when it is just a subset of a program's threads that are involved in the deadlock.

During schedule synthesis, ESD automatically detects mutex deadlocks by using a deadlock detector based on a resource allocation graph [71]. Deadlocks involving condition variables are more challenging to detect automatically—inferring whether a thread that is waiting on a condition variable will eventually be signaled by another thread is undecidable in general. However, ESD can check for the case when no thread can make any progress and, if all threads are waiting either to be signaled, to acquire a mutex, or to be joined by another thread, then ESD identifies the situation as a deadlock.

When searching for a schedule that reproduces a reported deadlock, ESD may encounter deadlocks that do not match the reported bug. This means ESD has discovered a different bug. It records the information on how to reproduce this deadlock, notifies the developer, rolls back to a previous snapshot, and resumes the search for the reported deadlock.

4.4.2 Synthesizing Thread Schedules for Data Race Bugs

To find paths to failures induced by data races, ESD takes an approach similar to the one for deadlocks: place preemptions at all the relevant places, then explore first those schedules most likely to reveal the data race. Snapshotting is used in much the same way, piggybacking on the copy-on-write mechanism for managing execution states. In addition to synchronization primitives, ESD also introduces preemptions before instructions flagged as potential data races.

ESD uses a dynamic data race detection algorithm similar to Eraser [105]: it keeps track of the lockset for each memory address and detects a data race when the intersection of all locksets used to synchronize the access to a particular memory address is void. ESD inserts preemption points wherever potentially harmful data races [95] are detected. Normally, dynamic data race detectors can miss races, because they only observe execution paths exercised by the given workload. However, by using symbolic execution, ESD can expose to the detection algorithm an arbitrary number of different execution paths, independently of workload.

In order to avoid unnecessary thread schedules early in the execution of the program, ESD uses an additional heuristic. It identifies the longest common prefix of the final thread call stacks in the coredump and inserts preemptions only in executions whose call stacks contain this prefix. If the last frame of the common prefix corresponds to procedure p, then p is set as an intermediate goal for each thread—for the example in Listing 4.1, p would be the entry into *CriticalSection*. Once all threads reach their respective goals (or when no threads can make any further progress), ESD's scheduler starts forking execution states on fine-grain scheduling decisions and checks for data races. We found this heuristic to work well in practice, especially considering that many applications run the same code in most of their threads.

For simplicity and clarity, we assume a sequential consistency model for memory shared among threads, an assumption present in most recent systems dealing with concurrency bugs (such as Chess [94]). An immediate consequence is that each machine instruction is assumed to execute atomically with respect to memory, which simplifies the exploration process. In the case of shared memory with relaxed consistency, ESD could miss possible paths, but will never synthesize an infeasible execution leading to a bug goal.

Data race detection can be turned on even when debugging non-race bugs. In this way, ESD can synthesize paths even to bugs (e.g., deadlocks, buffer overflows) that manifest only in the presence of data races. Moreover, as with deadlocks, unknown data races may be fortuitously discovered.

In summary, ESD's synthesis of bug-bound paths and schedules exploits features of the corresponding bug report to drastically reduce the search space. While ultimately equivalent to an exhaustive exploration, ESD uses heuristics to aggressively steer exploration toward those portions of the search space that have the highest likelihood of revealing the desired bug-bound execution.

4.5 Execution Playback

Once the execution synthesizer (Section 4.3–Section 4.4) reaches its goal, it generates an execution file containing the playback information. This file is read by the ESD playback environment—the subject of this section. The goal of playback is to provide developers an explanation of the bug symptoms, in a way that allows them to inspect the execution with a classic debugger.

4.5.1 The Output of Execution Synthesis

In order to achieve the highest possible fidelity, ESD plays back a reported bug using the native binary that was run by the end user. The synthesized execution file contains concrete values for all input parameters, all interactions with the external environment (e.g., through system calls), and the complete thread schedule. For all program input, including that coming from the environment, ESD solves the constraints found during execution synthesis and produces corresponding concrete values (such as *getchar()='m'* and *getenv("mode")[0]='Y'*). This is identical to what automated test generators do (like DART [55] and Klee [23]), except that these test generators do not produce thread schedules.

ESD saves the thread schedule of a synthesized execution in the form of happens-before relations [78] between specific program instructions. ESD can also save a strict schedule in the file, by recording the exact instructions on which the context was switched during synthesis, along with the switched thread identifiers. During the playback phase (Section 4.5.2), this strict schedule will enforce literally a serial execution of the program, whereas the schedule based on happens-before relations allows playback to proceed with the same degree of parallelism as the original execution.

4.5.2 Playing Back the Synthesized Execution

In order to steer a program into following the steps reflected in the synthesized execution file, ESD relies on two components: one for input playback and one for schedule playback. For input playback, ESD takes from the trace the values of command line arguments and passes them to the program. ESD also intercepts via a custom library the calls to the environment and returns the inputs from the execution file. To preserve the consistency of the execution, ESD also relies on Klee's symbolic filesystem and network models.

To play back the synthesized schedule, ESD gains control of the concrete execution by intercepting synchronization calls with a shim library and by selectively instrumenting the binary. The intercepted calls are then coordinated by ESD's cooperative scheduler underneath the program being played back. While, during execution synthesis, the threads of a program were emulated, during playback the program is permitted to create native threads and invoke the native synchronization mechanisms. The threads are context-switched only when this is necessary to satisfy the happens-before relations in the execution file.

ESD can also record and play back an execution serially. One single thread runs at a time, and all instructions execute in the exact same order as during synthesis. Serial execution playback makes it easier for a developer to understand how the bug is exercised, because the bug's causality chain is more obvious. Serial execution is also more precise, if the program happens to have race conditions. However, performance of parallel programs may be negatively affected by serialization, and in some cases this might matter.

Developers run the buggy program in the playback environment and can attach to it with a debugger at any time. They can repeat the execution over and over again, place breakpoints, inspect data structures, etc. After fixing the bug, ESD can be re-run, to check whether there still exists a path to the bug. This is particularly helpful for concurrency bugs, where patches often do not directly fix the underlying bug, but merely decrease its probability of occurrence [85]. If ESD can no longer synthesize an execution that triggers the bug, then the patch can be considered successful.

4.6 Complexity Analysis

This section analyzes execution synthesis from a theoretical perspective, in order to determine its expected performance and the bounds on various parameters that influence synthesis time, such as the length of the execution path and the number of branch statements in the synthesized path that depend on symbolic input. The final goal of this section is to derive bounds on how many seconds of original native execution time can be synthesized in a feasible amount of time.

4.6.1 Notation

We assume that we use execution synthesis to reproduce a failure of depth d, where d is the total number of executed branch statements from the start state of the program to the failure state. We assume that on the path that reproduces the failure there are b branch statements that depend on symbolic input. In our experience, few of the executed branches depend (directly or indirectly) on symbolic input, therefore $b \ll d$.

4.6.2 Analysis

We model the native execution that occurred in production as requiring $k \cdot d$ time to reach the failure state, where k is a constant that depends on the target program, the workload, and hardware on which the program is executing. Of course, the target program could have a different value of k compared to the value of k for the execution synthesis platform. For instance the program could be mostly idle, waiting for I/O to complete, while during execution synthesis, I/O could be equivalent to a no-op. However, for the purpose of this analysis, we consider that k is the same for both the target program and for the execution synthesis tool.

The Complexity of the Static Analysis Search

Execution synthesis first uses static analysis to trim the search space, and we assume that the static analysis phase depends on d and takes $SA(d)$ time.

The Complexity of Constraint Solving

After the static analysis phase, execution synthesis searches for the path that reproduces the bug. Each symbolic branch encountered during the search requires constraint solving (to determine if the path is feasible). In the best case, assuming an oracle searcher (i.e., no search has to be done to determine a feasible path), execution synthesis still must solve one constraint to determine the inputs that drive the program on the synthesized path. This constraint must have at least b terms. Assuming each branch condition adds a constant number of terms to the constraint, then the total time to solve the constraints is a function of b and we denote it by $CS(b)$.

Depending on the program structure, constraints may not involve many of the symbolic variables, or they can be simplified. In this case the constraint solving time may not be a function of b. Moreover, in practice one will time out if a constraint cannot be solved (the disadvantage of using timeouts is a loss of completeness: the inability to explore some execution paths).

The Complexity of Dynamic Search

In the dynamic phase, execution synthesis explores the search space using various heuristics. Dynamic search is also a function of b, denoted by $DS(b)$, and it is computed in the following way: for every branch that depends on symbolic input, we need to select the best state from the queue of states, solve the constraints associated with the chosen state, and to run the program along the chosen branch. This process needs to be repeated for each encountered symbolic branch, which is equivalent to the subset of the execution tree that is explored by the execution synthesis technique. In principle, assuming we only explore paths containing up to b branches, the worst case exploration time is proportional to 2^b (assuming all paths are explored) and the best case exploration time is constant and proportional to d (assuming an oracle searcher that explores a single path).

To estimate $DS(b)$ more precisely, we make the following notations and assumptions. We denote $SD(b)$ to be the time required to make a search decision (e.g., pick a state from the available states), $CS(b)$ is the time required for constraint solving during the synthesis process, $SET(b)$ is the number of states in the subset of the execution tree that is explored. Moreover, execution synthesis is typically done by running the target program in a special environment [133], which introduces overhead for each executed instruction compared to the native execution. We assume this overhead is a constant factor and is denoted by I. Since on average there are $\frac{d}{b}$ branches in between two symbolic branches, we obtain that the time spent running the program in the special environment is on average $\frac{I \cdot d \cdot k}{b}$ for each search decision. Thus, we get that $DS(b) = SD(b) + CS(b) + SET(b) \cdot \frac{I \cdot d \cdot k}{b}$.

Adding All Up

Thus, execution synthesis will require $SA(d) + DS(b) = SA(d) + SD(b) + CS(b) + \frac{I \cdot d \cdot k}{b} \cdot SET(b)$ to finish. Since the original execution takes $d \cdot k$, then execution synthesis will be $ES(b) = \frac{SA(d)+SD(b)+CS(b)}{d \cdot k} + \frac{I}{b} \cdot SET(b)$ times slower than the native execution.

Worst Case

In the worst case, execution synthesis time is exponential in b, since either CS(b) or SET(b) are exponential. Figure 4.2 shows how worst case execution synthesis time varies with the number of symbolic branches in the worst case scenario.

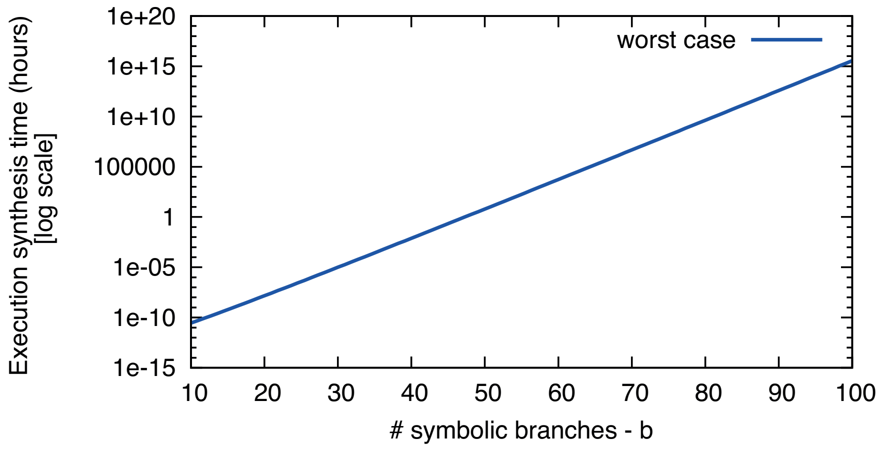


Figure 4.2: Worst case execution synthesis time vs. # symbolic branches (log scale). We make some assumptions about constants involved: i.e., $SA(d)$, $SD(b)$, and $CS(b)$ are negligible compared to the time of the original execution ($d \cdot k$), that $SET(b) = 10^{-9} \cdot 2^b$. Thus, we assumed that only the search space is exponential, and then obtain that $ES(b) > 10^{-9} \cdot 2^b$. This shows that, when the complexity is exponential, soon after b exceeds 60, execution synthesis takes too long to be practical.

Polynomial Case

We now make some assumptions about the complexity of the various parts of the execution synthesis algorithm. These assumptions are optimistic, given that all the analysis problems are exponential in the worst case, however, it does provide an interesting estimation for realistic cases in which these problems are polynomial. The purpose of this analysis is not to show the best case. In the best case, execution synthesis is faster than this example. Instead, the analysis shows that

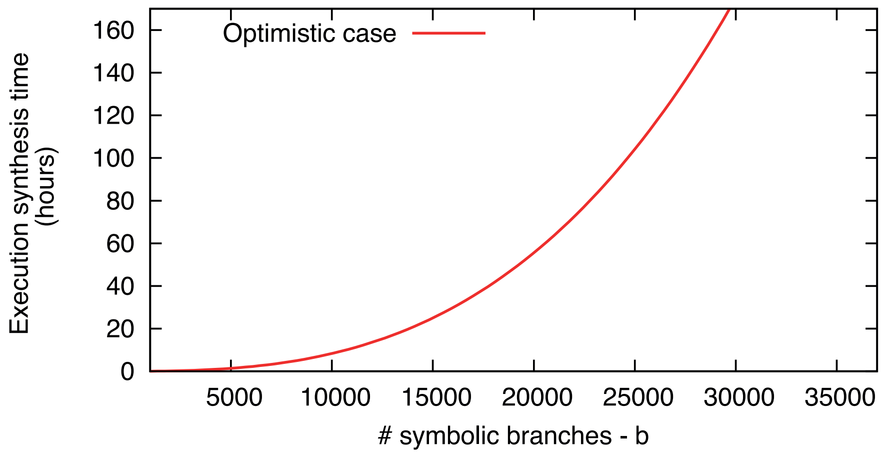


Figure 4.3: Example: execution synthesis time vs. # symbolic branches.

polynomial complexity may also be sufficient to limit the depth of the synthesized execution for which execution synthesis is practical.

Thus, we assume that:

- I is 2. This is currently an under-approximation, since ESD's interpreter typically introduces more than $100\times$ overhead. Running concretely in S2E [29] is faster, but still introduces $10-30\times$ overhead. Nevertheless, it may be feasible to achieve lower values than 2 for I, as shown in [107].

- We assume ESD's heuristics are efficient, therefore little of the execution tree is explored, such that $SET(b) = set \cdot b^4$, where set is a constant, which we assume to be 10^{-8}.

- $CS(b) = cs \cdot b^4$, where cs is a constant representing the time required to solve a constraint involving a query based on a *single branch* (e.g., $x > 0 \wedge x < 10$). We optimistically assume cs to be $\approx 10^{-8}$ seconds, which means we can approximate $CS(b)$ to $10^{-8} \cdot b^4$.

- We assume that the static analysis is very fast and we ignore it for the purpose of this analysis. Note that in practice, depending on the complexity of this analysis, $SA(d)$ could also be exponential in d.

- We assume the state selection process is fast, therefore $SD(b) \approx 10^{-8} \cdot b^2$.

- In order to estimate k, we developed a tool based on PIN [86] to count the number of branches in an execution. The tool found that in 10 seconds of native execution, on a 2 GHz quad-core Xeon E5405 CPU, there are typically about 10^9 branches, so $k = 10^{-8}$ seconds/branch. The analysis was averaged over Firefox, Chrome, and Apache.

- We optimistically assume that one in 10^6 branches is symbolic, therefore $d = 10^4 \cdot b$. This number largely depends on the structure of the program and the inputs it processes, etc, therefore it is hard to do an accurate estimation even for a single program.

Thus, we further obtain that execution synthesis will be $10^{-4} \cdot (b + b^3) + 2 \cdot 10^{-8} \cdot b^3$ slower than the native execution. Figure 4.3 shows how execution synthesis time varies with b in this optimistic case. This graph shows that more than 1 week is necessary to handle $b \geq 30,000$, which corresponds to a 3 second execution, given our assumption that $d = 10^4 \cdot b$ and that $k = 10^{-8}$.

Thus, the key to achieving scalability is to reduce b. We show in Chapter 5 how to reduce b using reverse execution synthesis.

4.7 Discussion

In this section we discuss how to use ESD for debugging, ESD's limitations, and how ESD can complement static analysis tools.

Usage: When developers are assigned a bug report, they would pass the reported coredump to ESD, along with a hint for the type of bug. For the current ESD prototype, this can be *crash*, *deadlock*, or *race condition*. ESD compiles the program source code with the standard LLVM tool chain and uses the resulting bitcode file. Developers can also instruct ESD to enable various types of detection (e.g., data races) during path synthesis, using the following command line:

```
$ esd <coredump file> <program>
      < --crash | --deadlock | --race >
      [--with-race-det] [--with-deadlock-det]
```

ESD then processes the coredump, extracts the necessary information, and computes the $<B,C>$ goals for synthesis. It then performs the path and schedule search, and produces the synthesized execution file. Developers then use the playback environment to reproduce the bug and optionally attach to the program with their favorite debugger:

```
$ esd-play <orig program binary> <synthetic exec file>
```

We discuss usage models in more detail in Section 6.5.

Limitations: Our approach is based on heuristics and static analysis to trim down the search space that would otherwise be too large to explore in a naive approach. Like any heuristic-based technique, ESD could be imprecise; lack of precision can increase the time to find a bug, thus hurting ESD's efficiency. We did not experience this situation for the bugs we reproduced with ESD, but the theoretical possibility exists. If ESD is used as part of a bug triage system, then the potentially long running times can be amortized by running them off the critical path of debugging, unlike when ESD is directly used by a developer.

Execution synthesis may not always be able to reproduce a bug. Symbolic execution has inherent limitations when solving complex constraints, such as finding a string m for which $hash_{SHA\text{-}2}(\mathrm{m}) = \mathtt{0xf8e28ed7b8db9a}$. As a result, ESD would have a hard time finding a program input that would exercise the then-branch of an if statement involving the above condition. If there is a bug that manifests only when this condition holds, ESD will likely not be able to reproduce it—doing so would amount to breaking the SHA-2 cryptographic hash function [96].

Some coredumps cannot be processed by ESD's automated analyzer. For example, if a bug corrupts the stack or the heap, ESD does not yet know how to repair the data structures before extracting them and using them for synthesis. However, in some cases, it may be possible to repair the stack trace by inspecting the control flow of the program. In other cases, obtaining from the coredump the size of a dynamically allocated buffer can be challenging. ESD can obtain the size of a dynamically allocated buffer by parsing the memory allocator metadata, but this requires inferring some of the heap characteristics. E.g., for the glibc memory allocator, metadata is stored relative to the base address of the allocated buffer and can be reliably retrieved only if the base address can be inferred from the coredump.

From a theoretical point of view, execution synthesis does not provide debug determinism (Chapter 3), since it only reproduces the failure and may reproduce a different root cause. Since execution synthesis does not have any false positives, $k_{fp} = 0$, therefore debugging fidelity $DF_{ESD} = \frac{k_{original} + \frac{k_{other}}{n}}{k_{original} + k_{other}}$. Execution synthesis can theoretically reproduce all possible n root causes (assuming unbounded resources), so assuming it reproduces each possible root cause only once, its debugging fidelity $DF_{ESD} = \frac{1 + \frac{n-1}{n}}{1+n}$, where n is the number of possible root causes. For instance, if $n = 2$, then $DF_{ESD} = 0.5$. In our evaluation (Chapter 6), we found that execution synthesis finds the correct root cause and (to the best of our knowledge) there is a single root cause for all the bugs found. Thus, for the bug reports we used in our evaluation, execution synthesis had maximum debugging fidelity, therefore it achieved debug determinism.

This chapter described an automated debugging technique that reproduces an full execution using only the program and the bug report. The work described in this chapter appeared in [133]. In the next chapter we describe reverse execution synthesis, a technique that synthesizes an execution suffix instead of a full execution, in order to enable debugging of arbitrarily long executions.

Chapter 5

Reverse Execution Synthesis

The previous chapter described a technique to reproduce an entire execution starting from the program and a bug report. This chapter describes an automated debugging technique that uses the same input to reproduce an execution suffix instead of the entire execution.

5.1 Definition

Definition 2 *Given a program P and an execution E of P that produces coredump Core as a result of a failure F, then reverse execution synthesis automatically computes a snapshot S of P's state and an execution suffix X of the unmodified program P, such that, executing X starting from S deterministically reproduces the same failure F. Reverse execution synthesis computes S and X without recording any information during the execution E.*

5.2 Motivation

A fundamental challenge for debugging is that the coredump does not contain enough information to reproduce the exact execution that led to the failure in the general case. However, this is not really necessary: for debugging, it is sufficient to produce *some* execution that reproduces the observed failure state and the root cause [131]. Execution synthesis accomplishes this by mimicking a human developer: it does a backward analysis starting from the coredump, identifies in the space of possible execution paths some key "reference points" that must be part of all failure-bound executions, and then uses forward dynamic search through the control flow graph of the program to find a path that passes through the reference points and produces the coredump.

The problem, though, is that this approach does not work for arbitrarily long executions—in fact, the longer the execution, the more ambiguity in the location of these reference points, and the harder it becomes (Section 4.6) to synthesize an execution all the way from the start of the execution to the end failure state.

For such executions, we advocate a new approach that turns execution synthesis on its head; we call it reverse execution synthesis (RES). The observation we leverage is that developers do not really need a full execution from start to finish, but just a suffix of the failure-bound execution—as long as developers can replay this suffix and it contains the root cause of the failure, it is sufficient to debug it [131].

In essence, RES reverse-executes the program and reproduces the last few milliseconds of the execution, enough to capture the root cause; the length of the full execution is irrelevant to this approach. Unlike backward static analysis (e.g., PSE [87]), RES's analysis provides an accurate execution suffix that can be run deterministically in a debugger. Unlike execution synthesis, RES interprets the entire coredump, not just the failure condition C (Section 4.3), which makes RES strictly more powerful.

The rest of this chapter describes the technique in more detail (Section 5.3), describes sequential execution suffix synthesis (Section 5.4) and thread schedule suffix synthesis (Section 5.5), and discusses the complexity of reverse execution synthesis (Section 5.6).

5.3 Overview

We need a tool that, for a given program P, can use a coredump *Core* to generate a suffix of a feasible execution E that causes program P to produce coredump *Core*. The key requirements are that (1) there is no recording at runtime; (2) the technique works for multi-threaded programs and concurrency bugs; (3) the suffix is of a feasible execution; (4) the suffix contains the root cause of the failure; (5) execution E deterministically leads to *Core*; and (6) no modifications are to be made to P. Since it is predicated on the presence of a coredump, this tool would work for failures that generate a coredump (e.g., crashes due to violations of memory safety properties, assertion violations, deadlocks, etc.). Our current design for RES meets requirements (1), (2), (5), (6), and aims to satisfy but cannot always guarantee (3) and relies on developers to achieve (4).

In proposing a technique for building such a tool, we rely on two enablers: First, E does not need to be the execution that actually occurred in production and led to coredump *Core*—any execution that reproduces the same root cause and failure is sufficient. Second, we assume that the root cause is located fairly close to the failure (e.g., 85% of the bugs analyzed in [137] were executed just a few instructions before the failure), so we expect a short execution suffix to suffice for debugging.

What Are the Inputs and Outputs?

Inputs: As suggested above, RES takes in the coredump *Core* that represents a snapshot of the failed program's state; this is typically a free by-product of a failed execution and is already being collected by production systems [54, 116]. In addition to *Core*, RES takes in the program source code P_S, which should be available to developers. Thus, the input is $< Core, P_S >$.

Outputs: RES produces a set of execution traces T_i that end with the program counter found in the coredump; corresponding to each instruction trace, a memory image M_i (Section 5.4.2) is also provided, representing the content of the program's address space just before the execution of the suffix—executing T_i starting with state M_i leads to a state identical to the coredump. The execution suffix T_i consists of the inputs (e.g., system call returns) and the thread schedule required to accomplish this. To replay a suffix in a debugger like gdb, a special environment is slipped underneath the debugger to instantiate M_i and replay T_i; to the developer it looks as if the program deterministically runs into the same failure as the original execution.

RES continues building up suffixes by moving backward through the execution until the user stops it. If allowed to run to completion, RES would eventually either reconstruct a full start-to-finish execution path, or conclude that no such path exists and therefore the coredump is likely due to hardware failure.

5.4 Sequential Path Synthesis

5.4.1 The Challenge of Inferring the Past Based on the Present

RES requires moving backward in time through the unknown execution that led to the failure. One thought might be to reverse the outcome of every instruction, but this is not feasible. For example, reversing a memory write in the general case requires knowledge of what value was in that location prior to the execution of the overwriting instruction. Further aspects that pertain mostly to CISC instruction sets like x86 make the reversing of other instructions hard as well. A method has been proposed for reverse-executing programs running on the RISC PowerPC [8], but even this method needed heavyweight recording to recover missing information.

The main challenge then is how to accurately reconstruct past program state without having recorded it. Prior work based on static analysis can compute backward program slices [87, 126] or derive weakest preconditions [21, 25] for given memory safety bugs. These techniques are typically imprecise, as they do not use the rich source of information present in the coredump. They also work only on sequential programs, because reasoning statically about concurrent executions is very hard.

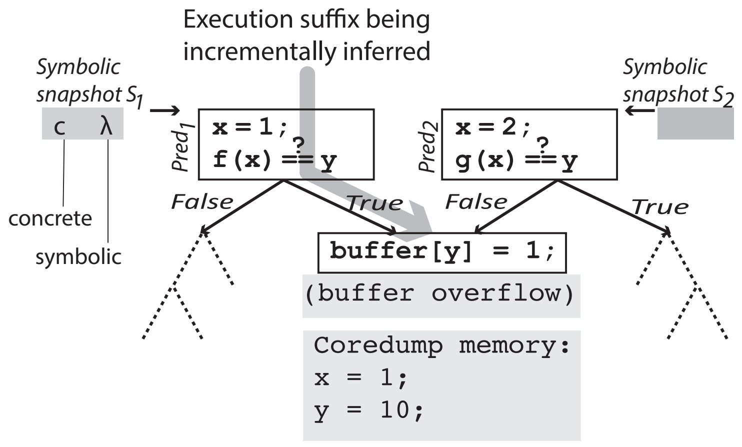


Figure 5.1: Simplified example illustrating the basics of RES on a program that crashed due to a buffer overflow. RES creates symbolic snapshots S_1 and S_2 that correspond to program state just prior to each possible predecessor basic block. Since $x = 1$ in the coredump, and only $Pred_1$ ever sets x to 1, then $Pred_1$ must be part of the correct execution suffix; RES discards the execution suffix that traverses $Pred_2$. A symbolic snapshot contains both concrete and symbolic memory (e.g., x has an unconstrained symbolic value in S_1 because $Pred_1$ overwrites x's value, so x prior to $Pred_1$ could be anything).

5.4.2 Representing Past Program State with Symbolic Snapshots

RES combines precise dynamic symbolic analysis with static information from the coredump and the control-flow graph of the program to reconstruct missing information. Unlike forward execution synthesis (Chapter 4), where the static analysis phase goes from the final state all the way to the start state before engaging in dynamic analysis, RES alternates between static and dynamic analysis for each basic block, incrementally producing a precise execution suffix. Because RES focuses both static and dynamic analysis on an execution suffix—which is substantially shorter than the length of the entire execution—it alleviates the path explosion problem of forward execution synthesis.

RES starts from the coredump and navigates P's control-flow graph backward until it reaches a basic block that has at least two predecessors ($Pred_1$ and $Pred_2$ in Figure 5.1). At this point, RES determines statically which predecessors are possible, and infers P's memory state just prior to executing each predecessor block.

To do this, RES creates *symbolic snapshots* (S_1 and S_2 in Figure 5.1), one for each predecessor basic block. A symbolic snapshot is a "hypothesis" of how program state may have looked prior to executing that predecessor block. It is an image of P's memory state in which some locations do not have concrete values, but rather have stand-ins for *any* possible value (these are called symbolic values, like in Chapter 4. Such symbolic values can also be subject to constraints, such as having to be positive, or being in a certain range. A symbolic snapshot in RES is a mix of known, concrete values and as-of-yet unknown, symbolic values. The program counter of a symbolic snapshot is set to the entry point of the corresponding predecessor basic block.

5.4.3 Reconstructing Past Program State

A symbolic snapshot S_{pre} can be thought of as an overapproximation of all possible program states just prior to executing the predecessor block B. At a high level, the idea is that, if S_{post} is the program state after executing B, then we can obtain S_{pre} from S_{post} by simply replacing every memory location overwritten by B with an unconstrained symbolic value.

If we now execute B with S_{pre} as a starting state, B will transform S_{pre} into S', a more constrained version of the symbolic snapshot S_{pre}. This is because, as B executes, it overwrites values in S_{pre} with values computed either based on other values in S_{pre} (which may be concrete or symbolic) or based on program inputs. For example, a variable z may be unconstrained prior to executing B, but be constrained to $z \in [0,10]$ after some arithmetic performed by B. Program inputs (e.g., incoming network packets, reads from disk) are handed to the program as unconstrained symbolic values, since these inputs refer to system state that is not contained in program memory.

After executing the last instruction in B, RES compares S_{post} and S', to check if the resulting S' is an overapproximation of S_{post}, meaning that the value of every location in S_{post} is a subset of the possible values of that location in S' (we denote this by $S' \supset S_{post}$). If it is, then the just-executed B is part of a feasible execution suffix, because it transformed program state in a way that is compatible with the post-B state. If $S' \not\supset S_{post}$, then it means that B cannot be part of the suffix.

This reverse synthesis process is applied recursively to B's predecessor block(s), incrementally forming an execution suffix, one block at a time. The first step of RES is the base case of the recursion, in which S_{post} is initialized with a copy of the coredump *Core*, and the first instance of block B is the last basic block of the execution suffix.

When deriving S_{pre} from S_{post}, the main challenge are memory read and write operations. RES's approach is described in Algorithm 2. When encountering a *memory write* instruction in B, there is no way of knowing what value was overwritten by the instruction, so RES sets the corresponding location in S_{pre} to an unconstrained symbolic value (line 6). When encountering a *memory read* instruction in B (line 8), RES faces two options: If that memory location will not be subsequently overwritten by an instruction in B, then RES knows exactly what value the read should return: the value is taken directly from S_{post}. If, however, that memory location will be overwritten somewhere in the remaining part of B, then RES cannot know what value resided there, so it returns from the read an unconstrained symbolic value. If there are any branch instructions that depend on symbolic values, RES forks the execution and follows both paths using a symbolic execution engine [22].

RES uses a combination of static and dynamic analysis (line 10) to determine which memory

Algorithm 2: Algorithm for reconstructing past program state

```
Input: coredump $Core$, snapshot $S_{pre}$
Output: computes $S_{post}$ (the state obtained after executing $S_{pre}$), updates $S_{pre}$ accordingly
1  $S' \leftarrow S_{pre}$
2  $(S_{post}, S_{pre}) \leftarrow$ execute($S'$) // execute $S'$ until it reaches the same program counter as Core.
3  return ($S_{post}$, $S_{pre}$)
4  /* execute() calls the beforeWrite and beforeRead functions before write and read instructions. */
5  function beforeWrite(memory object $mo$, address $addr$)
6    markSymbolicBytes($S_{pre}$, $addr$, $mo.size$)
7    return
8  function beforeRead(memory object $mo$, address $addr$)
9    // Determine if memory range will be overwritten
10   if checkNoOverwite($S'$, adddr, mo.size) then
11     return // memory range will not be overwritten, so it is identical in $S_{pre}$, $S'$, $S_{post}$, and Core.
12   else
13     markSymbolicBytes($S'$, $addr$, $mo.size$) markSymbolicBytes($S_{pre}$, $addr$, $mo.size$) return
```

will be overwritten in subsequent execution. This optimization aims to avoid the path explosion caused by merely returning unconstrained values for every memory read.

We have seen how RES synthesizes a sequential execution suffix. The next section describes the reverse execution synthesis of multi-threaded executions.

5.5 Thread Schedule Synthesis

The previous sections described RES for sequential executions. This section describes how RES synthesizes thread schedules for multi-threaded executions.

The gist of thread schedule synthesis is to use RES's sequential path synthesis algorithm and at the same time to explore multiple possible thread schedules and test if they can match the coredump. RES explores thread schedules by preempting threads at relevant points in their execution, such as before and after synchronization operations and accesses to shared memory. The algorithm used to explore thread schedules efficiently is inspired by CHESS [94].

For each possible execution suffix, RES uses three distinct analysis passes (Section 5.5.1, Section 5.5.2, Section 5.5.3). Each pass uses the information computed by the previous pass.

5.5.1 Identifying Shared Memory

The purpose of this initial analysis pass is to identify the write set of each thread for the execution suffix. This is an optimization pass meant to determine as many shared memory accesses as possible before running the second analysis pass. This pass can produce incomplete and inaccurate results, which are further refined in the next pass.

This pass analyzes a single path and a single thread schedule. It executes the program starting from a program snapshot that is almost identical to the coredump. The only difference between the initial snapshot and the coredump is that, in the snapshot, all threads have their PCs set back to the entry basic block of the last function on the call stack. At this stage RES does not infer what memory was overwritten in the snapshot, therefore the snapshot does not contain any symbolic memory and is unlikely to be a feasible program state. Executing this infeasible snapshot is likely to cause the program to crash (e.g., due to a buffer overflow) at arbitrary program locations. Thus, to be able to execute the program, RES uses a form of failure-oblivious computing [104] to execute the snapshot (i.e., it allocates buffers on-demand if the program accesses unallocated memory). Despite using failure-oblivious computing, the execution may not always reach the program locations in the coredump (the PCs of the threads may not be identical to the PCs of the threads in the coredump), so RES will stop threads from executing when it can determine that they can no longer reach a state that is aligned with the coredump or when the number of executed instructions by each thread exceeds a certain threshold.

Thus, the outcome of the initial analysis pass is a set of program counters at which different threads read or write to shared memory. These program counters are candidate preemption points for the next passes. This set is imprecise and it is neither an over- nor an under-approximation.

5.5.2 Synthesizing an Over-Approximate Thread Schedule

The goal of this second pass is to synthesize an over-approximate thread schedule. To achieve this, RES performs a backward analysis pass similar to reverse sequential path synthesis (Section 5.4), the main difference being that reads from shared memory locations return unconstrained symbolic values. Thus, both outcomes (i.e., *then* and *else*) of the branch conditions that depend on these symbolic values will be explored in this pass. Even though this pass explores a single thread schedule, it effectively over-approximates the effects of shared memory interactions between threads, therefore it over-approximates the possible states of the program under all thread schedules.

Any execution that matches the coredump is considered feasible, and the associated symbolic snapshots are passed to the next RES pass.

This pass detects additional shared memory locations that may have been missed by the previous pass. RES may need to backtrack in this pass if new shared memory is discovered. The number of times RES could backtrack is bounded (by the number of program locations in the program), but it may still be large, so RES caps it to a small number of iterations.

The outcome of this pass is an over-approximation of the possible execution suffixes of the program, since shared memory values are treated as being unconstrained symbolic memory.

5.5.3 Synthesizing an Accurate Thread Schedule Based on the Over-Approximation

The goal of this third pass is to synthesize an accurate thread schedule based on the over-approximation computed in the previous pass. This is the final pass of the thread schedule synthesis algorithm. The pass reasons explicitly about concurrency, determines how to interleave threads, and its outcome is an execution suffix that contains the synthesized thread schedule.

This pass starts from the feasible symbolic snapshots computed by the previous pass and executes starting from each of these snapshots.

The main difference from the previous pass is that it explores all possible thread schedules, within a configurable preemption bound. It introduces thread schedule preemptions at each shared memory location and synchronization primitive, and uses the technique described in Section 4.4.2 to search through the set of possible thread schedules while limiting the number of preemptions. Another difference from the previous pass is that reads from shared variables do not return symbolic unconstrained data, but rather their real values (which may be concrete or symbolic), similarly to the way ESD operates. Therefore, this analysis pass refines the possible symbolic snapshots computed by the previous pass.

In this final analysis pass, any execution that matches the coredump is considered feasible, and the execution suffix is presented to the user.

5.6 Complexity Analysis

This section analyzes reverse execution synthesis from a theoretical perspective, in order to determine its expected performance and the bounds on the length of the execution suffix for which reverse execution is practical.

5.6.1 Notation

We assume that we use reverse execution synthesis to reproduce the failure in a coredump *Core* of size *coreSize*, of which W bytes are symbolic in the symbolic snapshot at the beginning of the execution suffix (i.e., they have been overwritten by the execution suffix). We denote by d the number of instructions of the execution suffix required for debugging (i.e., from the root cause to the failure). We also denote p to be the number of unique predecessor basic blocks for all the basic blocks of the execution suffix.

5.6.2 Analysis

At each step of the reverse execution synthesis algorithm, RES performs a check to see if the candidate execution suffix matches the coredump. For the concrete bytes in the symbolic snapshot, this test is simply a comparison. For the symbolic bytes, the check requires solving a constraint, to determine if they can match the coredump. The number of symbolic bytes in the symbolic snapshot is at least equal to the number of bytes overwritten during the execution of the execution suffix, since RES over-approximates the set of overwritten bytes. The size of the constraint depends on the number of overwritten bytes W and the length of the execution suffix d, therefore we denote the total time required to solve constraints $CS(W,d)$. The constraint solving time depends on the structure of the program and on the actual d instructions.

Similarly to forward execution synthesis, reverse execution synthesis uses dynamic search, except that for reverse execution this search is backward from the coredump and is a function of the execution suffix length d, unlike the case of forward execution synthesis, for which d stands for the length of the entire execution. Another difference from forward execution synthesis is that RES searches through all predecessors of a basic block, while execution synthesis makes search decisions only at branches that depend on symbolic input. We denote the dynamic search time $DS(p)$.

Thus, reverse execution synthesis will require $RES(W,d,p) = CS(W,d) + DS(p)$ time to synthesize the correct execution suffix.

Similarly to forward execution synthesis, reverse execution synthesis is in the worst case exponential in d, since CS is exponential in d and DS is exponential in p, which means that RES is practical only for small values of d. This may still be sufficient for a short execution suffix, depending on the size of the constants. For instance, it may still be feasible to find the correct execution suffix for $d < 10$ and $p < 10$ in a reasonable amount of time.

Similarly to the complexity analysis of execution synthesis, we now analyze a scenario that is more optimistic than the worst case exponential case, since it assumes polynomial complexity. We do not analyze the best case, instead we assume polynomial complexity with a large exponent. We assume that both CS and DS are polynomial problems: For instance, $CS(W,d) = cs \cdot W^2 \cdot d^3$ and $DS(p) = ds \cdot p^4$ and $d = 10 \cdot p$, where cs, and ds are constants. With these assumptions, we get that $RES(W,d,p) = cs \cdot W^2 \cdot d^3 + 10^{-4} \cdot ds \cdot d^4$. If we further assume values for $cs = 10^{-8}$ seconds and $ds = 10^{-8}$ seconds, and that $W = 10 \cdot d$ (i.e., the number of overwritten bytes is proportional to the length of the execution suffix), we obtain that $RES(W,d,p) = 10^{-10} \cdot d^5 + 10^{-12} \cdot d^4$ (Figure 5.2). Thus, in this case, it is feasible to infer an execution suffix size of 1500 instructions in one week. Of course, this is just an example scenario and it does not represent the best case, therefore reverse execution synthesis can synthesize larger execution suffixes.

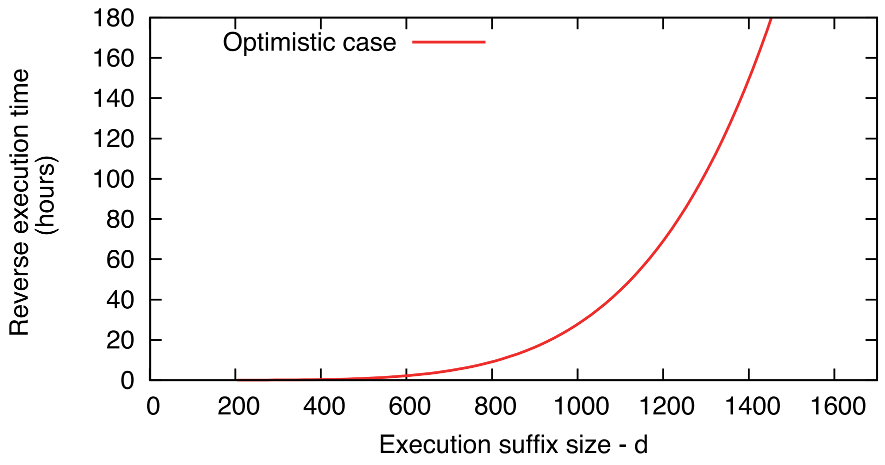


Figure 5.2: Example: reverse execution synthesis time vs. suffix length.

5.7 Discussion

The main limiting factor for RES is the size of the execution suffix. If the root cause of the failure is far from the failure, or the failure requires reproducing complex thread schedule interleavings, RES will encounter the unavoidable path explosion problem [18].

There are cases in which reversing executions requires inverting a difficult code construct (e.g., a hash function or a cryptographic function). RES, as described, might not be able to produce a suffix that goes beyond the difficult code construct. However, such code constructs may be regenerated otherwise, e.g., the inputs to the hash function may still be on the stack, and RES could re-execute the function instead of reverse-analyzing it.

Full coredumps may not always be available. RES can work with smaller dumps, but this may reduce its ability to compute long execution suffixes. However, RES could implement on-demand collection of additional data (e.g., broader memory dumps, log files), as in WER [54].

RES does not currently handle memory or stack corruption, because they may cause the CFG of the program to also be corrupted, and RES requires an accurate CFG and an accurate stack.

RES may not always identify the exact root cause that led to the observed failure, therefore it may not offer debug determinism (Chapter 3). However, RES's accuracy promises to be good, mainly owing to the fact that any execution suffix must match the full coredump exactly. In our experience, even small deviations from the real execution suffix lead to a different coredump. Furthermore, one could argue that every root cause of a failure should be fixed by developers; after fixing a root cause, RES can be run again to identify the other root cause, and so on until all root causes are fixed. Our initial experience with RES shows that it can provide accurate execution suffixes for complex bugs.

Theoretically—assuming unbounded resources—reverse execution synthesis can find all possible root causes, but it may also produce false positives (infeasible execution suffixes). Thus, unlike execution synthesis, k_{fp} can be > 0, so we cannot simplify it from the generic formula of debugging fidelity: $DF_{RES} = \frac{k_{original} + \frac{k_{other}}{n}}{k_{original} + k_{other} + k_{fp}}$. If RES is run until completion (i.e., until it finds full execution paths), it will produce the same outcome as ESD, therefore it will have the same debugging fidelity as well. Otherwise, due to the fact that RES may synthesize infeasible suffixes, it will have lower debugging fidelity than ESD.

This chapter showed how to synthesize an execution suffix in order to debug arbitrarily long executions. The work described in this chapter appeared in [135].

The next chapter describes the implementation and empirical evaluation of ESD and RES, as well as several uses cases of the two techniques.

Chapter 6

Evaluation and Use Cases

Having described the design of ESD and RES, in this chapter we describe the prototypes of these two techniques (Section 6.1) and their effectiveness in reproducing real bugs in real systems (Section 6.2). We also compare ESD to other approaches (Section 6.3), analyze ESD's performance (Section 6.4), and discuss several use cases for ESD and RES.

6.1 Prototypes

The ESD prototype currently works for C programs, and we verified that it works seamlessly with the gdb debugger. For symbolic execution, we adopted KLEE [23] and extended it in several ways; the most important one is support for multi-threaded symbolic execution. We first describe this extension in brief, and then provide a few details related to the implementation of synthesis and playback.

KLEE is a symbolic virtual machine designed for single-threaded programs. To allow ESD to explore various thread schedules, we added support for POSIX threads. We later improved support for POSIX threads in Cloud9 [22].

Our extended version supports most common synchronization API calls,such as thread, mutex and condition variable management, including thread-local storage functions. The new KLEE thread functions are handlers that hijack the program's calls to the native threads library. To create a simulated thread, ESD resolves at runtime the associated start routine and sets the thread's program counter to it, creates the corresponding internal thread data structures and a new thread stack, and adds the new thread to the ESD scheduler queue. ESD also maintains information on the state of mutex variables and on how threads are joined.

ESD runs one thread at a time. The decision of which thread to schedule next is made before and after each call to any of the synchronization functions, or before a load/store at a program location flagged as a potential data race. Each execution state has a list of the active threads. To schedule a thread, ESD replaces the stack and instruction pointer of the current state with the ones of the next thread to execute.

ESD preserves KLEE's abstraction of a process with an associated address space, and adds process threads that share this address space. As a result, the existing copy-on-write support for forked execution states can be leveraged to reduce memory consumption—this is key to ESD's scalability.

For the execution synthesis phase, ESD compiles the program to LLVM bitcode [79], a low-level instruction set in static single assignment form. We chose LLVM because KLEE operates on LLVM and because the associated compiler infrastructure provides rich static analysis facilities. LLVM provides load and store instructions up to word-level granularity, thus providing sufficient control for ESD to synthesize thread schedules that reproduce the desired data races.

We speed up the computation of the distance to the goal during synthesis by caching previously computed distances and using specialized data structures to track search state information. This optimization is crucial, because execution states in ESD can be switched at the granularity of individual instructions (i.e., is done frequently), so the selection of the next state to execute must be efficient.

For dynamic data race detection, ESD can use any existing algorithm. ESD implements a version of the Eraser [105] data race detection algorithm, modified to record, for each memory address, the last PC that accessed that particular memory address. This information is necessary to identify the program locations that are potentially racing, where ESD introduces preemption points.

During playback, ESD allows the program to create native threads and to call the real synchronization operations with the actual arguments passed by the program. The calls are intercepted in a library shimmed in via LD_PRELOAD. In here, synchronization operations can be delayed as needed to preserve the ordering from the synthesized execution file.

ESD prototype currently relies on a 32-bit version of KLEE, which means that it can address at most 4 GB of memory. For large real programs, this can cause ESD to run out of memory before finding the desired path. Porting ESD to 64-bit architectures will enable it to make use of the increasing amounts of physical memory available on modern machines.

We implemented a prototype of RES for LLVM [79] binaries (e.g., generated from C/C++ source code). RES supports multi-threaded programs and is implemented on top of the Cloud9 [22] symbolic execution engine. Currently, RES assumes sequential memory consistency when synthesizing execution suffixes, but we plan to lift this limitation in future work.

6.2 Effectiveness for Real Bugs in Real Systems

ESD succeeds in automating the debugging of real systems code. Table 6.1 shows examples of the programs we ran under ESD, ranging in size across three orders of magnitude: from over 100,000 LOC (SQLite) down to 100 LOC (mkfifo). All reported experiments ran on a 2 GHz quad-core Xeon E5405 CPU with 4GB of RAM, under 32-bit Linux. ESD had a total of 2GB of memory available.

System	**Bug manifestation**	**Execution synthesis time**
SQLite	hang	150 seconds
HawkNL	hang	122 seconds
ghttpd	crash	7 seconds
paste	crash	25 seconds
mknod	crash	20 seconds
mkdir	crash	15 seconds
mkfifo	crash	15 seconds
tac	crash	11 seconds

Table 6.1: ESD applied to real bugs: ESD synthesizes an execution in tens of seconds, while other tools cannot find a path at all in our experiments capped at 1 hour (see Section 6.3).

One class of bugs results in hangs. For example, bug #1672 in SQLite 3.3.0 is a deadlock in the custom recursive lock implementation. SQLite, an embedded database engine, is a particularly interesting target, since it has a reputation for being highly reliable and the developer-built test suites achieve 99% statement coverage [108]. This makes us believe that the remaining bugs are there because they are particularly hard to reproduce. Another hang bug appears in HawkNL 1.6b3 [3], a network library for distributed games. When two threads happen to call *nlClose()* and *nlShutdown()* at the same time on the same socket, HawkNL deadlocks.

Other bugs result in crashes. A security vulnerability in the *ghttpd* [2] Web server is caused by a buffer overflow when processing the URL for GET requests [52]. The overflow occurs in the `vsprintf` function when the request is written to the log. A bug in the *paste* UNIX utility [32] causes an invalid `free` for some inputs. The four bugs in the *tac*, *mkdir*, *mknod*, and *mkfifo* UNIX utilities [32] are all segmentation faults, with the last three occurring only on error handling paths. The UNIX utilities bugs are reported in [23].

ESD synthesized the bug-bound execution paths entirely automatically. For most bugs, ESD was able to automatically retrieve from the coredump the goal $<B,C>$ of the synthesized path. The only exception was ghttpd, whose coredump contained a corrupt call stack; it took a few minutes to manually reconstruct the correct call stack with gdb. ESD consistently synthesized an execution path to the bug under consideration, output the synthesized execution file (a couple MB in each case), and played it back deterministically.

Using ESD, we were able to play back each bug inside gdb. We perceived no overhead during playback, leading us to subjectively conclude that ESD does not hurt the developer's debugging experience. Even so, performance is rarely of importance, given that playback is repeatable and deterministic.

It is worth noting that ESD is effective not only for programs, but also for shared libraries, such as SQLite and HawkNL. Debugging libraries often has higher impact than debugging individual programs, because bugs inside libraries affect potentially many applications. For example, SQLite is used in Firefox, iPhone, Mac OS X, McAfee anti-virus software, Nokia's Symbian OS, PHP, Skype, and others [108]. In order to reproduce library bugs with ESD, one writes a program that exercises the library through the suspected buggy entry points; ESD then analyzes and symbolically executes these driver programs along with the library.

We evaluated RES on three synthetic concurrency bugs. These programs had 3 threads and around 100 LOC. The failures for these bugs were crashes due to buffer overflows. The root cause of these bugs were data races or atomicity violations. In all the cases RES was able to identify the correct root cause in less than 1 minute. RES only produced execution suffixes that reproduced the correct root cause, therefore it had no false positives.

6.3 Comparison to Alternate Approaches

Having seen ESD to be effective and fast, we now examine how it stacks up against alternate approaches.

The first approach to reproduce the bugs is brute force trial-and-error. To measure objectively, we ran several series of stress tests and random input testing for several hours. Neither of these efforts caused any of the bugs in Table 6.1 to manifest.

Bug finding tools, like KLEE [23] and Chess [94], can also be used to find paths to bugs—these tools produce test cases meant to reproduce the found bugs. Such a comparison is not entirely fair, for several reasons. On the one hand, ESD can synthesize execution paths for bugs that occur in production, away from ESD, whereas bug finding tools can only reproduce bugs that occur under their own close watch. On the other hand, bug finding tools are not guided toward a specific bug; their goal is to find previously unknown bugs and typically aim for high code coverage. Nevertheless, since we are not aware of other execution synthesis tools, we analyze the efficiency of ESD's search via this comparison.

We extended KLEE with support for multi-threading and implemented Chess's preemption-bounding approach for exploring multi-threaded executions [94]. We name the resulting tool KC—a hybrid system that embodies the KLEE and Chess techniques. We compare ESD to two different KC search strategies inherited directly from KLEE: *DFS*, which can be thought of as equivalent to an exhaustive search, and *RandPath*, a quasi-random strategy meant to maximize global path coverage. We augmented the corresponding strategies to encompass all active threads and limit preemptions to two, as done in [94].

We ran both KLEE and KC to find a path leading to each of the sample bugs in Table 6.1. After running for over one hour for each bug, neither tool found a path. The five bugs in UNIX utilities were originally found with KLEE and reported in [23]. Our experiments did not find them perhaps due to differences in the KLEE version and search strategies. ESD was built on top a KLEE code snapshot that was generously provided to us by its authors in Aug. 2008. In order to still have a practical baseline for comparison, we introduced four null-pointer-dereference bugs in the *ls* UNIX utility, for which KC does find a path in less than one hour. The *ls* utility has 3 KLOC.

Figure 6.1 shows the time it takes ESD to find a path vs. KC's two different search strategies. ESD is one to several orders of magnitude faster at finding the path to the target bug. We do not know if KC would eventually find a path to the bugs in Table 6.1 and, if it did, how long that would take.

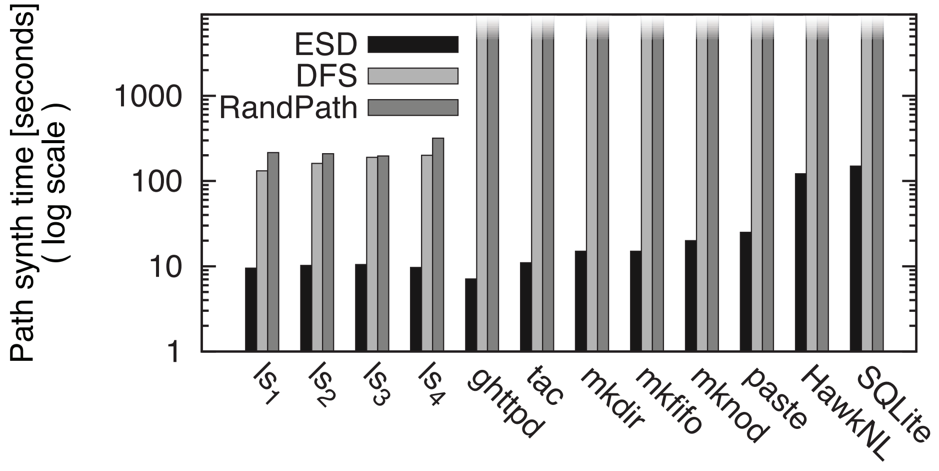


Figure 6.1: Comparison of time to find a path to the bug: ESD vs. the two variants of KC. Bars that fade at the top indicate KC did not find a path by the end of the 1-hour experiment.

6.4 Performance Analysis

In order to analyze the impact of zero-tracing execution path synthesis and the corresponding heuristics, we developed a microbenchmark, called BLIB (Branches-Inputs-Locks Benchmark). The main purpose of BLIB is to profile ESD without the measurements being influenced by environment interactions, such as library calls or system calls. For the more general case, BLIB can serve as a way to compare the performance of automated debugging tools like ESD.

BLIB produces synthetic programs that hang and/or crash. These programs have conditional branch instructions that depend on program inputs. When using more than one thread, the crash/hang scenarios depend on both the thread schedule and program inputs. BLIB allows direct control of five parameters for program generation: number of program inputs, number of total branches, number of branches depending (directly or indirectly) on inputs, number of threads, and number of shared locks.

We performed experiments with eight configurations of BLIB, comprising different program sizes. All data points correspond to programs with two threads and two locks, in which every branch instruction depends (directly or indirectly) on program inputs. There is one deadlock bug in each generated program. We varied the number of branch instructions from 2^3 to 2^{10}, which means that the number of possible branches varied from 2^4 to 2^{11}. We explored other benchmark configurations as well, but, given the results shown here, the results were as expected.

In an attempt to quantify the deadlock probability in the generated programs, we ran stress tests for one hour on each program. Neither of them deadlocked, suggesting that each program has a low probability of deadlocking "in practice," making these settings sufficiently interesting for our measurements. We then fed the programs to ESD and required it to synthesize an execution path exhibiting the deadlock bug. We confirmed that the synthesized executions indeed lead to the deadlock, by playing them back in gdb.

Figure 6.2 shows the time to synthesize an execution as a function of program complexity (in terms of branches). We find that ESD's performance varies roughly as expected; one exception is the jump from 2^8 to 2^9 branches—we suspect that structural features of the larger program presented an extra challenge for ESD's heuristics. Nevertheless, ESD performs well, keeping the time to synthesize a path to under 2 minutes, which is a reasonable amount of time for a developer to wait. We also included, for reference, the time taken by KC with the *RandPath* search strategy; it found a path within one hour only for the two simplest benchmark-generated programs. The *DFS* strategy did not find any path.

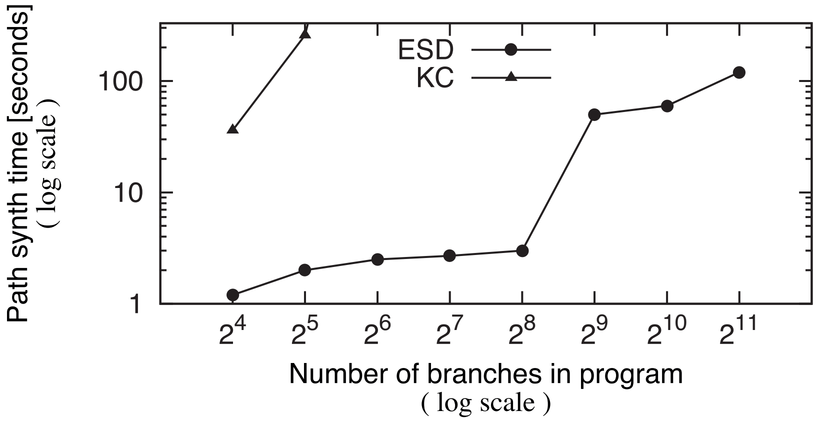


Figure 6.2: Synthesis time for programs of varying complexity for ESD and KC.

An alternate perspective on these results is to view them in terms of program size. Figure 6.3 shows the same data, but in terms of KLOC in the generated programs.

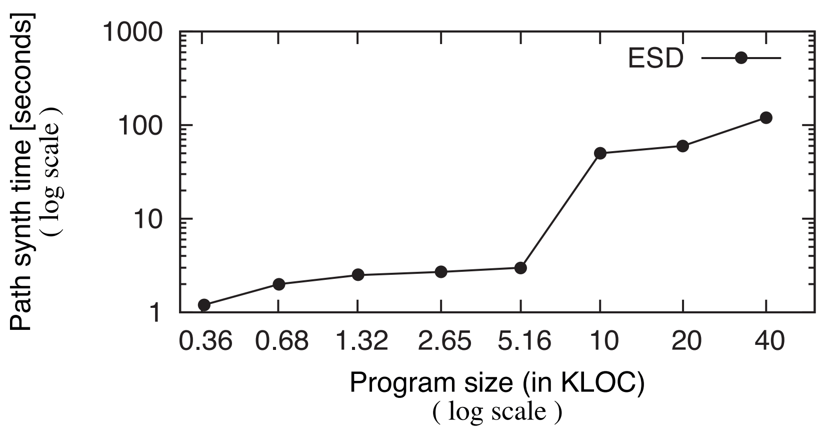


Figure 6.3: Synthesis time as a function of program size.

We conclude that ESD offers a practical way to automatically debug reported bugs, starting from just the corresponding bug report. Our evaluation shows that, whereas exhaustive or even improved random searches are unlikely to succeed in finding a path to the target bugs, ESD's execution synthesis heuristics are effective in guiding the search toward reproducing otherwise-elusive bugs.

6.5 Use Cases

We now present several use cases for execution synthesis and reverse execution synthesis: triaging bug reports (Section 6.5.1), detecting failures caused by hardware errors (Section 6.5.2), and automated debugging (Section 6.5.3). We compare the way execution synthesis and reverse execution synthesis address these uses cases.

6.5.1 Triaging Bug Reports

Debugging a large software development is hard, because the sheer volume of bug reports can be overwhelming [54]. In this context, accurately and automatically prioritizing reports from millions of users is particularly difficult yet crucial in cutting down the maintenance costs.

The main challenge in bug triaging is that a single bug can lead to different failures, and different bugs can lead to the same failure point. The state of the art in triaging bug reports is Windows Error Reporting (WER) [54]. Despite proving its utility in over ten years of operation, WER relies on simple heuristics and the law of large numbers. For instance, WER uses heuristics such as de-prioritizing reports that suggest bugs in core OS code, which is deemed to be correct. Thus, WER can incorrectly bucket up to 37% of the bug reports [54].

RES can complement WER by reconstructing the execution suffix and more precisely identifying the root cause of the failure. RES can process incoming bug reports and triage them based on the execution suffix and the likely root cause. Determining the root cause in the general case is hard; however, in several cases it is possible. For example, RES can detect reads from freed memory, which are likely to generate failures with different call stacks. A naive triaging technique that only looks at the call stack in the coredump would classify these failures in different buckets, while RES could improve accuracy by triaging based on the root cause. Similarly, a naive triaging might mis-triage bugs for which the root cause is not in the functions on the call stack. To cope with root causes that are hard to infer automatically, RES can use human feedback: once developers find the root cause of a failure, they can write RES annotations for the particular root cause, which would help RES triage other bug reports into the same bucket.

RES can also be used to classify bugs as exploitable by an attacker. For instance, say RES traces a failure to a buffer overflow and then further determines that the data copied to the buffer was tainted by external data that could be supplied by an attacker (e.g., a system call that reads a network packet). Such a verdict would automatically classify the bug as remotely exploitable and increase the priority level for the bug report. However, without RES, such a remotely exploitable bug, which typically generates many different failures (all with different call stacks), would be bucketed incorrectly (each failure in its own bucket). This could (1) cause the exploit to fly under the radar, because each instance of it would seem to be a different bug, and (2) burden the developers who have to inspect many buckets, all due in fact to the same bug.

This use case can also be addressed by execution synthesis, however, it is out of reach for ESD for the cases in which the execution is long (this is typically the case for executions that correspond to WER bug reports), therefore it is more suitable for RES.

6.5.2 Detecting Failures Caused by Hardware Errors

Hardware errors are common, correlated, and recurrent [97]. Machines that crash once due to a hardware error are two orders of magnitude more likely to crash a second time [97]. Moreover, hardware errors are not software bugs, therefore developers waste time debugging them instead of filtering them out. RES could be used to reduce this significant source of noise.

It is difficult to distinguish a hardware error from a software error, because both can manifest in similar ways. In some simple cases, as with machine check exception (MCE) CPU errors, it is easy to diagnose a hardware error, because the hardware detects the error in the first place. However, in other cases, such as memory errors, one cannot reliably differentiate between a software error (e.g., memory corruption) and a multi-bit DRAM failure or DMA writes from a faulty device.

Prior work [97] used manual post-hoc analysis to identify likely hardware failures in the CPU subsystem, one-bit memory flips, and disk system failures. These are cases in which manual analysis is easy. For instance, CPU errors are the ones that trigger an MCE and checks for one-bit memory flips are limited to the kernel image, which is meant to be read-only and can be compared to the vanilla kernel image.

The open question that could be solved with RES is how to automate this manual process and extend it to more challenging cases (e.g., for memory that is not read-only). For instance, while analyzing a coredump, RES can discover inconsistencies between the coredump and the execution of the program prior to generating the coredump, indicating that the likely explanation is a hardware error. One example are memory errors: if on all the possible paths to the coredump the program writes the value 1 to a certain memory address, but the coredump contains the value 0, this would likely indicate a memory error. Another example are CPU errors: say the CPU miscomputed an addition, and this led to a crash. If RES retrieves the result and the operands from the coredump, and on all possible suffixes it obtains a different result for the addition, it concludes the likely explanation for this is a hardware error. Of course, diagnosing a hardware error with full accuracy requires exploring all possible execution suffixes; this may be possible for short suffixes.

This use case is suitable for RES, but it is not suitable for ESD. The first reason is the limited ability of (forward) execution synthesis to deal with long executions (similarly to the previous case). The second reason is that execution synthesis cannot typically prove that a certain execution suffix is infeasible. To achieve this, execution synthesis would have to exhaustively explore all executions for which it cannot statically prove they cannot reach the coredump. This set may be arbitrarily large even for short executions, so execution synthesis would not be practical. On the other hand, reverse execution synthesis may only need to analyze a small set of execution suffixes to prove that the root cause is a hardware error.

6.5.3 Automated Debugging

ESD enables several debugging aids on top of traditional debuggers like gdb: synthesizing an execution, reconstructing past state (the symbolic snapshots), and the ability to do reverse debugging without the need to record the execution. Additionally, RES automatically computes the read and write sets of the execution suffix, therefore it automatically focuses developers' attention on the recently read or written state, which, for debugging, is more likely to be important than the rest of the coredump.

RES could also be used to automate the testing of various hypotheses formulated during debugging, such as "what was the program state when the program was executing at program counter X," or "was a thread T preempted before updating shared memory location M?"

Unlike ESD, RES reproduces any failure that generates a coredump, therefore it is not restricted to a particular type of bugs—even semantic bugs (e.g., captured by `assert` statements) can be reproduced.

Similarly to the previous two use cases, this use case may not always be amenable to execution synthesis due to the length of the execution.

This section discussed three uses cases for forward execution synthesis and reverse execution synthesis. By virtue of addressing arbitrarily long executions, reverse executions synthesis is more applicable to these use cases than forward execution synthesis.

Chapter 7

The Synthesis-Time vs. Runtime-Overhead Trade-Off

Identifying the cause of a bug based on a bug report is often difficult because there is a gap between the information available at the time of a software failure and the information actually shipped to developers in the bug report. Augmenting bug reports with recorded runtime information can help debugging, however, there is an inherent trade-off between runtime recording overhead and the fidelity/ease of subsequently reproducing bugs. The spectrum of solutions has full system replay [42] at one end, and execution synthesis at the other. This spectrum is still poorly understood, and an important question remains: which is the least amount of information that is practical to record at runtime, yet still makes it easy to diagnose bugs of a certain type?

The main challenge for execution synthesis is synthesizing deep paths. Similarly, the main challenge for reverse execution synthesis is synthesizing long execution suffixes. As described in Chapter4 and Chapter 5, the reason for both of these challenges is that the synthesis process relies on search algorithms that are often exponential in the depth of the synthesized path (or of the synthesized execution suffix in the case of reverse execution synthesis).

This chapter describes a hybrid technique that aims to make execution synthesis and reverse execution synthesis more efficient for deeper execution paths. This technique trades runtime overhead for execution synthesis time by recording small pieces of runtime information about the execution and using these pieces as intermediate goals during execution synthesis. We call the recorded pieces of information "execution breadcrumbs".

Intuitively, execution breadcrumbs have the same role in execution synthesis as the breadcrumbs used by the two main characters of the Hansel and Gretel story [59]: Hansel and Gretel used breadcrumbs to "record" a trace of their way back home. Even though the breadcrumbs were sparse, they were close enough for Hansel and Gretel to efficiently find a path back home.

This hybrid technique is different from execution synthesis, since the definition of execution synthesis (Section 4.1) does not include recording. Instead, this technique resembles record-replay systems that provide relaxed determinism (e.g., ODR [9], PRES [102], Oasis [35]).

Unlike record-replay systems that record all non-determinism, hybrid execution synthesis aims to perform little recording. The requirements we set forth are the following:

- Low runtime overhead:

  Overhead less than 1% is likely to be considered acceptable for production use, while overhead higher than 1% will impede wide adoption.

- No modifications to applications:

  Requiring developers to change applications will also impede wide adoption. Instead, modifying a shared library that can be linked with applications to make them more amenable to hybrid execution synthesis is acceptable.

- Improve synthesis time:

  Execution breadcrumbs should help reduce the execution synthesis time for deep execution paths.

This chapter describes how to record small pieces of runtime information to speed up the debugging of deadlocks (Section 7.1) and how the recorded thread schedule of an execution can be used in conjunction with execution synthesis to accurately classify data race bugs (Section 7.2).

7.1 Recording Bug Fingerprints to Speed Up the Debugging of Deadlock Bugs

We propose *bug fingerprints*—an augmentation of classic automated bug reports with runtime information about how the reported bug occurred in production. Bug fingerprints contain additional small amounts of highly relevant runtime information that helps understand how the bug occurred.

7.1.1 Problem Statement

Our observation is that, given a class of bugs, it is possible to record a small amount of bug-specific runtime information with negligible overhead, and this information can substantially improve debugging. Based on this observation, we propose *bug fingerprints*, small additions to classic bug reports that contain highly relevant "breadcrumbs" of the execution in which the bug occurred. These breadcrumbs ease the reconstruction of the sequence of events that led to the failure.

We show that this idea works for deadlocks, an important class of concurrency bugs. We built DCop, a prototype deadlock fingerprinting system for C/C++ software—it keeps track at runtime of each thread's lock set and the callstacks of the corresponding lock acquisitions; when a deadlock hangs the application, this information is added to the bug report. DCop's runtime overhead is negligible (e.g., less than 0.17% for the Apache Web server), yet these breadcrumbs enable faster, even automated, debugging.

Despite being frequent (e.g., 30% of the bugs reported in [85] are deadlocks), deadlock bug reports are scarce, because deadlocks do not produce a coredump—instead, they render the application unresponsive. Normal users restart the application without submitting a bug report, while expert users may attach a debugger to the program and capture each thread's callstack. Systems such as WER [54] can be used to create a coredump, but it is still hard to debug deadlocks based on this information that describes only the end state of the program.

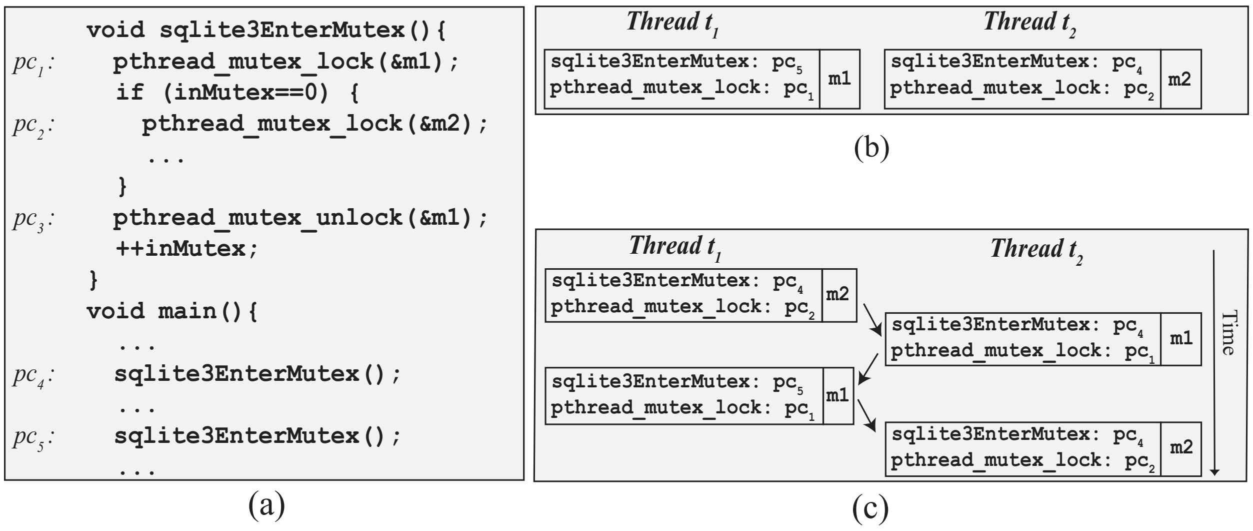
7.1.2 Design

Deadlocks become straightforward to debug if we have information on how the program acquired *every* mutex involved in the deadlock. In particular, the callstacks of the calls that acquired mutexes *held* at the time of deadlock, together with the callstacks of the *blocked* mutex acquisitions, provide rich information about how the deadlock came about. Alas, the former type of callstack information is no longer available at the time of the deadlock, and so it does not appear in the coredump.

Fortunately, it is feasible to have this information in every bug report: First, the amount of information is small—typically one callstack per thread. Second, it can be maintained with low runtime overhead, because most programs use synchronization infrequently. As it turns out, even for lock-intensive programs DCop incurs negligible overhead.

DCop's deadlock fingerprints contain precisely this information. Regular deadlock bug reports contain callstacks, thread identifiers, and addresses of the mutexes that are requested—but not held—by the deadlocked threads. We call these the *inner* mutexes, corresponding to the innermost acquisition attempt in a nested locking sequence. Additionally, deadlock fingerprints contain callstack, thread id, and address information for the mutexes that are already held by the threads that deadlock. We call these the *outer* mutexes, because they correspond to the outer layers of the nested locking sequence. Outer mutex information must be collected at runtime, because the functions where the outer mutexes were acquired are likely to have already returned prior to the deadlock.

We illustrate deadlock fingerprints with the code in Fig. 7.1a, a simplified version of the global mutex implementation in SQLite [108], a widely used embedded database engine. The bug occurs when two threads execute *sqlite3EnterMutex()* concurrently. Fig. 7.1b shows the classic bug report, and Fig. 7.1c shows the deadlock fingerprint.



```
       void sqlite3EnterMutex(){
pc1:     pthread_mutex_lock(&m1);
         if (inMutex==0) {
pc2:       pthread_mutex_lock(&m2);
           ...
         }
pc3:     pthread_mutex_unlock(&m1);
         ++inMutex;
       }
       void main(){
         ...
pc4:     sqlite3EnterMutex();
         ...
pc5:     sqlite3EnterMutex();
         ...
```

(a)
Thread t_1
Thread t_2
sqlite3EnterMutex: pc_5
pthread_mutex_lock: pc_1
m1
sqlite3EnterMutex: pc_4
pthread_mutex_lock: pc_2
m2
(b)
Thread t_1
Thread t_2
sqlite3EnterMutex: pc_4
pthread_mutex_lock: pc_2
m2
sqlite3EnterMutex: pc_4
pthread_mutex_lock: pc_1
m1
sqlite3EnterMutex: pc_5
pthread_mutex_lock: pc_1
m1
sqlite3EnterMutex: pc_4
pthread_mutex_lock: pc_2
m2
Time
(c)

Figure 7.1: **(a)** SQLite deadlock bug #1672. **(b)** Regular bug report. **(c)** DCop-style deadlock fingerprint.

A regular bug report shows the final state of the deadlocked program: t_1 attempted to lock mutex m_1 at pc_1 and t_2 attempted to lock mutex m_2 at pc_2—we invite the reader to diagnose how the deadlock occurred based on this information. The bug report does not explain how t_1 acquired m_2 and how t_2 acquired m_1, and this is not obvious, since there are several execution paths that can acquire mutexes m_1 and m_2.

The deadlock fingerprint (Fig. 7.1c) clarifies the sequence of events: t_1 acquired m_2 at pc_2 in a first call to *sqlite3EnterMutex*, and t_2 acquired m_1 at pc_1. This allows a developer to realize that, just after t_1 unlocked m_1 at pc_3 and before t_1 incremented the *inMutex* variable, t_2 must have locked m_1 at pc_1 and read variable *inMutex*, which still had the value 0. Thus, t_2 blocked waiting for m_2 at pc_2. Next, t_1 resumed, incremented *inMutex*, called *sqlite3EnterMutex* the second time, and tried to acquire m_1 at pc_1. Since m_1 was held by t_2 and m_2 was held by t_1, the threads deadlocked. This is an example of how DCop can help debug the deadlock and reveal the data race on *inMutex*.

To acquire this added information, DCop uses a lightweight instrumentation layer that intercepts the program's synchronization operations. It records the acquisition callstack for currently held mutexes in a per-thread event list. A deadlock detector is run whenever the application is deemed unresponsive, and it determines whether the cause is a deadlock.

The runtime monitor is designed to incur minimal overhead. First key decision was to avoid contention at all costs, so each thread records the callstack information for its lock/unlock events in a thread-local private list. The private lists are merged solely when a deadlock is found (and thus the application threads are stuck anyway). This avoids introducing any additional runtime synchronization.

A second design choice was to trim the private lists and keep them to the minimum required size: every time a mutex is unlocked, DCop finds the corresponding lock event in the list and discards it—mutexes that are no longer held cannot be involved in deadlocks. Thus, DCop only keeps track of mutexes that have not yet been released, and so the size of a per-thread event list is bounded by the maximum nesting level of locking in the program. In our experience, no event lists ever exceeded 4 elements.

As a result of this design, DCop's runtime overhead is dominated by obtaining the backtrace on each mutex acquisition. To reduce this overhead to a minimum, DCop resolves backtrace symbols offline, since this is expensive and need not be done at runtime.

The deadlock detection component of DCop is activated when the user stops an application due to it being unresponsive. The detector processes each thread's list and creates a resource allocation graph (RAG) based on the events in the lists. The RAG contains a vertex for each active thread and mutex, and edges correspond to mutex acquisitions (or acquisition requests that have not succeeded yet). Edges are labeled with the thread id of the acquiring thread and the callstack corresponding to the lock operation. Once the RAG is constructed, the detector checks for cycles in the graph—a RAG cycle corresponds to a deadlock. If a deadlock is found, the detector assembles the corresponding fingerprint based on the callstacks and thread identifiers found on the cycle's edges.

DCop's deadlock detector has zero false positives. Furthermore, since the size of the threads' event lists is small, assembling a deadlock fingerprint is fast.

For the cycle detection, performed in the monitor, execution time is linear in the average number of events in the per-thread list and linear in $N \cdot (|V| + |E|)$ (where $RAG = [V, E]$ and $N =$ the average number of active threads), because we use optimal colored DFS [76] for detecting cycles.

7.1.3 Implementation

We implemented DCop inside FreeBSD's `libthr` POSIX threads library; our changes added 382 LOC. One advantage of recording fingerprints from within the existing threading library is the opportunity to leverage existing data structures. For example, we added pointers to DCop's data structures inside the library's own thread metadata structure. An important optimization in DCop is the use of preallocated buffers for storing the backtrace of mutex acquisitions—this removes memory allocations from the critical path.

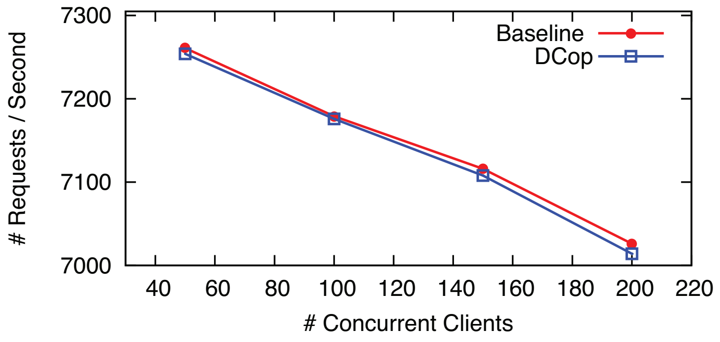


Figure 7.2: Comparative request throughput for the Apache 2.2.14 server at various levels of client concurrency.

7.1.4 Evaluation

Having discussed DCop's design and implementation, we now turn our attention to the key question of whether it is suitable for use in production? We evaluate DCop's performance on a workstation with two Intel 4×1.6GHz-core CPUs with 4GB of RAM running FreeBSD 7.0.

First, we employ DCop on interactive applications we use ourselves, such as the emacs text editor. There is no perceptible slowdown, leading to the empirical conclusion that user-perceived overhead is negligible. However, since recording mutex operations adds several instructions at each synchronization operation, (e.g., obtaining the backtrace for a lock operation), some lock intensive programs may exhibit more overhead.

Next, we use DCop for the Apache Web server with 50 worker threads. We vary the number of concurrent clients and, for each concurrency level, we execute 5×10^5 GET requests for a 44-byte file. In Fig. 7.2 we compare the aggregate request throughput to a baseline without DCop. The overhead introduced by DCop is negligible throughout, with the worst-case being a less than 0.17% drop in throughput for 200 concurrent clients. Both baseline and DCop throughput decrease slightly with concurrency level, most likely because there are more clients than worker threads. The maximum synchronization throughput (lock operations/second) reaches 7249 locks/second.

To analyze DCop's overhead in depth, we wrote a synchronization-intensive benchmark that creates 2 to 1024 threads that synchronize on 8 shared mutexes. Each thread holds a mutex for δ_{in} time, releases it, waits for δ_{out} time, then tries to acquire another mutex. δ_{in} and δ_{out} are implemented as busy loops, thus simulating computation done inside and outside a critical section. The threads randomly call multiple functions within the microbenchmark, in order to build up highly varied callstacks ("fingerprints").

We measure how synchronization throughput varies with the number of threads. In Fig. 7.3 we show DCop's overhead for δ_{in}=1 microsecond and δ_{out}=1 millisecond, simulating a program that grabs a mutex, updates in-memory shared data structures, releases the mutex, and then performs

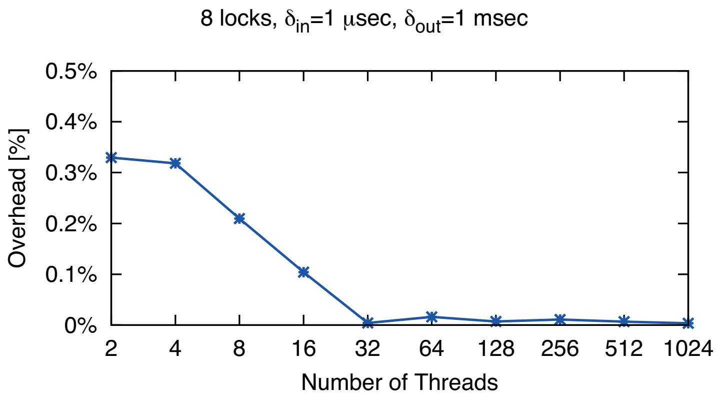


Figure 7.3: Overhead of collecting deadlock fingerprints as a function of the number of threads.

computation outside the critical section. The worst case overhead is less than 0.33% overhead. The decreasing overhead shows that indeed DCop introduces no lock contention. Instead, the application's own contention amortizes DCop's overhead.

We repeat the experiment for various combinations of $1 \leq \delta_{in} \leq 10^4$ and $1 \leq \delta_{out} \leq 10^4$ microseconds, simulating applications with a broad range of locking patterns. The measured overhead ranges from 0.06% in the best case to 0.77% in the worst case. The maximum measured synchronization throughput reaches 831,864 locks/second.

These results confirm that DCop introduces negligible runtime overhead, thus making it well suited for running in production, even for server applications. We hope this advantageous cost/benefit trade-off will encourage wider adoption of deadlock fingerprinting.

7.1.5 Discussion

Augmenting bug reports with bug fingerprints can substantially speed up debugging. For example, a developer debugging a deadlock can get from the deadlock fingerprint all mutexes involved in the deadlock and the callstacks corresponding to their acquisition calls. This allows the developer to insert breakpoints at all outer mutex locations and understand how the deadlock can occur.

Bug fingerprints can improve the efficiency of execution synthesis, since they help disambiguate between possible executions. Bug fingerprints contain clues that can substantially prune this search space. For example, a major challenge in execution synthesis for deadlocks is identifying the thread schedule that leads to deadlock. DCop's deadlock fingerprints narrow down the set of possible schedules, thus reducing search time. In preliminary measurements, we find that for a program with three threads and an average lock nesting level of three, the thread schedule synthesis phase of execution synthesis can be reduced by an order of magnitude.

Bug Fingerprints for Other Bug Types

Choosing what runtime information to include in a given fingerprint is specific to each class of bugs. We illustrate this process with two examples: data races and unchecked function returns.

A bug fingerprint for a data race-induced failure contains information on the races that manifested during execution prior to the failure in the bug report. This way, it is possible to determine which potential data races influenced the execution and which did not. However, monitoring memory accesses efficiently is not easy.

An efficient data race fingerprinting system employs static analysis to determine offline, prior to execution, which memory accesses are potential data races [73, 81]. It then monitors at runtime only these accesses. We see two options to perform such monitoring with low overhead: debug registers and transactional memory (TM). x86 debug registers [64] can be configured to deliver an interrupt to a monitor thread whenever two memory accesses to the same address are not ordered by a happens-before relation and at least one of the access is a write (i.e., a data race occurred). The corresponding program counters and memory address are then saved for later inclusion in the bug report, should a failure occur. One drawback is that today's CPUs can monitor only a small set of addresses at a time, so debug registers can be used to watch only a subset of the statically-discovered potential races. An alternative approach is to use the conflict detection mechanism of TM to detect data races, and record the fingerprint. If TM features are available in hardware, this can be done quite efficiently.

Another interesting class of bugs appears in code that "forgets" to check all possible return values of a library function. For example, not checking whether a socket `read()` call returned -1 can lead to data loss (if caller continues as if all data was read) or even memory corruption (if return value is used as an index). For such unchecked-return bugs, the fingerprint contains (a) the program locations where a library function call's return value was not checked against all possible return values, and (b) the actual return value. Such fingerprinting can be done with low overhead by statically analyzing the program binary to determine the places in the program where library calls are not properly checked (e.g., using the LFI callsite analyzer [88]), and monitoring at runtime only those locations.

For most bug types, a general solution is to incrementally record the execution index [122] and include it in the bug fingerprint. The execution index is a precise way to identify a point in an execution and can be used to correlate points across multiple executions. Such a bug fingerprint can be used to reason with high accuracy about the path that the program took in production, but has typically high recording overhead (up to 42% [122]). It is possible to reduce the overhead by recording only a partial execution index (e.g., by sampling) that, although less precise, can still offer clues for debugging.

It is practical to fingerprint any class of bugs, as long as the runtime information required to disambiguate possible executions that manifest the bug can be recorded efficiently. Fingerprinting mechanisms can leverage each other, so that collecting fingerprints for *n* classes of bugs at the same time is cheaper than *n* times the average individual cost.

This section described how to speed up the debugging of several types of bugs using small pieces of recorded runtime information. This work appeared in [134].

7.2 Recording the Thread Schedule to Automatically Classify Data Race Bugs

Even though most data races are harmless, the harmful ones are at the heart of some of the worst concurrency bugs. Alas, spotting just the harmful data races in programs is like finding a needle in a haystack: 76%-90% of the true data races reported by state-of-the-art race detectors turn out to be harmless [95].

We built Portend [72], a tool that not only detects races but also automatically classifies them based on their potential consequences: Could they lead to crashes or hangs? Could their effects be visible outside the program? Are they harmless? Portend achieves high accuracy by efficiently analyzing multiple paths and multiple thread schedules in combination.

Multi-path analysis in Portend is based on hybrid execution synthesis: Portend records the thread schedule of an execution and then uses execution synthesis to find alternative executions that follow the same thread schedule. The additionally synthesized executions help increase classification accuracy.

We ran Portend on 7 real-world applications: it detected 93 true data races and correctly classified 92 of them, with no human effort. 6 of them are harmful races. Portend's classification accuracy is up to 89% higher than that of existing tools, and it produces easy-to-understand evidence of the consequences of harmful races, thus both proving their harmfulness and making debugging easier. We envision Portend being used for testing and debugging, as well as for automatically triaging bug reports.

7.2.1 Problem Statement

Data races are some of the worst concurrency bugs. As programs become increasingly parallel, we expect the number of data races they contain to increase. Eliminating all data races still appears impractical. First, synchronizing all racing memory accesses would introduce performance overheads that may be considered unacceptable. For example, for the last year, developers have not fixed a race in memcached that can lead to lost updates—ultimately finding an alternate solution—because it leads to a 7% drop in throughput [91]. Performance implications led to 23 data races in Internet Explorer and Windows Vista being purposely left unfixed [95]. Similarly, several races have been left unfixed in the Windows kernel, because fixing those races did not justify the associated costs [69].

Another reason why data races go unfixed is that 76%–90% of data races are considered harmless by developers [69, 95, 44, 117]—*harmless races* do not affect program correctness, either fortuitously or by design, while *harmful races* lead to crashes, hangs, resource leaks, even memory corruption or silent data loss. Deciding whether a race is harmful or not involves a lot of human labor (with industrial practitioners reporting that it can take days, even weeks [57]), so time-pressed developers may not even attempt this high-investment/low-return activity. On top of all this, static race detectors can have high false positive rates (e.g., 84% of races reported by [117] were not true races), further disincentivizing developers. Alas, automated classifiers [66, 69, 95, 111] are often inaccurate (e.g., [95] reports a 74% false positive rate in classifying harmful races).

In this thesis we will only focus on how hybrid execution synthesis is used in Portend. A full description of Portend is available in [72] and is beyond the scope of this thesis. In the rest of this chapter we provide a high level overview of Portend (Section 7.2.2) and evaluate the contribution of hybrid execution synthesis to classification accuracy (Section 7.2.3).

7.2.2 Design

Classification Categories

Portend automatically classifies data races into four categories: “specification violated”, “single ordering”, “output differs”, and “k-witness harmless”. We illustrate this taxonomy in Fig. 7.4.

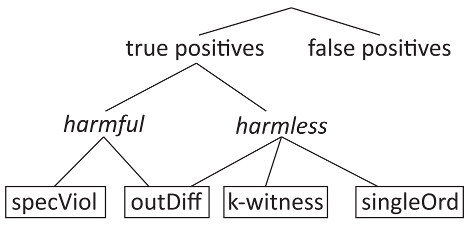


Figure 7.4: Portend taxonomy of data races.

“Spec violated” corresponds to races for which at least one ordering of the racing accesses leads to a violation of the program’s specification. These are, by definition, harmful. For example, races that lead to crashes or deadlocks are generally accepted to violate the specification of any program; we refer to these as “basic” specification violations.

“Output differs” is the set of races for which the two orderings of the racing accesses can lead to the program generating different outputs, thus making the output depend on scheduling. Such races are often considered harmful: one of those outputs is likely “the incorrect” one. However, "output differs" races can also be harmless, whether intentional or not. For example, a debug statement that prints the ordering of the racing memory accesses is intentionally order-dependent, thus an intentional harmless race.

"K-witness harmless" are races for which the harmless classification is performed with some quantitative level of confidence: the higher the *k*, the higher the confidence. Such races are guaranteed to be harmless for at least *k* combinations of paths and schedules; this guarantee can be as strong as covering a virtually infinite input space (e.g., a developer may be interested in whether the race is harmless for all positive inputs, not caring about what happens for zero or negative inputs). Depending on the time and resources available, developers can choose *k* according to their needs—in our experiments we found $k = 5$ to be sufficient to achieve 99% accuracy for all the tested programs.

"Single ordering" are races for which only a single ordering of the accesses is possible, typically enforced via ad-hoc synchronization [123]. In such cases, although no explicit synchronization primitives are used, the shared memory could be protected using busy-wait loops that synchronize on a flag. We consider this a race because the ordering of the accesses is not enforced using synchronization primitives, even though it is not actually possible to exercise both interleavings of the memory accesses (hence the name of the category). Such ad-hoc synchronization, even if bad practice, is frequent in real-world software [123]. Previous data race detectors generally cannot tell that only a single order is possible for the memory accesses, and thus report this as a race; such cases turn out to be a major source of harmless data races [66, 111].

Overview

The challenge in accurately classifying data races is finding multiple executions that exercise the same data race: more executions provide higher classification accuracy. Portend's race analysis starts by executing the target program and dynamically detecting data races using a dynamic happens-before [78] data race algorithm. For each of the data races, Portend records the thread schedule and uses it as a trace of execution breadcrumbs. Subsequently, Portend uses execution synthesis to find multiple executions that match the same breadcrumbs and thus increase data race classification accuracy.

First, Portend replays the schedule in the trace up to the point where the race occurs (Fig. 7.5a). Then, it explores two different executions: one in which the original schedule is followed (the *primary*) and one in which the alternate ordering of the racing accesses is enforced (the *alternate*). Some classifiers compare the primary and alternate program state immediately after the race, and, if different, flag the race as potentially harmful. Even if program outputs are compared rather than states, "single-pre/single-post" analysis (Fig. 7.5a) may not be accurate, as we will show below. Portend uses "single-pre/single-post" analysis to record the execution breadcrumbs and to

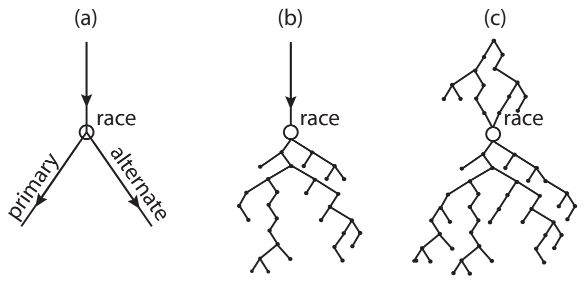


Figure 7.5: Increasing levels of completeness in terms of paths and schedules: [a. single-pre/single-post] $\ll$ [b. single-pre/multi-post] $\ll$ [c. multi-pre/multi-post].

determine whether the alternate schedule is possible at all. In other words, this stage identifies any ad-hoc synchronization that might prevent the alternate schedule from occurring.

If there is a difference between the primary and alternate post-race states, Portend does not consider the race as necessarily harmful. Instead, we allow the primary and alternate executions to run, independently of each other, and we observe the consequences. If, for instance, the alternate execution crashes, the race is harmful. Of course, even if the primary and alternate executions behave identically, it is still not certain that the race is harmless: there may be some unexplored pair of primary and alternate paths with the same pre-race prefix as the analyzed pair, but which does not behave the same. This is why single-pre/single-post analysis is insufficient, and we need to explore *multiple* post-race paths. This motivates "single-pre/multi-post" analysis (Fig. 7.5b), in which multiple post-race execution possibilities are explored—if any primary/alternate mismatch is found, the developer must be notified.

Even if all feasible post-race paths are explored exhaustively and no mismatch is found, one still cannot conclude that the race is harmless: it is possible that the absence of a mismatch is an artifact of the specific pre-race execution prefix, and that some different prefix would lead to a mismatch. Therefore, to achieve higher confidence in the classification, Portend uses hybrid execution synthesis to explores multiple feasible paths even in the pre-race stage, not just the one path witnessed by the race detector. This is illustrated as "multi-pre/multi-post" analysis in Fig. 7.5c. The advantage of doing this vs. considering these as different races is the ability to systematically explore these paths.

Portend combines multi-path analysis with *multi-schedule* analysis, since the same path through a program may generate different outputs depending on how its execution segments from different threads are interleaved. The branches of the execution tree in the post-race execution in Fig. 7.5c correspond to different paths that stem from both multiple inputs and schedules.

Hybrid Execution Synthesis

The "multi-pre" analysis in Portend is done using hybrid execution synthesis.

The goal of this step is to explore variations of the single paths found in the previous step (i.e., the primary and the alternate) in order to expose Portend to a wider range of execution alternatives.

First, Portend uses hybrid execution synthesis to find multiple primary paths that satisfy the input trace, i.e., they (a) all experience the same thread schedule (up to the data race) as the input trace, and (b) all experience the target race condition. These paths correspond to *different* inputs from the ones in the initial race report.

To synthesize multiple paths that traverse the same execution breadcrumbs (the recorded thread schedule), Portend now executes the primary *symbolically*. This means that the target program is given symbolic inputs instead of regular concrete inputs. When an expression with symbolic content is involved in the condition of a branch, *both* options of the branch are explored, if they are feasible. The resulting path(s) are annotated with a constraint indicating that the branch condition holds true (respectively false). Thus, instead of a regular single-path execution, we get a tree of execution paths, similar to the search space of execution synthesis.

During hybrid execution synthesis, Portend prunes the paths that do not obey the thread schedule (the execution breadcrumbs) in the trace, thus excluding the (many) paths that do not enable the target race. Moreover, Portend attempts to follow the original trace only until the second racing access is encountered; afterward, it allows execution to diverge from the original schedule trace. This enables Portend to find more executions that partially match the original schedule trace.

In this section we discussed how hybrid execution synthesis is used in Portend to find multiple executions that reproduce the same data race. In the next section we empirically evaluate the contribution of this technique to data race classification accuracy.

7.2.3 Evaluation

In this section we will briefly describe Portend's results and then analyze the influence that Portend's hybrid execution synthesis component has on classification accuracy.

We apply Portend to 7 applications: SQLite, an embedded database engine (used, for example, by Firefox, iOS, Chrome, and Android), that is considered highly reliable, with 100% branch coverage [108]; Pbzip2, a parallel implementation of the widely used bzip2 file compressor [53]; Memcached [47], a distributed memory object cache system (used, for example, by services such as Flickr, Twitter and Craigslist); Ctrace [90], a multi-threaded debug library; Bbuf [125], a shared buffer implementation with a configurable number of producers and consumers; Fmm, an n-body simulator from the popular SPLASH2 benchmark suite [120]; and Ocean, a simulator of eddy currents in oceans, from SPLASH2.

Portend classifies with 99% accuracy the 93 known data races we found in these programs, with no human intervention, in under 5 minutes per race on average.

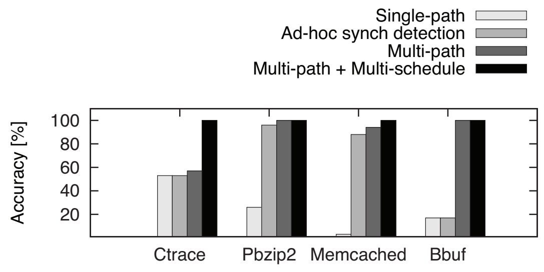


Figure 7.6: Breakdown of the contribution of each technique toward Portend's accuracy. We start from single-path analysis and enable one by one the other techniques: ad-hoc synchronization detection, multi-path analysis, and finally multi-schedule analysis.

Multi-path multi-schedule exploration—which uses hybrid execution synthesis—proved to be crucial for Portend's accuracy. Fig. 7.6 shows the breakdown of the contribution of each technique used in Portend: ad-hoc synchronization detection, multi-path analysis, and multi-schedule analysis. In particular, for 16 out of 21 "output differs" races (6 in bbuf, 9 in ctrace, 1 in pbzip2) and for 1 "spec violated" race (in ctrace), single-path analysis revealed no difference in output; it was only multi-path multi-schedule exploration that revealed an output difference (9 races required multi-path analysis for classification, and 8 races required also multi-schedule analysis). Without multi-path multi-schedule analysis, it would have been impossible for Portend to accurately classify those races by just using the available test cases.

This section described how to use hybrid execution synthesis to synthesize multiple executions that reproduce the same data race, in order to improve the accuracy of a data race classifier. This work was published in [72].

The following chapter describes future work ideas on how to improve the debugging techniques described in this thesis.

Chapter 8

Future Work

In this chapter we discuss several future work ideas related to both forward and reverse execution synthesis.

One drawback of the proximity-guided search used by execution synthesis is that it does not learn from the paths that are proven to be infeasible. To illustrate this, assume that execution synthesis finds a path that reproduces the failure, but this path would be proven infeasible by the constraint solver. The next path picked by ESD's dynamic searcher would be the one with the best proximity to the goal (e.g., the failure location). However, if this path proves to be infeasible for a similar reason (e.g., a previous *if* branch was taken by both paths in the execution tree), ESD does not attempt to identify the reason. Instead, ESD is agnostic to the structure of the constraint that makes paths infeasible. A solution to this problem is to gain visibility into the constraint solver and identify the reason why a path is infeasible. This can be done using existing algorithms that compute the UNSAT core [84, 31] of a constraint. UNSAT cores are the minimal unsatisfiable subsets of a constraint solver query. The UNSAT core can provide ESD a way to transform an infeasible execution path into another execution path that is more likely to reach the coredump. Thus, leveraging the UNSAT core can help execution synthesis better search through the execution tree for a path that explains the coredump.

We see a clear opportunity in using ESD to weed out false positives generated by static analysis tools, such as race and deadlock checkers [44]. Static analysis is powerful and typically complete, but these properties come at the price of soundness: static analyzers commonly produce large numbers of false positives, and selecting the true positives becomes a laborious human-intensive task. Fortunately, the output of such tools is already similar to a bug report, so ESD could be used "out of the box" to validate each suspected bug: if ESD finds a path to the bug, then it is a true positive. We plan to explore in future work the synergy between such tools and our execution synthesis technique.

Existing hardware can help record some runtime information without any overhead. However, this information is not currently leveraged for debugging (i.e., it is not part of the coredump). For instance, the Last Branch Record (LBR) in Intel CPUs stores the source and destination addresses of the last 16 branches with virtually no overhead. LBR provides a precise execution suffix that can substantially trim the search space in RES. The length of the trace provided by LBR can be extended by configuring the hardware to filter information that can be easily inferred offline (e.g., LBR could filter taken conditional branches, and RES would use the CFG of the program to reverse engineer the taken conditional branches). One challenge is saving the LBR logs from all the cores to the coredump right when the failure occurs.

In future work, we plan to augment coredumps with the Last Branch Record (LBR) logs. We plan to identify other sources of readily available short-term log information available in hardware, in addition to LBR. LBR is a suitable source of information because it does not incur any performance penalty at runtime. If runtime overhead is acceptable, we could use Cyrus [63], a hardware solution for recording the traces of thread interleaving with low runtime overhead. However the hardware on which Cyrus is based on is not yet available in commodity CPUs.

Our current RES prototype is implemented on top of Cloud9 [22] and LLVM [79], however we plan to implement it on top of S^{2}E [29] in order to make it applicable to program binaries. The main challenge in implementing a prototype based on S^{2}E is loss of semantic information from LLVM to x86, which will limit the effectiveness of the static analysis used by RES.

A prototype based on S^{2}E would also enable RES to analyze the coredumps produced by a virtual machine monitor (VMM), such as VMWare [115]. Focusing on VMM coredumps has several advantages compared to other programs. First, the scale of the problem is smaller: the execution paths are short (on the order of a few microseconds at most, because the VMM is designed to run infrequently for short periods of time, handle privileged operations and then cease control of the execution to the virtual machine or the guest operating system), the call stacks in the coredumps are typically a few frames, and the address space is small (e.g., 4MB for the 32-bit version of VMWare Workstation). Second, vendors such as VMWare have a large collection of VMM coredumps [116], and most of them have already been manually assigned to known bugs, which would facilitate evaluating RES on coredumps from a real deployment.

Chapter 9

Conclusion

This thesis introduces execution synthesis, a technique for automatically debugging real software. Execution synthesis starts from a bug report and automatically synthesizes an execution that causes the bug to manifest. Developers can then deterministically play back this execution in their favorite debugger as many times as necessary to generate a fix. Execution synthesis requires no program modifications and no runtime tracing, thus introducing no runtime overhead.

The thesis presents a theoretical evaluation and an empirical evaluation of execution synthesis, showing how ESD, an embodiment of the execution synthesis technique, can reproduce, with no human intervention, concurrency bugs and crashes reported in real applications. It took less than three minutes to synthesize explanations for these bugs, which suggests ESD is practical for frequent use during development and debugging. To the best of our knowledge, ESD is the first tool that can automatically synthesize fully accurate executions that can be played back to reproduce bugs that occurred in the field, without incurring the overhead of execution tracing.

In order to automatically debug arbitrarily long executions—the main limitation of execution synthesis—this thesis introduced reverse execution synthesis. Like execution synthesis, reverse execution synthesis automates the debugging of failures that occur in production systems, without having to resort to runtime recording. Reverse execution synthesis takes as input a program and a coredump, and outputs the suffix of an execution that leads that program to that coredump. By reproducing just an execution suffix instead of the entire execution, reverse execution synthesis is suitable for arbitrarily long executions for which the root case and the failure are close to each other. Reverse execution synthesis could be used to improve a wide range of debugging-related tasks, such as automatic triaging of bug reports, identifying failures caused by hardware faults, and automating debugging processes that are human labor-intensive.

This thesis also shows how execution synthesis can be coupled with the lightweight recording of execution breadcrumbs in order to accurately diagnose data race bugs or to speed up deadlock debugging.

Bibliography

[1] GDB - reverse debugging. http://www.gnu.org/software/gdb/news/reversible.html.

[2] Ghttpd. http://gaztek.sourceforge.net/ghttpd.

[3] HawkNL. http://hawksoft.com/hawknl.

[4] Hypertable. http://www.hypertable.org.

[5] UndoDB. http://undo-software.com.

[6] VMware vSphere 4 fault tolerance: Architecture and performance. http://www.vmware.com/resources/techresources/10058.

[7] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[8] Tankut Akgul and Vincent J. Mooney III. Assembly instruction level reverse execution for debugging. *Trans. Softw. Eng. Methodol.*, 2004.

[9] Gautam Altekar and Ion Stoica. ODR: Output-deterministic replay for multicore programs. In *Symp. on Operating Systems Principles*, 2009.

[10] Gautam Altekar and Ion Stoica. Focus replay debugging effort on the control plane. In *Workshop on Hot Topics in Dependable Systems*, 2010.

[11] Lars Ole Andersen. Program analysis and specialization for the c programming language. Technical report, University of Copenhagen, 1994.

[12] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. AEG: Automatic exploit generation. In *Network and Distributed System Security Symposium*, 2011.

[13] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. In *Symp. on Operating Sys. Design and Implem.*, 2010.

[14] Domagoj Babić, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. Statically-directed dynamic automated test generation. In *Intl. Symp. on Software Testing and Analysis*, 2011.

[15] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *ACM EuroSys European Conf. on Computer Systems*, 2006.

[16] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. Deterministic process groups in dos. In *Symp. on Operating Sys. Design and Implem.*, 2010.

[17] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. Framework for instruction-level tracing and analysis of program executions. In *Intl. Conf. on Virtual Execution Environments*, 2006.

[18] Peter Boonstoppel, Cristian Cadar, and Dawson R. Engler. RWset: Attacking path explosion in constraint-based test generation. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[19] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A binary analysis platform. In *Intl. Conf. on Computer Aided Verification*, July 2011.

[20] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. 2008.

[21] David Brumley, Hao Wang, Somesh Jha, and Dawn Song. Creating vulnerability signatures using weakest preconditions. In *Computer Security Foundations Symp.*, 2007.

[22] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel symbolic execution for automated real-world software testing. In *ACM EuroSys European Conf. on Computer Systems*, 2011.

[23] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symp. on Operating Sys. Design and Implem.*, 2008.

[24] Miguel Castro, Manuel Costa, and Jean-Philippe Martin. Better bug reporting with better privacy. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2008.

[25] Satish Chandra, Stephen J. Fink, and Manu Sridharan. Snugglebug: a powerful approach to weakest preconditions. In *Intl. Conf. on Programming Language Design and Implem.*, 2009.

[26] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber. Bigtable: A distributed storage system for structured data. *Proc. 7th USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[27] Shimin Chen, Michael Kozuch, Theodoros Strigkos, Babak Falsafi, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Olatunji Ruwase, Michael Ryan, and Evangelos Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. In *Intl. Symp. on Computer Architecture*, 2008.

[28] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Partial replay of long-running applications. In *Symp. on the Foundations of Software Eng.*, 2011.

[29] Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, and George Candea. Selective symbolic execution. In *Workshop on Hot Topics in Dependable Systems*, 2009.

[30] Jim Chow, Tal Garfinkel, and Peter M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX Annual Technical Conf.*, 2008.

[31] Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. A simple and flexible way of computing small unsatisfiable cores in SAT modulo theories. In *Intln. Conf. on Theory and Applications of Satisfiability Testing*, 2007.

[32] Coreutils. http://www.gnu.org/software/coreutils/.

[33] Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. Bouncer: Securing software by blocking bad input. In *Symp. on Operating Systems Principles*, 2007.

[34] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-end containment of Internet worms. In *Symp. on Operating Systems Principles*, 2005.

[35] Olivier Crameri, Ricardo Bianchini, and Willy Zwaenepoel. Striking a new balance between program instrumentation and debugging time. In *ACM EuroSys European Conf. on Computer Systems*, 2011.

[36] Heming Cui, Jingyue Wu, John Gallagher, Huayang Guo, and Junfeng Yang. Efficient deterministic multithreading through schedule relaxation. In *Symp. on Operating Systems Principles*. ACM, 2011.

[37] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in polynomial time. In *Intl. Conf. on Programming Language Design and Implem.*, 2002.

[38] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. DMP: deterministic shared memory multiprocessing. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2009.

[39] Joseph Devietti, Jacob Nelson, Tom Bergan, Luis Ceze, and Dan Grossman. RCDC: a relaxed consistency deterministic computer. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2011.

[40] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[41] Isil Dillig, Thomas Dillig, and Alex Aiken. Sound, complete and scalable path-sensitive analysis. In *Intl. Conf. on Programming Language Design and Implem.*, 2008.

[42] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Symp. on Operating Sys. Design and Implem.*, 2002.

[43] George W. Dunlap, Dominic Lucchetti, Peter M. Chen, and Michael Fetterman. Execution replay on multiprocessor virtual machines. In *Intl. Conf. on Virtual Execution Environments*, 2008.

[44] Dawson Engler and Ken Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Symp. on Operating Systems Principles*, 2003.

[45] Dawson Engler and Daniel Dunbar. Under-constrained execution: making automatic code destruction easy and scalable. In *Intl. Symp. on Software Testing and Analysis*, 2007.

[46] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, February 2001.

[47] Brad Fitzpatrick. Memcached. http://memcached.org, 2013.

[48] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. *SIGPLAN Not.*, 2005.

[49] Jason Gait. A probe effect in concurrent programs. *Software: Practice and Experience*, 16(3), 1986.

[50] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: Global comprehension for distributed replay. In *Symp. on Networked Systems Design and Implem.*, 2007.

[51] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay debugging for distributed applications. In *USENIX Annual Technical Conf.*, 2006.

[52] Ghttpd Log Function Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/5960, 2010.

[53] Jeff Gilchrist. Parallel BZIP2. http://compression.ca/pbzip2, 2013.

[54] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. Debugging in the (very) large: ten years of implementation and experience. In *Symp. on Operating Systems Principles*, 2009.

[55] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Intl. Conf. on Programming Language Design and Implem.*, 2005.

[56] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security Symp.*, 2008.

[57] Patrice Godefroid and Nachiappan Nagappan. Concurrency at Microsoft – An exploratory survey. In *Intl. Conf. on Computer Aided Verification*, 2008.

[58] Google. Open-source multi-platform crash reporting system. https://code.google.com/p/google-breakpad.

[59] Brothers Grimm. *Hansel and Gretel*.

[60] 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 *Symp. on Operating Sys. Design and Implem.*, 2008.

[61] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Software verification with BLAST. In *Intl. SPIN Workshop*, 2003.

[62] G. J. Holzmann and D. Bosnacki. Multi-core model checking with SPIN. In *Intl. Parallel and Distributed Processing Symp.*, 2007.

[63] Nima Honarmand, Nathan Dautenhahn, Josep Torrellas, Samuel T. King, Gilles Pokam, and Cristiano Pereira. Cyrus: unintrusive application-level record-replay for replay parallelism. *SIGPLAN Notices*, 48(4), March 2013.

[64] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 2. 2011.

[65] Franjo Ivancic, Zijiang Yang, Malay K. Ganai, Aarti Gupta, Ilya Shlyakhter, and Pranav Ashar. F-soft: Software verification platform. In *Intl. Conf. on Computer Aided Verification*, 2005.

[66] Ali Jannesari and Walter F. Tichy. Identifying ad-hoc synchronization for enhanced race detection. In *Intl. Parallel and Distributed Processing Symp.*, 2010.

[67] Wei Jin and Alessandro Orso. Bugredux: reproducing field failures for in-house debugging. In *Intl. Conf. on Software Engineering*, 2012.

[68] Wei Jin and Alessandro Orso. F3: fault localization for field failures. In *Intl. Symp. on Software Testing and Analysis*, 2013.

[69] Sebastian Burckhardt John Erickson, Madanlal Musuvathi and Kirk Olynyk. Effective data-race detection for the kernel. In *Symp. on Operating Sys. Design and Implem.*, 2010.

[70] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *Intl. Conf. on Programming Language Design and Implem.*, 2009.

[71] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *Symp. on Operating Sys. Design and Implem.*, 2008.

[72] Baris Kasikci, Cristian Zamfir, and George Candea. Data races vs. data race bugs: Telling the difference with Portend. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2012.

[73] Baris Kasikci, Cristian Zamfir, and George Candea. RaceMob: Crowdsourced data race detection. In *Symp. on Operating Systems Principles*, 2013.

[74] James C. King. A new approach to program testing. In *Intl. Conf. on Reliable Software*, 1975.

[75] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX Annual Technical Conf.*, 2005.

[76] Donald E. Knuth. *The Art of Computer Programming*, volume III: Sorting and Searching. Addison-Wesley, 1998.

[77] Oren Laadan, Nicolas Viennot, and Jason Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. *ACM SIGMETRICS Conf.*, 38(1), June 2010.

[78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), 1978.

[79] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Intl. Symp. on Code Generation and Optimization*, 2004.

[80] Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Computers*, 1987.

[81] Dongyoon Lee, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Chimera: Hybrid program analysis for determinism. In *Intl. Conf. on Programming Language Design and Implem.*, 2012.

[82] Jeremias Rößler, Andreas Zeller, Gordon Fraser, Cristian Zamfir, and George Candea. Reconstructing core dumps. In *Intl. Conf. on Software Testing Verification and Validation*, 2013.

[83] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *Intl. Conf. on Programming Language Design and Implem.*, 2003.

[84] Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning*, 40(1), January 2008.

[85] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes – A comprehensive study on real world concurrency bug characteristics. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2008.

[86] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. PIN: building customized program analysis tools with dynamic instrumentation. In *Intl. Conf. on Programming Language Design and Implem.*, 2005.

[87] Roman Manevich, Manu Sridharan, Stephen Adams, Manuvir Das, and Zhe Yang. PSE: explaining program failures via postmortem static analysis. In *Symp. on the Foundations of Software Eng.*, 2004.

[88] Paul D. Marinescu, Radu Banabic, and George Candea. An extensible technique for high-precision testing of recovery code. In *USENIX Annual Technical Conf.*, 2010.

[89] Paul Dan Marinescu and Cristian Cadar. High-coverage symbolic patch testing. In *Intl. SPIN Workshop*, 2012.

[90] Cal McPherson. Ctrace. http://ctrace.sourceforge.net, 2012.

[91] Memcached issue 127, 2009. http://code.google.com/p/memcached/issues/detail?id=127.

[92] Microsoft. !exploitable crash analyzer - MSEC debugger extensions. http://msecdbg.codeplex.com/, 2013.

[93] Pablo Montesinos, Matthew Hicks, Samuel T. King, and Josep Torrellas. Capo: a software-hardware interface for practical deterministic multiprocessor replay. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2009.

[94] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *Symp. on Operating Sys. Design and Implem.*, 2008.

[95] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically classifying benign and harmful data races using replay analysis. *Intl. Conf. on Programming Language Design and Implem.*, 2007.

[96] National Institute of Standards and Technology. Secure hash standard (SHS). FIPS PUB 180-3, October 2008.

[97] Edmund B. Nightingale, John R. Douceur, and Vince Orgovan. Cycles, cells and platters: an empirical analysis of hardware failures on a million consumer PCs. In *ACM EuroSys European Conf. on Computer Systems*, 2011.

[98] Aditya V. Nori, Sriram K. Rajamani, Saideep Tetali, and Aditya V. Thakur. The Yogi project: Software property checking via static analysis and testing. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2009.

[99] Ohloh SQLite code analysis. https://www.ohloh.net/p/sqlite/analyses/467152, 2009.

[100] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: efficient deterministic multithreading in software. *SIGPLAN Not.*, 44(3), March 2009.

[101] Soyeon Park, Shan Lu, and Yuanyuan Zhou. Ctrigger: exposing atomicity violation bugs from their hiding places. In *ASPLOS*, 2009.

[102] Soyeon Park, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, Shan Lu, and Yuanyuan Zhou. Do you have to reproduce the bug at the first replay attempt? – PRES: Probabilistic replay with execution sketching on multiprocessors. In *Symp. on Operating Systems Principles*, 2009.

[103] David Ramos and Dawson Engler. Practical, low-effort equivalence verification of real code. In *Intl. Conf. on Computer Aided Verification*, 2011.

[104] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee, Jr. Enhancing server availability and security through failure-oblivious computing. In *Symp. on Operating Sys. Design and Implem.*, 2004.

[105] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4), 1997.

[106] Koushik Sen. Race directed random testing of concurrent programs. *Intl. Conf. on Programming Language Design and Implem.*, 2008.

[107] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A fast address sanity checker. In *USENIX Annual Technical Conf.*, 2012.

[108] SQLite. http://www.sqlite.org/, 2013.

[109] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: a lightweight extension for rollback and deterministic replay for software debugging. In *USENIX Annual Technical Conf.*, 2004.

[110] Kunal Taneja, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. eXpress: guided path exploration for efficient regression test generation. In *Intl. Symp. on Software Testing and Analysis*, 2011.

[111] Chen Tian, Vijay Nagarajan, Rajiv Gupta, and Sriraman Tallam. Dynamic recognition of synchronization operations for improved data race detection. In *Intl. Symp. on Software Testing and Analysis*, 2008.

[112] Nikolai Tillmann and Jonathan De Halleux. Pex – white box test generation for .NET. *Tests and Proofs*, 2008.

[113] Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou. Triage: diagnosing production run failures at the user's site. In *Symp. on Operating Systems Principles*, 2007.

[114] Valgrind, 2011. http://valgrind.org/.

[115] Vmware. http://www.vmware.org/.

[116] VMware, Inc. Collecting diagnostic information for VMware products (KB 1008524), 2013.

[117] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. RELAY: Static race detection on millions of lines of code. In *Symp. on the Foundations of Software Eng.*, 2007.

[118] Dasarath Weeratunge, Xiangyu Zhang, and Suresh Jagannathan. Analyzing multicore dumps to facilitate concurrency bug reproduction. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2010.

[119] Mark Weiser. Program slicing. In *Intl. Conf. on Software Engineering*, 1981.

[120] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: characterization and methodological considerations. *Intl. Symp. on Computer Architecture*, 1995.

[121] Tao Xie, Nikolai Tillmann, Peli de Halleux, and Wolfram Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Intl. Conf. on Dependable Systems and Networks*, 2009.

[122] Bin Xin, William N. Sumner, and Xiangyu Zhang. Efficient program execution indexing. In *Intl. Conf. on Programming Language Design and Implem.*, 2008.

[123] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad-hoc synchronization considered harmful. In *Symp. on Operating Sys. Design and Implem.*, 2010.

[124] Min Xu, Rastislav Bodik, and Mark D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Intl. Symp. on Computer Architecture*, 2003.

[125] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. Distributed dynamic partial order reduction based verification of threaded software. In *Intl. SPIN Workshop*, 2007.

[126] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. SherLog: error diagnosis by connecting clues from run-time logs. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2010.

[127] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, M. Michael Lee, Yuanyuan Zhou, and Stefan Savage. Be conservative: Enhancing failure diagnosis with proactive logging. In *Symp. on Operating Sys. Design and Implem.* USENIX Association, 2012.

[128] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. Characterising logging practices in open-source software. In *Intl. Conf. on Software Engineering*, June 2012.

[129] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving software diagnosability via log enhancement. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, March 2011.

[130] Anna Zaks and Rajeev Joshi. Verifying multi-threaded C programs with SPIN. In *Intl. SPIN Workshop*, 2008.

[131] Cristian Zamfir, Gautam Altekar, George Candea, and Ion Stoica. Debug determinism: The sweet spot for replay-based debugging. In *Workshop on Hot Topics in Operating Systems*, 2011.

[132] Cristian Zamfir, Gautam Altekar, George Candea, and Ion Stoica. Automating the debugging of datacenter applications with ADDA. In *Intl. Conf. on Dependable Systems and Networks*, 2013.

[133] Cristian Zamfir and George Candea. Execution synthesis: A technique for automated debugging. In *ACM EuroSys European Conf. on Computer Systems*, 2010.

[134] Cristian Zamfir and George Candea. Low-overhead bug fingerprinting for faster debugging. In *Intl. Conf. on Runtime Verification*, 2010.

[135] Cristian Zamfir, Baris Kasikci, Johannes Kinder, Edouard Bugnion, and George Candea. Automated debugging for arbitrarily long executions. In *Workshop on Hot Topics in Operating Systems*, 2013.

[136] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 2002.

[137] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. ConSeq: Detecting concurrency bugs through sequential errors. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2011.

[138] Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou, and Josep Torrellas. iWatcher: Efficient Architectural Support for Software Debugging. In *Intl. Symp. on Computer Architecture*, 2004.

Cristian Zamfir

PhD Candidate
Ecole Polytechnique Fédérale de Lausanne (EPFL)

EPFL - IC - DSLAB INN 319, Station 14	+41 (21) 693 81 89
1015 Lausanne	cristian.zamfir@epfl.ch
Switzerland	http://dslab.epfl.ch/people/zamfir

Research Interests

I am interested in dependable systems, with an emphasis on automated debugging. My main focus is building automated debuggers that start from a bug report and automatically synthesize an execution that reproduces the respective bug. I am also exploring ways to augment classic bug reports with small amounts of runtime information that can make the automated debugging process orders of magnitude faster.

I envision that automation is the key to reducing the manual effort and high cost of software development. My long term goal is to create the next-generation debuggers that are capable of fully automating the process of removing bugs from software. Next-generation debuggers will be able to automatically triage, reproduce, diagnose, and fix bugs, without compromising on the performance and user experience of the software.

Education

Ecole Polytechnique Fédérale de Lausanne (EPFL)
Ph.D. candidate in Computer Science 2008–present
Thesis: Execution Synthesis: A Technique for Automated Debugging
Advisor: Prof. George Candea

University of Glasgow
M.Sc. by Research in Computer Science 2006–2007
Thesis: Live Migration of User Environments Across Wide Area Networks
Advisors: Dr. Peter Dickman and Dr. Colin Perkins

University Politehnica of Bucharest (UPB)
B.Eng. in Electrical Engineering & Computer Science 2000–2005
Thesis: Security and Trust Management in Distributed Recommender Systems
Advisors: Prof. Wolfgang Nejdl and Prof. Valentin Cristea

Work Experience

Ecole Polytechnique Fédérale de Lausanne (EPFL) *Lausanne, Switzerland*
Research Assistant 2008–present
Developed execution synthesis, a technique for automated debugging, that can reproduce concurrency and memory safety bugs starting from just a bug report, in just a few minutes.

University of California, Berkeley *Berkeley, CA*
Visiting Researcher 2010–2011
Developed together with with Prof. Ion Stoica and Gautam Altekar a system for replay-debugging data center applications. We also defined and proposed ways to achieve "debug determinism", the sweet spot between runtime recording overhead and replay fidelity for replay debugging systems.

Microsoft Research *Cambridge, UK*
Research Intern (Systems & Networking Group) 2007
Developed and analyzed various parts of a background transfer service, a receiver-side controller designed to detect and reduce the interference of background traffic on normal priority TCP flows.

New Jersey Institute of Technology *Newark, NJ*
Research Assistant 2006
Developed a hybrid routing protocol for vehicular networks that aims to reduce routing overhead and improve TCP performance.

L3S Research Center *Hanover, Germany*
Research Intern 2005
Developed metrics for identifying shilling attacks in distributed recommender systems and an algorithm that prevents large groups of shilling attackers from interfering with a recommender system.

National Center for Information Technology *Bucharest, Romania*
Research Assistant 2003–2005
Developed an efficient electronic cash platform for smartphones based on Merkle Hash Trees and symmetric cryptography.
Developed an autonomous robot for motion detection-based video surveillance of remote sites.
Implemented the IMS QTI specification for the UPB E-LEARNING platform.

Professional Service

Member of Program Commmittees

BigDataCloud – Intl. Workshop on Big Data Management in Clouds	2013
JSEP – Journal of Software: Evolution and Process	2012
CloudCP – Intl. Workshop on Cloud Computing Platforms	2012
SFMA – Intl. Workshop on Systems for Future Multi-core Architectures	2012
CloudCP – Intl. Workshop on Cloud Computing Platforms	2011
NBiS – Intl. Conference on Network-Based Information Systems	2011
DOLAP – ACM Intl. Workshop On Data Warehousing and OLAP	2010
3PGCIC – Intl. Conference on P2P, Parallel, Grid, Cloud and Internet Computing	2010

ShadowPC Committee Member

EuroSys–ACM SIGOPS/EuroSys European Conference on Computer Systems	2010

External Reviewer

CIDR – Conference on Innovative Data Systems Research	2013
SOCC – ACM Symposium on Cloud Computing	2012
EuroSys – ACM SIGOPS/EuroSys European Conference on Computer Systems	2009, 2011, 2012
SOSP – Symposium on Operating Systems Principles	2011
USENIX Annual Technical Conference	2011
SPIN – Intl. SPIN Workshop on Model Checking of Software	2011
ASPLOS – ACM Conference on Architectural Support for Programming Languages and Operating Systems	2010

Peer-reviewed Conference and Journal Publications

[1] RaceMob: Crowdsourced Data Race Detection. Baris Kasikci, Cristian Zamfir, and George Candea. *Symp. on Operating Systems Principles*, Farmington, PA, November 2013.

[2] Automating the Debugging of Datacenter Applications with ADDA. Cristian Zamfir, Gautam Altekar, George Candea, and Ion Stoica. *Intl. Conf. on Dependable Systems and Networks*, Budapest, June 2013.

[3] Reconstructing Core Dumps. Jeremias Rößler, Andreas Zeller, Gordon Fraser, Cristian Zamfir, and George Candea. *Intl. Conf. on Software Testing Verification and Validation (ICST)*, Luxembourg, March 2013.

[4] Data Races vs. Data Race Bugs: Telling the Difference with Portend. Baris Kasikci, Cristian Zamfir, and George Candea. *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, London, UK, March 2012.

[5] Parallel Symbolic Execution for Automated Real-World Software Testing. Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. *ACM EuroSys European Conf. on Computer Systems (EUROSYS)*, Salzburg, Austria, April 2011.

[6] Low-Overhead Bug Fingerprinting for Faster Debugging. Cristian Zamfir and George Candea. *Intl. Conf. on Runtime Verification (RV)*, Malta, November 2010.

[7] Automated Software Testing as a Service. George Candea, Stefan Bucur, and Cristian Zamfir. *Symp. on Cloud Computing (SOCC)*, Indianapolis, IN, June 2010.

[8] Execution Synthesis: A Technique for Automated Debugging. Cristian Zamfir and George Candea. *ACM EuroSys European Conf. on Computer Systems (EUROSYS)*, Paris, France, April 2010.

[9] Deadlock Immunity: Enabling Systems To Defend Against Deadlocks. Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea. *Symp. on Operating Sys. Design and Implem. (OSDI)*, San Diego, CA, December 2008.

[10] An Efficient Electronic Cash Platform for Smart Phones. Cristian Zamfir, Ionut Constandache, Andrei Damian, and Valentin Cristea. *E-COMM-LINE International Conference*, Bucharest, Romania, 2004.

[11] Implementing the IMS Question and Test Interoperability Specification: An Architectural Overview. Cristian Zamfir, Ionut Constandache, and Octavian Udrea. *E-COMM-LINE International Conference*, Bucharest, Romania, 2003.

Peer-reviewed Workshop Publications

[12] Automated Debugging for Arbitrarily Long Executions. Cristian Zamfir, Baris Kasikci, Johannes Kinder, Edouard Bugionon, and George Candea. *Workshop on Hot Topics in Operating Systems (HOTOS)*, Santa Ana Pueblo, NM, May 2013.

[13] CoRD: A Collaborative Framework for Distributed Data Race Detection. Baris Kasikci, Cristian Zamfir, and George Candea. *Workshop on Hot Topics in Dependable Systems (HOTDEP)*, Hollywood, CA, October 2012.

[14] Debug Determinism: The Sweet Spot for Replay-based Debugging. Cristian Zamfir, Gautam Altekar, George Candea, and Ion Stoica. *Workshop on Hot Topics in Operating Systems (HOTOS)*, Napa, CA, May 2011.

[15] Cloud9: A Software Testing Service. Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. *Workshop on Large Scale Distributed Systems and Middleware*, Big Sky, MT, October 2009.

[16] Selective Symbolic Execution. Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, and George Candea. *Workshop on Hot Topics in Dependable Systems (HOTDEP)*, Cascais, Portugal, June 2009.

[17] Preventing Shilling Attacks in Online Recommender Systems. Paul-Alexandru Chirita, Wolfgang Nejdl, and Cristian Zamfir. *ACM International Workshop on Web Information and Data Management (WIDM)*, Bremen, Germany, 2005.

Invited Talks

1. "Classifying Data Races with Portend" at *Microsoft ICES Workshop, Cambridge, UK*, 2012
2. "Classifying Data Races with Portend" at *Imperial College, London, UK*, 2012
3. "Execution Synthesis: A technique for Automated Debugging" at *U.C. Berkeley, CA*, 2010
4. "Execution Synthesis: A technique for Automated Debugging" at *Intel Research, Berkeley, CA*, 2010

Talks at Peer-Reviewed Conferences and Workshops

1. "Debug Determinism: The Sweet Spot for Replay-based Debugging" at the *Workshop on Hot Topics in Operating Systems (HotOS)*, Napa, CA, 2011.
2. "Execution Synthesis: A Technique for Automated Debugging" at the *European Conf. on Computer Systems (EuroSys)*, Paris, France, 2010.
3. "Low-Overhead Bug Fingerprinting for Faster Debugging" at the *Intl. Conf. on Runtime Verification (RV)*, Malta, 2010.
4. "Selective Symbolic Execution" at the *Workshop on Hot Topics in Dependable Systems (HotDep)*, Cascais, Portugal, 2009.
5. "An Efficient Electronic Cash Platform for Smart Phones" at the *E-COMM-LINE International Conference*, Bucharest, Romania, 2004.

Patents and Patent Applications

- "Automatic Generation of Program Execution that Reaches a Given Failure Point", Cristian Zamfir and George Candea, *US PTO #13/304,041* (pending)
- "Parallel automated testing platform for software systems", Stefan Bucur, George Candea, and Cristian Zamfir, *US PTO #61/430,191* (pending)
- "Low-Overhead Deterministic Replay for Datacenter Applications", Gautam Altekar, Cristian Zamfir, George Candea, and Ion Stoica (pending)

Awards

Intel Doctoral Student Honor Programme – recipient of $35,000 Intel award	2012
Faculty Scholarship – University of Glasgow	2006
Overseas Research Scholarship – University of Glasgow	2006
Merit Scholarship (top 5% of my class) – University Politehnica of Bucharest	2001–2005
2nd prize in UPB Annual Project Competition – Advanced Remote Surveillance System	2004
3rd prize in the National Mathematics Olympiad – Romania	2000
2nd prize in the National Mathematics Olympiad – Romania	1999
1st prize in National Mathematics Summer Camp – Romania	1998
Silver Medal in International Mathematics Project Competition – Turkey	1998
1st prize in the National Mathematics Olympiad – Romania	1996
2nd prize in the National Mathematics Olympiad – Romania	1995

Teaching

Software Engineering (3rd year undergraduate level at EPFL)
Teaching assistant 2008–2012

Software Systems for Computer Networks (5th year undergraduate level at UPB)
Teaching assistant 2005

Co-supervised M.Sc. thesis (Hanover University) 2005
Title: "Distributed Recommender Systems", student Jurgen Belizki

Co-supervised M.Sc. thesis (Masaryk University) 2012
Title: "Execution Synthesis: Formal Description and Core Dump Interface", student Martin Milata

Additional Qualifications

Venture Leaders Entrepreneurial training	*Boston* 2013
Venture Challenge Entrepreneurial training	*Lausanne* 2012
4th International School on Foundations of Security Analysis and Design	2004
Cisco Networks – taken 2 semesters of CCNA courses	2003
IBM WebSphere Application Server Development Workshop	2003

Undergraduate Academic Projects

Device Drivers 2004
Linux and Windows device drivers for serial port, firewall, and virtual disk drive.

Robot online searching in star-shaped polygons. 2002

Miscellaneous

Interests

Playing guitar and singing
Photography
Rock climbing
Table tennis
Cycling

Languages

English (fluent)
Spanish (fair)
Romanian (mother tongue)
French (beginner)