Automated Debugging for Arbitrarily Long Executions

Cristian Zamfir, Baris Kasikci, Johannes Kinder, Edouard Bugnion, George Candea
School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Abstract

One of the most energy-draining and frustrating parts of software development is playing detective with elusive bugs. In this paper we argue that automated post-mortem debugging of failures is feasible for real, in-production systems with no runtime recording. We propose *reverse execution synthesis* (RES), a technique that takes a coredump obtained after a failure and automatically computes the *suffix* of an execution that leads to that coredump. RES provides a way to then play back this suffix in a debugger deterministically, over and over again. We argue that the RES approach could be used to (1) automatically classify bug reports based on their root cause, (2) automatically identify coredumps for which hardware errors (e.g., bad memory), not software bugs are to blame, and (3) ultimately help developers reproduce the root cause of the failure in order to debug it.

1 Introduction and Motivation

Debugging software deployed in the real world is hard, frustrating, and typically requires deep knowledge of the code. Bug reports rarely provide sufficient information, so developers must turn into detectives in search of an explanation of how the program could have reached the reported failure state. It would be great if developers had a better way to triage, analyze, and debug these failures.

One way to do this is deterministic record-replay: 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 [2, 14, 15, 17, 19].

Record-replay systems, however, are not ideal, mainly because of performance and storage overheads. For example, making a multi-threaded execution on a multi-core 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 [14] and 60% for ODR [2], even for a 2-core CPU). Several systems choose to trade some of the reproducibility guarantees for lower runtime overhead [2, 5, 23], but this trade-off hurts their utility for debugging [27]. When building a record-replay system for datacenter applications [28], a big challenge is that they 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), we believe such performance and storage overheads make record-replay impractical for debugging failures in production systems.

Another option would be to use deterministic execution systems [3, 4, 11, 12], but they too are prohibitively heavyweight, especially for multi-CPU systems.

We set 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, our 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, thus enabling developers to home in on the root cause and fix it. This tool would essentially automate what developers do manually today.

A fundamental challenge 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 [27]. The execution synthesis technique [29] we proposed in the past 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 symbolic execution [9] on 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 to synthesize an execution all the way from the start of the execution to the end failure state.

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 [27].

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 [20]), 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 a minidump, which makes RES strictly more powerful.

We now describe the technique in more detail (§2) and present three possible use cases (§3): automatic classification of bug reports, automatic identification of failures likely caused by hardware errors (such as memory bit flips or CPU bugs), and helping developers debug the failed program.

2 Design of Reverse Execution Synthesis

We need a tool that, for a given program P, can use a coredump C to generate a suffix of a feasible execution E that causes program P to produce coredump C. 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 C; and (6) no modifications are to be made to P. This would make the tool indeed useful for debugging failures that occur in real-world production systems. Since it is predicated on the presence of a coredump, this tool would work for failures whose state can be snapshotted in a coredump (e.g., crashes, deadlocks). Our current design for RES meets requirements (1), (2), (5), (6), and aims to satisfy but cannot always guarantee (3) and (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 C—any execution that reproduces the same root cause and failure is sufficient [27]. Second, we assume that the root cause is located fairly close to the failure (e.g., 85% of the bugs analyzed in [30] were executed just a few instructions before the failure), so we expect a short execution suffix to suffice for debugging.

2.1 What Are the Inputs and Outputs?

Inputs: As suggested above, RES takes in the coredump C 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 [16, 25]. In addition to C, RES takes in the program source code P_S, which should be available to developers. Thus, the input is $< C, P_S >$.

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 partial memory image M_i (§2.3) 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 compatible with 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.

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.

2.2 The Challenge of Inferring the Past

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 [1], 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 [20, 26] or derive weakest preconditions [7, 10] for given vulnerabilities. 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.

2.3 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, where the

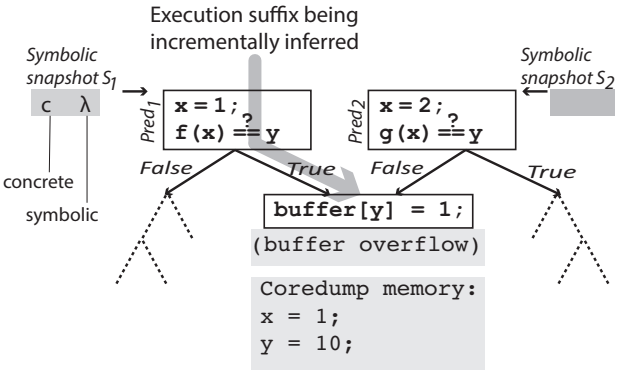

Figure 1: Simplified example illustrating the basics of RES on a program that crashed due to a buffer overflow. RES creates symbolic snapshots S_1 and S_2 that correspond to program state just prior to each possible predecessor basic block. Since $x = 1$ in the coredump, and only $Pred_1$ ever sets x 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).

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 approaches.

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 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 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 [9]). 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 currently unknown, symbolic values. The program counter of a symbolic snapshot is set to the entry point of the corresponding predecessor basic block.

2.4 Reconstructing Program State

A symbolic snapshot S_{pre} can be thought of as an over-approximation 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 over-approximation 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 C, 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. 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. When encountering a *memory read* instruction in B, 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.

Due to space constraints, we omit our preliminary ideas on how to reconstruct thread schedules, how to execute reads and writes with symbolic addresses (pointers), and how to handle function pointers.

Execution breadcrumbs: RES can benefit from coredumps augmented with runtime information that is cheap to collect after the crash. For instance, existing error logs can provide RES with useful, coarse-grained "breadcrumbs" of the execution trace. Another example is the Last Branch Record (LBR) in Intel CPUs, which 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).

3 Use Cases

We now present several use cases where employing reverse execution synthesis can help.

3.1 Triaging Bug Reports

Debugging in the large is hard, because the number of deployed systems is big, and the sheer volume of bug reports can be overwhelming [16]. In this context, accurately and automatically prioritizing reports from millions of users is particularly difficult yet crucial in cutting down the development 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) [16]. Despite proving its utility in over ten years of operation, WER relies on ad-hoc heuristics and the law of large numbers. For instance, WER uses heuristics such as deprioritizing 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 [16].

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. 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.

3.2 Failures Caused by Hardware Errors

Hardware errors are common, correlated, and recurrent [22]. Machines that crash once due to a hardware error are two orders of magnitude more likely to crash a second time [22]. Moreover, hardware errors generate noise, and 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. 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 [22] 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.

3.3 Debugging

RES enables several debugging aids on top of traditional debuggers like gdb: synthesizing the execution suffix, reconstructing past state (the symbolic snapshots), and the ability to do reverse debugging without the need to record the execution. Moreover, since it computes the read and write sets of the execution suffix, RES 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?"

Since RES reproduces the coredump, it is not restricted to a particular type of bugs—even semantic bugs (e.g., captured by assert statements) can be reproduced.

4 Preliminary Prototype

We are in the early stages of implementing a prototype of RES for LLVM [18] binaries (e.g., generated from C/C++ source code). RES supports multi-threaded programs and is implemented on top of the Cloud9 [8] symbolic execution engine. Currently, RES assumes sequential memory consistency when synthesizing execution suffixes, but we plan to lift this limitation.

We evaluated RES on three synthetic concurrency bugs. 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.

5 Related Work

!exploitable [21] is a debugging tool that assigns exploitability ratings to crashes. *!exploitable* uses heuristics, and unfortunately this can lead to both false positives and false negatives. By providing an execution suffix, RES can improve the accuracy of this classification.

In some sense, RES is like computing weakest preconditions [13] 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 [7, 10] do not work for concurrent programs, do not leverage the coredump, and assume some level of recording [7]. The full use of the coredump, the accurate memory handling, and the support for concurrent programs are RES's key differentiators from work on weakest precondition computation.

RES's approach to executing symbolic snapshots was inspired by UC-KLEE [24], which uses under-constrained execution for equivalence checking.

6 Known Challenges

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 [6].

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.

RES may not always identify the exact root cause that led to the observed failure, therefore it may not offer *debug determinism* [27]. However, RES's accuracy promises to be good, mainly owing to the fact that any execution suffix must match the full coredump exactly. Typically, 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; after fixing it, RES can be run again to identify the other root cause, and so on until all root causes are fixed.

RES does not currently handle control flow through invalid pointers and memory or stack corruption, because these cases may cause the CFG of the program to also be corrupted, and the current RES prototype requires an accurate CFG and stack.

7 Conclusion

We argued that it is conceivable to automate the debugging of failures that occur in production systems, without having to resort to runtime recording. We proposed an initial design for reverse execution synthesis, which takes as input a program and a coredump, and outputs the suffix of an execution that leads that program to that coredump. With this approach we hope 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.

References

[1] T. Akgul and V. J. Mooney III. Assembly instruction level reverse execution for debugging. *Trans. Softw. Eng. Methodol.*, 2004.

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

[3] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *Symp. on Operating Sys. Design and Implem.*, 2010.

[4] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dos. In *Symp. on Operating Sys. Design and Implem.*, 2010.

[5] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *Intl. Conf. on Virtual Execution Environments*, 2006.

[6] P. Boonstoppel, C. Cadar, and D. 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.

[7] D. Brumley, H. Wang, S. Jha, and D. Song. Creating vulnerability signatures using weakest preconditions. In *Computer Security Foundations Symp.*, 2007.

[8] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel symbolic execution for automated real-world software testing. In *ACM EuroSys European Conf. on Computer Systems*, 2011.

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

[10] S. Chandra, S. J. Fink, and M. Sridharan. Snugglebug: a powerful approach to weakest preconditions. In *Intl. Conf. on Programming Language Design and Implem.*, 2009.

[11] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: deterministic shared memory multiprocessing. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2009.

[12] J. Devietti, J. Nelson, T. Bergan, L. Ceze, and D. Grossman. RCDC: a relaxed consistency deterministic computer. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2011.

[13] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[14] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Symp. on Operating Sys. Design and Implem.*, 2002.

[15] G. W. Dunlap, D. Lucchetti, P. M. Chen, and M. Fetterman. Execution replay on multiprocessor virtual machines. In *Intl. Conf. on Virtual Execution Environments*, 2008.

[16] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: ten years of implementation and experience. In *Symp. on Operating Systems Principles*, 2009.

[17] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. *ACM SIGMETRICS Conf.*, 38(1), June 2010.

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

[19] D. Lee, P. M. Chen, J. Flinn, and S. Narayanasamy. Chimera: hybrid program analysis for determinism. Intl. Conf. on Programming Language Design and Implem., 2012.

[20] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang. PSE: explaining program failures via postmortem static analysis. In *Symp. on the Foundations of Software Eng.*, 2004.

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

[22] E. B. Nightingale, J. R. Douceur, and V. 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.

[23] S. Park, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, S. Lu, and Y. 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.

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

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

[26] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. SherLog: error diagnosis by connecting clues from run-time logs. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2010.

[27] C. Zamfir, G. Altekar, G. Candea, and I. Stoica. Debug determinism: The sweet spot for replay-based debugging. In *Workshop on Hot Topics in Operating Systems*, 2011.

[28] C. Zamfir, G. Altekar, G. Candea, and I. Stoica. Automating the debugging of datacenter applications with ADDA. In *Intl. Conf. on Dependable Systems and Networks*, 2013.

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

[30] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps. ConSeq: Detecting concurrency bugs through sequential errors. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2011.