Execution Reconstruction: Harnessing Failure Reoccurrences for Failure Reproduction

Gefei Zuo
gefeizuo@umich.edu
University of Michigan, USA

Jiacheng Ma
jcma@umich.edu
University of Michigan, USA

Andrew Quinn
arquinn@umich.edu
University of Michigan, USA

Pramod Bhatotia
pramod.bhatotia@in.tum.de
TU Munich, Germany

Pedro Fonseca
pfonseca@purdue.edu
Purdue University, USA

Baris Kasikci
barisk@umich.edu
University of Michigan, USA

Abstract

Reproducing production failures is crucial for software reliability. Alas, existing bug reproduction approaches are not suitable for production systems because they are not simultaneously efficient, effective, and accurate. In this work, we survey prior techniques and show that existing approaches over-prioritize a subset of these properties, and sacrifice the remaining ones. As a result, existing tools do not enable the plethora of proposed failure reproduction use-cases (e.g., debugging, security forensics, fuzzing) for production failures.

We propose Execution Reconstruction (ER), a technique that strikes a better balance between efficiency, effectiveness and accuracy for reproducing production failures. ER uses hardware-assisted control and data tracing to shepherd symbolic execution and reproduce failures. ER's key novelty lies in identifying data values that are both inexpensive to monitor and useful for eliding the scalability limitations of symbolic execution. ER harnesses failure reoccurrences by iteratively performing tracing and symbolic execution, which reduces runtime overhead. Whereas prior production-grade techniques can only reproduce short executions, ER can reproduce any reoccuring failure. Thus, unlike existing tools, ER reproduces fully replayable executions that can power a variety of debugging and reliabilty use cases. ER incurs on average 0.3% (up to 1.1%) runtime monitoring overhead for a broad range of real-world systems, making it practical for real-world deployment.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06…$15.00
https://doi.org/10.1145/3453483.3454101

Keywords: debugging, symbolic execution

ACM Reference Format:
Gefei Zuo, Jiacheng Ma, Andrew Quinn, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. 2021. Execution Reconstruction: Harnessing Failure Reoccurrences for Failure Reproduction. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3453483.3454101

1 Introduction

Failure reproduction is critical for software reliability. Not only is failure reproduction the first step in effective debugging [5, 21, 87, 89, 99], it also enables tools across many software reliability domains including security forensics [49], configuration management [34], and directed testing and fuzzing [42]. These tools are particularly valuable when used on *production failures*, because such failures may be elusive and difficult to replicate in house, outside of specific deployment scenarios. For instance, eliminating a production bug can drastically improve availability [8–10, 61], or a security audit of a production breach can help with leak assessment [97, 102, 112].

Many techniques automate the process of reproducing failures in software, yet, none offer comprehensive support for production usage. Prior systems sacrifice at least one of three key properties that are necessary for failure reproduction in production settings: (1) *efficiency*, which relates to the amount of resources expended to reproduce failures, (2) *effectiveness*, which relates to the capability to reproduce different kinds of bugs (e.g., concurrency bugs [78] and bugs where the failure and root cause are distant [54]), and (3) *accuracy*, which relates to whether the control flow and the data values of an execution can be recovered correctly. Our motivating study (§2) reveals that existing bug reproduction techniques over-prioritize a subset of the efficiency, effectiveness and accuracy properties and thus over-sacrifice in the other properties. Consequently, when used in production systems, existing techniques cannot enable the plethora of use cases (e.g., debugging, security forensics, fuzzing, etc.) that are otherwise available.

For instance, record/replay systems [32, 63–65, 82, 88, 92, 95, 98, 108] can accurately reproduce any execution and are thus very effective in reproducing bugs, however, they incur high overheads (e.g., up to 2× for a state-of-the-art system [108]), making them unsuitable for production use cases. Many techniques reduce runtime overhead by reproducing failures using a combination of online (runtime) recording and offline program analysis [47, 55, 93, 111, 113, 117]. For instance, REPT [111] and POMP [113] use hardware tracing to efficiently record a program's control flow at runtime and recover the program's data values via static analysis. However, since programs overwrite data values frequently, these tools cannot accurately reproduce full executions with only control-flow information and the final program state. It is difficult to debug programs with unreliable control flow and data information, and, even worse, these systems cannot be used in many domains including security forensics, directed testing and fuzzing, and configuration management.

We present Execution Reconstruction (ER), a hybrid offline/online failure reproduction technique that explores a sweet spot in the efficiency-effectiveness-accuracy trade-off space. Compared to prior work, ER's main contribution lies in determining a subset of the execution information that is both inexpensive to monitor online and useful for failure reproduction. ER is efficient enough to be deployed in production (i.e., it incurs on average 0.3% runtime overhead); effective in reproducing different kinds of bugs; and accurate in reproducing data values and control flow that lead to the original failure, which enables a variety of debugging and analysis use cases for production scenarios.

At the heart of ER lies *shepherded symbolic execution*, which uses dynamic control flow and data value information to reconstruct an execution that leads to a production failure. Rather than exploring exponentially many control-flow paths, ER follows a single recorded path that leads to a failure, thereby eliminating the notorious path explosion problem of symbolic execution. Symbolic execution computes constraints on program inputs that lead to the failure. Once ER completes symbolic execution along the entire path that was recorded during the failing production run, it invokes a constraint solver to compute failure-inducing program inputs.

Unfortunately, for real-world programs, eliminating path explosion is not sufficient to reconstruct an execution via symbolic execution. The constraints on input values grow very complex, which prevents symbolic execution from making progress (i.e., symbolic execution stalls) and eventually causes constraint solving to timeout (see § 4 for how stalls occur in practice). Our evaluation shows that shepherded symbolic execution using only a control-flow trace stalls on 11/13 failures (§ 5). ER overcomes this challenge by carefully selecting and recording a minimal set of data values that simplify constraint solving and thus eliminate slowdowns in symbolic execution without rendering ER too inefficient for production use.

Ideally, ER would select the data values that optimally simplify constraint solving, but this is intractable. Instead, ER turns to a novel heuristic to identify data values that drastically simplify constraint solving in practice. ER builds and analyzes a constraint graph during symbolic execution to determine *key data values* (e.g., instruction operands, return values, addresses, parts of the input), which are involved in complex constraint dependencies. In particular, ER identifies two patterns in the constraint graph that often cause constraint solver timeouts: long chains of symbolic writes, in which each write in the chain is dependent upon a previous write in the chain, and accesses to large symbolic memory objects, which can be difficult to reason about, as many memory locations may be accessed. If known, the data values from these patterns substantially simplify constraint solving. In our evaluation, the selection heuristic identifies a small set of data values that simplify constraint solving such that ER can reconstruct failures with negligible overhead.

Anticipating the key data values and recording them before a failure is difficult. Key data values depend on the constraints in symbolic execution, which ER cannot predict without symbolically executing the program. Thus, to reproduce a production failure, ER introduces a novel iterative algorithm that leverages frequent failure reoccurrences in large scale deployments, similar to existing production monitoring systems [58, 71–73]. In each iteration, ER gathers more data about the reoccurring failure, uses symbolic execution to determine constraints of the failing execution, selects additional data values that can help symbolic execution complete, and collects more data from programs in the next iteration. To bootstrap this process, ER uses always-on control-flow recording. When symbolic execution completes, ER determines program inputs that lead to the original failure and stops iterating. Using this iterative process, ER guarantees reconstruction of an entire failing execution for any reoccuring failure. In certain cases (i.e., 2/13 cases in our evaluation), ER can reproduce a failing execution even after a single occurrence of a failure. In practice, ER reproduces failures in only a few occurrences (only requiring an average of ~3.5 occurrences in our evaluation).

ER resides at the sweet spot in the efficiency-effectiveness-accuracy trade-off. By using hardware tracing, ER is *efficient* enough for production use, as it incurs on average 0.3% (up to 1.1%) runtime overhead during online monitoring. By using the iterative approach to shepherded symbolic execution, ER is *effective* in reproducing failures in long-running real-world program executions. Finally, by using shepherded symbolic execution, ER is *accurate* enough for a plethora of software reliability use cases (debugging, security forensics, fuzzing, etc.). ER generates program inputs that are guaranteed to lead to the original production failure. While ER-generated inputs may differ from actual inputs that led to the production failure, we empirically validate ER is useful for debugging

and other software reliability use cases, including a case-study that uses likely program invariant computation to perform automated failure localization.

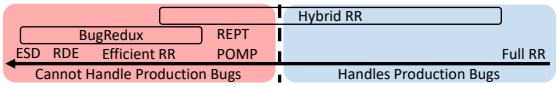
To summarize, we propose Execution Reconstruction (ER), an end-to-end technique for reproducing production failures and make the following contributions:

- *Shepherded Symbolic Execution*, which follows a control flow and data value trace recorded in production to eschew path explosion and solver stalls that hinder traditional symbolic execution.
- *Key Data Value Selection*, which identifies data values that drastically speed up constraint solving and can be monitored efficiently with existing hardware support.
- An evaluation of ER on 13 bugs in real systems (e.g., memcached, SQLite, python, etc.) shows that ER is efficient (on average 0.3% and up to 1.1% runtime overhead), effective, and accurate enough for production
- An invariant-based failure localization case-study that shows that ER can enable production use of software reliability tools.

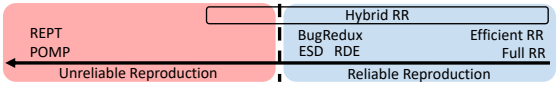
2 Motivation

Ideally, production-grade failure reproduction systems would achieve three key properties: (1) *efficiency*, which is the ability to incur low overhead and use few resources, (2) *effectiveness*, which is the ability to reproduce different kinds of failures (e.g., due to concurrency bugs or latent bugs where the failure (e.g., a crash) and the bug (e.g., an overflow) are distant), (3) *accuracy*, which is the ability to recover the control flow (branches executed) and data (values read/written) of the failing execution correctly. Unfortunately, all known approaches that achieve effectiveness and accuracy on multi-threaded applications (e.g., record/replay systems) record detailed execution information, which imposes high overhead. As a result, practical failure reproduction systems for production must trade-off one or more desirable properties.

(a) Efficiency. Systems on right have production-grade overhead.



(b) Effectiveness. Systems on right reproduce all production bugs.



(c) Accuracy. Systems on right reliably reproduce failures.

Figure 1. Prior techniques on a spectrum for each reproduction property. RR stands for record /replay. Hybrid RR and BugRedux are shown as boxes spanning a property range.

We study prior failure reproduction systems and place them on a spectrum for each failure reproduction property (see Fig. 1). For each property, we identify a *usability-boundary* (shown as a vertical dotted-line in Fig. 1), an inflection point at which a failure reproduction system shifts from an undesirable trade-off to an acceptable one. While many systems are usable in one or two properties, no existing failure reproduction system is simultaneously efficient enough for production, able to handle common production bugs, and able to reliably reproduce an execution. Thus, no existing failure reproduction system provides *production* support for the powerful software reliability tools that have been developed for debugging, security forensics, fuzzing, etc. Below we place systems onto each spectrum and construct a usability-boundary for each property.

2.1 Efficiency

Efficiency, or low runtime performance overhead, is paramount for a production-grade system. A few failure reproduction techniques maximize efficiency by using an offline analysis. For instance, ESD [114, 115] takes as input a failure (e.g., a deadlock) location, statically computes intermediate statements that the program must execute before the failure (e.g., locks before a deadlock), and steers symbolic execution towards intermediate statements and then towards the failure. Similarly, RDE [109] imposes negligible overhead because it uses existing or light-weight data from the production environment (application logs, system call sequences) to guide symbolic execution towards a failure-inducing input.

Purely offline approaches struggle to achieve effectiveness (see § 2.2), so many systems record execution information at the cost of efficiency. Record/replay systems [29, 63–65, 81, 82, 86, 88, 95, 98, 108] record all program inputs and other non-deterministic events. Full record/replay systems [29, 108] record all inputs and all non-deterministic events resulting in high overhead that is unsuitable for production usage (up to 2× for a state-of-the-art system [108]); efficient record/replay systems [49, 81, 86] record fewer events but are less effective than full record/replay systems (i.e., they cannot handle data races, see § 2.2).

Hybrid online/offline techniques provide a middle-ground; they achieve better effectiveness than offline techniques by recording some execution information, while achieving better efficiency than record/replay by offloading analysis to offline processing. BugRedux [69] uses symbolic execution to synthesize an execution that reaches the locations traced by one of two approaches: complete tracing records all control flow (up to 10× overhead), or the more efficient call sequence tracing, which records each function call (between 2%–50% overhead). Similarly, hybrid record/replay techniques (e.g., PRES [93] and ODR [32]) search for an execution that reaches execution states monitored using recording granularities that occupy different points in the efficiency-effectiveness-accuracy trade-off space: the finest-grained tracing records

full program traces and has high overhead (3×-20×) but good effectiveness/accuracy, while coarse-grained modes impose lower overhead by only recording non-deterministic input and synchronization operations, but have poor effectiveness/accuracy. In Fig. 1, we plot ranges for BugRedux and hybrid record/replay systems.

Recent hybrid approaches, REPT [111] and POMP [113], use hardware-assisted tracing (e.g., Intel Processor Trace (PT) [67]) to record control-flow information with low overhead (below 5% [71, 111]). These approaches use static analyses that perform reverse and/or forward execution along the trace to reconstruct program state.

To identify a usability boundary, we use the analysis of always-on record/replay systems [49, 86], which suggest that run-time overhead of 10% or less is acceptable in production. We consider offline, recent hybrid approaches (REPT and POMP), and efficient record/replay systems to be efficient-enough for production use. Full record/replay and BugRedux are too inefficient for production use. Finally, hybrid record/replay systems are efficient-enough when using coarse-grained recording, but are too inefficient when using fine-grained recording. Fig. 1b depicts this spectrum.

2.2 Effectiveness

A failure reproduction system is only useful if it can effectively reproduce common production failures. Full record/replay systems [29, 108] are maximally effective since they can reproduce any error in any application. Efficient record/replay systems [49, 81, 86] have unacceptable effectiveness for production failure reproduction since they cannot replay executions that contain data-races, an important class of production bugs [54, 99]. REPT has limited effectiveness since it does not guarantee that it can reconstruct data for arbitrary failures. In particular, REPT can only reconstruct short execution fragments (15-60% of the data values are incorrectly recovered for traces longer than 100K dynamic instructions) because programs overwrite data values frequently. Thus, REPT is not effective for latent bugs, where failures are distant from the root cause. Latent bugs are an important and complex class of production incidents [45, 46, 54, 60, 104, 105]. Prior techniques that rely on symbolic execution (ESD, BugRedux, and RDE) are not guaranteed to reproduce failures because constraint solvers may timeout on real failures. Finally, hybrid record/replay systems are efficient when using fine-grained recording, but can struggle to reproduce an execution when using coarse-grained recording.

We find that very few solutions are effective enough for production usage. Existing solutions that rely on symbolic execution are not guaranteed to reproduce failures and are undesirably ineffective. In addition, the inability of efficient record/replay systems to handle data races and the latent bug restrictions of REPT render these tools similarly ineffective, since these bug classes are prevalent in production. We identify a usability-boundary inspired by the coarse interleaving hypothesis introduced by Snorlax [71] and adopted by REPT [111]. The coarse interleaving hypothesis observes that most events in concurrency bugs [74, 76] are interleaved coarsely (e.g., 10s of microseconds separate the events). Thus, we consider failure reproduction tools that can reproduce any bug not violating the coarse interleaving hypothesis to be effective-enough for production use, since these tools can faithfully reproduce errors that arise in production (shown in Fig. 1b). While REPT reproduces executions that satisfy the coarse interleaving hypothesis, it cannot support latent bugs and is thus not effective enough for production bugs.

2.3 Accuracy

All record/replay systems are accurate since they faithfully reproduce all state in an execution [29, 49, 81, 86, 108]. ESD and other offline tools achieve a useful accuracy property—while ESD does not always reproduce the same data and control flow as the failure, it reproduces an execution that will lead to the same failure. Hybrid record/replay systems offer accuracy between record/replay and offline tools, depending on the recording granularity. REPT guesses data values during post-mortem analysis, which yields incorrect register or memory values in the majority of the reconstructed executions—in its original evaluation, all failure reproductions contain incorrect values [111]. REPT is especially inaccurate for bugs with long traces; for traces longer than 100K instructions, 15%-60% of the values were incorrectly recovered by REPT. Even worse, incorrect values are not guaranteed to be consistent with recorded control-flow trace and may not be detectable without reproducing the failure, thereby misleading developers.

Record/replay systems, offline techniques like ESD, and hybrid record/replay techniques produce control flow and data that lead to the failure. Thus, developers can trust the output of these tools. In contrast, best-effort tools, such as REPT, produce output with missing and/or inconsistent data and control-flow. Inaccuracy is a major obstacle in practice—the reliability of the tools and the accuracy of their results is particularly important for developers [73], because testing and debugging often requires significant effort and time [87]. Moreover, the output from best-effort failure reproduction systems is not executable, which makes it impossible to leverage dynamic tools on top of these systems (see § 2.4). We identify a usability-boundary at this inflection point: a tool is acceptably accurate if it is guaranteed to reproduce a replayable execution with the same failure.

2.4 Summary

Existing bug reproduction tools are not sufficiently efficient, effective, and accurate to (1) ensure low overheads that are compatible with the performance requirements of production settings, (2) reproduce complex failures found in

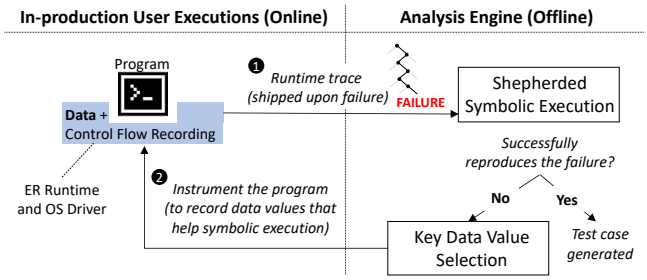


Figure 2. The high-level design and usage model of ER

production environments, and (3) provide reliable and executable output. We find that prior systems over-prioritize one or more properties at the expense of the others. Offline and hybrid solutions, like REPT, are overly efficient; they record little to no information, which imposes negligible run-time overhead but sacrifices efficiency/accuracy. Full record/replay systems are overly effective; they support bugs which violate the coarse interleaving hypothesis, even though those bugs do not appear in practice [71]. Efficient record/replay systems are overly accurate; they are guaranteed to reproduce the exact control flow and data values from a failure, but, in software reliability domains (e.g., debugging, fuzzing, security forensics), it is sufficient to reproduce any execution that leads to the same failure [114] (see § 5).

In contrast, ER lies at the usability-boundary of each failure reproduction property. Thus, ER reliably reproduces production failures with production-grade efficiency and adds production support for the wide-range of dynamic tools that are built on top of failure reproduction systems. For example, ER enables tools for debugging [63], configuration management [34], security forensics [49], automated failure localization [100, 120], large-scale parallelism [96], and bug detection [42]. In § 5, we provide a case study that uses ER for invariant-based failure localization.

3 Design

ER resides at the sweet spot in the design space of bug reproduction systems with respect to the aforementioned key properties of efficiency, effectiveness, and accuracy. ER uses execution information (control flow and data values) in conjunction with symbolic execution to reconstruct a failure. Through careful combination of these techniques, ER simultaneously mitigates the efficiency problems of purely online approaches and the effectiveness problems of purely offline approaches. At a high-level, ER follows an iterative algorithm that monitors production failures and performs symbolic execution to either generate a failure-inducing input or identify the reason why inputs cannot be generated. In the latter case, ER selects new data values to monitor which allow symbolic execution to generate a failing input during future failure occurrences.

Fig. 2 shows the high-level design of ER. ER provides a runtime that monitors an execution to produce a trace of control flow and data values as per the direction of ER's Analysis Engine (§ 3.1). When a failure occurs, the runtime ships the failure trace to the analysis engine to perform shepherded symbolic execution. Shepherded symbolic execution uses the runtime trace recorded in production to reconstruct an execution that reproduces the failure (§ 3.2). Control-flow information allows shepherded symbolic execution to eschew the notorious path-explosion problem in symbolic execution. However, the constraints on inputs nevertheless grow complex and stall symbolic execution, eventually leading to constraint solver timeouts. So, ER performs key data value selection, which analyzes the constraints from symbolic execution to identify a small set of data values that will simplify constraints and thus eliminate stalls in symbolic execution(§ 3.3). ER instruments the program to record selected data values, redeploys the program in production, and waits until the failure reoccurs. This process continues until ER successfully generates an input that leads to the failure.

We describe how ER performs online monitoring (§ 3.1), shepherded symbolic execution (§ 3.2), and key data value selection (§ 3.3), and how ER supports concurrency (§ 3.4).

3.1 Online Monitoring

ER uses the runtime and driver support to monitor a production failure using efficient hardware support (via Intel PT [67]). ER records the control flow and data values of production executions, as per the directions of ER's key data value selection (§ 3.3). By default, ER always traces control-flow, because control-flow information is useful to eliminate path explosion (§ 3.2) and existing hardware techniques enable non-intrusive control-flow recording (i.e., executions can be traced without any modifications). If desired, developers can configure ER to enable tracing only after a failure is observed multiple times.

When a failure is detected, ER ships the runtime trace including the control-flow trace and any data values to shepherded symbolic execution (§ 3.2, step ❶). The failure can either be fail-stop (e.g. crashes, hangs, etc.) or programmatically detectable by other criteria (e.g. semantic bugs checked by developer-specified assertions).

3.2 Shepherded Symbolic Execution

Shepherded symbolic execution generates program inputs that reproduce a production failure by following the runtime trace produced during online monitoring and gathering constraints on input variables. The runtime trace includes the control flow of the failing execution (i.e., the branches, calls, and returns) and any data values (e.g., arguments to a function, return values, etc.) selected by key data value selection (§ 3.3). The trace of a failure reoccurrences may have subtle discrepancies with previous traces for the same failure; shepherded symbolic execution always uses the latest trace.

```
1 uint32 V[256] = {0};
2 foo(uint32 a, uint32 b, uint32 c, uint32 d) {
3   uint32 x = (a + b);
4   if (x < 256 && c < 256 && d < 256) {
5     V[x] = 1;
6     if (V[c] == 0) // x != c
7       V[c] = 512;
8     V[V[x]] = x;
9     if (c < d)     // d != c
10      if (V[V[d]] == x)
11        abort();
12  }
13 }
```

Figure 3. Running example

To demonstrate ER's end-to-end operation, we use the listing in Fig. 3. If this program aborts (Line 11), then `x` equals `d`, due to the statements `V[V[x]] = x` (Line 8) and `if (V[V[d]] == x)` (Line 10). Although this program is simple, it shows the code patterns that cause symbolic execution stalls in real-world applications, as discussed in § 3.3.

In our example, we make a few assumptions: (1) `foo` is called as `foo(0,2,0,2)`, which satisfies all the branch conditions, (2) the program crashes on Line 11 due to the `abort`, and (3) this is the first time the failure has occurred (so the runtime trace only contains control-flow information). The control-flow trace of the failing execution (i.e., the line numbers of the executed statements after each taken branch) in Fig. 3 is **2→ 5 → 7 → 10 → 11 (FAILURE)**. The ER runtime ships this control-flow trace to the analysis engine.

By following a control-flow trace from a production failure, shepherded symbolic execution eliminates the path explosion challenge, since it only explores a single program state that follows the trace. ER still executes the program with symbolic inputs and builds up a *path constraint*. A path constraint is a collection of constraints on the program input at any given point in symbolic execution.

In our example, ER starts executing `foo` with four symbolic (unconstrained) arguments, λ_a, λ_b , λ_c and λ_d (Line 2). Since all the operands are symbolic, the value of `x` computed on Line 3 is also symbolic. The branch on Line 4 compares symbolic inputs (λ_c, λ_d) and the symbolic value `x` to 256. Since the next instruction executed in the recorded control-flow trace is the memory write at Line 5 (see the above trace in bold), the branch on Line 4 was taken and ER will update the path constraint to include the outcome of the branch condition, i.e. $((\lambda_a + \lambda_b) < 256) \wedge (\lambda_c < 256) \wedge (\lambda_d < 256)$.

Line 5 uses `x` to dereference an entry in `V` and creates a symbolic memory address (the base address of `V`, offset by $(\lambda_a + \lambda_b)$). ER cannot determine the addresses of `V[x]`, but the fact that the control-flow trace has more elements indicates that dereferencing `V[x]` does not lead to the failure. The control-flow trace indicates that the branch on Line 6 was also taken. Since `V` is updated at a symbolic address on Line 5, the branch condition on Line 6 needs to treat array `V` symbolically: $(\text{Read}(\text{Write}(V, x, 1), c) == 0)$. Here, we use a standard representation of accesses to symbolic memory [48, 56]: 1) $\text{Read}(A, i)$ returns the value at location $A[i]$, where A is an array and i is the index. 2) $\text{Write}(A, i, v)$ updates the i-th entry in the array A with value v.

ER continues shepherded symbolic execution by updating the path constraint until it reaches the failure, or until symbolic execution stalls because the underlying constraint solver cannot handle the complex constraints that have been gathered. During shepherded symbolic execution, ER invokes a constraint solver every time the program accesses symbolic memory (e.g., $(\text{Read}(\text{Write}(V, x, 1), c)$ in our example) to determine the set of concrete memory locations that may be accessed. This approach prevents the symbolic execution engine from assuming that a write operation may write to any address, which would complicate the path constraints and lead to other scalability issues in symbolic execution.

If shepherded symbolic execution reaches the failure that occurred in production (i.e., at the end of the trace), ER invokes a constraint solver to determine concrete program inputs that would lead to the failure. If constraint solving can determine a satisfying assignment to the path constraint, the failure is reproduced and a full test case is generated for the developer. The generated test case may not include the same inputs that caused the production failure, but is guaranteed to lead the program along the same recorded control flow and reproduce the same failure. As we show in our evaluation, in a few cases (2 out of 13), ER can reproduce the failure in the first attempt.

However, for most cases (11/13 in our evaluation), shepherded symbolic execution is not able to reproduce the failure in the first attempt. In these cases, ER identifies a solver timeout (§ 4) and constructs a constraint graph (explained below) that depicts the dependencies among data values and constraints. ER passes the constraint graph to key data value selection (§ 3.3).

Constraint Graph Construction When there is a solver timeout, ER constructs a constraint graph, which succinctly describes the dependencies between values and constraints that are established as the program is symbolically executed. This graph representation is inspired by the internal data structures of the STP constraint solver [56]. The nodes in the graph represent operations (arithmetic, logic, memory), constants, program inputs (which are symbolic and therefore unknown), objects in the symbolic memory model (e.g., an array allocated in the program), and symbolic memory addresses. The edges represent the dependencies among the nodes and point from a node to its input dependencies. The goal of a constraint solver is to identify a concrete assignment for each symbolic input node that simultaneously satisfies all the constraints in the graph.

We use our example from Fig. 3 to explain how ER constructs the constraint graph. Recall that shepherded symbolic execution follows the control-flow trace (**2→ 5 → 7 → 10 → 11 (FAILURE)**) recorded by running `foo`, and tries to reproduce the crash (due to `abort`) on Line 11. Fig. 4 shows the

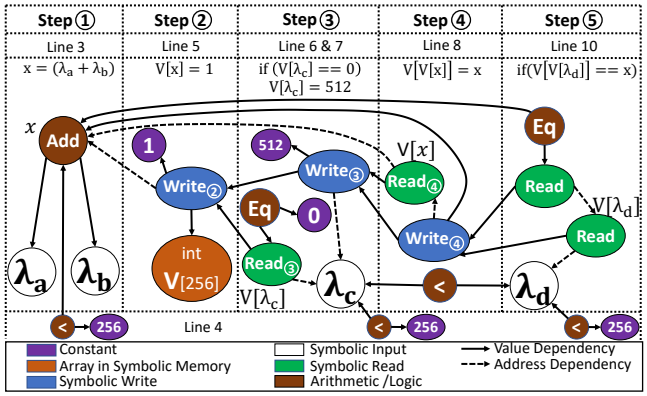


Figure 4. Step-by-step construction of the constraint graph for the example in Fig. 3.

example constraint graph that is generated right before `foo` aborts. Since ER is executing the program symbolically, `foo`'s arguments are represented as four symbolic input nodes (λ_a, λ_b, λ_c, λ_d). As shown in step ①, the value computed by $x = (a + b)$ on Line 3 is an arithmetic "addition" node, `Add`, that depends on the values of the symbolic inputs λ_a and λ_b.

Using the recorded control-flow trace, ER knows that the branch on Line 4 was taken. Subsequently, in step ②, the value `x` is used to index into the array `V` and write 1 to an element (Line 5). The write operation ($\texttt{Write}_②$[1]) depends on the target array (`V`), the destination address (`x`), and the value to write (1). Note that the graph representation of the write operation is equivalent to the representation we introduced above, namely Write(`V`, `x`, 1).

In step ③ (Lines 6-7), the branch condition includes a read operation ($\texttt{Read}_③$) , which depends on: (i) a symbolic array to read from (which is represented by $\texttt{Write}_②$, as this node represents the last state of the array `V` that was written to in step ② and (ii) a source address (λ_c). Note that the graph representation of the read operation is equivalent to the representation we introduced above, namely Read($\texttt{Write}_②$, λ_c). The path constraint that ER builds by following the control-flow trace (i.e., the branches that are taken) is represented in the graph using the logic `Eq` nodes. $\texttt{Eq}_③$ denotes that $\texttt{Read}_③$ was equal to 0. Finally, $\texttt{Write}_③$ represents the value 512 being written to the symbolic array `V` (whose final state before being written to is again represented by $\texttt{Write}_②$).

Constraint graph generation is similar for steps ④, ⑤.

3.3 Key Data Value Selection

Key data value selection eliminates symbolic execution stalls by analyzing the constraint graph and determining the key data values that will simplify the path constraints. We describe the main sources of constraint solving complexity (in §3.3.1) and how ER determines the set of key data values that will remove stalls in symbolic execution (in §3.3.2).

[1]We use subscripts to distinguish different nodes in different steps.

3.3.1 Sources of Constraint Complexity.

Recall from section 3.2 that ER invokes the constraint solver every time a symbolic memory location is accessed to simplify reasoning about subsequent memory accesses. We now describe how constraints on symbolic memory accumulate and provide intuition about how these constraints lead to symbolic execution stalls. We then explain the two key contributors to constraint complexity, namely (1) the length of symbolic write chains, and (2) the size of the accessed symbolic memory.

Examples of complex constraints. Consider the example in Fig. 3, and the associated constraint graph in Fig. 4. On Line 5, (step ② in the constraint graph), the program writes the value 1 to a symbolic address (`V[x]`), because x is a symbolic value, `x`= $(\lambda_a + \lambda_b)$. Depending on the value of x, the access will either be within the bounds of the array `V` and simplify reasoning, or, outside the bounds of the array and require that ER update other memory objects. Since the branch on Line 4 was taken, the path constraint contains the constraint $\lambda_a + \lambda_b < 256$, and the solver deduces that the array access on Line 5 is within the bounds of `V`, which has 256 elements. ER represents the state of the symbolic memory in step ② as Write(V, x, 1), or $\texttt{Write}_②$ for short.

Symbolic execution can stall when encountering chains of symbolic addresses. Consider the statement `V[V[x]]=x` on Line 8 (step ④ in the constraint graph). This statement first reads `V[x]` from the most recent update to `V` (i.e. $\texttt{Write}_③$) with offset `x`, where $x = \lambda_a + \lambda_b$. By the logic of the previous paragraph, the solver determines that the offset `x` is always within the bounds of the array `V`. We denote this read operation in step ④ as $\texttt{Read}_④$ = Read($\texttt{Write}_③$, x), where $\texttt{Write}_③$ = Write($\texttt{Write}_②$, λ_c, 512). Expanding $\texttt{Write}_②$ provides $\texttt{Write}_③$ = Write(Write(V, x, 1), λ_c, 512). Together with the initialization of `V` on Line 1, this chain of writes identifies that `V[x]` is 0, 1 or 512. Thus, it is difficult to determine if the access on line 8 (`V[V[x]]`) is within the bounds of the array `V`; the solver will have to combine constraints gathered from Lines 5 (`V[x] = 1`) and 6 (`if (V[c]==0)`) to determine that `c` cannot be equal to `x` and thus `V[x]` cannot be equal to 512. The memory read and the branch condition on Line 10 (`if(V[V[d]] == x)`) involves similar non-trivial reasoning on the solver's part.

While modern solvers, and hence ER, are able to reason about cases such as Lines 8 and 10, as chains of symbolic memory accesses increases, constraint solving becomes a challenge. In addition, constraint solving is complicated by accesses to large memory objects, since the solver needs to reason about accesses to more memory locations. Therefore, for the purpose of illustration, we assume that the accesses on Line 8 and Line 10 are challenging for solvers to resolve and thus cause symbolic execution to stall.

Key contributors of constraint complexity. An obvious way to avoid all constraint solving complexity would be to record all program inputs, similar to what record/replay

engines do. Unfortunately, record/replay can incur high overhead, therefore, ER aims to determine key data values that causes the constraints to become complex and record them. We observe that there are two key contributors to the complexity of symbolic memory constraints:

1) Length of symbolic write chains. Updates (i.e., writes) to symbolic memory add constraints to the memory state, complicating constraint solving. In our example, the chain of three `Write` operations in Fig. 4 can be represented as:

$$\texttt{Write}_{④} = \text{Write}(\text{Write}(\text{Write}(V, x, 1), \lambda_c, 512), \\ \text{Read}((\text{Write}(\text{Write}(V, x, 1), \lambda_c, 512), x), x)$$

This chain bottlenecks shepherded symbolic execution, when the solver gets invoked for the read operations in steps ④ and ⑤ (Lines 8 and 10).

2) Size of the accessed symbolic memory. If the size of the symbolic memory that is accessed is large, the solver needs to reason about a large number of locations that can be accessed, which is complicated. An example is the three `Write` nodes in Fig. 4 that modify the 1024-byte array `V` (256×4), which is larger than any other memory object in the graph.

3.3.2 Computing the Bottleneck and Recording Sets.

ER exhaustively searches in the constraint graph to identify the longest symbolic write chain and the write chain that updates the largest symbolic memory object. (Note that these two chains can be the same.) We call the set of all symbolic values that are read/written by operations in these two chains as the *bottleneck set*; collectively, this set represents key data values that should be recorded to simplify constraints and resolve symbolic execution stalls. In our example, where we assume constraint solving stalls on Line 8 in step ④, there is a single write chain ($\texttt{Write}_{④} \rightarrow \texttt{Write}_{③} \rightarrow \texttt{Write}_{②}$) that updates the symbolic array `V`, which is the largest symbolic memory object. The bottleneck set comprises all the symbolic values that are referred to by these three writes (see the dashed address dependency arrows depicted in Fig. 4), namely {`x`, λ_c, `V[x]`}.

Reducing the Cost of Recording. A naive strategy to simplify constraint solving would be to record all the elements in the bottleneck set the next time the failure occurs. In Fig. 4, this strategy would correspond to recording the concrete values for node `x`, λ_c and `V[x]` ($3 \times 4 = 12$ bytes total), which will make it easier for the constraint solver to satisfy all the constraints in the first four steps.

Unfortunately, this approach has high overhead. So, ER reduces the amount of data it records by identifying an alternative set of data values to record from which the bottleneck set can be inferred. ER achieves this with the following algorithm: Initially, ER assigns the recording set of data values, $\mathbf{E} = \{E_0, E_1, \ldots, E_k\}$, to be the bottleneck set. ER assigns a cost (C_i) to recording each element that is equal to the size of the element times the number of times the node is referenced in the recorded control-flow trace, i.e., $C_i = sizeof(E_i) \times Count_{E_i}$. ER's goal is to minimize the total cost of recording, $C = \sum_{i=1}^{k} C_i$.

For each element E_i in $\mathbf{E}$, ER performs a depth-first search in the constraint graph to determine if it can record a node (or multiple nodes) with a lower cost than C_i that ER can use to determine the value of E_i. ER continues searching until it can no longer reduce C, the total cost of recording.

Consider the symbolic elements of the bottleneck set {`x`, λ_c, `V[x]`} in Fig. 4, where each element has a reference count of 1 since all the statements in `foo` executed once. The algorithm first considers the element `x`, which has a recording cost of 4 (4 bytes of data × referred 1 time). ER continues the depth first search and determines that it cannot record `x` with a lower cost (the cost of recording both λ_a and λ_b—sum of which equals `x`—is 8). Similarly, when ER does a depth-first search starting with λ_c, it does not update the recording set (since λ_c is a leaf node, i.e., an input). Finally, when ER performs a search starting with `V[x]`, it finds out that `V[x]` can be deduced given the values of `x` and λ_c. This happens, because given the values of `x` and λ_c, all the memory writes in steps ②–④ write a concrete value to a concrete address, therefore, $\texttt{Read}_{④}$ reads a concrete value from a concrete address as well. Consequently, ER removes `V[x]` from the recording set. At this stage of the search, the elements in the recording set do not change anymore and the set of key values ER will record is {`x`, λ_c}.

3.3.3 Recording Key Data Values.

Based on the final recording set, ER instructs its runtime to record the elements (e.g., values, addresses). The runtime records an element at a point in the execution corresponding to where the element is introduced in the constraint graph. For the recording set {`x`, λ_c} in our example, ER records the value `x` (to concretize $\lambda_a + \lambda_b$), when `x` is first computed (Line 3 in Fig. 3, step ① in Fig. 4). ER records the input value `c` (to concretize λ_c) when it is introduced in the graph in the expression `if (V[c] == 0)` (Line 6 in Fig. 3 and step ③ in Fig. 4).

3.3.4 Iterative Operation.

ER continues the iterative process of recording runtime information, shepherded symbolic execution, and key data value selection until it can reproduce the failure. In our example, a second occurrence of the failure will ship the control-flow trace and values of x and λ_c to shepherded symbolic execution. Symbolic execution will stall when calculating the memory read on Line 10 and key data selection will identify a recording set of {`x`, λ_c, λ_d}. After a third occurrence of the failure, shepherded symbolic execution is able to reproduce the failure on Line 11.

3.4 Handling Concurrency

Traditionally, symbolic execution reasons about different thread interleavings in a program by encoding the thread schedule in the path constraint of an execution [39]. Unfortunately, in order to explore the potential thread interleavings,

a symbolic execution engine has to fork program states after every instruction in the program, exacerbating the path explosion problem. Although this naive method can be improved by identifying data races in the program and only forking program states at racing memory accesses [31, 75, 77], it is still not a tractable approach for large systems [114].

ER instead relies on observations of prior works [71, 111] that suggest that in many cases, it is possible to use a coarse-grained timer to track the execution order of shared memory accesses in a multi-threaded program. Specifically, ER relies on time tracking capabilities of modern hardware (timestamp packets in Intel PT [67]) to record a partial order of instructions across multiple threads. During shepherded symbolic execution, ER executes chunks of instructions from different threads according to the partial order described by the timer packets in the recorded trace. If ER is unable to establish a total order among the chunks of instructions because the associated timestamps overlap, ER arbitrarily selects a sequence of instructions and tries to reconstruct the execution.

If the granularity of the timer packets is fine enough to capture all the data races and synchronization operations, ER reconstructs the execution reliably. For the multi-threaded programs in our evaluation (§5), ER is able to reconstruct failing executions. However, if a program exhibits a high-degree of fine-grained racing accesses, ER may not be able to reconstruct the execution. In such cases, ER could use state-space exploration techniques [32, 93] to determine the order of finer grained racing accesses.

4 Implementation

Runtime and OS Driver. ER uses hardware-assisted control and data tracing in Intel PT [67] to collect the runtime traces to quantify performance overhead. ER configures the Intel PT Linux driver and records the control-flow, timing information, key data values, in a 64MB ring buffer for each monitored application. The buffer size is decided by the largest trace we collect from the evaluated failures (§5.1). Mapping x86_64 control-flow traces into LLVM IR (which is required by Klee to symbolically execute code) introduces inaccuracies due to optimizations. In our evaluation, we observe that only 91.5% of the control-flow events (branches, calls, returns) in the x86_64 executions can be mapped back to LLVM IR. Meanwhile, recorded key data values can be mapped fully accurately. Our prototype currently makes up for the control-flow inaccuracy by tracing both control-flow and key data values within Klee.

Even though symbolic execution in Klee can deal with partially-recovered LLVM traces at the expense of slight path explosion [106], we intend to tackle this problem by either (1) instrumenting the `clang` optimization passes that cause information loss to save metadata about optimizations in order to increase the accuracy of x86_64-to-LLVM mapping, or (2) turning to a binary symbolic execution engine [44].

Recording Key Data Values. We implemented an LLVM compiler pass (156 LoC) to instrument programs with the `ptwrite` instructions [27], which records data values identified by key data value selection into the Intel PT trace. Instrumentation requires redeploying a new version of the application to record additional values, which suits the rapid development cycles of modern software [101] well. Dynamic binary instrumentation [37, 38, 83] presents a potentially less invasive solution for less frequently updated applications.

Shepherded Symbolic Execution. We implemented a prototype shepherded symbolic execution engine on top of Klee [40] in 13.8K LoC. We modified Klee to follow a control-flow trace and extended its POSIX environment model based upon the changes from Cloud9 [39]. The extended POSIX environment treats thread-interleavings and system calls (e.g. the content of input files, packets from network sockets, clock information, etc.) as sources of non-determinism. Our prototype treats these non-deterministic values as symbolic (unknown). The shepherded symbolic execution engine detects the reoccurrence of a failure based on matching the program counter and the call stack where the failure occurs.

Detecting Symbolic Execution Stalls Complex constraints prevent symbolic execution from progressing. Some constraints (e.g. floating-point) are not expressible in Klee; certain classes of constraints are known to be undecidable[35]. For constraints supported and solvable in Klee, ER determines if shepherded symbolic execution is stalled by identifying solver timeouts. The ideal timeout depends upon the frequency of a failure. For failures that occur frequently, ER should be configured with a relatively short timeout so that shepherded symbolic execution can quickly simplify symbolic constraints using recorded data values; for infrequent failures, ER should use a longer timeout to reduce the number of reoccurrences required to reproduce a failure. For the failures in our evaluation, we found that a 30 second timeout provides a good balance between the time spent on symbolic execution and the number of failure reoccurrences required to reproduce a failure.

Key Data Value Selection. We implemented the key data value selection algorithm in Python (1.3K LoC). When the solver times out, ER passes the path constraint from Klee to the analyzer. The analyzer generates a list of LLVM IR registers as the key data values to record.

5 Evaluation

In this section, we first describe our experimental setup (§5.1) as well as our benchmark selection. We then evaluate our prototype of ER by answering the following questions:

Effectiveness and Accuracy (§5.2) Can ER reproduce failures in complex, long-running executions? Is ER able to accurately reproduce failing executions? Does key data value selection choose to record useful data values? How useful is data value recording for shepherded symbolic execution?

Table 1. Bugs used in the evaluation of ER along with bug IDs and types. "MT" denotes if the program is multithreaded (Y for Yes, N for No). "LoC" denotes the lines of code of each application. "#Instr(x86_64)" denotes the total number of executed x86_64 instructions in the failing execution."#Occur" denotes the number of failure occurrences ER needs to reproduce the failure. "Symbex Time" represents the total time (sum for all iterations) ER spends on shepherded symbolic execution.

Application-BugID	Bug Type	MT	LoC	#Instr(x86_64)	#Occur	Symbex Time	Performance Benchmark
PHP-2012-2386	Integer overflow [7]	N	968,607	5,460,436	6	1.7 min	Benchmark Script [6]
PHP-74194	Heap buffer overflow [13]	N	1,303,868	5,791,278	10	111 min	Benchmark Script [6]
SQLite-7be932d	NULL pointer dereference [14]	N	413,846	1,408,411	3	3.3 min	Official fuzz test
SQLite-787fa71	Inconsistent data-structure [17]	N	221,771	1,115,003	4	61.3 min	Official fuzz test
SQLite-4e8e485	NULL pointer dereference [18]	N	302,653	1,349,129	3	21.8 min	Official fuzz test
Nasm-2004-1287	Stack buffer overrun [2]	N	224,147	1,480,285	3	12.5 min	Assemble a large asm file
Objdump-2018-6323	Integer overflow [16]	N	1,077,896	323,788	3	0.06 min	Disassemble a large binary
Matrixssl-2014-1569	Stack buffer overrun [20]	N	160,447	4,448,948	6	6.5 min	Official test
Memcached-2019-11596	NULL pointer dereference [19]	Y	151,716	1,840,258	2	3.1 min	memtier_benchmark [22]
Libpng-2004-0597	Buffer overflow [1]	N	73,442	71,752	1	0.2 min	resvg-test-suite [23]
Bash-108885	NULL pointer dereference [12]	N	335,176	866,668	1	0.3 min	Quicksort in Bash script
Python-2018-1000030	Shared data corruption [15]	Y	1,020,698	36,108,946	2	23.5 min	From PyPy benchmarks [24]
Pbzip2	Use-after-free [11]	Y	13,052	6,937,510	2	2.6 min	Compress a .tar file

Efficiency (§5.3) What is the runtime performance overhead incurred by ER due to online control and data recording? How does ER's runtime overhead compare against record/replay? What is ER's offline computational and memory overhead?
Case Study (§5.4) Can ER provide production support for software reliability tools? How does ER's support for these tools differ from existing systems?

5.1 Experimental Setup

Target Programs and Bugs. As shown in Table 1, we evaluate ER using 13 failures from a broad range of 10 real-world programs including the PHP interpreter; the Python interpreter; SQLite database; memcached, a widely-used distributed object store; the Bash shell; the binary analysis tool objump; NASM, a popular assembler in Linux; libpng, a pervasive image processing library; MatrixSSL, a TLS/SSL implementation, and pbzip2, a parallel compression tool.

Bug IDs and bug types are also shown in Table 1. We choose programs and bugs from closely-related work [71, 73, 111] and CVE exploits [25, 26] that were supported by the POSIX environment model of the KLEE symbolic execution engine [40], which ER uses for shepherded symbolic execution. The failures are caused by inconsistent data-structures, integer overflow, buffer overflow, shared data corruption, use-after-free, etc.
Workloads for Performance Evaluation. When available, we use existing benchmarking and testing suites to assess the performance impact of data and control-flow recording by ER. To evaluate the performance overhead of NASM and objdump, we assemble and disassemble the SQLite binary, respectively. To evaluate pbzip's performance, we compress the SQLite codebase which is a 71 MB .tar file. We ran each performance experiment 10 times and report averages and standard error.
Software and Hardware Configuration. We evaluate ER on two servers with Intel Xeon Silver 4114 CPU and Intel Pentium Silver J5005 CPU. The servers have 187 GB and 8 GB of memory, respectively. The Linux kernel version on both machines is 5.5.2.
Baseline. We compare ER against REPT [111], a state-of-the-art deployed bug reproduction system, and rr [29], a state-of-the-art record-and-replay system. Since REPT is not publicly available, we are only able to compare with the results reported in the REPT paper.

5.2 Effectiveness and Accuracy of ER

Length and Complexity of Reproduced Executions. Table 1 shows all the failures that ER reproduced along with the number of instructions in the executions that ER reconstructed. As shown in the "#Instr(x86_64)" column, ER was able to reconstruct failures in executions of up to ~36 million dynamic x86_64 instructions, which is more than two orders of magnitude (361×) longer than the executions that REPT [111] can reproduce (~100K).
Accuracy of Reproduced Executions. ER accurately reconstructs all data values of a failing execution, given a trace of the control flow and a few key data values recorded in production. Even though the input generated by ER may not be the same input that caused the in-production failure, ER guarantees the generated input will lead to the same control flow as the failure and reproduce the same failure. Thus, ER is able to provide developers a concrete test case (input + control flow) that they can run in a debugger to debug root causes. For example, the SQL queries recovered by ER in three SQLite bugs ([14, 17, 18]) differ from the original input that leads to these bugs in terms of SQL keywords (e.g. `sEleCT` instead of `SELECT`), identifier names (e.g. different table/field names), data values, etc. Despite these differences, the generated input follows the same control-flow because 1) keywords are case-insensitive; 2) renaming identifiers does not change query semantics; 3) recovered data satisfies all control-flow conditions in the trace.

In some instances, developers may be able to debug an issue using an inaccurate execution, such as the one provided by REPT, without relying on a full test case. However, REPT has a substantial number of unknown and inaccurate values as trace length extends beyond 100K instructions. To make matters worse, a developer cannot know which values are inaccurate, since they lack the ground truth. To demonstrate when this may be an issue, we investigated the MatrixSSL bug (for which thorough developer patches and documentation were available) to determine the root cause of the failure. We found the last instruction from the patch that fixes the bug to be executed 3 million instructions prior to the failure. So, REPT would likely not have been able to provide information about the data values of the variables used by the patch and not accurate enough for a developer to debug this issue (we only provide a qualitative comparison, as REPT is closed-source and only available on Windows).

Key Data Value Selection Effectiveness. To determine the effectiveness of key data value selection, we compared key data value selection with a version of ER that uses a random data recording strategy. The random recording strategy records the same amount of data that ER records, but selects the data randomly among all the data elements in the constraint graph. ER with random data recording only reproduces one failure among the failures that require data value recording (Nasm-2004-1287). In all other cases, ER with random data recording encounters symbolic execution stalls which it is not able to simplify with the random data, and thus cannot complete symbolic execution.

Benefits of Data Value Recording. We show the benefits of data value recording on shepherded symbolic execution. Fig. 5 compares the progress of shepherded symbolic execution on the PHP-74194 bug when using a control-flow trace compared to using traces containing control flow and the data values selected during the first and second iterations of key data value selection. We disable the solver timeout and let shepherded symbolic execution execute the same number of instructions in all three cases. Our results show that shepherded symbolic execution with no data values, the first iteration data values and the second iteration data values take 11468, 5006, and 1800 seconds, respectively, to symbolically execute the same number of instructions. We conclude that shepherded symbolic execution drastically benefits from data values and that ER's iterative approach is effective in current production environments where bugs often reoccur.

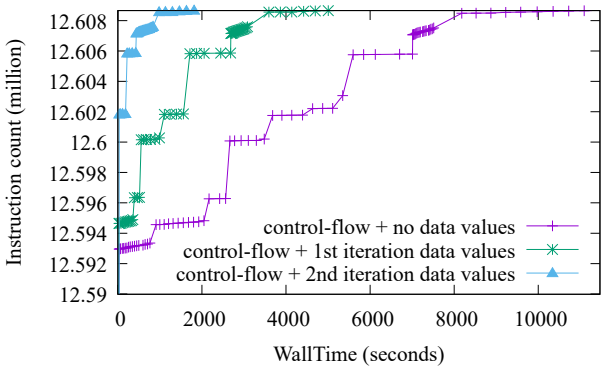


Figure 5. Number of instructions executed and the time spent on shepherded symbolic execution for PHP-74194.

5.3 Efficiency of ER

Runtime Performance Overhead. We measure the runtime performance overhead of online control flow and data value tracing. For each program, we report the recording overhead of ER for the last occurrence of the failure needed to reproduce the failure. We choose the last occurrence because ER records the most data in the last iteration. Similar to prior work [57], our sensitivity analysis shows no statistical difference in the runtime overhead for buffer sizes of 4KB, 64KB, 1MB, 16MB, 64MB (using a 90% confidence interval).

As shown in Fig. 6, ER incurs on average 0.3% (and up to 1.1%) runtime performance overhead across all the programs we evaluate. The overhead numbers for some applications are sufficiently low to be affected by the variability in other factors such as disk I/O (e.g., Libpng's performance benchmarks open and read about 1000 files). These results are in line with prior work that demonstrated that Intel PT is efficient enough to be deployed in production [71, 73, 111]. In our work, we also demonstrate that recording a few key data values does not increase the overhead incurred by Intel PT.

Comparison to Record/Replay. A key question we set out to explore in this paper is whether we can achieve the same level of effectiveness (i.e., failure reproduction ability) as a record replay engine, with better efficiency (i.e., by incurring lower runtime performance overhead).

Fig. 6 also displays the recording overhead incurred by a state-of-the-art record/replay engine, Mozilla rr. rr imposes an average of 48.0% (and up to 142.2%) runtime performance overhead. The overhead of rr is prohibitive, even for single-threaded applications (objdump, libpng, etc.). We conclude that ER is able to effectively reconstruct all the failures in our evaluation, and incur lower overhead than a state-of-the-art record/replay system.

Offline Memory and Computational Overhead of ER. The maximum amount of memory consumed by shepherded symbolic execution and iterative constraint reduction is 10 GB, which we believe is a reasonable amount given the decreasing trends of memory cost [28, 62]. The largest constraint graph in our evaluation had about 40K nodes, from which computing bottleneck sets and recording sets took at most 15 seconds. The average and the maximum total shepherded symbolic execution time was 19 and 111 minutes, respectively.

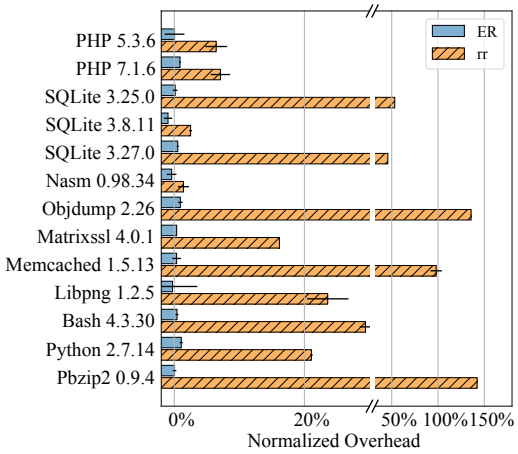


Figure 6. Runtime performance overhead incurred by ER's control and data flow recording as well as recording using rr [29]. The error bars display standard error.

5.4 Invariant-Based Failure Localization

ER allows software reliability tools across many domains (e.g., security forensics, fuzzing, etc.) to leverage production failures. In this case study, we show how ER provides production-support to MIMIC [120], an invariant-based automated failure localization technique. MIMIC uses Daikon [52] to calculate likely invariants, predicates which are observed during successful executions. When presented with a failure, MIMIC identifies the likely invariants that are violated by the failure and proposes these as potential root causes. In our case study, MIMIC gathers likely invariants offline, using existing integration and unit tests. When a production failure occurs, ER reconstructs the failure and passes it to MIMIC, which identifies potential root causes.

We test this approach using the coreutil bugs (od [3] and pr [4]) from MIMIC. For both applications, we generate likely invariants using 4 successful executions. We then use ER to reconstruct a failing execution that exhibits the bugs, and pass the reconstructed execution to Daikon. Daikon identifies the same potential root causes (i.e., invariant violations) when using the reproduced execution from ER as it does when using the failing test case directly.

As discussed in Fig. 1, existing techniques for reproducing bugs cannot support tools like MIMIC in production scenarios (although they may help with the relatively simple programs, od and pr, discussed above). Full record/replay systems, hybrid record/replay systems, and BugRedux are too expensive for production deployment. Offline approaches (e.g., ESD) and efficient record/replay cannot handle all production bugs and are not guaranteed to reproduce the failure to pass to MIMIC. Finally, REPT does not produce output that MIMIC can execute to produce invariants. If MIMIC could generate invariants from the inaccurate data values recovered by REPT, MIMIC might identify root causes that are both false positives (i.e. inaccurate values violate a likely invariant that should be satisfied) and false negatives (i.e. inaccurate values satisfy a likely invariant that should be violated).

6 Related Work

Reverse execution of failed executions. REPT [111] is the state-of-the-art approach to recover the data flow of a program given the control flow. We compare our work with REPT in §2 and subsequent sections, in particular, we discuss its limitations in terms of effectiveness and accuracy. Similarly, RETracer [47] reverse-executes a program to determine the root causes of memory corruption. RETracer is efficient in that it only uses a core dump, without data or control-flow information, however its scope is limited to reasoning about memory corruption bugs within a stack trace.

Record-replay. iDNA [36], PinPlay [94], ReVirt [51], rr [92] and many others [32, 63–65, 81, 82, 86, 88, 95, 98, 108] record at run-time all non-deterministic events (i.e., system calls, thread scheduling, etc.) that impact program execution. They allow users to replay and analyze the execution afterwards. Although useful during development, they are not commonly used in production due to high overheads, e.g., rr has an overhead that can range from 49% to 685% [92]. Other approaches [90] propose specialized hardware for record-replay, but adoption is slow and difficult.

Schedule record-replay. Several approaches specifically target the challenges posed by multi-threaded applications [53], which depend not just on program input but also on the thread interleavings. For instance, H3 [66] records the control flow of programs (with overheads between 1.4%-14.7%) and uses this information to determine the thread interleavings that lead to concurrency bugs. Notably, PRES [93] and HOLMES [43] record select run-time information, such as the total order of synchronization calls, system calls, or function call invocations, to constrain the interleaving space exploration during post-failure analysis. Unlike ER, these tools assume that the failure-inducing input is already known by employing uni-processor record-replay techniques, hence these approaches are orthogonal to ours.

Symbolic execution. Symbolic execution techniques [79] reason about programs symbolically using SMT solvers (e.g., Z3 [48]) and are often used to explore the input space of programs and conduct path-sensitive analysis [40, 44, 59, 91]. To reproduce a particular failure, state-of-the-art approaches [41, 69, 114] take a stack trace where the program crashed, and steer symbolic execution towards the failure location, which traditionally leads to path explosion. Symbolic backward execution [50, 84] is a variant technique that tries to reason about programs from a source code location (e.g., an assertion) backwards to the initial program state. Such approaches are not able to address the constraint complexity challenges like ER. Concolic testing approaches augment

symbolic execution with concrete executions to automatically generate test cases [59, 85, 103]. Although concolic testing can simplify the path constraints by recording concrete values from random testing, it is generally designed for path exploration, and does not focus on a specific failure trace like ER.

Failure analysis and diagnosis. A large body of work has developed techniques for failure diagnosis and root cause analysis [30, 33, 68, 70, 71, 73, 80, 100, 107, 110, 116, 119], which can complement ER, since failure reproduction can assist failure diagnosis. Other work [113] has explored different approaches to analyze core dumps of failed executions and extract useful information. Pensieve [118] reproduces the set of events that are relevant to a failure in a distributed system. ER can be used in conjunction with Pensieve to thoroughly reason about failures in a single node.

7 Conclusion

In this paper, we presented Execution Reconstruction (ER), a technique to reproduce production failures. ER strives for a sweet spot among efficiency, effectiveness and accuracy. ER uses shepherded symbolic execution to leverage dynamic control flow to eschew path explosion. In addition, to avoid solver stalls, ER uses key data value selection, which analyzes the constraints generated by shepherded symbolic execution to identify a set of data values that can accelerate symbolic execution. Using these techniques, ER will eventually generate concrete test cases that reproduce complex failures. We have implemented an end-to-end prototype to demonstrate the effectiveness of our approach based on Intel PT and Klee. Real world applications show that ER is able to reproduce failures with similar efficiency to state-of-the-art efficient failure reproduction tools (on average 0.3%) while providing similar accuracy/effectiveness as state-of-the-art accurate/effective failure reproduction tools. In addition, we show that ER allows software reliability tools to leverage production failures.

Acknowledgements

We thank the anonymous reviewers and our shepherd, Ben Liblit, for their valuable feedback. We thank Petros Maniatis for his feedback on an earlier draft and Jörg Thalheim for his initial prototype of the Intel PT recording tool. This work is supported by the NSF CAREER award 1942218, the NSF DGE award 1256260, a Google Faculty Award, and the Applications Driving Architectures (ADA) Research Center (a JUMP Center co-sponsored by SRC and DARPA).

References

[1] 2004. CVE Detail. https://nvd.nist.gov/vuln/detail/CVE-2004-0597
[2] 2004. NASM 0.98.x - Error Preprocessor Directive Buffer Overflow. https://www.exploit-db.com/exploits/25005
[3] 2007. Coreutils, Fault in OD. https://lists.gnu.org/archive/html/bug-coreutils/2007-08/msg00034.html
[4] 2008. Coreutils, Fault in PR. https://lists.gnu.org/archive/html/bug-coreutils/2008-04/msg00177.html
[5] 2012. Fixing Bugs - If You Can't Reproduce a Bug, You Can't Fix It - DZone Agile. Retrieved 06/08/2019 from https://dzone.com/articles/if-you-cant-reproduce-bug-you
[6] 2012. Free PHP Benchmark Performance Script. http://www.php-benchmark-script.com
[7] 2012. Sec Bug #61065: Secunia SA44335 - arbitrary code execution. https://bugs.php.net/bug.php?id=61065
[8] 2013. https://issues.apache.org/jira/browse/HBASE-10210
[9] 2013. https://issues.apache.org/jira/browse/HBASE-10237
[10] 2013. https://issues.apache.org/jira/browse/HDFS-5690
[11] 2013. pbzip2. https://github.com/jieyu/concurrency-bugs/tree/master/pbzip2-0.9.4
[12] 2015. sr #108885: 4-byte script triggers null ptr deref and segfault. https://savannah.gnu.org/support/index.php?108885
[13] 2017. Sec Bug #74194: a heap-buffer-overflow when serializing ArrayObject. https://bugs.php.net/bug.php?id=74194
[14] 2018. Adverse interaction between .stats and .eqp in the CLI. https://www.sqlite.org/src/tktview/7be932d
[15] 2018. CVE-2018-1000030: Python 2.7 readahead feature of file objects is not thread safe. https://bugs.python.org/issue31530
[16] 2018. CVE Detail. https://www.cvedetails.com/cve/CVE-2018-6323/
[17] 2019. Assertion fault when multi-use subquery implemented by co-routine. https://www.sqlite.org/src/tktview/787fa71
[18] 2019. Crash on query using an OR term in the WHERE clause. https://www.sqlite.org/src/tktview/4e8e485
[19] 2019. CVE Detail. https://nvd.nist.gov/vuln/detail/CVE-2019-11596
[20] 2019. MatrixSSL < 4.0.2 - Stack Buffer Overflow Verifying x.509 Certificates. https://www.exploit-db.com/exploits/46435
[21] 2019. What does debugging a program look like? - Julia Evans. Retrieved 07/04/2019 from https://jvns.ca/blog/2019/06/23/a-few-debugging-resources/
[22] 2020. Retrieved 05/27/2020 from https://github.com/RedisLabs/memtier_benchmark
[23] 2020. Retrieved 05/27/2020 from https://github.com/RazrFalcon/resvg-test-suite/tree/master/png
[24] 2020. Retrieved 5/27/2020 from https://foss.heptapod.net/pypy/benchmarks/-/tree/branch/default/unladen_swallow/performance
[25] 2020. CVE Database. https://www.cvedetails.com
[26] 2020. Exploit Database Archive. https://www.exploit-db.com/
[27] 2020. Intel® 64 and IA-32 Architectures Software Developer's Manual. https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html
[28] 2020. Memory Prices Decreasing with Time. https://jcmit.net/mem2015.htm
[29] 2020. rr. https://rr-project.org
[30] R. Abreu, P. Zoeteweij, and A. J. c. Van Gemund. 2006. An Evaluation of Similarity Coefficients for Software Fault Localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. 39–46. https://doi.org/10.1109/PRDC.2006.18
[31] Adil Ahmad, Sangho Lee, Pedro Fonseca, and Byoungyoung Lee. 2021. KARD: Lightweight Data Race Detection with Per-Thread Memory Protection. In *Proceedings of the International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS'21)*.
[32] Gautam Altekar and Ion Stoica. 2009. ODR: Output-Deterministic Replay for Multicore Debugging. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 193–206. https://doi.org/10.1145/1629575.1629594

[33] Joy Arulraj, Po-Chun Chang, Guoliang Jin, and Shan Lu. 2013. Production-Run Software Failure Diagnosis via Hardware Performance Counters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. Association for Computing Machinery, New York, NY, USA, 101–112. https://doi.org/10.1145/2451116.2451128

[34] Mona Attariyan, MIchael Chow, and Jason Flinn. 2012. X-ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, Hollywood, CA, 307–320.

[35] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *Comput. Surveys* 51, 3 (May 2018), 50:1–50:39. https://doi.org/10.1145/3182657

[36] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. 2006. Framework for Instruction-Level Tracing and Analysis of Program Executions. In *Proceedings of the 2nd International Conference on Virtual Execution Environments* (Ottawa, Ontario, Canada) *(VEE '06)*. Association for Computing Machinery, New York, NY, USA, 154–163. https://doi.org/10.1145/1134760.1220164

[37] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An Infrastructure for Adaptive Dynamic Optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* IEEE Comput. Soc, San Francisco, CA, USA, 265–275. https://doi.org/10.1109/CGO.2003.1191551

[38] Bryan Buck and Jeffrey K. Hollingsworth. 2000. An API for Runtime Code Patching. *Int. J. High Perform. Comput. Appl.* (2000). https://doi.org/10.1177/109434200001400404

[39] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. 2011. Parallel Symbolic Execution for Automated Real-World Software Testing. In *Proceedings of the Sixth Conference on Computer Systems*. https://doi.org/10.1145/1966445.1966463

[40] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 209–224. http://dl.acm.org/citation.cfm?id=1855741.1855756.

[41] N. Chen and S. Kim. 2015. STAR: Stack Trace Based Automatic Crash Reproduction via Symbolic Execution. *IEEE Transactions on Software Engineering* 41, 2 (2015), 198–220. https://doi.org/10.1109/TSE.2014.2363469

[42] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. 2020. Savior: Towards Bug-Driven Hybrid Testing. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1580–1596. https://doi.org/10.1109/SP40000.2020.00002

[43] Trishul M. Chilimbi, Ben Liblit, Krishna Mehra, Aditya V. Nori, and Kapil Vaswani. 2009. HOLMES: Effective Statistical Debugging via Efficient Path Profiling. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, USA, 34–44. https://doi.org/10.1109/ICSE.2009.5070506

[44] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A Platform for In Vivo Multi-Path Analysis of Software Systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA). https://doi.org/10.1145/1950365.1950396

[45] Marcello Cinque, Catello Di Martino, and Alessandro Testa. 2012. Analyzing and modeling the failure behavior of wireless sensor networks software under errors. In *Wireless Communications and Mobile Computing Conference (IWCMC), 2012 8th International*. IEEE, 1136–1141. https://doi.org/10.1109/IWCMC.2012.6314366

[46] Matthew Tan Creti. 2015. *Software and hardware approaches for record and replay of wireless sensor networks*. Ph.D. Dissertation. Purdue University.

[47] Weidong Cui, Marcus Peinado, Sang Kil Cha, Yanick Fratantonio, and Vasileios P. Kemerlis. 2016. RETracer: Triaging Crashes by Reverse Execution from Partial Memory Dumps. In *International Conference on Software Engineering*. https://doi.org/10.1145/2884781.2884844

[48] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Budapest, Hungary) *(TACAS'08/ETAPS'08)*. Springer, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

[49] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen. 2014. Eidetic Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*.

[50] Peter Dinges and Gul Agha. 2014. Targeted Test Input Generation Using Symbolic-Concrete Backward Execution. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. Association for Computing Machinery, New York, NY, USA, 31–36. https://doi.org/10.1145/2642937.2642951

[51] George W Dunlap, Samuel T King, Sukru Cinar, Murtaza A Basrai, and Peter M Chen. 2002. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review* 36, SI (2002), 211–224. https://doi.org/10.1145/1060289.1060309

[52] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. 2001. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Transactions on Software Engineering* 27, 2 (February 2001), 99–123. https://doi.org/10.1109/32.908957

[53] Pedro Fonseca, Cheng Li, and Rodrigo Rodrigues. 2011. Finding Complex Concurrency Bugs in Large Multi-threaded Applications. In *Proceedings of the 6th European Conference on Computer Systems (EuroSys'11)*.

[54] Pedro Fonseca, Cheng Li, Vishal Singhal, and Rodrigo Rodrigues. 2010. A study of the internal and external effects of concurrency bugs. In *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*. https://doi.org/10.1109/dsn.2010.5544315

[55] Pedro Fonseca, Rodrigo Rodrigues, and Björn B. Brandenburg. 2014. SKI: Exposing Kernel Concurrency Bugs Through Systematic Schedule Exploration. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*.

[56] Vijay Ganesh and David L. Dill. 2007. A decision procedure for bit-vectors and arrays. In *CAV*.

[57] Xinyang Ge, Weidong Cui, and Trent Jaeger. 2017. GRIFFIN: Guarding Control Flows Using Intel Processor Trace. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 585–598. https://doi.org/10.1145/3037697.3037716

[58] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. 2009. Debugging in the (Very) Large: Ten Years of Implementation and Experience. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 103–116. https://doi.org/10.1145/1629575.1629586

[59] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. *Queue* 10, 1, Article 20 (January 2012), 8 pages. https://doi.org/10.1145/2090147.2094081

[60] Wei Guan and KJ Ray Liu. 2012. Mitigating error propagation for wireless network coding. *IEEE Transactions on Wireless Communications* 11, 10 (2012), 3632–3643. https://doi.org/10.1109/TWC.2012.083112.112053

[61] Haryadi S Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J Eliazar,

Agung Laksono, Jeffrey F Lukman, Vincentius Martin, et al. 2014. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the ACM Symposium on Cloud Computing*. 1–14. https://doi.org/10.1145/2670979.2670986

[62] Jim Handy. 2019. DRAM Prices Hit Historic Low. https://thememoryguy.com/dram-prices-hit-historic-low/.

[63] Nima Honarmand and Josep Torrellas. 2014. Replay Debugging: Leveraging Record and Replay for Program Debugging. *SIGARCH Comput. Archit. News* 42, 3 (June 2014). https://doi.org/10.1145/2678373.2665737

[64] Derek R. Hower and Mark D. Hill. 2008. Rerun: Exploiting Episodes for Lightweight Memory Race Recording. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA '08)*. IEEE Computer Society, USA, 265–276. https://doi.org/10.1109/ISCA.2008.26

[65] Jeff Huang, Charles Zhang, and Julian Dolby. 2013. CLAP: Recording Local Executions to Reproduce Concurrency Failures. *SIGPLAN Not.* 48, 6 (June 2013). https://doi.org/10.1145/2491956.2462167

[66] Shiyou Huang, Bowen Cai, and Jeff Huang. 2017. Towards Production-Run Heisenbugs Reproduction on Commercial Hardware. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 403–415.

[67] Intel Corporation. 2013. Intel Processor Trace. https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing.

[68] Guoliang Jin, Aditya Thakur, Ben Liblit, and Shan Lu. 2010. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *International Conference on Object Oriented Programming Systems Languages and Applications*. https://doi.org/10.1145/1869459.1869481

[69] Wei Jin and Alessandro Orso. 2012. BugRedux: Reproducing Field Failures for in-House Debugging. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, 474–484. https://doi.org/10.1109/ICSE.2012.6227168

[70] James A. Jones and Mary Jean Harrold. 2005. Empirical Evaluation of the Tarantula Automatic Fault-localization Technique. In *IEEE/ACM International Conference on Automated Software Engineering*. https://doi.org/10.1145/1101908.1101949

[71] Baris Kasikci, Weidong Cui, Xinyang Ge, and Ben Niu. 2017. Lazy Diagnosis of In-Production Concurrency Bugs. In *SOSP*. Shanghai, China. https://doi.org/10.1145/3132747.3132767

[72] Baris Kasikci, Cristiano Pereira, Gilles Pokam, Benjamin Schubert, Malandal Musuvathi, and George Candea. 2015. Failure Sketches: A Better Way to Debug. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. USENIX Association, Kartause Ittingen, Switzerland.

[73] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. 2015. Failure Sketching: A Technique for Automated Root Cause Diagnosis of In-Production Failures. In *SOSP*. Monterey, CA. https://doi.org/10.1145/2815400.2815412

[74] Baris Kasikci, Cristian Zamfir, and George Candea. 2012. CORD: A Collaborative Framework for Distributed Data Race Detection. In *Eighth Workshop on Hot Topics in System Dependability (HotDep 12)*. USENIX Association, Hollywood, CA.

[75] Baris Kasikci, Cristian Zamfir, and George Candea. 2012. Data Races vs. Data Race Bugs: Telling the Difference with Portend. In *ASPLOS*. London, UK. https://doi.org/10.1145/2150976.2150997

[76] Baris Kasikci, Cristian Zamfir, and George Candea. 2013. RaceMob: Crowdsourced Data Race Detection. In *SOSP*. Farmington, PA. https://doi.org/10.1145/2517349.2522736

[77] Baris Kasikci, Cristian Zamfir, and George Candea. 2015. Automated Classification of Data Races Under Both Strong and Weak Memory Models. *ACM Trans. Program. Lang. Syst.* (2015). https://doi.org/10.1145/2734118

[78] Ali Kheradmand, Baris Kasikci, and George Candea. 2014. Lockout: Efficient Testing for Deadlock Bugs.. In *5th Workshop on Determinism and Correctness in Parallel Programming*. Salt Lake City, UT.

[79] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394. https://doi.org/10.1145/360248.360252

[80] Benjamin Robert Liblit. 2004. *Cooperative Bug Isolation*. Ph.D. Dissertation. University of California, Berkeley.

[81] Hongyu Liu, Sam Silvestro, Wei Wang, Chen Tian, and Tongping Liu. 2018. iReplayer: In-situ and Identical Record-and-replay for Multithreaded Applications. *SIGPLAN Not.* 53, 4 (June 2018). https://doi.org/10.1145/3192366.3192380

[82] Peng Liu, Xiangyu Zhang, Omer Tripp, and Yunhui Zheng. 2015. Light: Replay via Tightly Bounded Recording. *SIGPLAN Not.* 50, 6 (June 2015). https://doi.org/10.1145/2737924.2738001

[83] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *ACM SIGPLAN Notices* 40, 6 (June 2005), 190–200. https://doi.org/10.1145/1064978.1065034

[84] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. 2011. Directed Symbolic Execution. In *Proceedings of the 18th International Conference on Static Analysis* (Venice, Italy) *(SAS'11)*. Springer-Verlag, Berlin, Heidelberg, 95–111.

[85] R. Majumdar and K. Sen. 2007. Hybrid Concolic Testing. In *29th International Conference on Software Engineering (ICSE'07)*. 416–426. https://doi.org/10.1109/ICSE.2007.41

[86] Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. 2017. Towards Practical Default-On Multi-Core Record/Replay. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. https://doi.org/10.1145/3037697.3037751

[87] Steve McConnell. 2004. *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA.

[88] Pablo Montesinos, Luis Ceze, and Josep Torrellas. 2008. DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently. In *International Symposium on Computer Architecture*. https://doi.org/10.1145/1394608.1382146

[89] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*.

[90] Satish Narayanasamy, Gilles Pokam, and Brad Calder. 2005. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. *SIGARCH Comput. Archit. News* 33, 2 (May 2005), 284–295. https://doi.org/10.1145/1080695.1069994

[91] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. 2020. AGAMOTTO: How Persistent is your Persistent Memory Application?. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association.

[92] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering Record and Replay for Deployability. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 377–389.

[93] Soyeon Park, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, Shan Lu, and Yuanyuan Zhou. 2009. PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. In *SOSP*. https://doi.org/10.1145/1629575.1629593

[94] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. 2010. PinPlay: A Framework for Deterministic Replay and Reproducible Analysis of Parallel Programs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Toronto, Ontario, Canada) *(CGO '10)*. Association

for Computing Machinery, New York, NY, USA, 2–11. https://doi.org/10.1145/1772954.1772958

[95] Gilles Pokam, Cristiano Pereira, Shiliang Hu, Ali-Reza Adl-Tabatabai, Justin Gottschlich, Jungwoo Ha, and Youfeng Wu. 2011. CoreRacer: A Practical Memory Race Recorder for Multicore x86 TSO Processors. In *IEEE/ACM International Symposium on Microarchitecture*. https://doi.org/10.1145/2155620.2155646

[96] Andrew Quinn, Jason Flinn, and Michael Cafarella. 2018. Sledgehammer: Cluster-Fueled Debugging. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 545–560.

[97] Vaibhav Rastogi, Zhengyang Qu, Jedidiah McClurg, Yinzhi Cao, and Yan Chen. 2015. Uranine: Real-time privacy leakage monitoring without system modification for android. In *International Conference on Security and Privacy in Communication Systems*. Springer, 256–276. https://doi.org/10.1007/978-3-319-28865-9_14

[98] Michiel Ronsse and Koen De Bosschere. 1999. RecPlay: A Fully Integrated Practical Record/Replay System. *ACM Trans. Comput. Syst.* 17, 2 (May 1999). https://doi.org/10.1145/312203.312214

[99] Caitlin Sadowski and Jaeheon Yi. 2014. How Developers Use Data Race Detection Tools. In *Workshop on Evaluation and Usability of Programming Languages and Tools*. https://doi.org/10.1145/2688204.2688205

[100] Swarup Kumar Sahoo, John Criswell, Chase Geigle, and Vikram Adve. 2013. Using Likely Invariants for Automated Software Fault Localization. *SIGPLAN Not.* 48, 4 (March 2013). https://doi.org/10.1145/2490301.2451131

[101] Tony Savor, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. 2016. Continuous deployment at Facebook and OANDA. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 21–30. https://doi.org/10.1145/2889160.2889223

[102] Julian Schütte, Dennis Titze, and José María De Fuentes. 2014. Appcaulk: Data leak prevention by injecting targeted taint tracking into android apps. In *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 370–379. https://doi.org/10.1109/TrustCom.2014.48

[103] Koushik Sen. 2007. Concolic Testing. In *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering* (Atlanta, Georgia, USA) *(ASE '07)*. Association for Computing Machinery, New York, NY, USA, 571–572. https://doi.org/10.1145/1321631.1321746

[104] Matthew Tancreti, Mohammad Sajjad Hossain, Saurabh Bagchi, and Vijay Raghunathan. 2011. Aveksha: A hardware-software approach for non-intrusive tracing and profiling of wireless embedded systems. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems (Sensys)*. ACM, 288–301. https://doi.org/10.1145/2070942.2070972

[105] Matthew Tancreti, Vinaitheerthan Sundaram, Saurabh Bagchi, and Patrick Eugster. 2015. TARDIS: Software-only system-level record and replay in wireless sensor networks. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks*. ACM, 286–297. https://doi.org/10.1145/2737095.2737096

[106] David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. 2018. Chopped symbolic execution. In *Proceedings of the 40th International Conference on Software Engineering*. 350–360. https://doi.org/10.1145/3180155.3180251

[107] Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou. 2007. Triage: Diagnosing Production Run Failures at the User's Site. *SIGOPS Oper. Syst. Rev.* 41, 6 (October 2007). https://doi.org/10.1145/1294261.1294275

[108] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2011. DoublePlay: Parallelizing Sequential Logging and Replay. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. https://doi.org/10.1145/1950365.1950370

[109] Peipei Wang, Hiep Nguyen, Xiaohui Gu, and Shan Lu. 2016. RDE: Replay DEbugging for Diagnosing Production Site Failures. In *2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS)*. 327–336. https://doi.org/10.1109/SRDS.2016.050

[110] Yan Wang, Harish Patil, Cristiano Pereira, Gregory Lueck, Rajiv Gupta, and Iulian Neamtiu. 2014. DrDebug: Deterministic Replay Based Cyclic Debugging with Dynamic Slicing. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*. https://doi.org/10.1145/2544137.2544152

[111] Xinyang Ge Weidong Cui, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. 2018. REPT: Reverse Debugging of Failures in Deployed Software. In *OSDI*.

[112] Mingyuan Xia, Lu Gong, Yuanhao Lyu, Zhengwei Qi, and Xue Liu. 2015. Effective real-time android application auditing. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 899–914. https://doi.org/10.1109/SP.2015.60

[113] Jun Xu, Dongliang Mu, Xinyu Xing, Peng Liu, Ping Chen, and Bing Mao. 2017. Postmortem program analysis with hardware-enhanced post-crash artifacts. In *26th USENIX Security Symposium (USENIX Security 17)*. 17–32.

[114] Cristian Zamfir and George Candea. 2010. Execution Synthesis: A Technique for Automated Software Debugging. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*. https://doi.org/10.1145/1755913.1755946

[115] Cristian Zamfir, Baris Kasikci, Johannes Kinder, Edouard Bugnion, and George Candea. 2013. Automated Debugging for Arbitrarily Long Executions. In *14th Workshop on Hot Topics in Operating Systems (HotOS XIV)*. USENIX Association, Santa Ana Pueblo, NM.

[116] Andreas Zeller and Ralf Hildebrandt. [n.d.]. Simplifying and Isolating Failure-Inducing Input. *Transactions on Software Engineering* ([n. d.]). https://doi.org/10.1109/32.988498

[117] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. 2011. ConSeq: Detecting Concurrency Bugs Through Sequential Errors. *SIGARCH Comput. Archit. News* 39, 1 (March 2011). https://doi.org/10.1145/1950365.1950395

[118] Yongle Zhang, Serguei Makarov, Xiang Ren, David Lion, and Ding Yuan. 2017. Pensieve: Non-intrusive failure reproduction for distributed systems using the event chaining approach. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 19–33. https://doi.org/10.1145/3132747.3132768

[119] Yongle Zhang, Kirk Rodrigues, Yu Luo, Michael Stumm, and Ding Yuan. 2019. The inflection point hypothesis: a principled debugging approach for locating the root cause of a failure. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 131–146. https://doi.org/10.1145/3341301.3359650

[120] Daniele Zuddas, Wei Jin, Fabrizio Pastore, Leonardo Mariani, and Alessandro Orso. 2014. MIMIC: Locating and Understanding Bugs by Analyzing Mimicked Executions. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. Association for Computing Machinery, New York, NY, USA, 815–826. https://doi.org/10.1145/2642937.2643014