Execution Synthesis: A Technique for Automated Software Debugging

Cristian Zamfir and George Candea

School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Abstract

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

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

Our technique requires no runtime tracing or program modifications, thus incurring no runtime overhead and being practical for use in production systems. We evaluate ESD—a debugger based on execution synthesis—on popular software (e.g., the SQLite database, ghttpd Web server, HawkNL network library, UNIX utilities): starting from mere bug reports, ESD reproduces on its own several real concurrency and memory safety bugs in less than three minutes.

Categories and Subject Descriptors D.2.2.5 [*Software Engineering*]: Testing and Debugging—Debugging aids

General Terms Reliability

Keywords automated debugging, symbolic execution

1. Introduction

Developing software is a challenging endeavor, and debugging it is even harder. With increasing parallelism in both hardware and software, the classic problem of bugs in sequential execution is now being compounded by concurrency bugs and other hard-to-reproduce behavior.

To fix a bug, developers traditionally try to reproduce it and observe its manifestation in a debugger. Alas, this approach is often challenging, especially for concurrency bugs—in a recent survey, almost 75% of respondents considered reproducibility to be hard or very hard [17]. There are multiple reasons for this: First, complex sequences of low-probability events (e.g., a particular thread schedule) are required for a concurrency bug to manifest, and programmers do not have the means of directly controlling such events. Second, the probe effect—unintended alteration of program behavior through the introduction of instrumentation and breakpoints [15]—can make concurrency bugs "vanish" when hunted with a debugger. Third, variations in the OS and runtime environment (e.g., kernel or library version differences) may make it practically impossible to reproduce a bug exactly as it occurred at the end user's site.

Modern software is increasingly parallel, which makes it that much more difficult to debug. Currently, more than 70% of concurrency bugs take many days or even months to analyze and diagnose [17]. This increases the cost of maintaining and evolving parallel software, and consumers must wait a long time before fixes become available. Moreover, the large amount of guesswork involved in debugging leads to error-prone patches, with many concurrency bug fixes either introducing new bugs or, instead of fixing the underlying bug, merely decreasing its probability of occurrence [27]. Increasingly parallel hardware causes software to experience increasingly concurrent executions, making latent bugs more likely to manifest, yet no easier to fix.

In this paper, we introduce *execution synthesis*, a technique for automatically finding "explanations" for hard-to-reproduce bugs. Execution synthesis takes as input a program plus a bug report and produces an execution of that program that causes the reported bug to manifest deterministically. Our technique requires no tracing or execution recording at the end user's site, making it well suited for debugging long-running, performance-sensitive software, like

Web servers, database systems, e-mail servers, application servers, game servers, etc.

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

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

We prototyped the proposed technique in ESD, a tool that automatically analyzes common elements provided in bug reports (coredumps, stack traces, etc.), synthesizes an execution that leads to the reported misbehavior, and allows developers to play back this execution in a debugger. ESD is practical and scales to real systems. For example, it takes less than three minutes to synthesize an execution for a deadlock bug in SQLite, an embedded database engine with over 100 thousand lines of C/C++ code [32] used in Firefox, Skype, Mac OS X, Symbian OS, and other popular software [36].

In this paper, we give an overview of ESD (§2), describe sequential path synthesis (§3), thread schedule synthesis (§4), and execution playback (§5). We then present the ESD implementation (§6), an experimental evaluation (§7), discussion (§8), related work (§9), and conclusion (§10).

2. Overview

The input to ESD consists of the coredump associated with a bug report and the program the developer is trying to debug. ESD then outputs a trace that can be played back in a debugger with the ESD runtime environment. Given a class of bugs, ESD can extract from the coredump all information it needs to find a way to reproduce that class of bugs (e.g., for debugging deadlocks, it extracts the call stacks of the deadlocked threads). At the end user site, the buggy program is run normally, i.e., without instrumentation or special environments, no annotations, and no debug symbols.

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

Listing 1: Example of a deadlock bug. Two threads executing this code may deadlock if the condition on line 10 is true and one thread is preempted right after executing statement 11.

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

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

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

We use the example in Listing 1 to illustrate how execution synthesis works. In this example, two threads executing *CriticalSection()* concurrently may deadlock if the condition on line 10 is true. An execution in which the threads deadlock is the following: one thread runs up to line 11 and is preempted right after the unlock call, then a second thread executes up to line 9 and blocks waiting for mutex M_2, then the first thread resumes execution and blocks waiting for M_1 on line 12. The program is now deadlocked.

The bug report for this deadlock would likely contain the final stack trace of each thread, but would be missing several important pieces of information needed for debugging, such as the return values of external calls—*getchar()* and *getenv()*—and the interleaving of threads. ESD "fills in the

blanks" and infers two key aspects of the buggy execution: a program path in each thread from the beginning to where the bug occurs, and a schedule that makes this path feasible.

To synthesize the path through the program for each thread, ESD first statically analyzes the program and then performs a dynamic symbolic analysis. In the static analysis phase, ESD computes the control flow graph (CFG) and performs intra- and inter-procedural data flow analysis to identify the set of paths through the graph that reach the bug location. For the example in Listing 1, ESD's static analysis identifies two paths that could lead the first thread to statement 12: $1 \to 2 \to 3 \to 4 \to 7 \to ... \to 12$ and $1 \to 3 \to 4 \to 7 \to ... \to 12$, both of which require *getenv("mode")* to return a string starting with *'Y'*. Since ESD cannot decide statically whether statement 2 is part of the path to statement 12 or not, both alternatives are considered possible. For the second thread, a similar analysis finds four possible paths to statement 9.

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

Symbolic execution suffers from the notorious "path explosion" problem [3]. Execution synthesis therefore incorporates a number of techniques to cope with the large number of paths that typically get explored during symbolic execution. The foremost of these techniques is the use of a *proximity heuristic* to guide symbolic execution on those paths most likely to reach the bug. ESD uses the CFG to estimate the distance (in basic blocks) from any given node in the CFG to the bug location. Using this estimate, the exploration of paths is steered toward choices that have a shorter distance to the bug, thus enabling ESD to find a suitable path considerably faster than mere symbolic execution.

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

The novelty of execution synthesis is its ability to reconstruct bug-bound executions without recording program inputs or events that induce non-determinism in program execution, and without requiring any alteration of the program or its environment. This makes ESD practical for real software running in performance-critical environments.

3. Sequential Path Synthesis

In this section we describe how ESD finds a sequential bug-bound execution path within each thread of a program: first it identifies a search goal (§3.1), then performs static analysis (§3.2), and finally a dynamic search (§3.3).

3.1 Determining the Goal

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

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

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

3.2 Static Phase of Path Search

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

ESD first computes the full inter-procedural CFG of the program and applies all the optimizations that were applied to the program version that experienced the reported bug. It performs alias analysis and resolves as many function pointers as possible, replacing them with the corresponding direct calls; this can substantially simplify the CFG. ESD can handle the case when not all function pointers are resolved, though it may lose precision. In this case, subsequent analyses will still be sound and complete, but may take longer to

execute. ESD also eliminates all basic blocks that cannot be reached from the start of the program (i.e., dead code) and all basic blocks from which there is no path to B.

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

ESD identifies the critical edges by starting from the goal block and working backward, in a manner similar to backward slicing [37]. Starting from B, the algorithm finds at each step a predecessor node in the CFG. For each such node, if only one of its outgoing edges can lead to B, then that edge is marked as critical. The current version of ESD can only explore one predecessor for each node, so as soon as a block with multiple predecessors is found, the marking of critical edges stops and ESD moves to the next step. A more effective, but potentially slower, algorithm would explore all predecessors and identify multiple sets of critical edges.

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

To determine intermediate goals, ESD relies on the critical edges. For each critical edge, the corresponding branch condition and its desired value (true or false) are retrieved. For each variable $x, y, \ldots$ in the branch condition, ESD finds the sets of instructions $\mathbb{D}_x, \mathbb{D}_y, \ldots$ that are reaching definitions [1] of the variable. It then looks for combinations of instructions from $\mathbb{D}_x, \mathbb{D}_y, \ldots$ that would give the branch condition the desired value, i.e., instructions for which there is a static guarantee that, if they were executed, the critical edge would be followed. When such a combination is found, the basic blocks that contain the reaching definitions in the combination are marked as intermediate goals. Should more than one combination exist, the corresponding sets of instructions are marked as disjunctive sets of intermediate goals.

While condition C in goal $<B,C>$ is not explicitly used in the above algorithms, ESD does use C in its analyses. To a first degree of approximation, basic block B is replaced in the program with a statement of the form *if (C) then BugStrikes*, and the static analysis phase runs on the transformed program, with *BugStrikes* as the goal basic block. By finding a path along which the program executes *BugStrikes*, ESD will have found a path that executes block B while condition C holds, i.e., a path that reaches the original goal $<B,C>$. Some conditions, however, cannot be readily expressed in this way. For example, a deadlock condition is a property

that spans the sequential execution paths of multiple threads. For such cases, ESD has special-case handling to check condition C during the dynamic phase; this will be further described in §4.

3.3 Dynamic Phase of Path Search

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

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

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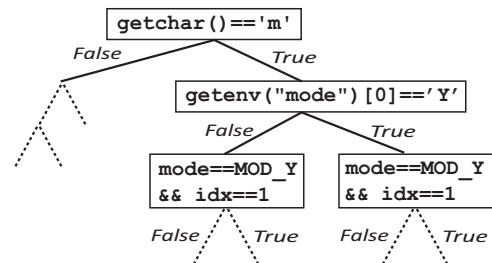

Figure 1: Execution tree for the example in Listing 1.

An execution state consists of a program counter, a stack, and an address space. Such states can be "executed," i.e., the instruction pointed to by the program counter is executed and may cause corresponding updates to the state's stack and

address space. We chose this representation for compatibility with the Klee symbolic execution engine [6], since the ESD prototype relies on (a modified version of) Klee.

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

There are two key challenges, though: the execution tree grows very fast (the notorious path explosion problem [3]), and determining the satisfiability of constraints at every branch condition, in order to determine which of the branches are feasible, is CPU-intensive. These two properties make symbolic execution infeasible for large programs. For ESD to be practical, the search for a path to the goal must be very focused: the less of the tree is expanded and searched, the less CPU and memory are consumed.

ESD uses three key techniques to focus the search: First, it uses statically derived intermediate goals (§3.2) as anchor points in the search space, to divide a big search into several small searches. Second, ESD leverages the information about critical edges (§3.2) to promptly abandon during symbolic execution paths that are statically known to not lead to the goal. Third, ESD orders the priority queue of execution states based on each state's estimated proximity to the next intermediate goal. In this way, the search is consistently steered toward choosing and exploring executions that appear to be more likely to reach the intermediate goal soon.

We refer to this latter technique as proximity-guided search and describe it in the next section. Due to space constraints, we omit details on path elimination based on critical edges and on the evaluation of complex goal conditions.

3.4 Proximity-Guided Path Search

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

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

When the goal is inside the currently executing procedure, function *distance* computes the proximity. If there are no calls to other procedures, the distance is the length of the path to the goal with the fewest number of instructions (lines 9-12). If, however, any of the instructions along the

---

Algorithm 1: Heuristic Proximity to Goal (Simplified)

Input: Execution state S, goal G (potentially intermediate)
Output: Estimate of S's distance to G

1 $d_{min} \leftarrow distance(S.pc, G)$
2 **if** $d_{min} = \infty$ **then**
3   **foreach** *procedure* $\pi \in S.callStack$ **do**
4          $I_{ra} \leftarrow$ instruction to be executed after π call returns
5          $d \leftarrow dist2ret(S.pc) + distance(I_{ra}, G) + 1$
6          $d_{min} \leftarrow \min(d_{min}, d)$

7 **return** d_{min}

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

16 **return** d_{min}

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

---

path are calls to other procedures, then ESD factors in the costs of executing those procedures by adding to the path length the cost of the calls (lines 13-14).

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

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

Each execution state S in ESD has n distances associated with it, corresponding to S's distance to the $G_1, ..., G_{n-1}$ intermediate goals inferred through static analysis and to the final goal $G_n = B$. The closer an intermediate goal truly

is, the more accurate the distance estimate. ESD maintains n "virtual" priority queues $Q_1, ..., Q_n$, which provide an ordering of the state's distance to the respective goal: the state at the front of Q_i has the shortest estimated distance to goal G_i. We refer to these queues as "virtual" because the queue elements are just pointers to the execution states. Each state can be found on each of the virtual queues.

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

Once a state has reached the final goal (i.e., $S.pc = B$) the search completes: ESD has found a feasible path that explains the buggy behavior. ESD solves the constraints that accumulated along the path and computes all the inputs required for the program to execute that path, in a way similar to automated test generation [6, 18]. ESD relies on symbolic models of the filesystem [6] and the network stack to ensure all symbolic I/O stays consistent.

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

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

We found that the three techniques of focusing the search—proximity-based guidance, the use of intermediate goals, and path abandonment based on critical edges—can speed up the search by several orders of magnitude compared to other search strategies (§7).

Nevertheless, further techniques could be employed to improve the search strategy. For instance, if the initialization phase of the program can be reproduced by other means, such as from an existing test case (ESD does not require existing test cases), ESD could run concretely the initialization

phase and automatically switch from concrete to symbolic execution later in the execution of the program [8, 19], thus reducing execution synthesis time.

4. Thread Schedule Synthesis

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

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

Preemption points of interest are before and after concurrency-sensitive operations: load instructions, store instructions, and calls to synchronization primitives. While conceptually the sequential path synthesis phase is separate from schedule synthesis, ESD overlaps them and synthesizes one "global" sequential path, by exploring the possible thread preemptions as part of the sequential path synthesis.

Just as for sequential path synthesis, ESD employs heuristics to make the search for a thread schedule efficient. It is substantially easier to choose the right heuristic if ESD knows the kind of concurrency bug it is trying to debug, and this can often be inferred from the coredump. ESD can synthesize schedules for deadlocks (§4.1) and data races (§4.2).

4.1 Deadlocks

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

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

For the deadlock example in Listing 1, a coredump would indicate call stacks that (in stylized form) would look like

---

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

$T_1 : [\dots 12]$ and $T_2 : [\dots 9]$, meaning that thread T_1 was blocked in a *lock* call made from line 12, while T_2 was blocked in a *lock* call made from line 9. The call stack shows the call sequence that led to the lock request that blocked the thread. This lock request appears in the last frame, and we refer to it as the thread's *inner lock*. We call *outer locks* those that are already held by the deadlocked thread. This naming results from the fact that a deadlock typically arises from nested locks [27], where an inner lock is requested while holding an outer lock. At the time of deadlock, the acquisitions of the outer locks are not visible in the call stack anymore.

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

Any time ESD encounters a *lock* or *unlock* operation, it forks off an execution state in which the current thread is preempted. The running execution state maintains a pointer to that forked state, in case ESD needs to return to it to explore alternate schedules. More generally, we augment each execution state S with a map $\mathbb{K}_S : mutex \rightarrow execution\ state$. An element $<M, S'> \in \mathbb{K}_S$ indicates that S is exploring one schedule outcome connected with the acquisition of mutex M, while S' is the starting point for exploring alternative scheduling outcomes. A snapshot entry $<M, S'>$ is deleted as soon as M is unlocked. The size of $\mathbb{K}_S$ is therefore bounded by the program's maximum depth of lock nesting. ESD leverages Klee's copy-on-write mechanisms at the level of memory objects to maximize memory sharing between execution states (§6.1). As a result, snapshots are cheap.

We augment execution state S with $S.scheduleDistance$, an estimate of how much context switching is required to reach the deadlock. For the case of two-thread deadlocks, this schedule distance can take one of two values: *far* or *near*. ESD computes a weighted average of the path distance estimate (§3.4) and the schedule distance estimate, with a heavy bias toward schedule distance. The virtual state priority queues are kept sorted by this weighted average. The bias ensures that low-schedule-distance execution states are selected preferentially over low-path-distance states.

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

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

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

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

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

We illustrate on the example from Listing 1, for which the search goals are $T_1 :<12, T2@9>$ and $T_2 :<9, T_1@12>$.

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

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

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

During schedule synthesis, ESD automatically detects mutex deadlocks by using a deadlock detector based on a re-

source allocation graph [22]. Deadlocks involving condition variables are more challenging to detect automatically—inferring whether a thread that is waiting on a condition variable will eventually be signaled by another thread is undecidable in general. However, ESD can check for the case when no thread can make any progress and, if all threads are waiting either to be signaled, to acquire a mutex, or to be joined by another thread, then ESD identifies the situation as a deadlock.

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

4.2 Data Races

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

ESD uses a dynamic data race detection algorithm similar to Eraser [34] and inserts preemption points wherever potentially harmful data races [30] are detected. Normally, dynamic data race detectors can miss races, because they only observe execution paths exercised by the given workload. However, by using symbolic execution, ESD can expose to the detection algorithm an arbitrary number of different execution paths, independently of workload.

In order to avoid unnecessary thread schedules early in the execution of the program, ESD uses an additional heuristic. It identifies the longest common prefix of the final thread call stacks in the coredump and inserts preemptions only in executions whose call stacks contain this prefix. If the last frame of the common prefix corresponds to procedure p, then p is set as an intermediate goal for each thread—for the example in Listing 1, p would be the entry into *CriticalSection*. Once all threads reach their respective goals (or when no threads can make any further progress), ESD's scheduler starts forking execution states on fine-grain scheduling decisions and checks for data races. We found this heuristic to work well in practice, especially considering that many applications run the same code in most of their threads.

For simplicity and clarity, we assume a sequential consistency model for memory shared among threads, an assumption present in most recent systems dealing with concurrency bugs (such as Chess [29]). An immediate consequence is that each machine instruction is assumed to execute atomically with respect to memory, which simplifies the exploration process. In the case of shared memory with relaxed

consistency, ESD could miss possible paths, but will never synthesize an infeasible execution leading to a bug goal.

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

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

5. Execution Playback

Once the execution synthesizer (§3–§4) reaches its goal, it generates an execution file containing the playback information. This file is read by the ESD playback environment—the subject of this section. The goal of playback is to provide developers an explanation of the bug symptoms, in a way that allows them to inspect the execution with a classic debugger.

5.1 Synthesized Execution File

In order to achieve the highest possible fidelity, ESD plays back a reported bug using the native binary that was run by the end user. The synthesized execution file contains concrete values for all input parameters, all interactions with the external environment (e.g., through system calls), and the complete thread schedule. For all program input, including that coming from the environment, ESD solves the constraints found during execution synthesis and produces corresponding concrete values (such as *getchar()=‘m’* and *getenv("mode")[0]=‘Y’*). This is identical to what automated test generators do (like DART [18] and Klee [6]), except that these test generators do not produce thread schedules.

ESD saves the thread schedule of a synthesized execution in the form of happens-before relations [25] between specific program instructions. ESD can also save a strict schedule in the file, by recording the exact instructions on which the context was switched during synthesis, along with the switched thread identifiers. During the playback phase (§5.2), this strict schedule will enforce literally a serial execution of the program, whereas the schedule based on happens-before relations allows playback to proceed with the same degree of parallelism as the original execution.

5.2 Playback Environment

In order to steer a program into following the steps reflected in the synthesized execution file, ESD relies on two components: one for input playback and one for schedule playback. For input playback, ESD reuses Klee's driver, which takes from the trace the values of command line arguments and

passes them to the program. This driver also intercepts via a custom library the calls to the environment and returns the inputs from the execution file. To preserve the consistency of the execution, ESD also relies on Klee's symbolic filesystem and network models.

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

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

Developers run the buggy program in the playback environment and can attach to it with a debugger at any time. They can repeat the execution over and over again, place breakpoints, inspect data structures, etc. After fixing the bug, ESD can be re-run, to check whether there still exists a path to the bug. This is particularly helpful for concurrency bugs, where patches often do not directly fix the underlying bug, but merely decrease its probability of occurrence [27]. If ESD can no longer synthesize an execution that triggers the bug, then the patch can be considered successful.

6. Implementation

The ESD prototype currently works for C programs, and we verified that it works seamlessly with the gdb debugger. For symbolic execution, we adopted Klee [6] and extended it in several ways; the most important one is support for multi-threaded symbolic execution. After describing this extension in brief (§6.1), we provide a few details related to the implementation of synthesis and playback (§6.2).

6.1   Multi-threaded Symbolic Execution

Klee [6] is a symbolic virtual machine designed for single-threaded programs. To allow ESD to explore various thread schedules, we added support for POSIX threads.

Our extended version supports most common operations, such as thread, mutex and condition variable management, including thread-local storage functions. The new Klee thread functions are handlers that hijack the program's calls to the real threads library. To create a simulated thread, ESD

resolves at runtime the associated start routine and points the thread's program counter to it, creates the corresponding internal thread data structures and a new thread stack, and adds the new thread to the ESD scheduler queue. ESD also maintains information on the state of mutex variables and on how threads are joined.

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

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

6.2   Execution Synthesis and Playback

For the execution synthesis phase, ESD compiles the program to LLVM bitcode [26], a low-level instruction set in static single assignment form. We chose LLVM because Klee operates on LLVM and because the associated compiler infrastructure provides rich static analysis facilities. LLVM provides load and store instructions up to word-level granularity, thus providing sufficient control for ESD to synthesize thread schedules that reproduce the desired data races.

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

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

Our current prototype can play back deadlocks involving mutexes and condition variables with negligible overhead. We are in the process of implementing playback for data races using PIN [28] for binary instrumentation; ESD can then control the interleaving of threads' memory accesses.

7. Evaluation

In this section we evaluate ESD's effectiveness in reproducing real bugs in real systems (§7.1). We also compare ESD to other approaches (§7.2) and analyze ESD's performance (§7.3). All reported experiments ran on a 2 GHz quad-core Xeon E5405 CPU with 4GB of RAM, under 32-bit Linux. ESD had a total of 2GB of memory available.

7.1 Real Bugs in Real Systems

ESD succeeds in automating the debugging of real systems code. Table 1 shows examples of the programs we ran under ESD, ranging in size across three orders of magnitude: from over 100,000 LOC (SQLite) down to 100 LOC (mkfifo).

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

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

One class of bugs results in hangs. For example, bug #1672 in SQLite 3.3.0 is a deadlock in the custom recursive lock implementation. SQLite, an embedded database engine, is a particularly interesting target, since it has a reputation for being highly reliable and the developer-built test suites achieve 99% statement coverage [36]. This makes us believe that the remaining bugs are there because they are particularly hard to reproduce. Another hang bug appears in HawkNL 1.6b3, a network library for distributed games. When two threads happen to call *nlClose()* and *nlShutdown()* at the same time on the same socket, HawkNL deadlocks.

Other bugs result in crashes. A security vulnerability in the *ghttpd* Web server is caused by a buffer overflow when processing the URL for GET requests [16]. The overflow occurs in the `vsprintf` function when the request is written to the log. A bug in the *paste* UNIX utility causes an invalid `free` for some inputs. The four bugs in the *tac*, *mkdir*, *mknod*, and *mkfifo* UNIX utilities are all segmentation faults, with the last three occurring only on error handling paths. The UNIX utilities bugs are reported in [6].

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

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

It is worth noting that ESD is effective not only for programs, but also for shared libraries, such as SQLite and HawkNL. Debugging libraries often has higher impact than debugging individual programs, because bugs inside libraries affect potentially many applications. For example, SQLite is used in Firefox, iPhone, Mac OS X, McAfee antivirus software, Nokia's Symbian OS, PHP, Skype, and others [36]. In order to reproduce library bugs with ESD, one writes a program that exercises the library through the suspected buggy entry points; ESD then analyzes and symbolically executes these driver programs along with the library.

7.2 Comparison to Alternate Approaches

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

The first approach to reproduce the bugs is brute force trial-and-error. To measure objectively, we ran several series of stress tests and random input testing for several hours. Neither of these efforts caused any of the bugs in Table 1 to manifest.

Bug finding tools, like Klee [6] and Chess [29], can also be used to find paths to bugs—these tools produce test cases meant to reproduce the found bugs. Such a comparison is not entirely fair, for several reasons. On the one hand, ESD can synthesize execution paths for bugs that occur in production, away from ESD, whereas bug finding tools can only reproduce bugs that occur under their own close watch. On the other hand, bug finding tools are not guided toward a specific bug; their goal is to find previously unknown bugs and typically aim for high code coverage. Nevertheless, since we are not aware of other execution synthesis tools, we analyze the efficiency of ESD's search via this comparison.

We extended Klee with support for multi-threading and implemented Chess's preemption-bounding approach for exploring multi-threaded executions [29]. We name the resulting tool KC—a hybrid system that embodies the Klee and Chess techniques. We compare ESD to two different KC search strategies inherited directly from Klee: *DFS*, which can be thought of as equivalent to an exhaustive search, and *RandomPath*, a quasi-random strategy meant to maximize global path coverage. We augmented the corresponding strategies to encompass all active threads and limit preemptions to two, as done in [29].

We ran both Klee and KC to find a path leading to each of the sample bugs in Table 1. After running for over one hour for each bug, neither tool found a path.[2] In order to still have a practical baseline for comparison, we introduced four null-pointer-dereference bugs in the *ls* UNIX utility, for which KC does find a path in less than one hour. The *ls* utility has 3 KLOC.

Figure 2 shows the time it takes ESD to find a path vs. KC's two different search strategies. ESD is one to several orders of magnitude faster at finding the path to the target

---

[2] The five bugs in UNIX utilities were originally found with Klee and reported in [6]. Our experiments did not find them perhaps due to differences in the Klee version and search strategies. ESD was built on top a Klee code snapshot that was generously provided to us by its authors in Aug. 2008.

bug. We do not know if KC would eventually find a path to the bugs in Table 1 and, if it did, how long that would take.
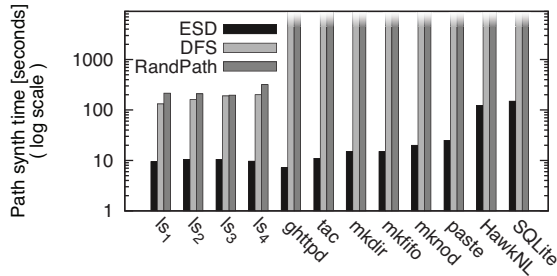

Figure 2: Comparison of time to find a path to the bug: ESD vs. the two variants of KC. Bars that fade at the top indicate KC did not find a path by the end of the 1-hour experiment.

7.3   Performance Analysis

In order to analyze the impact of this paper's contributions—zero-tracing execution path synthesis and the corresponding heuristics—we developed a microbenchmark, called BPF. The main purpose of BPF is to profile ESD without the measurements being influenced by environment interactions, such as library calls or system calls. For the more general case, BPF can serve as a way to compare the performance of automated debugging tools like ESD.

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

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

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

Figure 3 shows the time to synthesize an execution as a function of program complexity (in terms of branches). We find that ESD's performance varies roughly as expected; one exception is the jump from 2^8 to 2^9 branches—we suspect that structural features of the larger program presented an extra challenge for ESD's heuristics. Nevertheless, ESD performs well, keeping the time to synthesize a path to under 2 minutes, which is a reasonable amount of time for a developer to wait. We also included, for reference, the time taken by KC with the *RandPath* search strategy; it found a path within one hour only for the two simplest benchmark-generated programs. The *DFS* strategy did not find any path.
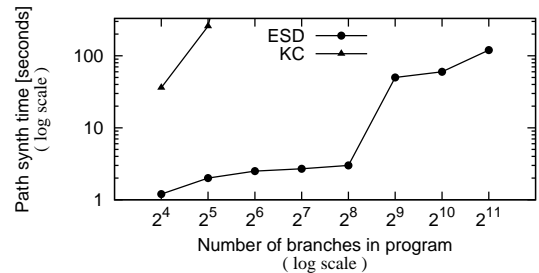

Figure 3: Synthesizing a bug-bound path for programs of varying complexity with ESD and KC.

An alternate perspective on these results is to view them in terms of program size. Figure 4 shows the same data, but in terms of KLOC in the generated programs.
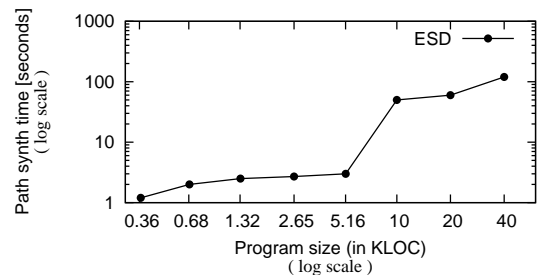

Figure 4: Synthesis time as a function of program size.

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

8.   Discussion

In this section we discuss ESD's usage models, its limitations, and how ESD can complement static analysis tools.

Usage Models: We envision ESD being used in at least two modes: developers can employ it directly during debugging, or it can be used for automated bug triage and deduplication.

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

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

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

```
esdplay <orig program binary> <synthetic exec file>
```

Another usage model is one in which ESD is part of an automated bug reporting/triage system, where each incoming bug report is passed through ESD, to produce a deterministic execution file that gets attached to the bug report. At the same time, ESD can be used to automatically identify reports of the same bug: if two synthesized executions are identical, then they correspond to the same bug.

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

Execution synthesis may not always be able to reproduce a bug. Symbolic execution has inherent limitations when solving complex constraints, such as finding a string m for which $hash_{SHA-2}(m)$=0×f8e28ed7b8db9a. As a result, ESD would have a hard time finding a program input that would exercise the then-branch of an if statement involving the above condition. If there is a bug that manifests only when this condition holds, ESD will likely not be able to reproduce it—doing so would amount to breaking the SHA-2 cryptographic hash function [31].

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

ESD currently relies on a 32-bit version of Klee, which means that it can address at most 4 GB of memory. For large real programs, this can cause ESD to run out of memory before finding the desired path. We have recently ported Klee to 64-bit architectures, but have not yet ported ESD to the new version of Klee. Once we do so, we expect ESD to be able to make use of the increasing amounts of physical memory available on modern machines.

Complementing Static Analysis Tools with ESD: We see a clear opportunity in using ESD to weed out false positives generated by static analysis tools, such as race and deadlock checkers [14]. Static analysis is powerful and typically complete, but these properties come at the price of soundness: static analyzers commonly produce large numbers of false positives, and selecting the true positives becomes a laborious human-intensive task. Fortunately, the output of such tools is already similar to a bug report, so ESD could be used "out of the box" to validate each suspected bug: if ESD finds a path to the bug, then it is a true positive. We plan to explore in future work the synergy between such tools and our execution synthesis technique.

9. Related Work

In this section we review related work. Some of these prior works provided inspiration for ESD, while others are related by virtue of addressing similar problems. We broadly divide the body of related work into bug finding tools (some of which focus on inferring inputs, while others focus on finding schedules) and record/replay systems.

Bug Finding Tools – Inputs: There is a rich body of work focused on discovering bugs in programs [6, 14, 18, 19, 34], with recent tools typically employing symbolic execution. ESD builds upon techniques developed for these systems, most notably Klee [6].

In combining static analysis with symbolic execution, we were inspired by a series of systems [7, 10, 11] which compute inputs that take a program to a specified undesired state, such as the location of a crash. Unlike ESD, these systems are targeted at program states that can be deterministically reached when given the right program arguments. ESD was specifically motivated by the difficulty of reproducing elusive "non-deterministic" bugs, hence our emphasis on inferring not only program arguments, but also inputs from the program's environment and scheduling decisions. Moreover, these prior tools require recording of certain program inputs and/or events; in ESD we go to the extreme of zero program tracing, in order to be practical for production systems.

Static analysis and symbolic execution were used to create vulnerability signatures [4] and to show that it is possible to automatically create exploits from patches [5]. ESD is similar to this work in that it aims to create inputs that execute the program toward a certain vulnerability. However, ESD addresses bugs more broadly than just input validation bugs and is able to handle multi-threaded programs.

Bug Finding Tools – Schedules: Even though program testing is different from debugging, we drew inspiration for schedule synthesis from tools that search for concurrency bugs, like Chess [29] and DeadlockFuzzer [21]. Still, there exist major differences. These tools exercise target programs in a special environment and, when a bug occurs, the tools are able to replay those bugs. In contrast, ESD reproduces bugs discovered in the field by end users, in which case requiring the program to run in a special setting is not feasible. Chess and DeadlockFuzzer also require the existence of a test case that provides all required program input, whereas ESD automatically infers this input.

Another important difference appears in the use of heuristics. Chess, for example, employs a technique called iterative context bounding (ICB) [29]. ICB assumes that prioritizing executions with fewer context switches is an efficient way to find concurrency bugs, so Chess repeatedly runs an existing test case, each time with a different schedule, and limits the total number of possible context switches, as in ICB. When searching for a specific bug, we found ESD's approach to be much faster. Also, ESD achieves scalability without having to bound the number of context switches. We repeat, however, that Chess's goals are different from ESD's, so direct performance comparisons must be done carefully.

Similarly to RaceFuzzer [35], ESD dynamically detects potential data races and performs context switches before memory accesses suspected to be in a race. However, ESD's approach is more precise, because it is targeted at a specific bug and uses checkpoints to explore alternate thread interleavings, unlike RaceFuzzer's random scheduler. Moreover, by using symbolic execution, ESD can achieve substantially higher coverage for data race detection.

Record/Replay: A classic way of reproducing bugs is by using whole-system replay: the application is run inside a specialized virtual machine, which captures all relevant details of an execution, enabling it to be replayed later [12, 13]. This approach works well for bugs that occur relatively frequently. However, concurrency bugs in production are rare occurrences, so the performance and space overhead of always-on recording of the entire execution offers less payback. Reverse debugging [24] uses VMs to travel back and forth in an execution, which is useful in dealing with hard-to-reproduce bugs; this approach typically incurs prohibitive recording overhead for bugs that occur infrequently. In contrast, ESD requires no tracing, so it presents unique advantages in dealing with rare events, such as concurrency bugs.

Higher-level replay systems, like R2 [20], can record library interactions and replay them. These approaches typically incur lower overhead than whole-system replay. ESD's playback environment uses similar techniques, extending them with the ability to play back asynchronous events (such as thread preemptions) that are crucial to reproducing concurrency bugs. The goal of R2 is to observe the interactions that lead to a particular problem. ESD, on the other hand, can synthesize the desired execution, thus obviating the need for any runtime observations.

Recent work looked at replaying concurrency bugs, such as data races, while aiming to minimize the amount of user-side recording [2, 33]. While similar in spirit to ESD, these tools still require recording all program inputs and the order of synchronization operations, thus adding overheads as high as 50%, which is hard to justify in production systems.

Aftersight [9] is an efficient way to observe and analyze the behavior of running programs on production workloads. Aftersight decouples analysis from normal execution by logging non-deterministic VM inputs and replaying them on a separate analysis platform. ESD, on the other hand does not monitor the running program, rather merely requires a coredump to perform its analysis at the developer's site.

10. Conclusion

This paper introduced *execution synthesis*, a technique for automatically debugging real software. We presented ESD, a practical tool that embodies this technique and alleviates the burden of fixing hard-to-reproduce bugs. ESD starts from a bug report and automatically synthesizes an execution that causes the bug to manifest. Developers can then deterministically play back this execution in their favorite debugger as many times as necessary to generate a fix. ESD requires no program modifications and no runtime tracing, thus introducing no runtime overhead.

We showed ESD can reproduce, with no human intervention, concurrency bugs and crashes reported in real applications. It took less than three minutes to synthesize explanations for these bugs, which suggests ESD is practical for frequent use during development and debugging. To our knowledge, ESD is the first tool that can automatically synthesize executions to reproduce bugs that occurred in the field, without incurring the overhead of execution tracing.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

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

[3] P. Boonstoppel, C. Cadar, and D. R. Engler. RWset: Attacking path explosion in constraint-based test generation. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

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

[5] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *IEEE Symp. on Security and Privacy*, 2008.

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

[7] M. Castro, M. Costa, and J.-P. Martin. Better bug reporting with better privacy. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2008.

[8] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea. Selective symbolic execution. In *Workshop on Hot Topics in Dependable Systems*, 2009.

[9] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX Annual Technical Conf.*, 2008.

[10] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of Internet worms. In *Symp. on Operating Systems Principles*, 2005.

[11] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: Securing software by blocking bad input. In *Symp. on Operating Systems Principles*, 2007.

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

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

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

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

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

[17] P. Godefroid and N. Nagappan. Concurrency at Microsoft – An exploratory survey. In *CAV Workshop on Exploiting Concurrency Efficiently and Correctly*, 2008.

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

[19] P. Godefroid, M. Y. Levin, and D. Molnar. Automated White-box Fuzz Testing. In *Network and Distributed System Security Symp.*, 2008.

[20] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An application-level kernel for record and replay. In *Symp. on Operating Systems Design and Implementation*, 2008.

[21] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *Conf. on Programming Language Design and Implementation*, 2009.

[22] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In *Symp. on Operating Systems Design and Implementation*, 2008.

[23] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 1976.

[24] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX Annual Technical Conf.*, 2005.

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

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

[27] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes – a comprehensive study on real world concurrency bug characteristics. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2008.

[28] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. PIN: building customized program analysis tools with dynamic instrumentation. In *Conf. on Programming Language Design and Implementation*, 2005.

[29] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *Symp. on Operating Systems Design and Implementation*, 2008.

[30] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data races using replay analysis. *Conf. on Programming Language Design and Implementation*, 2007.

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

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

[33] S. Park, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, S. Lu, and Y. Zhou. Do you have to reproduce the bug at the first replay attempt? – PRES: Probabilistic replay with execution sketching on multiprocessors. In *Symp. on Operating Systems Principles*, 2009.

[34] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4), 1997.

[35] K. Sen. Race directed random testing of concurrent programs. *Conf. on Programming Language Design and Implementation*, 2008.

[36] SQLite. SQLite. http://www.sqlite.org/, 2010.

[37] M. Weiser. Program slicing. In *Intl. Conf. on Software Engineering*, 1981.