BugRedux: Reproducing Field Failures for In-House Debugging

Wei Jin and Alessandro Orso
Georgia Institute of Technology
Email: {weijin|orso}@gatech.edu

***Abstract*—A recent survey conducted among developers of the Apache, Eclipse, and Mozilla projects showed that the ability to recreate field failures is considered of fundamental importance when investigating bug reports. Unfortunately, the information typically contained in a bug report, such as memory dumps or call stacks, is usually insufficient for recreating the problem. Even more advanced approaches for gathering field data and help in-house debugging tend to collect either too little information, and be ineffective, or too much information, and be inefficient. To address these issues, we present BUGREDUX, a novel general approach for in-house debugging of field failures. BUGREDUX aims to synthesize, using execution data collected in the field, executions that mimic the observed field failures. We define several instances of BUGREDUX that collect different types of execution data and perform, through an empirical study, a cost-benefit analysis of the approach and its variations. In the study, we apply BUGREDUX to 16 failures of 14 real-world programs. Our results are promising in that they show that it is possible to synthesize in-house executions that reproduce failures observed in the field using a suitable set of execution data.**

I. INTRODUCTION

Quality-assurance activities, such as software testing and analysis, are notoriously difficult, expensive, and time-consuming. As a result, software products are typically released with faults or missing functionality. The characteristics of modern software are making the situation even worse. Because of the dynamic nature, configurability, and portability of today's software, deployed applications may behave very differently in house and in the field. In some cases, these different behaviors may be totally legitimate behaviors that simply were not observed during in-house testing. In other cases, however, such behaviors may be anomalous and result in *field failures*, failures of the software that occur after deployment, while the software is running on user machines.

Field failures are both difficult to foresee and difficult, if not impossible, to reproduce outside the time and place in which they occurred. In fact, a recent survey among developers of the Apache, Eclipse, and Mozilla projects revealed that most developers consider information on how to reproduce failures (*e.g.,* stack traces, steps to follow, and ideally even inputs) to be the most valuable and difficult to obtain piece of information in a bug report [1]. This pressing need is demonstrated by the emergence, in the last decade, of several reporting systems that collect information (*e.g.,* stack traces and register dumps) when a program crashes and send it back to the software producer (*e.g.,* [2], [3]). Although useful, the information collected by these systems is often too limited to allow for reproducing a failure and is typically used to identify correlations among different crash reports or among crash reports and known failures.

Researchers have also investigated more sophisticated techniques for capturing data from deployed applications that can help debugging (*e.g.,* [4]–[10]). Among these techniques, some collect only limited amounts of information (*e.g.,* sampled branch profiles for CBI [6], [7]). These techniques have the advantage of collecting types of data that are unlikely to be sensitive, which makes them more likely to be accepted by the user community. Moreover, given the amount of information collected, it is conceivable for users to manually inspect the information before it is sent to developers.

Unfortunately, subsequent research has shown that the usefulness of the information collected for debugging increases when more (and more detailed) data is collected. Researchers have therefore defined novel techniques that gather a wide spectrum of richer data, ranging from path profiles to complete execution recordings (*e.g.,* [4], [5], [11], [12]). Complete execution recordings, in particular, can address the issue of reproducibility of field failures. User executions, however, have the fundamental drawbacks that they (1) can be expensive to collect and (2) are bound to contain sensitive data. While the former issue can be alleviated with suitable engineering (*e.g.,* [5], [11]), the latter issue would make the use of these techniques in the field problematic. Given the sheer amount of data collected, users would not be able to manually check the data before they are sent to developers, and would therefore be unlikely to agree on the collection of such data. Although some techniques exist whose goal is to sanitize or anonymize collected data, they are either defined for a different goal, and would thus eliminate sensitive data only by chance (*e.g.,* [13], [14]), or are still in their early phase of development and in need of a more thorough evaluation (*e.g.,* [15], [16]).

The overall goal of this work is to address these limitations of existing techniques by developing novel approaches for reproducing field failures in house without imposing too much overhead on the users and without violating the users' privacy. More precisely, we aim to develop a general technique that can synthesize, given a program P, a field execution E of P that results in a failure F, and a set of execution data D for E, an in-house execution E' as follows. *First*, E' should result in a failure F' that is analogous to F, that is, F' has the same observable behavior of F. If F is the violation of an assertion at a given location in P, for instance, F' should violate the same assertion at the same point. *Second*, E' should be an

actual execution of P, that is, the approach should be sound and generate an actual input that, when provided to P, results in E'. *Third*, the approach should be able to generate E' using only P and D, without the need for any additional information. *Finally*, D should not contain sensitive data and should be collectable with low overhead on E.

As a first step towards our goal, in this paper we present BUGREDUX, a general technique for (1) collecting different kinds of execution data and (2) using the collected data to synthesize in-house executions that can reproduce failures observed in the field. Intuitively, BUGREDUX can be seen as a general framework parameterized along two dimensions: the kind of execution data D collected and the technique used for synthesizing execution E'. We present four variations, or instances, of BUGREDUX that all share the same synthesis technique (*i.e.,* symbolic execution) but differ in the kind of execution data they use. Specifically, we consider four types of increasingly rich execution data: points of failure, call stacks, call sequences, and complete program traces.

We also present an empirical investigation in which we assess the tradeoffs that characterize the variations of BUGREDUX with respect to (1) the cost of the data collection, in terms of space and time overhead (and, indirectly, likelihood to contain sensitive data), and (2) the ease of synthesizing a failing execution starting from such data. In the evaluation, we used an implementation of BUGREDUX developed for the C language and applied it to 16 failures of 14 real-world programs. For each failure, we collected the four different types of execution data, measured the overhead of the collection, and tried to synthesize an execution that reproduced the failure using such data. Interestingly, our results show that the richest data, beside being the most expensive to collect and the most problematic in terms of potential privacy violation, is not necessarily the most useful when used for synthesizing executions. Our results also confirm that, at least for the cases we considered, information that is traditionally collected by crash-report systems, such as the call stack at the point of failure, is typically not enough for recreating field failures.

For the current incarnation of BUGREDUX, we found that the best option in terms of cost-benefit ratio is the use of call sequences. As our study shows, using call-sequence data BUGREDUX was able to recreate all of the 16 failures considered, while imposing an acceptable time and space overhead. We believe that these results, albeit preliminary in nature, are encouraging and motivate further research in this direction. In fact, as we discuss in the final part of the paper, we have already identified several opportunities for further reducing the cost of the data collection while maintaining the same ability of recreating field failures.

This paper provides the following novel contributions:

- A general approach for collecting execution data in the field and using the data to synthesize executions that reproduce field failures.
- The instantiation of the approach for four different kinds of execution data and one execution synthesis technique and its implementation in a freely-available prototype tool (see http://www.cc.gatech.edu/~orso/software/bugredux.html).
- An empirical study that performs a cost-benefit analysis of the approach and its variations in terms of data collection costs and ability to synthesize failing executions.

II. MOTIVATING EXAMPLE

We introduce an example that we use in the rest of the paper to motivate our work, show the challenges involved in reproducing observed failures, and illustrate our technique. Our example, shown in Figure 1, is taken from the Coreutils library [17]. Specifically, we selected a piece of code that contains a fault and simplified it to make it self contained and easier to understand for the reader.

The program takes a string argument from the command line and has five functions: `main`, `process`, `uppercase`, `replaceescape`, and `printresult`. Function `main` first checks that exactly one command-line argument is present (lines 39–40) and that the length of the input parameter is less than 256 characters (lines 41–42). It then allocates an array of 256 characters, which will be used to store the result of the execution, and invokes function `process` with the input argument and the newly created array as parameters (line 44).

Function `process` scans each character in its input string and adds it to the output string after processing it in one of three ways. If the character is alphabetical and lower-case, it is replaced with the corresponding upper-case character using function `uppercase` (lines 26–27). If the character is part of an escape sequence, it is replaced using function `replaceescape` (lines 28–30). This function replaces the character with either a new line or a tab, if the escape sequence is one of $\backslash n$ or $\backslash t$, or with $null$ otherwise. All other characters are copied to the output string unmodified (lines 31–32). After all input characters have been processed, function `process` calls function `printresult`, which prints out the output string.

The fault in the code is at line 30, in function `process`. After processing an escape sequence, the code increments the index of the output array `out` instead of that of the input array `in`. The first consequence of this fault is that one character is skipped in the output string, and an extra character is added to the string. For a sequence "$\backslash n$", for instance, both a newline character and character "n" would be added to the output, with an undefined character in between. Another effect is that the index of the output array will grow larger than the index of the input array by one for each escape character processed. Therefore, if the number of escape characters plus the length of the input array were to exceed 256, which is the size of the output array, this fault will cause a memory error.

This is a simple, yet interesting fault, as the memory error would be triggered only by an input with the following characteristics: (1) the input must contain less than 256 characters, to pass the initial test, and (2) the sum of the length of the input plus the number of escape characters it contains must be greater than 256. Note that this also implies that the input must contain at least two escape characters, as the input string can contain at most 255 characters, whereas the output

```
1.  char  replaceescape(char e) {
2.    switch (e) {
3.      case 'n':
4.        return 10;
5.      case 't':
6.        return 9;
7.      default:
8.        return 0;
9.    }
10. }
11. char  uppercase(char l) {
12.   return l-'a'+'A';
13. }
14. void  printresult(char* str,int length) {
15.   int i;
16.   for (i=0;i<length;i++) {
17.     if (str[i]!=0)
18.       printf("%c",str[i]);
19.   }
20. }
21. void  process(char* source, char* dest) {
22.   int out=0;
23.   int in=0;
24.   int srclength = strlen(source);
25.   while (in<srclength) {
26.     if (source[in]>='a'&&source[in]<='z') {
27.       dest[out]=uppercase(source[in]);
28.     } else if (source[in]=='\\') {
29.       dest[out]=replaceescape(source[in+1]);
30.       out++;  // correct version: in++;
31.     } else
32.       dest[out]=source[in];
33.     out++;
34.     in++;
35.   }
36.   printresult(dest,out);
37. }
38. int  main(int argc, char *argv[]) {
39.   if (argc!=2)
40.     exit(0);
41.   if (strlen(argv[1])>=256)
42.     exit(0);
43.   char* outputstr=malloc(256);
44.   process(argv[1],outputstr);
45. }
```

Figure 1. Example of faulty program.

array's capacity is 256. Because it requires a specially-crafted input to be triggered, such a fault may not be revealed by an in-house test suite (even one that covers all branches in the program) and could therefore result in a field failure.

Assume that the program is released with the fault at line 30 we just discussed, and that a user provides an input that triggers the fault and results in a memory error at line 27, when the program tries to write the 257^{th} character in the output array. In this situation, the runtime system would generate a crash report such as the one shown in Figure 2. As the figure shows, the crash report lists the *point of failure (POF)* for F (line 27 of `example.c`), and the call stack at the moment of failure, with one entry per function on the stack.

A developer who is assigned this bug report and wants to investigate the problem would likely try to reproduce the failure, which is far from trivial. It would be unlikely, for instance, that random testing could generate an input that satisfies the failing conditions for F by pure chance. Even more sophisticate approaches, such as those based on symbolic execution or some other verification techniques, would have a hard time triggering the faulty behavior without any guidance. Given that (1) the length of the input string plus the number of escape characters should be greater than 256, and (2) the loop in function `process` increments the output array's index by at most two characters per iteration, the shortest failing path would be one that traverses the loop 128 times and, for all iterations but the last one, follows path $\{26, 28, 29, 30, 33, 34\}$. Finding this path using symbolic execution would not be possible if the number of loop iterations were bounded to some small value, as it is typically the case to make the exploration feasible (*e.g.,* [18]–[20]). With unbounded loop exploration, on the other hand, symbolic

```
Error: memory error
File: example.c
Line: 27
Stack:
        #0 00000388 in process (source=\\
           185417824, dest=186177720) at example.c:27
        #1 00000492 in main (argc=\\
           2, argv=180717480) at example.c:44
```

Figure 2. Crash report for our example program.

execution may have to explore three paths for each iteration, which would result in 3^n paths explored for n iterations—a number that would quickly grow to impractically large values.

For this example, thus, POF and call stack are unlikely to provide enough information to help developers reproduce and debug the reported field failure. To do so, developers would need additional information about the length of the input and the number of escape characters in it. This information could be provided in many ways, and by collecting many different kinds of data (*e.g.,* profiles, input values, invariants). Most importantly, the kind of data needed is likely to depend on the specific failure considered. As a first step towards defining a technique for reproducing field failures in house, it is therefore important to understand the usefulness of different kinds of execution data in this context. To this end, we defined a general approach for synthesizing executions that (1) mimic executions that resulted in field failures and (2) reproduce such failures. We instantiated several variants of our approach that differ in the kind of execution data they use, and studied the effectiveness of these different variants. The next sections discuss our approach and our empirical investigation. Note that, for space reasons, we do not provide here background information on some basic program analysis and symbolic execution concepts used in the discussion of our approach. Expanded discussions, definitions, and examples appear in the companion technical report [21].

III. Our Approach for Recreating Field Failures

As we stated in the Introduction, our overall goal is to recreate field failures faithfully (*i.e.,* in a way that allows for debugging them) by using execution data collected in the field that can be gathered without imposing too much space and time overhead on field executions. To achieve this goal, we developed a general approach that we call BugRedux. Intuitively, BugRedux operates by (1) collecting different kinds of execution data and (2) using the collected data to synthesize in-house executions that reproduce failures observed in the field. Figure 3 provides a high-level overview of BugRedux and of the scenario we target.

As the figure shows, BugRedux consists of two main components. The first one is the *instrumenter*, which takes as input an *application* provided by a *software developer* and generates an *instrumented application* that can collect execution data and add such execution data to *crash reports* from the field. The second component is the *analyzer*, which takes as input a *crash report* and tries to generate a *test input* that, when provided to the *application*, results in the same failure that was observed in the field. A *software tester* can then use the generated input to recreate and debug the field failure. This general approach can be defined in different ways

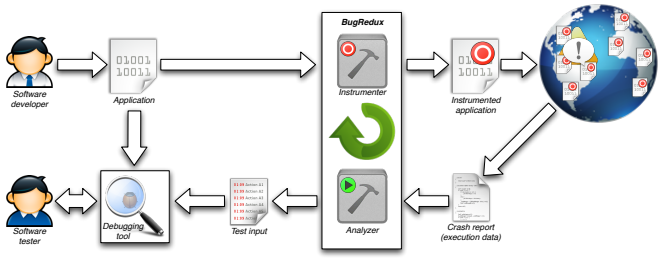

Figure 3. Intuitive high-level view of BUGREDUX.

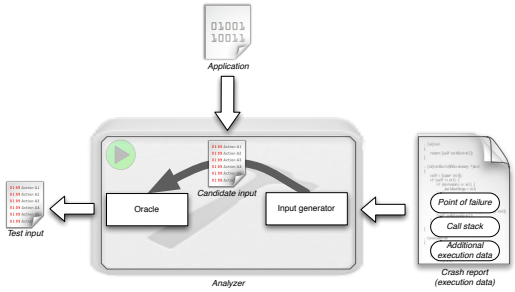

Figure 4. The analysis component of BUGREDUX.

depending on the kind of execution data collected and on the technique used for synthesizing execution.

A. Instrumenter and Analyzer Components

Instrumentation is a well assessed technology, so we do not discuss this part of the approach further. It suffices to say that BUGREDUX adds probes to the original program that, when triggered at runtime, generate the execution data of interest. Conversely, the analyzer is the core part of the approach and the most challenging to develop. Figure 4, which provides a more detailed view of the analysis component of BUGREDUX, puts the problem in context and lets us discuss how we addressed this challenge. As the figure shows, the inputs to the analyzer are an application program P, whose execution E produces failure F that we want to reproduce, and a crash report C for F. The goal of the analyzer is to generate a test input that would result in an execution E' that "mimics" E and would fail in the same way.

Given crash report C, the *input generator* would analyze program P and try to generate such test input. The exact definition of mimicking depends on the amount of information about the failing execution E that is available. If only the POF were available, for instance, E' would mimic E if it reaches the POF. Conversely, if a complete trace were to be used, E' would have not only to reach the POF but also to follow the same path as E. This concept of mimicking is defined within the *input generator*, which receives the execution data in the form of a sequence of goals (or statements) to be reached and tries to generate executions that reach such goals in the right order. If successful, the *input generator* would generate a candidate input, and the *oracle* would check whether that input actually fails in the same way as E.

In theory, any automated input generation technique could be used in this context, as long as it can be guided towards a goal (*e.g.,* the point of failure, the entry point of a function on the failure's call stack, or a branch within the program). In this work, we decided to use an approach based on symbolic execution [18]. Specifically, we use a symbolic execution algorithm customized with an ad-hoc search strategy that leverages the execution data available expressed as a set of goals. Our algorithm, GenerateInputs, is shown in Algorithm 1. GenerateInputs takes as input $icfg$, the Interprocedural Control Flow Graph (ICFG) [22] for program P, and $goals\_list$, an ordered list of statements to be reached during the execution. (We discuss the exact content of $goals\_list$ in Section III-B.)

Initially, GenerateInputs performs some initializations (lines 2–4). First, it initializes sym_state_0 with the initial

Algorithm 1: GenerateInputs

Input : $icfg$: ICFG for program P
$goals\_list$: an ordered list of statements $G_0, ...G_n$
Output: $input_f$: candidate input for synthesized run

```
1  begin
2      sym_state_0 ← initial symbolic values of program inputs
3      states_set ← (icfg.entry, true, sym_state_0, G_0)
4      curr_goal ← G_0
5      while true do
6          curr_state ← null
7          while curr_state == null do
8              curr_state ←
               SelNextState(icfg, states_set, curr_goal)
9              if curr_state == null then
10                 if curr_goal ≠ G_0 then
11                     curr_goal ← previous goal in goals_list
12                     continue
13                 else
14                     return null
15                 end
16             end
17         end
18         if curr_state.cl == curr_goal then
19             if curr_goal == G_n then
20                 input_f ← solver.getSol(curr_state.pc)
21                 if input_f is found then
22                     return input_f
23                 else
24                     remove(curr_state, states_set)
25                     continue
26                 end
27             else
28                 curr_goal ← next target in goals_list
29                 curr_state.goal ← curr_goal
30             end
31         else
32             if curr_state.cl ∈ goal_list then
33                 remove(curr_state, states_set)
34                 continue
35             end
36         end
37         if curr_state.cl is a conditional statement then
38             curr_state.pc ←
               addConstr(curr_state.pc, pred, true)
39             curr_state.cl ← getSucc(curr_state.cl, true)
40             if solver.checkSat(curr_state.pc) == false then
41                 remove(curr_state, states_set)
42             end
43             false_pc ← addConstr(curr_state.pc, pred, false)
44             false_cl ← getSucc(curr_state.cl, false)
45             if solver.checkSat(curr_state.pc) ≠ false then
46                 new_state ←
                   (false_cl, false_pc, curr_state.ss, curr_state.goal)
47                 insert(new_state, state_set)
48             end
49         else
50             curr_state.ss ←
               symEval(curr_state.ss, curr_state.cl)
51             curr_state.cl ← getSucc(curr_state.cl)
52         end
53     end
54 end
```

symbolic state, where all inputs are marked as symbolic. Then, it initializes $states\_set$, a set that will be used to store search states during the execution, with the initial search state. Entries in $states\_set$ are quadruples $\langle cl, pc, ss, goal \rangle$, where cl is a code location, pc the Path Condition (PC) for the path followed to reach location cl, ss the symbolic state right before cl, and $goal$ the current target for this state (used to enforce the order in which goals are reached). The initial search state consists of the entry of the program for cl, PC `true`, symbolic state sym_state_0, and goal G_0. Next, the algorithm assigns to $curr\_goal$ the first goal from $goals\_list$.

The algorithm then enters its main loop. At the beginning of each loop iteration, GenerateInputs invokes algorithm SelNextState, shown in Algorithm 2. SelNextState looks for the most promising state to explore in $states\_set$. (At the first invocation of SelNextState, only the initial state is in the $states\_set$. The number of states will increase in subsequent invocations, when more of the program has been explored symbolically.) SelNextState selects states based on the minimum distance $mindis$, computed in terms of number of statements in the ICFG, between each state's cl and $curr\_goal$. To avoid selecting states that have not reached goals that precede $curr\_goal$ in $goals\_list$, SelNextState only considers states whose target is $curr\_goal$ (line 5 in Algorithm 2). If none of these states can reach $curr\_goal$ (*i.e.,* there's no path between the state's cl and $curr\_goal$ in the ICFG), SelNextState returns `null` to GenerateInputs. Otherwise, the selected state is returned (line 15 in Algorithm 2).

When GenerateInputs receives the candidate state from SelNextState, it first checks whether the returned state is `null`, which means that no state in $states\_set$ with target $curr\_goal$ can actually reach such target. If so, GenerateInputs backtracks by updating $curr\_goal$ to the previous goal in the $goals\_list$ and looking for another path that can reach that goal (line 11). Conversely, if $curr\_state$ is not null, GenerateInputs continues the execution of its main loop.

If $curr\_state$'s code location corresponds to $curr\_goal$, GenerateInputs updates both global goal $curr\_goal$ and local goal $curr\_state.goal$ to the next goal in $goals\_list$ (lines 28–29). It then continues the symbolic execution. If the last goal G_n is reached, the algorithm stops the symbolic execution, feeds the current PC to the SMT solver, and asks the solver to find a solution for the PC (line 20). If a solution is not found, the generation of the candidate input is deemed unsuccessful. If $curr\_state$'s code location is not $curr\_goal$ but another goal in $goal\_list$, the algorithm removes $curr\_state$ from $state\_set$ and goes back to the beginning of the main loop (lines 32–34). It does so to avoid that the execution reaches the goals in the goal list in a different order from the one observed in the failing execution. If $curr\_state$'s code location is a conditional statement $pred$ that involves symbolic values, the algorithm performs one execution step along both branches, that is, it updates states' current location and path condition, checks the feasibility of both branches using the SMT solver, and removes (or does not add) infeasible states

Algorithm 2: SelNextState

```
   Input  : icfg : ICFG for program P
            states_set: set of symbolic states
            curr_goal: next goal
   Output: ret_state: candidate state for exploration
1  begin
2      mindis ← +∞
3      ret_state ← null
4      foreach State_i ∈ states_set do
5          if State_i.goal == curr_goal then
6              if State_i.loc can reach curr_goal in ICFG then
7                  nd ← shortest distance from State_i.loc to curr_goal
                   in ICFG
8                  if nd < mindis then
9                      mindis ← nd
10                     ret_state ← State_i
11                 end
12             end
13         end
14     end
15     return ret_state
16 end
```

from $states\_set$ (lines 38–47). (If the SMT solver did not provide an answer for PC, the algorithm would consider the corresponding state feasible and continue.) Finally, if $curr\_state$'s code location is any statement other than a conditional, the algorithm suitably updates the symbolic state and the current location of $states\_set$ (lines 50–51).

The algorithm terminates when either there are no more states to explore (*i.e.,* it tries to backtrace from G_0 (line 14)) or a candidate input is successfully generated (line 22). In the former case, our algorithm fails to find an input that can mimic the observed execution. In the latter case, conversely, the algorithm successfully produces such input.

In summary, our symbolic execution technique has two key aspects. First, it uses the execution data from the field to identify a set of intermediate goals that can guide the exploration of the solution space. Second, it uses a heuristic based on distance to select which states to consider first when trying to reach an intermediate goal during the exploration. In theory, the more data (*i.e.,* number of intermediate goals) available, the more directed the search, and the higher the likelihood of synthesizing a suitable execution. On the other hand, collecting too much data can have negative consequences in terms of overhead and introduce privacy issues. To study this tradeoff, we define several variants of our approach that differ on the kind of execution data they consider. We describe these variants in the next section.

B. Execution Data

In selecting the execution data to consider, we aimed to cover a broad spectrum of possibilities. To this end, we selected four kinds of data characterizing a failure: point of failure (POF), call stack, call sequence, and complete program trace. These types of data are representative of scenarios that go from knowing as little as possible about the failing execution to knowing almost everything about it. In addition, POFs and call stacks are types of data that are very commonly available for crashes, as they are normally included in crash reports. Call sequences, and program traces, on the other hand, are not normally available and represent data that, if

they were shown to be useful, would require changes in the way programs are monitored and crash reports are generated.

For each of these four types of execution data, we instantiated a variation of BUGREDUX that collected and used that type of data. As far as data collection is concerned, the first two types of execution data do not require any modification of the program being monitored, as they can be extracted them from existing reports. The other two types of data can be collected using well-understood program instrumentation techniques. To collect call sequences, BUGREDUX instruments all call sites and entry points (these latter to account for the possible presence of function pointers), whereas to collect program traces it instrument all branches.

Customizing BUGREDUX so that it uses the different data is also relatively straightforward, as it amounts to suitably generating the $goals\_list$ set to be passed to BUGREDUX's input generator. For POF, $goals\_list$ would contain a single entry—the POF itself. For a failure's call stack, there would be an entry in the set for each function on the stack, corresponding to the first statement of the function, plus an additional entry for the POF. Call sequences would result in a $goals\_list$ that contains an entry for each call, corresponding to the call statement. Also in this case, there would be an additional, final entry for the POF. Finally, the $goals\_list$ for a program trace would consist of an entry per branch, corresponding to the statements that is the destination of the branch, and the usual entry for the POF.

In the next section, we discuss how we used these four variants of BUGREDUX to study the tradeoffs involved with the use of different kinds of information and assess the general usefulness of the proposed approach.

IV. EMPIRICAL INVESTIGATION

We investigated the following research questions:

- **RQ1:** Can BUGREDUX synthesize executions that are able to reproduce field failures?
- **RQ2:** If so, which types of execution data provide the best tradeoffs in terms of cost benefit?

To address these questions, we implemented the four variants of BUGREDUX discussed in the previous section and applied them to a set of real-world programs.

A. BUGREDUX *Implementation*

Our implementation of BUGREDUX works on C programs and consists of three modules that correspond to the three components shown in our high-level view of the approach (see Figures 3 and 4): instrumenter, input generator, and oracle. BUGREDUX's instrumenter performs static instrumentation (*i.e.,* probes are added to the code at compile time) by leveraging the LLVM compiler infrastructure (http://llvm.org/). The input generator in BUGREDUX is built on top of KLEE [23], a symbolic execution engine for C programs. We implemented Algorithms 1 and 2 as a custom search strategy for KLEE and also made a few modifications to KLEE's code. Finally, BUGREDUX's oracle module is implemented as a Perl script that operates as follows: (1) it takes as input program P, an input I for P, and a crash report C corresponding to failure F; (2) it runs P against I and collects any crash report generated as a result of the execution; and (3) if either no report is generated or the call stack and POF in the generated report do not match those in C, it reports that the approach failed, whereas it reports a success otherwise.

Table I
SUBJECT PROGRAMS USED IN OUR STUDY.

Name	Repository	Description	Size (kLOC)	# Faults
sed	SIR	stream editor	14	2
grep	SIR	pattern-matching utility	10	1
gzip	SIR	compression utility	5	2
ncompress	BugBench	(de)compression utility	2	1
polymorph	BugBench	file system "unixier"	1	1
aeon	exploit-db	mail relay agent	3	1
glftpd	exploit-db	FTP server	6	1
htget	exploit-db	file grabber	3	1
socat	exploit-db	multipurpose relay	35	1
tipxd	exploit-db	IPX tunneling daemon	7	1
aspell	exploit-db	spell checker	0.5	1
exim	exploit-db	message transfer agent	241	1
rsync	exploit-db	file synchronizer	67	1
xmail	exploit-db	email server	1	1

B. Programs and Faults Considered

To investigate our research questions in a real(istic) setting, we used a set of real, non-trivial programs that contained one or more faults and had test cases that could reveal such faults. We considered programs from three public repositories that have been used extensively in previous research: SIR [24], BugBench [25], and exploit-db [26]. Specifically, we selected three programs from SIR, two from BugBench, and nine from exploit-db. Table I shows the relevant information about each program: name, repository from which it was downloaded, size, and number of faults it contains. As the table shows, the program sizes range from 0.5 kLOC to 241 kLOC, and each program contains one or two faults. The faults in the BugBench and exploit-db programs are real faults, whereas the ones in the programs from SIR are seeded.

We selected these programs because they have been used in previous research [16], [26] and because of the representativeness of their faults. The faults in exploit-db and BugBench are *real faults mostly discovered by users in the field*, whereas the faults in SIR are seeded by researchers but are carefully designed to simulate real faults.

We excluded from our study three programs from SIR and four from BugBench because the version of KLEE we used could not handle some of the constructs in these programs (*e.g.,* complex interactions with the environment and network inputs). As far as faults are concerned, we selected faults that caused a program crash, rather than just generating an incorrect result. This choice was made for convenience and to minimize experimental bias—with crashes, failures can be objectively identified and do not require the manual encoding of the failure condition as an assertion.

We also performed a preliminary check on the programs and faults that we selected by feeding them to an unmodified version of KLEE and letting it run for 72 hours. The goal of this check was to assess whether these faults could have been discovered by a technique that blindly tries to explore as much of the programs as possible. If so, this would be

an indication that the faults are too easy to reveal to be good candidates for our study. The unmodified KLEE was unable to reveal the faults in the programs except for one case: iwconfig. We therefore removed iwconfig from our set of subjects. It is worth noting that we decided not to use the benchmarks used in Reference [27] for the same reason—all of those failures could be recreated through plain, unguided symbolic execution, as also shown in Reference [23]. (Moreover, the benchmarks we selected are more representative, as programs are larger and 9 out of 16 faults are real faults reported by users, rather than faults found in-house by KLEE.)

C. Experimental Setup

In order to collect the data needed for our investigation, we proceeded as follows. To simulate the occurrence of field failures, we used the test cases distributed with our subject programs as proxies for real users. For each fault f considered, we ran the test cases until a test case t_f failed and generated a program crash; we associated t_f to f as its failing input. We then reran all the failing inputs on all the corresponding faulty programs three times. The first time, we ran them on the unmodified programs, the second time on the programs instrumented by BUGREDUX to collect call sequences, and the third time on the programs instrumented by BUGREDUX to collect complete program traces. For each such execution, we measured the duration of the execution and the size of the execution data generated.

With this information available, we used the four variants of BUGREDUX to synthesize a failing execution starting from a suitable set of goals (*i.e.,* POF, call stack at the time of failure, call sequence, and complete program trace). For each run of BUGREDUX, we recorded whether the generation was successful (*i.e.,* whether a candidate input was generated at all) and how long it took. We set a timeout of 72 hours for the generation, after which we marked the run as unsuccessful. We also recorded whether the candidate input, if one was generated, could reproduce the original failure according to BUGREDUX's oracle.

D. Results and Discussion

This section presents the results of our empirical study and discusses the implication of the results in terms of our two research questions. We present the results using two tables, where the first table contains the data related to the cost of the approach (*i.e.,* the time and space overhead imposed by BUGREDUX), and the second table shows the data about the effectiveness of the approach (*i.e.,* whether BUGREDUX was able to synthesize an execution and whether such execution could be used to reproduce an observed failure). These two tables present the results for each of the 16 failing executions considered, identified by the name of the failing program possibly followed by a fault ID, and for each of the variants of BUGREDUX, identified by the kind of execution data on which it operates.

Table II shows the time and space overhead imposed by BUGREDUX on the subjects for each of the four types of execution data collected. Time overhead is measured as the percentage increase of the running time due to instrumentation, whereas space overhead is measured as the size of the different kinds of execution data collected by BUGREDUX. We discuss the two types of overheads separately.

Table II
TIME (%) AND SPACE (KB) OVERHEAD IMPOSED BY BUGREDUX.

Name	POF		Call stack		Call sequence		Complete trace	
	time	space	time	space	time	space	time	space
sed.fault1	0%	0.8	0%	0.8	4.5%	5.8	27.2%	54.4
sed.fault2	0%	0.9	0%	0.9	12.5%	10.2	87.5%	261.9
grep	0%	0.7	0%	0.7	47%	3.4	182%	716.1
gzip.fault1	0%	0.8	0%	0.8	10.3%	2	72%	176
gzip.fault2	0%	0.8	0%	0.8	12%	2.5	308%	1784.6
ncompress	0%	0.7	0%	0.7	2%	0.9	16%	33.1
polymorph	0%	0.5	0%	0.5	1%	0.7	8%	1.5
aeon	0%	1	0%	1	50%	1.1	1066%	3
glftpd	0%	1.5	0%	1.5	9%	3.2	45%	130
htget	0%	0.7	0%	0.7	9%	2.7	287%	2814
socat	0%	0.8	0%	0.8	21%	9.6	110%	451
tipxd	0%	0.6	0%	0.6	2%	0.7	36%	19
aspell	0%	0.6	0%	0.6	18.8%	30.5	143%	566
rsync	0%	1	0%	1	3%	11.4	66%	521
xmail	0%	0.8	0%	0.8	22.6%	84.8	290%	2361
exim	0%	0.9	0%	0.9	17.4%	100.7	389%	14897

Time overhead. Because POFs and call stacks are collected by the runtime system at the moment of the failure, and do not require any additional instrumentation, collecting them incurs no overhead. The situation is different for call sequences and complete traces, which both require BUGREDUX to instrument the programs (see Section III-B). As expected, the overhead imposed by complete-trace collection is almost an order of magnitude higher than that for call sequences. We also observe that the overhead for collecting call sequences depends on program size and execution length. To correctly interpret these results, it is important to consider that this data was collected with a naive instrumentation that writes events to the log as soon as they occur; the use of caching techniques could decrease the overhead dramatically. Because the goal of this initial investigation was more exploratory, and the numbers are acceptable, we left optimizations for future work. Moreover, record-replay techniques (*e.g.,* [5], [11]) could be used to (1) efficiently record field executions and (2) collect execution data while replaying—offline and when free cycles are available on the user machines.

Space overhead. The data size for POFs and call stacks is the same because our current implementation of BUGREDUX extracts both of them from the crash reports generated by the runtime system. We therefore report the size of the crash reports for these two types of data. Also in this case, the size of the complete-trace data is at least an order of magnitude larger than that of the call-sequence data, and in some cases the difference is even more extreme. For instance, in the case of gzip.fault2, the reason for the large gap is that the number of function calls is low but there is a large number of loop iterations within functions. Overall, however, for the executions in this study, the size of the execution data is fairly contained, and it would be practical to collect them.

Table III addresses the core question of the effectiveness of our approach. The table shows, for each failing execution fe considered and each type of execution data ed, the time it took BUGREDUX to generate inputs that mimicked fe

using *ed* (or "N/A" if BUGREDUX was unable to generate such inputs in the allotted time) and whether the mimicked execution reproduced the observed failure ("Y" or "N").

As expected, symbolic execution guided only by the POF was unsuccessful for most programs. A manual examination of the programs for which POFs are enough to reproduce failures showed that all such failures have two common characteristics: (1) the POFs are close to the entry of the programs and are easy to reach; (2) the failures can be triggered by simply reaching the POFs. For these failures, developers could easily identify the corresponding faults if provided with traditional crash reports. As also expected, the larger the amount of data available (in the form of intermediate goals) to guide the exploration, the better the performance of the approach. Using call stacks, BUGREDUX could mimic 10 out of the 16 failing executions, and using call sequences, it was able to mimic all failing executions.

We observe that, in some cases (*e.g.,* htget, tipxd), the time needed to synthesize an execution using call stacks was larger than the time needed when using call sequences (when they are both successful). The reason for this behavior is that the additional information provided by call sequences can better guide symbolic execution and avoid the exploration of many irrelevant paths. One surprising finding, however, is that this trend is not confirmed when complete traces are used. We further analyzed this result and found that this happened for two reasons. One reason is that, intuitively, following complete traces can result in conditions that the SMT solver is unable to handle. The second reason is a mostly practical one: KLEE uses a simplified implementation of the system libraries when symbolically executing a program, which makes it impossible in some cases to follow exactly the same path that was followed in the original execution. Conversely, a looser, yet informative guidance, such as a call sequence, leaves more degrees of freedom to the input generator and increases its chances of success. For example, paths that result in constraints that are beyond the capabilities of the SMT solver could be dropped in favor of simpler paths that may still reach the targeted goals. In a sense, among the execution data we considered, call sequences represent a sweet spot between providing too little and too much information to the search.

It is important to stress that the executions synthesized by BUGREDUX are executions that reach all of the intermediate goals extracted from the execution data and provided to the input generator, but they are not guaranteed to reproduce the observed failure. Consider again our initial example in Figure 1. It is easy to synthesize an execution that reaches line 44 in function `main` and line 27 in function `process`, but that execution is unlikely to fail, as we discussed in Section II. This is especially true when considering more limited types of execution data, such as POFs and call stacks, which provide little guidance to the search. The results in Table III clearly illustrate this issue. As shown in the table, for the six of the failures in our set, reaching the POF is enough to trigger the original failure. Conversely, for the four failures in grep, gzip, htget and aspell, BUGREDUX was able to synthesize executions that generated the same call stacks as the failing executions, but such synthetic executions did not reproduce the considered failures. Finally, all of the 16 synthetic executions successfully generated from call sequences were able to reproduce the original failures.

Table III
EFFECTIVENESS AND EFFICIENCY OF BUGREDUX IN SYNTHESIZING EXECUTIONS STARTING FROM COLLECTED EXECUTION DATA.

Name	POF		Call stack		Call sequence		Complete trace	
sed.fault1	N/A		N/A		98s	Y	N/A	
sed.fault2	N/A		N/A		17349s	Y	N/A	
grep	N/A		16s	N	48s	Y	N/A	
gzip.fault1	3s	Y	18s	Y	11s	Y	N/A	
gzip.fault2	20s	N	28s	N	25s	Y	N/A	
ncompress	155s	Y	158s	Y	158s	Y	N/A	
polymorph	65s	Y	66s	Y	66s	Y	N/A	
aeon	1s	Y	1s	Y	1s	Y	1s	Y
rysnc	N/A		N/A		88s	Y	N/A	
glftpd	5s	Y	5s	Y	4s	Y	N/A	
htget	53s	N	53s	N	9s	Y	N/A	
socat	N/A		N/A		876s	Y	N/A	
tipxd	27s	Y	27s	Y	5s	Y	N/A	
aspell	5s	N	5s	N	12s	Y	N/A	
xmail	N/A		N/A		154s	Y	N/A	
exim	N/A		N/A		269s	Y	5624s	Y

E. Discussion

The results of our investigation, albeit preliminary, let us address our two research questions and make some observations. For RQ1, our results show that, for the programs and failures considered, BUGREDUX can reproduce observed failures starting from a set of execution data. For RQ2, the results provide initial but clear evidence that call sequences represent the best choice, among the ones considered, in terms of cost-benefit tradeoffs: using call sequences, BUGREDUX was able to reproduce all of the observed failures; even using an unoptimized instrumentation, BUGREDUX was able to collect call sequences with an acceptable time and space overhead; and we believe that call sequences are unlikely to reveal sensitive or confidential information about an execution. (Although this is just anecdotical evidence, we observed that none of the inputs generated when synthesizing executions from call sequences corresponded to the original input that caused the failure.) Unlike complete traces, which may provide enough information to reverse engineer the execution and identify the inputs that caused such execution, call sequences are a much more abstract model of executions.

An additional observation that can be made on the results is that POFs and call stacks do not seem to be particularly helpful for reproducing failures. Manual examination of the faults we considered showed that the points where the failure is observed tend to be distant from the fault. Therefore, most such failures are triggered only when the program executes the faulty code and the incorrect program state propagates to the POF. In these cases, POFs and call stacks are unlikely to help because the faulty code may be nowhere near the POF or the functions on the stack at the moment of the crash. If confirmed, this would be an interesting finding, as these are two types of execution data normally collected in crash

Table IV
MINIMAL NUMBER OF ENTRIES IN CALL SEQUENCES THAT ARE NEEDED TO REPRODUCE OBSERVED FAILURES.

Name	Original Length	Minimal Length
sed.fault1	73	12
sed.fault2	146	7
grep	31	2
xmail	1142	363
gzip.fault2	27	2
rysnc	23	2
aspell	516	256
socat	62	3
htget	25	2
exim	1029	326

reports. Extending crash reports with additional information may make them considerably more useful to developers.

As a further step towards understanding the usefulness of different execution data, we performed an additional exploratory study in which we removed entries in the collected call sequences and checked whether the partial sequences still contained enough information to recreate observed failures. More precisely, we selected from our original list the ten failures that could only be reproduced using call sequences. For each failure and corresponding call sequence, we then used a straw-man greedy algorithm that considers one entry in the call sequence at a time, starting from the beginning. If BugRedux can reproduce the failure without that entry in the sequence, the entry is removed. Table IV shows the result of this study in terms of number of entries in the call sequences before and after reduction. For example, only 2 of the 31 entries in the original call sequence are needed to reproduce the observed failure in grep. The results show that, in most cases, only a small subset of calls in the sequences is actually necessary to suitably guide the exploration. We can further observe that the number of entries needed seems to increase with the complexity of the input needed to trigger the fault, which makes sense intuitively. For instance, xmail's fault can only be triggered by an input file that includes a valid email address, and aspell's fault can only be triggered by an input of a given length. For these two faults, the reduction in the call sequence is less substantial than for the other faults considered. These additional results motivate further research in this direction, as we discuss in Section VI.

F. Limitations and Threats to Validity

The main limitation of BUGREDUX is that it relies on symbolic execution, an inherently complex and expensive approach. However, recent results have shown that, if suitably defined, tuned, and engineered, symbolic execution can scale even to large systems [28]. Moreover, as we discuss in future work, BUGREDUX can leverage different input generation techniques. Another limitation is the potential overhead involved in collecting field-execution data. For this reason, as also discussed in future work, we are currently investigating the use of alternative execution data. One final limitation is that BUGREDUX currently does not explicitly handle concurrency and non-determinism. In this initial phase of the research, we chose to focus on a smaller domain, and get a better understanding of that domain, before considering additional issues.

Like for all studies, there are threats to the validity of our results. To mitigate threats of internal validity related to errors in our implementation, we tested BUGREDUX on small examples and spot checked most of the results presented in the paper. In terms of external validity, our results may not generalize to other programs and failures. However, we studied 16 failures and 14 programs from three different software repositories. The subjects we used are real-world programs, several of which are widely used both by real users in the field and by researchers as experimental subjects. Another issue with the empirical results is that the ultimate evidence of the usefulness of the technique would require its use in a real setting and with real users. Although such an evaluation would be extremely useful, and we plan to do it in the future as we did for earlier work [29], we believe that it would be premature at this point. Moreover, when successful, BUGREDUX would generate an actual execution that reproduces the observed field failure. We expect such an execution to be usable, like any other failing execution, to debug the problem causing the failure.

Overall, we believe that our initial results show that the approach is promising and identify several research directions that it would be worth pursuing; directions that could be investigated by building on the work presented in this paper, as we discuss in Section VI.

V. RELATED WORK

Debugging is an extremely prolific area of research, and the related work is consequently vast. In this section, we focus on the work that is most closely related to our approach.

Our work is related to automated test-input generation techniques, such as those based on symbolic execution (*e.g.,* [19], [20], [23], [30]) and random generation (*e.g.,* [31], [32]). Generally, these techniques aim to generate inputs to discover faults, and they are not directly applicable to the problem we are targeting, as we have discussed in Section IV-B.

Techniques that capture program behaviors by monitoring or sampling field executions are also related to ours (*e.g.,* [5], [7], [11]). These techniques usually record execution events and possibly interactions between programs and the running environment to later replay or analyze them in house. These approaches tend to either capture too much information, and thus raise practicality and privacy issues, or too little information, and thus be ineffective in our context.

More recently, researchers started investigating approaches to reproduce field failures using more limited information. For example, some researchers used weakest preconditions to find inputs that can trigger certain types of exceptions in Java programs [33]–[35]. These approaches, however, target only certain types of exceptions and tend to operate locally at the module level. Another approach, SherLog [36], makes use of run-time logs to reconstruct paths near logging statements to help developers to identify bugs. LogEnhancer [37], a followup work, improves the diagnose ability of SherLog by adding more information to log messages. These approaches differ from ours because they do not aim to generate program

inputs, but rather to highlight code areas potentially related to a failure. Artzi and colleagues present ReCrash [38], a technique that records partial object states at method levels dynamically to recreate an observed crash at different levels of stack depth. Although this approach can help recreate a crash, the recreated crash can be very different from the original one: if the stack depth is low, the information is in most cases too local to help (*e.g.,* a null value passed as a parameter); conversely, if the stack depth is high, the technique may have to collect considerable amounts of program state, which can make it impractical and raise privacy issues.

The two approaches most related to ours are ESD, by Zamfir and Candea [27] and the technique by Crameri, Bianchini, and Zwaenepoel [39]. ESD is a technique for automated debugging based on input generation. Given a POF, ESD uses symbolic execution to try to generate inputs that would reach the POF. As we showed in this paper, without additional guidance, symbolic execution techniques are unlikely to be successful in this context. In fact, as we stated in Section IV-B, the programs and failures used in Reference [27] can also be recreated through unguided symbolic execution. Unlike BUGREDUX, however, one of the strengths of ESD is that it can recreate concurrency-related failures. With this respect, BUGREDUX and ESD are complementary, and it would be interesting to investigate a combination of the two techniques. (Another approach that aim to reproduce concurrency bugs, and is thus also complementary to BUGREDUX, is PRES, by Park and colleagues [40].) The approach by Crameri and colleagues improves ESD by using partial branch traces—where the relevant branches are identified through a combination of static and dynamic analysis—to guide symbolic execution for field failures reproduction. Although their approach can reduce dramatically the number of branches considered, we found in our experience that the use of branch-level traces can be problematic. Their empirical evaluation is also performed on programs whose failures can be reproduced using unguided symbolic execution. It is therefore unclear whether their approach would work on larger programs and harder-to-reproduce failures.

It is nowadays common practice to use software (*e.g.,* Breakpad [41]) or OS capabilities (*e.g.,* Windows Error Reporting [2] and Mac OS Crash Reporter [3]) to automatically collect crash reports from the field. As we discussed earlier, these reports can be used to correlate different failures reported from the field. DebugAdvisor [42], for instance, is a tool that analyzes crash reports to help find a solution to the reported problem by identifying developers, code, and other known bugs that may be correlated to the report. Although these techniques have been shown to be useful, they target a different problem, and the information they collect is too limited to allow for recreating field failures.

VI. Conclusion and Future Work

The ability to reproduce an observed failure has been reported as one of the key elements of debugging. Whereas recreating failures that occur during in-house testing is usually easy, doing so for failures that occur in the field, on user machines, is unfortunately an arduous task. To address this problem, we have presented BUGREDUX, a general approach for supporting in-house debugging of field failures. Our approach works by (1) collecting data about failing program executions in the field, (2) extracting from the collected execution data a sequence of intermediate goals (*i.e.,* statements in the program), and (3) using input generation techniques to synthesize, in house, executions that reach such goals and mimic the observed failures.

To better understand the tradeoffs between amount of information collected and effectiveness of the approach, we have performed an empirical investigation and studied the performance of four instances of BUGREDUX that leverage different kinds of execution data. We have applied these four instances to a set of 16 failures for 14 real-world programs and compared their cost and effectiveness. Our results are encouraging and provide evidence that BUGREDUX can reproduce observed failures starting from a suitable set of execution data. In addition, the analysis of the results led to several findings, some of which unexpected (*e.g.,* more information is not always better). Finally, our results provide insight that can guide future work in this area.

Our immediate plan for future work is to perform additional experiments on a larger set of programs and failures. We will also investigate the use of other kinds of execution data. Despite the good performance of call sequences in our initial investigation, collecting execution data whose size is bounded in the size of the program would be preferable in many cases. One possibility would be to use of execution data consisting of dynamic models of the program, such as dynamic call graphs, to prune the search space during input generation. Another alternative would be the investigation of efficient (and possibly partial) ways to represent potentially unbounded data, for example using some form of automata. Our approach currently assumes that all parts of a failing execution are equally relevant when trying to reproduce the failure. Intuitively, some parts of the execution, or even of a program in general, may be more relevant than others. We believe that collecting information at different levels of details for different parts may allow for an accurate reproduction of the failure even in the presence of less data. Our preliminary study on the use of partial call sequences provides supporting evidence for this research direction. Finally, symbolic execution is just one possible way to generate execution that can reproduce an observed failure. We will investigate alternative techniques for synthesizing failing executions, such as techniques based on backward (rather than forward) exploration (*e.g.,* [33]), techniques based on genetic algorithms (*e.g.,* [43]), and techniques that leverage existing test inputs using some form of fuzzing (*e.g.,* [14]).

Acknowledgments

This work was supported in part by NSF awards CCF-0916605 and CCF-0964647 to Georgia Tech, and by funding from IBM Research and Microsoft Research.

REFERENCES

[1] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schröter, and C. Weiss, "What makes a good bug report?" *IEEE Transactions on Software Engineering*, vol. 36, no. 5, pp. 618–643, September 2010.

[2] "Windows Error Reporting: Getting Started," March 2011, http://www.microsoft.com/whdc/maintain/StartWER.mspx.

[3] "Technical Note TN2123: CrashReporter," March 2011, http://developer.apple.com/technotes/tn2004/tn2123.html.

[4] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani, "HOLMES: Effective statistical debugging via efficient path profiling," in *Proceedings of the 31st International Conference on Software Engineering*, 2009, pp. 34–44.

[5] J. Clause and A. Orso, "A technique for enabling and supporting debugging of field failures," in *Proceedings of the 29th international conference on Software Engineering*, 2007, pp. 261–270.

[6] B. Liblit, "Cooperative Bug Isolation," Ph.D. dissertation, University of California, Berkeley, 2004.

[7] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable Statistical Bug Isolation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005, pp. 15–26.

[8] D. M. Hilbert and D. F. Redmiles, "Extracting Usability Information from User Interface Events," *ACM Computing Surveys*, vol. 32, no. 4, pp. 384–421, Dec 2000.

[9] C. Pavlopoulou and M. Young, "Residual Test Coverage Monitoring," in *Proceedings of the 21st International Conference on Software Engineering*, 1999, pp. 277–284.

[10] S. Elbaum and M. Diep, "Profiling Deployed Software: Assessing Strategies and Testing Opportunities," *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 312–327, 2005.

[11] "The Amazing VM Record/Replay Feature in VMware Workstation 6," March 2011, http://blogs.vmware.com/sherrod/2007/04/the_amazing_vm_.html.

[12] L. Jiang and Z. Su, "Context-aware Statistical Debugging: From Bug Predictors to Faulty Control Flow Paths," in *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, 2007, pp. 184–193.

[13] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[14] "tmin: Fuzzing Test Case Optimizer," April 2009, http://code.google.com/p/tmin/.

[15] M. Castro, M. Costa, and J.-P. Martin, "Better Bug Reporting with Better Privacy," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008, pp. 319–328.

[16] J. Clause and A. Orso, "Camouflage: automated anonymization of field data," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 21–30.

[17] "Coreutils - GNU core utilities," http://www.gnu.org/software/coreutils/, GNU.

[18] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[19] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," in *Proceedings of the 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2005, pp. 263–272.

[20] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming language design and implementation*, 2005, pp. 213–223.

[21] W. Jin and A. Orso, "BugRedux: Reproducing Field Failures for In-house Debugging," Georgia Institute of Technology, Tech. Rep. GIT-CERCS-11-06, August 2011, http://www.cercs.gatech.edu/tech-reports/.

[22] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[23] C. Cadar, D. Dunbar, and D. Engler, "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*, 2008, pp. 209–224.

[24] "Software-artifact Infrastructure Repository," http://sir.unl.edu/, university of Nebraska - Lincoln.

[25] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "Bugbench: Benchmarks for evaluating bug detection tools," in *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[26] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "Aeg: Automatic exploit generation," in *Proceedings of the 18th Network and Distributed System Security Symposium*, February 2011.

[27] C. Zamfir and G. Candea, "Execution synthesis: a technique for automated software debugging," in *Proceedings of the 5th European conference on Computer systems*, 2010, pp. 321–334.

[28] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated Whitebox Fuzz Testing," in *Proceedings of the Network and Distributed System Security Symposium*, 2008.

[29] C. Parnin and A. Orso, "Are Automated Debugging Techniques Actually Helping Programmers?" in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, Toronto, Canada, July 2011, pp. 199–209.

[30] W. Visser, C. S. Păsăreanu, and S. Khurshid, "Test Input Generation with Java PathFinder," *SIGSOFT Software Engineering Notes*, vol. 29, no. 4, pp. 97–107, 2004.

[31] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for Java," in *Proceedings of the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, 2007, pp. 815–816.

[32] C. Pacheco, S. K. Lahiri, and T. Ball, "Finding errors in .Net with feedback-directed random testing," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, 2008, pp. 87–96.

[33] S. Chandra, S. J. Fink, and M. Sridharan, "Snugglebug: a powerful approach to weakest preconditions," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming language design and implementation*, 2009, pp. 363–374.

[34] M. G. Nanda and S. Sinha, "Accurate interprocedural null-dereference analysis for java," in *Proceedings of the 31st International Conference on Software Engineering*, 2009, pp. 133–143.

[35] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for java," in *Proceedings of the 2002 ACM SIGPLAN Conference on Programming language design and implementation*, 2002, pp. 234–245.

[36] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "Sherlog: error diagnosis by connecting clues from run-time logs," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010, pp. 143–154.

[37] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011, pp. 3–14.

[38] S. Artzi, S. Kim, and M. D. Ernst, "Recrash: Making software failures reproducible by preserving object states," in *Proceedings of the 22nd European conference on Object-Oriented Programming*, 2008, pp. 542–565.

[39] O. Crameri, R. Bianchini, and W. Zwaenepoel, "Striking a new balance between program instrumentation and debugging time," in *Proceedings of the 6th European conference on Computer systems*, 2011, pp. 199–214.

[40] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu, "Pres: probabilistic replay with execution sketching on multiprocessors," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 177–192.

[41] "Breakpad," http://code.google.com/p/google-breakpad/, google and Mozilla.

[42] B. Ashok, J. Joy, H. Liang, S. K. Rajamani, G. Srinivasa, and V. Vangala, "Debugadvisor: a recommender system for debugging," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2009, pp. 373–382.

[43] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global, and hybrid search," *IEEE Transactions on Software Engineering*, vol. 36, no. 2, pp. 226 –247, March–April 2010.