BugRedux: Reproducing Field Failures for In-house Debugging

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

***Abstract*—When a software system fails in the field, on a user machine, and the failure is reported to the developers, developers in charge of debugging the failure must be able to reproduce the failing behavior in house. Unfortunately, reproducing field failures is a notoriously challenging task that has little support today. Typically, developers are provided with a bug report that contains data about the failure, such as memory dumps and, in the best case, some additional information provided by the user. However, this data is usually insufficient for recreating the problem, as recently reported in a survey conducted among developers of the Apache, Eclipse, and Mozilla projects. Even more advanced approaches for gathering field data and help in-house debugging tend to collect either too little information, which results in inexpensive but often ineffective techniques, or too much information, which makes the techniques effective but too costly. To address this issue, we present a novel general approach for supporting in-house debugging of field failures, called BUGREDUX. The goal of BUGREDUX is 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 use a tool that implements our approach to recreate 17 failures of 15 real-world programs. Our results are promising and lead to several findings, some of which unexpected. In particular, they show that by collecting a suitable yet limited set of execution data the approach can synthesize in-house executions that reproduce the observed failures.**

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 not only difficult to foresee, but they are also often difficult, if not impossible, to reproduce outside the time and place in which they occurred. In fact, a recent survey among many 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 test cases) as the most valuable—and also most 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 such as 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.

To better support developers in their debugging activities, researchers have also investigated more sophisticated techniques for capturing data from deployed applications (*e.g.,* [4]–[10]) and using them for debugging. Among these techniques, some collect only limited amounts of information (*e.g.,* sampled branch profiles in the case of 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, and researchers have 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 (1) they can be expensive to collect and (2) they 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, which means that they would eliminate sensitive data only by chance (*e.g.,* [13], [14]), or are still in their early phase of development and in need of 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, where by analogous we mean that 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 program P, that is, the approach should be sound and generate an actual input that, when provided to P, results in execution E' and, thus, failure F'. *Third*, the approach should be able to generate E' using only P and D, without the need of 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 (and a tool that implements the 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 considered four types of increasingly rich execution data: points of failure, stack traces, 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 on 17 failures of 15 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 a call stack at the point of failure, is typically not enough for recreating failures in house, and richer information is needed.

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 show, using call-sequence data, BUGREDUX was able to recreate 16 out of the 17 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 framework for collecting execution data in the field and using the data to synthesize executions that can help reproduce field failures.
- The instantiation of the framework for four different kinds of execution data and one execution synthesis technique and its implementation in a tool that is freely available for download as open source (http://www.cc.gatech.edu/~orso/software/bugredux.html).
- An empirical study in which we analyze the tradeoffs offered by the different types of data considered in terms of cost of the data collection and ease of synthesizing failing executions.

II. BACKGROUND AND TERMINOLOGY

Before discussing our approach, we briefly provide some necessary background information on symbolic execution and define some terms that we use in the rest of the paper.

Symbolic Execution: In its most general formulation, symbolic execution is a technique that executes a program using symbolic instead of concrete inputs. At any point in the computation, the program state consists of a *symbolic state* expressed as a function of the inputs; and the conditions on the inputs that cause the execution to reach that point is expressed as a set of constraints in conjunctive form called the *path condition (PC)* [18]. More formally, the symbolic state can be seen as a map $\mathcal{S} : \mathcal{M} \mapsto \mathcal{E}$, where $\mathcal{M}$ is the set of memory addresses for the program, and $\mathcal{E}$ is the set of possible symbolic values, that is, expressions in some theory $\mathcal{T}$ such that all free variables are input values.

Both the symbolic state and the PC are built incrementally during symbolic execution, with PC initialized to `true`, each input expressed as a symbolic variable, and $\mathcal{S}$ initialized according to the semantics of the language. (In C, for instance, memory addresses not yet initialized could be mapped to $\perp$ to indicate that they are undefined.) Every time a statement $stmt$ that modifies the value of a memory location m is executed, the new symbolic value e' of m is computed according to $stmt$'s semantics, and $\mathcal{S}$ is updated by replacing the old expression for m with e' ($\mathcal{S}' = \mathcal{S} \oplus [m \mapsto e']$, where $\oplus$ indicates an update). Conversely, when a predicate statement $pred$ that modifies the flow of control is executed, symbolic execution forks and follows both branches. Along each branch, the PC is augmented with an additional conjunct that represents the input condition, expressed in terms of symbolic state, that makes the predicate in $pred$ `true` or `false` (depending on the branch). Note that, for simplicity, we assume that conditional statements only have two branches; extending to the case of more than two branches is trivial. We also assumed that both branches are feasible; a Satisfiability Modulo Theories (SMT) solver could be invoked at every branch to check whether the PC for that branch has a solution

```
function foo(int a, int b, int c) {
 1. int d = a + 4
 2. if (d < b)
 3.    //do something
 4. if (b > 5)
 5.    //do something
 6. else if (a < 5)
 7.   if (d < c)
 8.     //do something
 9.   else
10.     //do something
11. else
12.   //do something
13. return
}
```

Figure 1. Simple code example to illustrate symbolic execution.

and, if the solver provides a response, eliminate the branch from consideration if this is not the case.

Symbolic execution, when successful, can be used to compute an input that would cause a given path to be executed or a given statement to be reached. To do so, at program exit or at a point of interest in the code, the PC for that point would be fed to an SMT solver, which would try to find a solution for PC. Such a solution would consist of an assignment to the free variables in PC (*i.e.,* , the inputs) that satisfies PC. If such a solution is found, the corresponding input is exactly the input that we wanted to compute. To illustrate symbolic execution with an example, consider the code snippet in Figure 1. We indicate the symbolic inputs for the parameters a, b, and c with a_0, b_0, and c_0. When symbolic execution follows path $\langle 1, 2, 3, 4, 6, 7, 8, 13\rangle$, for instance, the symbolic state at statement 13 is $\{[a \mapsto a_0], [b \mapsto b_0], [c \mapsto c_0], [d \mapsto a_0 + 4]\}$, and the corresponding PC would be $(a_0 + 4 < b_0) \wedge (b_0 <= 5) \wedge (a_0 < 5) \wedge (a_0 + 4 < c_0)$, which corresponds to the conjunction of the predicates for branches $2T$, $4F$, $6T$, and $7T$. A possible solution for this PC is the set of assignments $a_0 = 0$, $b_0 = 5$, and $c_0 = 5$, which correspond to an input $i = \{0, 5, 5\}$ that causes path $\langle 1, 2, 3, 4, 6, 7, 8, 13\rangle$ to be followed.

Terminology: A *control flow graph (CFG)* for a function f is a directed graph $G = \langle N, E, entry, exit\rangle$ where N is a set of nodes that represent statements in f and $E \subseteq N \times N$ is a set of edges that represent the flow of control between nodes, and $entry \in N$ and $exit \in N$ are the unique entry and exit points, respectively, for the CFG.

An *interprocedural control flow graph (ICFG)* is a graph built by composing a set CFGs. To build an ICFG, CFGs are connected based on call relationships between the functions they represent. If a function f_1 calls a function f_2, the two CFGs for f_1 and f_2, G_1 and G_2, are connected as follows: the node n in f_1 representing the call site to f_2 is replaced by two nodes n_c (call node) and n_r (return node), such that all predecessors of n are connected to n_c, and n_r is connected to all successors of n. Then, n_c is connected to G_2's entry node, and G_2's exit node is connected to n_r. This process is repeated for every call site in the program.

Given a program P, a failing execution E of P for a given input I, and the resulting failure F, we define the following terms. We call F a *field failure* if it occurred on a user machine, after P has been deployed. A *point of failure (POF)* is the statement in P where F manifests itself. For the sake of the discussion, and without loss of generality, we assume that a failure corresponds to a failing assertion, and that POF is the statement in which the assertion fails (all failure conditions can be expressed in the form of assertions in the code). A *failure call stack* for F is the ordered list of functions that were on the call stack when F occurred. Each entry in the list consists of a function and a location in the function (*i.e.,* either the location of the call to the next function in the list or, for the last entry, the location of the failure). In the rest of the paper, we refer to the failure call stack for F as F's call stack or simply call stack, except for cases where the term may be ambiguous. A *call sequence* for E is the sequence of calls executed (*i.e.,* call sites traversed) during E. A *complete trace* for E is the sequence of all branches (*i.e.,* program predicates and their outcomes) exercised during E. Obviously, complete traces subsume call sequences. We use the term *execution data* for E to refer to any dynamic information collected during E. Therefore, call sequences and complete traces are examples of execution data. Finally, a *crash report* for F is a record that is produced when F occurs and can be later sent to P's developers. Although crash reports can have different formats and contents, we assume that a crash report contains at least a POF and a call stack, and possibly some additional execution data.

III. 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 2, 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 example program takes a string argument from the command line and consists of 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 by invoking 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 the $null$ character otherwise. All other characters are simply copied to the output string unmodified (lines 31–32). After all input characters have been

```
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 2. Example of faulty program.

processed, function `process` calls function `printresult`, which prints out the generated 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, probably more serious, 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 array

```
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 3. Crash report for our example program.

can contain 256. Because it requires a specially-crafted input to be triggered, such a fault that 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.

Let us assume that the program is released with the fault at line 30 that 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 257th character in the output array. In this situation, the runtime system would generate a crash report such as the one shown in Figure 3. As the figure shows, the crash report lists the POF for F (line 27 of `example.c`), and the call stack at the moment of failure, with one entry per call 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. Imagine, for instance, that the developer tried to use purely random testing to reach the POF and generate F. It would be unlikely that such an approach could generate, by chance, an input that satisfies the failing conditions for F.

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 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, the POF and the 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 and, specifically, information about the length of the input and the number of escape characters in it. However, 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

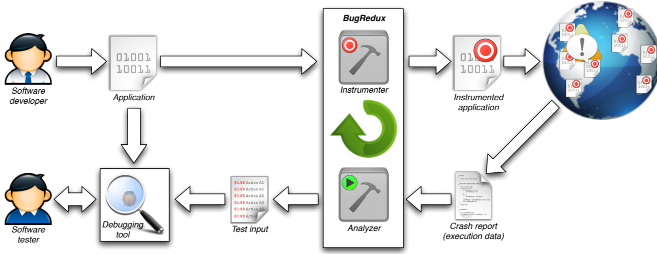

Figure 4. Intuitive high-level view of BUGREDUX.

execution data in this context. To investigate this issue, we defined a general approach for synthesizing executions that (1) mimic executions that resulted in field failures and (2) try to 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. We also show how (one instance of) the approach would be able to recreate the failure in our example.

IV. A General Approach for Recreating Field Failures

As stated in the Introduction, our overall goal is to be able to recreate field failures in a faithful way (*i.e.,* in a way that allows for debugging the failures) by using information collected in the field (*i.e.,* execution data) that can be gathered without imposing too much space and time overhead to the 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 4 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 component is the *instrumenter*, which takes as input an *application* provided by a *software developer* and generates an *instrumented application* that can collect execution data while being executed in the field and add the 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 try to debug the field failure. This general approach can be defined in different ways 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 5, which

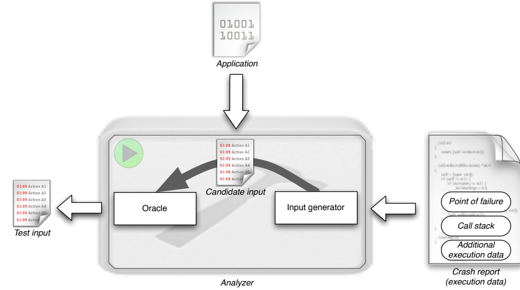

Figure 5. The analysis component of BUGREDUX.

provides a more detailed view of the analysis component of BUGREDUX, puts the problem in context and let 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]. (It is worth noting that we also considered the use of weakest precondition analysis as an alternative to symbolic execution. Because these two techniques are almost dual, with one being guided from program entry to the point of failure, and the other from the point of failure to program entry, we opted for the approach for which a more mature infrastructure is available.)

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 ICFG 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 IV-C.)

Before starting the symbolic execution, GenerateInputs performs some initializations (lines 2–4). First, it initializes

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) == true 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
```

sym_state_0 with the initial 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 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 reaches). 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 iteration of the loop, GenerateInputs invokes algorithm SelNextState, shown in Algorithm 2. SelNextState checks all the states in the $states_set$, looking for the most promising state to explore. (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 $mindis$, 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 has a cl that can reach $curr_goal$, 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`. If so, all states in $states_set$ with target $curr_goal$ cannot reach $curr_goal$. Therefore, GenerateInputs would backtrack by updating $curr_goal$ to the previous goal in the $goals_list$ and thus looking for another path that can reach such goal (line 11).

If GenerateInputs finds a candidate state $curr_state$ whose code location corresponds to $curr_goal$, it 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 GenerateInputs returns a candidate state $curr_state$ whose code location is not $curr_goal$ but another goal in $goal_list$, the algorithm removes $curr_state$ from $state_set$ (lines 32–34) because $curr_state$ would reach the goals in the goal list in a different order from the one observed in the failing execution.

When the symbolic execution reaches 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, and remove infeasible states from $states_set$ (lines 38–47).

For statements other than conditional statements, 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 back trace from G_0 (line 14)) or a candidate input is successfully generated (line 22). In the former case, our algorithm fails to find a suitable test input. The latter case, conversely, corresponds to a successful run of the algorithm.

In summary, our symbolic execution technique has two

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
```

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. Applying the algorithm on example code

We concisely illustrate how we can use a call sequence for a failing execution involving our example code in Figure 2 to demonstrate how our algorithm works in this case. An example of failing input is an input of the program starting with two occurrences "$\backslash n$" and followed by 251 characters of "a". A call sequence for this input is $\{44, 29, 27, 29, 27, 27, ..., 27\}$.

Our algorithm will start with a state that can execute the call site at line 44. Then, because our algorithm always selects the state with target $curr_goal$, a same state will always be picked by our algorithm for finding the shortest path to reach goals in $goal_list$ in the given order. The state that reaches these goals will be selected and executed during the search. After the selected state reaches all goals in the list, the PC would be $(input[0] = \backslash\backslash) \wedge (input[2] = \backslash\backslash) \wedge (input[1] \geq a) \wedge (input[1] \leq z) \wedge (input[3] \geq a) \wedge (input[3] \leq z) \wedge ... \wedge (input[255] \geq a) \wedge (input[255] \leq z)$. By feeding this path condition to the constraint solver, the algorithm can generate an input that has a similar string structure as the original input and triggers the same failure.

C. Execution Data

In selecting the execution data to consider, we wanted to cover a broad spectrum of possibilities. To this end, we selected four kinds of data: POFs, call stacks, call sequences, and complete program traces. Note that the 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.

Given these four kinds of execution data, we instantiated four variations of BUGREDUX that collected and used the four 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. All that is needed to use them is simply to extract them from existing reports. The other two types of execution data can be collected by means of well-understood program instrumentation. To collect call sequences, BUGREDUX instruments all call sites in the program,[1] whereas to collect program traces it instrument all branches within the program.

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.

V. EMPIRICAL INVESTIGATION

In our empirical investigation, we addressed the following research questions:

- **RQ1:** Can BUGREDUX synthesize executions that are able to reproduce field failures starting from a set of execution data?
- **RQ2:** If so, which types of field 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 sections and applied them to a set of real-world programs. In the rest of this section, we discuss our implementation of BUGREDUX, the program and failures we used, our experimental setup, and the results of our study.

[1]Note that, for engineering reasons related to the possible presence of function pointers, BUGREDUX actually instruments also function entry points.

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 4 and 5): 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 [21], a symbolic execution engine for C programs. KLEE is an ideal choice for us because it provides all of the basic symbolic-execution functionality, which we could reuse, and is easily extensible and customizable. We were able to implement Algorithms 1 and 2 as a custom search strategy for KLEE and only had to perform minor modifications to the underlying code. Finally, BUGREDUX's oracle module is implemented as a simple 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. (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; it reports a success otherwise.

B. *Program Subjects*

To investigate our research questions in a real(istic) setting, we decided to use a set of real, non-trivial programs that contained one or more faults and had test cases that could reveal such faults. To do this, we considered programs from three public repositories that have been used extensively in previous research: SIR [22], BugBench [23], and exploit-db [24]. 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 and, inparentheses and only for exploit-db, the ID of the corresponding repository entry; size; and the number of faults it contains. As the table shows, the program sizes range between 0.5 kLOC and 241 kLOC, and each program contains one or two faults. The faults in the BugBench's and exploit-db's 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 [24], [25] 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 our current prototype, and more specifically its underlying symbolic execution engine, 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 output. This choice was simply 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.

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 (CVE-2005-1019)	mail relay agent	3	1
iwconfig	exploit-db (CVE-2003-0947)	wireless tool	11	1
glftpd	exploit-db (OSVDB-ID#16373)	FTP server	6	1
htget	exploit-db (CVE-2004-0852)	file grabber	3	1
socat	exploit-db (CVE-2004-1484)	multipurpose relay	35	1
tipxd	exploit-db (OSVDB-ID#12346)	IPX tunneling daemon	7	1
aspell	exploit-db (CVE-2004-0548)	spell checker	0.5	1
exim	exploit-db (EDB-ID#796)	message transfer agent	241	1
rsync	exploit-db (CVE-2004-2093)	file synchronizer	67	1
xmail	exploit-db (CVE-2005-2943)	email server	1	1

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 the faults considered could have been discovered by a technique that blindly tries to cover as much of the programs as possible. If so, this would be an indication that the faults are easy to reveal and, thus, are probably not good candidates for our study. An unmodified version of KLEE was able to discovered the faults in only one of the subjects: iwconfig.

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)

Table II
TIME OVERHEAD IMPOSED ON THE SUBJECT PROGRAMS.

Name	POF	Call stack	Call sequence	Complete trace
sed.fault1	0	0	4.5%	27.2%
sed.fault2	0	0	12.5%	87.5%
grep	0	0	47%	182%
gzip.fault1	0	0	10.3%	72%
gzip.fault2	0	0	12%	308%
ncompress	0	0	2%	16%
polymorph	0	0	1%	8%
aeon	0	0	50%	1066%
iwconfig	0	0	7%	128%
glftpd	0	0	9%	45%
htget	0	0	9%	287%
socat	0	0	21%	110%
tipxd	0	0	2%	36%
aspell	0	0	18.8%	143%
exim	0	0	17.4%	389%
rsync	0	0	3%	66%
xmail	0	0	22.6%	290%

Table III
SIZE OF EXECUTION-DATA COLLECTED (KB).

Name	POF	Call stack	Call sequence	Complete trace
sed.fault1	0.8	0.8	5.8	54.4
sed.fault2	0.9	0.9	10.2	261.9
grep	0.7	0.7	3.4	716.1
gzip.fault1	0.8	0.8	2.0	176
gzip.fault2	0.8	0.8	2.5	1784.6
ncompress	0.7	0.7	0.9	33.1
polymorph	0.5	0.5	0.7	1.5
aeon	1	1	1.1	3
iwconfig	1	1	3	594
glftpd	1.5	1.5	3.2	130
htget	0.7	0.7	2.7	2814
socat	0.8	0.8	9.6	451
tipxd	0.6	0.6	0.7	19
aspell	0.6	0.6	30.5	566
exim	0.9	0.9	100.7	14897
rsync	1	1	11.4	521
xmail	0.8	0.8	84.8	2361

and how long it took for the synthesis. 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 four tables, where the first two tables contain the data related to the cost of the approach (*i.e.,* the time and space overhead imposed by BUGREDUX), and the last two tables show 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). We provide a set of tables that present the results for each of the 17 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 overhead imposed by BUGREDUX on the subject programs, measured as the percentage increase of the running time due to the instrumentation added by BUGREDUX. Because POF 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 IV-C). 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 the results, it is important to consider that this data was collected with a straw-man instrumentation that simply 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 such optimization for future work.

Table III shows the size of the different kinds of execution data collected by BUGREDUX. Note that the data size for POFs and call stacks is the same because our current implementation of BUGREDUX extracts them both 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, for the executions considered, the size of the execution data is fairly contained, and it would be practical to collect them.

The last two tables are the most relevant for our investigation, as they address the core question of the effectiveness of the approach. Specifically, Table IV shows how many failing executions BUGREDUX was able to mimic and, for those, how long it took for the input generation. Entries of type "N/A" in the table indicate cases where the input generator was not able to produce a candidate input within the time limit assigned (72 hours). As expected, symbolic execution guided only by the POF is unlikely to be successful 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 more information is available in the form of intermediate goals that can guide the exploration, the better the performance of the approach. Using stack traces, BUGREDUX can mimic 11 out of the 17 failing executions, and using call sequences, it can mimic all but 1 failing execution (the one in program exim).

In some cases (*e.g.,* htget, tipxd), the time needed to synthesize an execution using call stacks is larger than the time needed in the case of call sequences, when they are both successful. The reason for this result is that the additional information provided by call sequences can better guide sym-

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

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

Table V
EFFECTIVENESS OF SYNTHETIC EXECUTIONS IN REPRODUCING OBSERVED FAILURES.

Name	POF	Call stack	Call sequence	Complete trace
sed.fault1	N	N	Y	N
sed.fault2	N	N	Y	N
grep	N	N	Y	N
gzip.fault1	Y	Y	Y	N
gzip.fault2	N	N	Y	N
ncompress	Y	Y	Y	N
polymorph	Y	Y	Y	N
aeon	Y	Y	Y	Y
rsync	N	N	Y	N
iwconfig	Y	Y	Y	N
glftpd	Y	Y	Y	N
htget	N	N	Y	N
socat	N	N	Y	N
tipxd	Y	Y	Y	N
aspell	N	N	Y	N
xmail	N	N	Y	N
exim	N	N	N	N

bolic execution and avoid many irrelevant paths. The reduced number of explored paths would obviously result in less time needed for synthesizing an execution.

One surprising finding is that this trend is not confirmed when complete traces are used as a guide. We investigated this behavior and found that the main reason for this behavior is that, intuitively, complete traces somehow provide too much guidance to the input generator. Having a complete trace forces symbolic execution to follow exactly the same path as the original failing execution, which is useful in terms of imitating the failure's behavior, but in most cases results in potentially long paths that contain conditions that the SMT solver cannot handle. 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 would be dropped in favor of simpler paths that may still reach the targeted goal. 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 these synthetic executions 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 2. 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 III. This is especially true when considering the more limited types of execution data, such as POFs and call stacks, which provide little guidance to the search. The results in Table V clearly illustrate this issue. In the table, a "Y" (resp., "N") in a cell indicates that, for the failure corresponding to that row, the synthetic execution generated by BUGREDUX using the execution data of the corresponding column was able (resp., was unable) to reproduce the failure. (The values in the last column are all "N" except for aeon because, as we saw in Table IV, BUGREDUX was not able to generate synthetic executions for most of the failures when using complete traces.)

As shown in the table, for the seven simple programs in our set reaching 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. All synthetic executions but one (for one of exim's failures) generated from call sequences were able to reproduce the original failures, which provides clear evidence of the usefulness of this type of execution data.

E. Discussion

The results of our investigation, albeit preliminary, let us address our two research questions and make some observations. As far as RQ1 is concerned, our results provide initial evidence 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 16 out of 17 observed failures; even using an unoptimized instrumentation, BUGREDUX was able to collect call sequences with an acceptable time and space overhead; and in general, 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 by synthesizing call sequences corresponds 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 an execution. The fact that BUGREDUX was able to synthesize failing executions using call sequences and not complete traces should not be misleading in this respect. Consider that (1) the inputs to the failing executions may have little to do with those of the synthesized execution[2] and (2) a complete trace may

[2]For example, in program ncompress, the original input that makes the program fail contains 1520 "a" characters, whereas the input generated by BUGREDUX is a sequence of more than 1600 random characters.

Table VI
MINIMAL NUMBER OF ENTRIES IN CALL SEQUENCES REQUIRED TO REPRODUCE FAILURES

Name	Original Length	Minimal Length
sed.fault1	81	45
grep	34	3
gzip.fault1	17	1
gzip.fault2	27	2
rysnc	23	2
aspell	516	256

be difficult to handle for an algorithm, but may reveal much more information to a human being.

An additional observation that can be made based on the results is that POFs and call stacks do not appear to be very useful in reproducing failures. Manual examination of the faults 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 point of failure. 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 two types of execution data are normally collected in crash report. Extending crash reports with additional information may make them considerably more useful to developers.

In general, it is well known that the problem of synthesizing (inputs for) an execution that reaches a given point in a program is undecidable, as it can easily be reduced to the termination problem. Therefore, intuitively, we would expect that more guidance in the search for such an execution could only improve the chances of finding it. One interesting observation that we can derive from our results contradicts this intuition. As we discussed earlier in this section, providing too much guidance may be just as problematic as providing too little of it. In other words, these results seem to suggest that a sweet spot may be found that would allow enough freedom for the search to explore different possibilities, but also enough guidance to direct the search to its target.

As a further step towards the identification of such sweet spot, we performed an additional exploratory study in which we removed entries in call sequences and studied whether the partial sequences still contained enough information to recreate observed failures. More precisely, we selected six failures from the original list and, for each failure and corresponding call sequence, removed entries from the sequence until BUGREDUX was unable to reproduce the failure at hand using the resulting reduced sequence. Table VI shows the result of this study in terms of number of entries in the call sequences before and after reduction. For example, only 3 of the 34 entries in the original call sequence are needed to reproduce the observed failure in grep. From the results, we can observe 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 program input. For instance, sed.fault1 can only be triggered by an input file that include a sed script, 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 VII.

Limitations and Threats to Validity: The research presented is still in a relatively early stage. As such, there are limitations to the approach proposed and threats in the empirical investigation we performed. One limitation of BUGREDUX is due to the fact 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 system [26]. Moreover, as we discuss in our future work, our approach can leverage different execution-synthesis techniques. Another limitation of BUGREDUX is that we currently do not explicitly handle concurrency and non-determinism. Conceptually, we could have considered execution data that included concurrency related information, and extended BUGREDUX accordingly. However, 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. As far as our implementation of BUGREDUX is concerned, we tested it on small examples and spot checked many of the results presented in the paper, which mitigated the risk of reporting erroneous results. In terms of external validity, our results may not generalize to other programs and failures. However, we studied 17 failures and 15 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, we believe that it would be premature at this point. In fact, it is typical for most research in the area of debugging to perform extensive evaluations in-vitro to understand and fine tune the techniques before performing an expensive human study.

Overall, we believe that our results, albeit preliminary in nature, are promising and motivate further research in this area. If confirmed by additional studies and research, these results may result in new and more effective ways of debugging and improving software quality.

VI. 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]–[21], [27]) and random generation (*e.g.,* [28], [29]). Generally, these techniques target the problem of generating

inputs that trigger or discover faults in a program. As our results show, these techniques are generally not applicable to the problem we are targeting.

Techniques that capture program behaviors by monitoring or sampling field executions are also related to ours (*e.g.,* [5], [7], [11]). These techniques usually capture 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 replay 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 [30]–[32]. These approaches, however, target only certain types of exceptions and tend to operate locally at the module level. Another approach, SherLog [33], makes use of run-time logs to reconstruct paths near logging statements to help developers to identify bugs. This approach does not aim to generate program inputs, but rather to highlight code areas potentially related to a failure. Zamfir and Candea introduce a technique for automated debugging based on input generation [34]. Given a POF, they use 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. Unlike our approach, however, they can handle multi-threaded programs; it would be interesting to investigate a combination of the two techniques. Another approach that targets concurrency issues is that by Park and colleagues, who presented PRES (probabilistic replay via execution sketching) to help reproduce concurrency bugs [35]. Their technique mostly focuses on replaying issues related to concurrency, so it is also complementary to our approach.

It is nowadays common practice to use software (*e.g.,* Breakpad [36]) 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 [37], 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.

VII. Conclusion and Future Work

It is well known that the ability of reproducing an observed failure is 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, even when bug reports are available. To address this important problem, we have presented BugRedux, a general approach for supporting in-house debugging of field failures. At a high level, our approach is based on the idea of collecting data about program runs in the field, sending this information back to developers when a failure is observed, and using this information in house to mimic the failing execution and try to reproduce the observed failure. To do this, BugRedux extracts from the execution data collected in the field a sequence of intermediate goals (*i.e.,* statements in the program) and tries to generate an execution that reaches such goals. BugRedux is currently implemented for the C language and freely available for download as open source.[1]

To better understand the tradeoffs between amount of information collected and effectiveness of the approach, we performed an empirical investigation in which we studied the performance of four instances of BugRedux that leverage different kinds of execution data. We applied these four instances to a set of 17 failures for 15 real-world programs and compared their cost and effectiveness. Our results are encouraging and provide evidence that BugRedux, when operating on a suitable set of data, can successfully reproduce, in house, failures observed in a different context. In addition, some of the findings confirm our intuition, whereas other are unexpected (*e.g.,* more information is not always better than less information). Finally, the study provide insight that can guide future work in this area.

In the immediate, we will perform additional experiments to assess whether our initial results are confirmed when BugRedux is applied to a larger set of programs and failures. Another topic for future work is the identification of other kinds of execution data that may be useful in this context. Among the data we considered so far, call sequences appear to be an optimal choice. Despite their performance in our initial investigation, however, there may be cases where collecting execution data whose size is bounded in the size of the program would be preferable. This topic could be investigated along several directions. One direction could involve the use of execution data consisting of dynamic models of the program, such as dynamic call graphs, and use these models 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. If so, 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 evidence that support this research direction. Finally, symbolic execution is only 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.,* [30]), techniques based on genetic algorithms (*e.g.,* [38]), and techniques that take advantage of existing test inputs using some fort of fuzzing (*e.g.,* [14]).

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. Chilimbi, B. Liblit, K. Mehra, A. 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 IEEE and ACM SIGSOFT 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 Twenty-second 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 Sanitization of Field Data," Georgia Tech, Tech. Rep. Technical Report GIT-CERCS-09-14, 2009.

[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 / 13th ACM SIGSOFT International Symposium on 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 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005, pp. 213–223.

[21] C. Cadar, D. Dunbar, and D. Engler, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI'08: Proceedings of the 8th USENIX conference on Operating systems design and implementation*. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224.

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

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

[24] 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 (NDSS 2011)*, 2011.

[25] J. Clause and A. Orso, "Camouflage: Automated anonymization of field data," in *Proceedings of the Intenational Conference on Software Engineering (ICSE 2011)*, May 2011, to appear.

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

[27] 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.

[28] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for Java," in *OOPSLA '07: Proceedings of the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, 2007, pp. 815–816.

[29] C. Pacheco, S. K. Lahiri, and T. Ball, "Finding errors in .Net with feedback-directed random testing," in *ISSTA '08: Proceedings of the 2008 International Symposium on Software Testing and Analysis*, 2008, pp. 87–96.

[30] S. Chandra, S. J. Fink, and M. Sridharan, "Snugglebug: a powerful approach to weakest preconditions," in *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2009, pp. 363–374.

[31] M. G. Nanda and S. Sinha, "Accurate interprocedural null-dereference analysis for java," in *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 133–143.

[32] 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 ACM SIGPLAN 2002 Conference on Programming language design and implementation*, ser. PLDI '02. New York, NY, USA: ACM, 2002, pp. 234–245. [Online]. Available: http://doi.acm.org/10.1145/512529.512558

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

[34] C. Zamfir and G. Candea, "Execution synthesis: a technique for automated software debugging," in *Proceedings of the 5th European conference on Computer systems*, ser. EuroSys '10. New York, NY, USA: ACM, 2010, pp. 321–334. [Online]. Available: http://doi.acm.org/10.1145/1755913.1755946

[35] 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*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 177–192. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629593

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

[37] B. Ashok, J. Joy, H. Liang, S. K. Rajamani, G. Srinivasa, and V. Vangala, "Debugadvisor: a recommender system for debugging," in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE 2009*, 2009, pp. 373–382.

[38] 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.