RDE: Replay DEbugging for Diagnosing Production Site Failures

Peipei Wang
North Carolina State University
pwang7@ncsu.edu

Hiep Nguyen*
Google Inc.
hiepnguyen@google.com

Xiaohui Gu
North Carolina State University
gu@csc.ncsu.edu

Shan Lu
University of Chicago
shanlu@cs.uchicago.edu

***Abstract*—Online service failures in production computing environments are notoriously difficult to debug. One of the key challenges is to allow the developer to replay the failure execution within an interactive debugging tool such as GDB. Previous work has proposed in-situ approaches to inferring the production-run failure path within the production environment. However, those tools may sometimes suggest failure execution paths that are infeasible to reach by any program inputs. Moreover, production site often does not record or provide failure-triggering inputs due to the user privacy concern. In this paper, we present RDE, a Replay DEbug system that can replay a production-site failure at the development site within an interactive debugging environment without requiring user inputs. RDE takes an inferred production failure path as input and performs *execution synthesis* using a new *guided symbolic execution* technique. RDE can tolerate imprecise or inaccurate failure path information by navigating the symbolic execution along a set of selected paths. RDE synthesizes an input from the selected symbolic execution path which can be fed to a debugging tool to replay the failure. We have implemented an initial prototype of RDE and tested it with a set of coreutils bugs. The results show that RDE can successfully replay all the tested bugs within GDB.**

I. Introduction

Production systems that provide online interactive services typically require high reliability and availability. However, it is a great challenge to release a bug-free system due to the scale and complexity of the modern software. Despite extensive software testing and analysis, many failures still occur during the production run. To diagnose a production-run failure, software developers desire to be able to replay the failure at their site and use interactive debugging tools (e.g., GDB [2]) to understand what happened during the production run. Unfortunately, offline production-run failure debugging remains challenging, especially for those non-crashing failures (e.g., incorrect or unexpected results).

Much effort has been conducted to reduce the recording overhead while maintaining *debug determinism* [30] in which the replay exhibits the same failure symptom and includes the same root cause. Researchers have proposed different techniques, ranging from complete execution recording [14], [15], [17], [27] to partial record-replay [3], [8], [18], [19], [24], [28], [31], [32], in order to achieve debug determinism. However, production infrastructures are often reluctant to adopt these approaches due to the recording overhead, deployment complexity, and privacy concerns. Particularly, the original user input that triggered the production site failure is generally not provided to developers because it may contain sensitive user information [7].

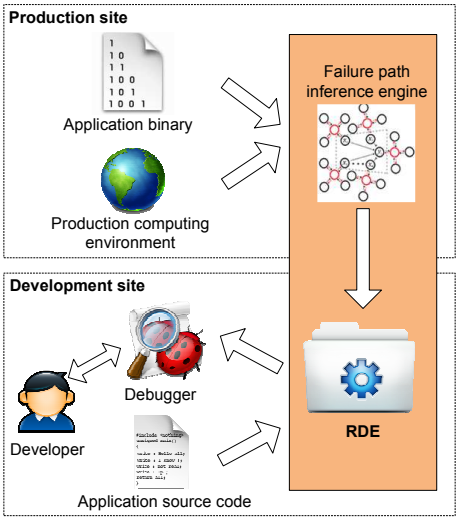


Fig. 1: Overview of RDE.

In our previous work, we have presented Insight [23], an in-situ failure path inference tool that discovers the execution path of a production-site failure immediately after a failure is detected without requiring any intrusive system recording. Although Insight can infer high fidelity failure paths, providing valuable information for understanding the failure, it does not offer any explicit support for reproducing the failure at the development site. Moreover, Insight intentionally skips the constraint checking during the failure path inference in order to achieve fast path search. As a result, Insight might occasionally generate infeasible paths which cannot be replayed in an interactive debugger.

In this paper, we introduce RDE, a Replay DEbugging system that can synthesize the production failure execution and replay the execution within an interactive debugging tool, shown by Figure 1. To protect private and confidential information, a failure report on the production site usually do not collect user information which makes bug replaying on the development site difficult. One key challenge to replay the production site failure at the development site is to handle unavailable user input and production environment information (e.g., settings from a configuration file). Figure 1 shows our approach to handle this cross-site system diagnosis problem. We leverage the failure execution path inferred on the production site by Insight. This path does not contain any user-specified

*This work has been done while the author was a PhD candidate at North Carolina State University.

information. RDE takes the failure execution path disc by Insight as the input to synthesize the failure-triggeri input and the production environment data to reprodu production-site failure execution at the development s RDE achieves practicality without requiring any user i intrusive system recording.

To synthesize the user input and the environment mation, RDE leverages symbolic execution [6], [9], execute the program symbolically along the failure ex path discovered by Insight. During the symbolic exe RDE collects a set of path constraints that the symboli must satisfy in order to cause the program to foll path. RDE then uses a Satisfiability Modulo Theories solver [12], [13] to infer concrete values for the inp environment data by solving the path constraints. Sy execution suffers from the "state space explosion" lem because the execution tree grows exponentially [computing satisfiability over a constraint set at each point is CPU-intensive. In order to perform efficie search, RDE first identifies which branches have deter condition values (i.e., non-flippable branches) accor the annotation provided by Insight[1]. During the sy execution, RDE skips exploring non-flippable branch uses them as anchor points to narrow down the search

As mentioned before, the execution path inferred situ tools may be infeasible because of skipping the co check. However, developers cannot run an infeasible ex path in an interactive debugging tool because the co solver cannot find any solution that can satisfy the con of an infeasible path. To address this problem, RDE pro new *guided* symbolic execution exploration scheme that find a feasible execution path that can reproduce the production-site failure and be replayed for debugging purposes.

We have implemented a prototype of the RDE system and evaluated it using a set of real software bugs found in online bug repositories. The results show that RDE can successfully reproduce the failure executions that exhibit the same failure symptom and cover the root causes for all the tested software bugs. Specifically, this paper makes the following contributions:

- We propose a new failure reproduction approach that can reproduce a production-site failure execution at the development site with the information collected by an in-situ failure path inference tool. By combining Insight with RDE, it provides a complete solution for cross-site system diagnosis.
- We present a guided symbolic execution exploration scheme that can synthesize the failure-triggering user inputs and environment data to reproduce a complete execution of a production-site failure.

[1]Insight [23] decides which branches are non-flippable based on the runtime console logs, system call traces, and interaction results with the production environment.

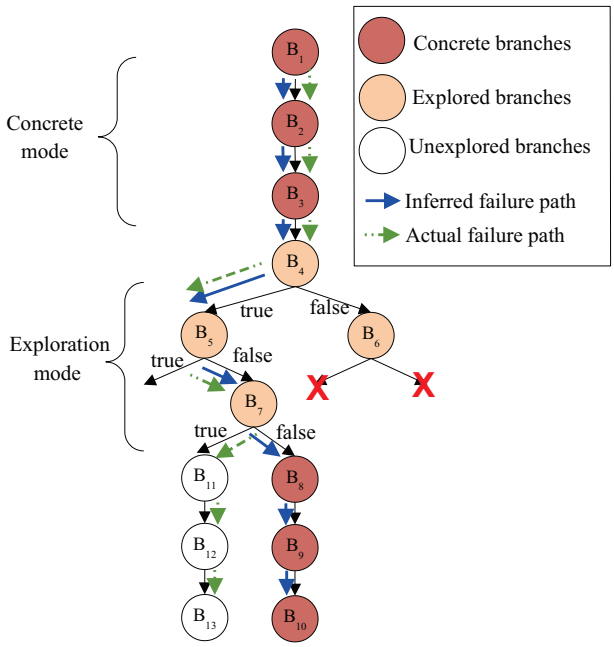


Fig. 2: Annotated branches in the path discovered by Insight.

- We evaluate RDE using five coreutils bugs and show that our approach is feasible for reproducing failure executions without requiring failure-triggering user inputs.

The rest of the paper is organized as follows. Section II presents the design of the RDE system. Section III describes the implementation of RDE. Section IV presents our experimental results. Section V compares our work with related work. Section VI discusses the limitation and the future work. Finally, the paper concludes in Section VII.

II. System Design

In this section, we first give an overview about the in-situ failure path inference tool Insight. We then describe our symbolic execution scheme for synthesizing missing production-site information (e.g., user inputs, environment data). After that, we describe how we handle the infeasible path problem. Next, we describe how developers can use RDE for debugging production-site failures.

A. In-Situ Failure Path Inference Overview

When a failure is detected in a production system, Insight dynamically creates a shadow component of the production server and performs failure path inference on the shadow component.

Insight infers the execution path of a failed service request on the production site immediately after a failure is detected. By inferring the failure path on the production site, Insight leverages both environment data (e.g., input logs, configuration files, states of interacting components) and runtime outputs (e.g., console logs, system calls) to reduce the failure path search scope. Insight performs binary-based failure path search

by replaying a recent input and matching the outputs from the replayed execution with those of the failed production run. If the replayed execution produces a mismatched console log message or system call, Insight rolls back the replay process to the previous branch point and flips its condition values to search a different path.

During the path inference process, depending on whether the environment data was changed or not after the failure occurred, Insight operates in two modes: concrete mode and exploration mode. Correspondingly, the recorded branch points are annotated with either *concrete* or *explored* shown by Figure 2. Concrete branches indicate that the environment data which influence branch conditions was unchanged before and after the failure. Thus, Insight does not need to infer the condition values of those branches using exploration. The explored branches indicate that the environment was changed after the failure occurred. Insight has to infer their condition values by exploring different paths and selecting the matched path based on recorded console logs and system calls. Therefore, we mark concrete branches as *non-flippable* since their condition values are exact and explored branches as *flippable* since their condition values come from runtime inference and maybe inaccurate.

We also rely on console log messages to determine whether a branch is flippable. Production systems often produce console log messages for debugging production-run failures. Since console log messages are inserted by software developers for recording operations considered to be important, they often can provide useful clues about program execution. Thus, we mark all the branches right before the logging statements as non-flippable on the inferred path in addition to those we annotated in the previous step.

The failure path discovered by Insight consists of an ordered list of branch points associated with their condition values and annotations indicating whether they are flippable or non-flippable branches.

B. Production-Site Data Synthesis

The goal of RDE is to be able to reproduce production failures at the development site. RDE relies on the failure path produced by Insight, called *inferred path*, to achieve practical symbolic execution for real programs with reduced scope. RDE performs symbolic execution along the inferred path to compute path constraints and synthesize the production site data (e.g., user inputs, environment data) by solving all the path constraints. Our approach is based on the premise that any input that satisfies the path constraints will lead the program to reach the failure point. We compute path constraints from the point where the input is first received by the program to the point where the failure is detected (i.e., an error message is produced). These constraints are then fed into an SMT solver [12], [13] to compute concrete values for the input.

To compute path constraints, the symbolic execution engine first makes the input of the program symbolic. It then executes the program along the path provided by Insight while keeping track of instructions on symbolic operands. When executing an instruction, if at least one of its operands is symbolic, the instruction is executed symbolically. Moreover, when the symbolic execution engine encounters a branch instruction where the branch condition depends on at least one symbolic variable (i.e., *symbolic branch*), it adds a new path constraint. Symbolic execution also generates path constraints when executing indirect call or jump if the target operand is symbolic. Similarly, constraints are also generated when load and store instructions are executed on a symbolic address operand.

Symbolic execution maintains *symbolic states* when executing the program. Each symbolic state consists of a register file, stack, heap, program counter, and path constraints on the input that cause the execution to reach a given point in the program. RDE performs symbolic execution along the inferred path provided by Insight and uses the recorded value to determine the branch condition at each branch instruction. However, branch condition value cannot be determined if it is not recorded in the inferred path (e.g., flippable branches or external library functions[2]) and the current path constraints are not enough to compute a definite concrete value for the branch condition. When the symbolic execution engine encounters a symbolic branch where the branch condition cannot be determined, it forks two symbolic states to follow both branch directions.

RDE finishes executing the program when there is one symbolic state reaching the end of the inferred path. The path constraints of the symbolic state contain all the predicates that the input and environment data must satisfy. RDE then uses the SMT solver to solve the path constraints and extract appropriate inputs and environment data. The current prototype of RDE uses an SMT solver called STP [13].

C. Infeasible Path Handling

The inferred path produced by Insight might be infeasible due to missing constraint check. The STP solver would not find any possible solution that satisfies all the path constraints if we follow the inferred path strictly. Under those circumstances, RDE uses the inferred path as guidance to find a similar but feasible execution path. To make sure the reproduced path still covers the root cause, RDE first annotates all branches in the inferred path to identify which branch is flippable and which branch is not. It then uses those annotated branches to perform guided symbolic execution exploration.

During the symbolic execution, RDE uses the recorded value for the non-flippable branch to determine the branch condition. For the flippable branch, RDE does not follow the

[2]Insight does not record the branches that belong to external libraries (e.g. $libc$) to avoid the mismatch between the execution environment of the production site and that of the development site (e.g., $libc$ vs. $ulibc$).

inferred path strictly. Instead, RDE performs normal symbolic execution for the flippable branch by forking two symbolic states corresponding to two possible branch condition values. However, the symbolic state corresponding to the recorded value of the flippable branch in the inferred path has a higher priority of being executed first compared to the other state.

Given an annotated path, RDE uses symbolic execution exploration to find a feasible path that must go through all the non-flippable branches in the path. Those branches are used as intermediate goals during the exploration.

To start the guided symbolic execution, RDE initializes the input as symbolic. It then executes the program using this initialized symbolic input. During the exploration, RDE intercepts conditional branch instructions and checks whether the branch is non-flippable. If the branch is non-flippable, RDE uses the recorded branch condition value in the inferred path to determine the direction of the execution. For the flippable branches on the inferred path, RDE performs normal symbolic execution by forking symbolic states to follow possible branch condition values.

At any time, the symbolic execution engine may be executing a large number of symbolic states. One question for RDE is which state needs to be executed first in order to reach the failure point. RDE maintains the information about which non-flippable branches each symbolic state already matched and which next non-flippable branch (i.e., the next goal) that the symbolic state needs to match. Among the candidate symbolic states that may be executed, RDE selects a state that matches the maximum number of non-flippable branches. If there are multiple ones, RDE chooses a state randomly to avoid getting stuck in an infinite loop or going too deep into an irrelevant path.

We now describe how RDE handles infeasible path using the simple example shown by Figure 2. Suppose the inferred input path is $B_1 \rightarrow B_2 \rightarrow B_3 \rightarrow B_4 \rightarrow B_5 \rightarrow B_7 \rightarrow B_8 \rightarrow B_9 \rightarrow B_{10}$ which is an infeasible path because the condition of B_7 is the false instead of the true. Let us call the first state created at the beginning of the symbolic execution the *RootState* and the set of candidate states *curStateSet*. The symbolic execution selects new states from the curStateSet and visits them one by one until the curStateSet is empty.

RDE first reads the inferred path and finds that B_4, B_5, and B_7 are flippable branches and B_1, B_2, B_3, B_8, B_9, and B_{10} are non-flippable branches. The symbolic execution begins with the RootState since the RootState is the only element in the curStateSet. The first branch RDE encounters is B_1. Since B_1 is a non-flippable branch, only one state is created for B_1 and added to the curStateSet. The RootState is removed from the curStateSet. Because of the same reason, the state for B_1 is selected to be the next state to visit. Similarly, states for B_2 and B_3 are created, added into the curStateSet, and selected to be next state in turn, separately. The symbolic execution then encounters B_4 and starts path exploration. It creates state

```
   void foo (int a)
1: if a >= 2 then
2:     //do something
3: if a <= 2 then
4:     //do something
```

Fig. 3: Simple code example to illustrate input synthesis.

S_1 for true condition value and state S_2 for false condition value, and add both of them into the curStateSet. RDE selects S_1 as the next state to visit because on the inferred path B_4 is annotated with true condition value and thus S_1 has a priority than S_2. Similarly, states S_3 and S_4 for B_5 and S_5 and S_6 are added to the curStateSet. The symbolic execution selects S_3 and S_5 because they have the false condition values annotated on the input path while the other states S_2 and S_4 have true condition value. However, the constraint solver gets an empty set through solving condition constraints. Therefore, RDE realizes that S_5 is an invalid state and terminates it immediately. RDE then selects from the remaining states in the curStateSet according to its search strategy. In the end, symbolic execution follows a feasible path although its input execution path is infeasible.

D. Replay Debugging

Once the symbolic execution finishes executing a complete path, RDE solves the path constraints to generate concrete values for all the required program inputs and uses them to replay the failure execution. To illustrate the input synthesis, consider the code snippet in Figure 3. If the symbolic execution follows path $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$, the corresponding constraint is $(a \geq 2) \wedge (a \leq 2)$. A concrete value of a that satisfies the path constraints is 2. To perform replay, we use a driver which reads the synthesized input and passes it to the program during the replay. More specifically, the driver intercepts access to symbolic variables and replaces them with concrete values. Developers can run the buggy program with the synthesized input and attach it with a debugger (e.g. GDB) to perform debugging. They can also rerun the program with the synthesized input after fixing the bug to ensure that the program patch is correct.

III. Implementation

The current prototype of RDE is built on top of KLEE [6], a symbolic execution engine for C programs. The first implementation challenge is the path alignment which needs to map the branch instructions in the binary to the branch representations in KLEE. Since KLEE works on the LLVM bitcode [21], RDE has to map branches on the inferred binary path (i.e., conditional jump instructions such as `JE`, `JL`) to corresponding branches in the LLVM bitcode (i.e., `br`). For each branch

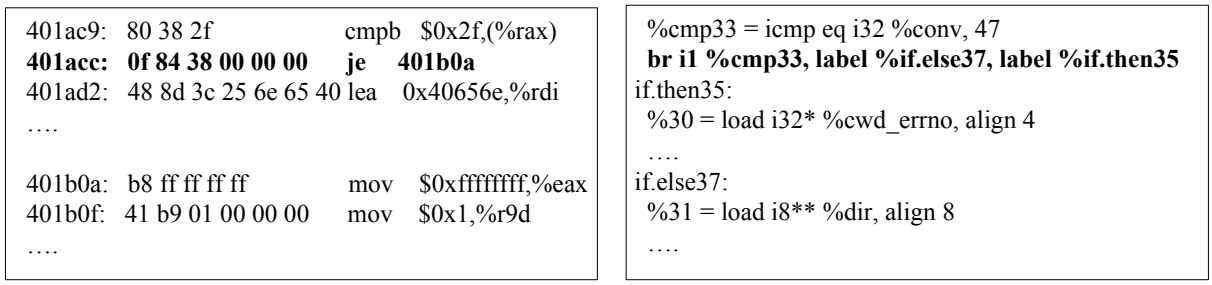


Fig. 4: The binary branch and its corresponding LLVM branch.

TABLE I: Real system failures used in our experiments. All the failures have one error log message produced during the failure run.

System name	LOC	Failure path length		Num. of console log msgs	Num. of system calls	Failure description	Root cause function
		Num. of functions	Num. of branches				
mkdir (5.92)	400	2	42	2	202	**Input failure**. The program rejects a valid input ending with "/.".	*mkdir_dir_parent*
rmdir (4.5.1)	200	2	23	3	198	**Input failure**. The program does not handle a valid input with trailing slashes at the end when using with the "-p" option.	*remove_parents*
ln (4.5.1)	600	2	43	2	186	**Option failure**. The program does not handle option "target-directory" correctly.	*do_link*
touch (7.6)	500	1	7	1	188	**Input failure**. The program rejects a valid input with leap second.	*main*
cp (6.10)	1900	13	116	2	199	**Input failure**. The program rejects a valid input if the destination of the copy command is already present in the system.	*copy_internal*

on the inferred path, RDE needs to translate it into the corresponding LLVM branch and its condition value in the LLVM bitcode of the program. We perform the branch mapping based on the semantics of the branch instruction and the basic code block layout information of both the binary and the LLVM bitcode. We assume that both the binary and the LLVM bitcode are compiled using the same compiler (e.g., Clang [1]), thus having similar basic code block layout.

All the branch instructions in the binary are conditional jump instructions which jump to a target branch or go to the next instruction depending on the EFLAGS register value. During the in-situ failure path inference, Insight interprets a branch instruction with a `true` condition value if the execution jumps to the target of the branch, otherwise, it assigns a `false` condition value for the branch. In addition to the branch value, Insight also records the branch target address (i.e., the location of basic code block corresponding to the `true` condition value) and the next instruction address (i.e., the location of the basic code block corresponding to the `false` condition value).

Using the branch value and the basic code block layout information, RDE determines the corresponding branch and its condition value in the LLVM bitcode. Similar to the conditional jump instruction in the binary, each conditional branch instruction in the LLVM bitcode is also associated with two basic blocks corresponding to the `true` and `false` condition values. RDE identifies the order of these basic blocks and leverages the basic code block order of a branch in the inferred binary path to determine the branch value for the corresponding LLVM branch. Figure 4 shows an example of a binary branch and the corresponding LLVM branch. In this example, the basic code block address corresponding to the `true` condition value of the binary branch is greater than the basic code block address corresponding to the `false` condition value. Thus, if the condition value of the binary branch is `true`, the condition value of the LLVM branch is also `true` because the basic block `if.then37` is after the basic block `if.then35`.

For the input replay, we use KLEE's driver that can replace symbolic inputs with synthesized values recorded in the trace. This is done through a custom library which intercepts the function called `klee_make_symbolic` and returns the value read from the log file. Developers simply need to link the buggy program with our custom library before attaching it to an interactive debugging environment such as GDB.

IV. System Evaluation

In this section, we present the experimental evaluation for the RDE system. We first describe our evaluation methodology. Next, we present our experimental results along with bug case studies.

A. Evaluation Methodology

We evaluate RDE using real software bugs in GNU coreutils. Table I shows the list of failures we use in our experiments. We report the number of function calls and branch points contained in the original failure execution path of each failure. Each failure contains one error message. We detect

failures by intercepting error messages: console log messages containing *error* and are written into *stderr*.

We evaluate the precision and efficiency of RDE using the following metrics: 1) *Call path difference* which is measured by the string edit distance between the original failure call path and the call path reproduced by RDE and denotes the deviation of the call path discovered by RDE from the original call path of the failed production run; 2) *Branch difference* which is also measured by the string edit distance and denotes the deviation at the branch level between the path reproduced by RDE and the original failure path. This metric is a more fine-grained comparison than the call path difference metric. We also show the branch difference between Insight and RDE to show the precision improvement of RDE; 3) *Number of explored paths* which defines the number of paths that RDE traverses before synthesizing the input; and 4) *Overhead* which evaluates the time taken by RDE to perform path alignment and input synthesis.

We also evaluate the effectiveness of RDE when it is given an infeasible path. As we mentioned before, Insight may output an infeasible path under certain circumstances [23]. In the experiment setup, we provide two types of inputs to the Insight failure path inference engine: original input and alternative input. *Original input* denotes the original failure-triggering input which is provided to Insight for the path inference. *Alternative input* denotes a different input but of the same type as the original input which does not trigger the failure.

We use alternative inputs which cause Insight to output infeasible execution paths, to examine whether RDE could find similar but feasible execution paths. The alternative inputs of the five coreutils programs are defined as follows. 1) $mkdir$ *failure*: a valid directory path that does not end with "/."; 2) $rmdir$ *failure*: a valid directory path without any trailing slash at the end; 3) ln *failure*: ln command without the "target-directory" option; 4) $touch$ *failure*: an input that does not specify a leap second; and 5) cp *failure*: a destination input which does not exist.

Our experiments were conducted on a computer cluster in our lab. Each cluster node is equipped with a quad-core Xeon 2.53GHz processor, 8GB memory, and is connected to Gigabyte network. Each host runs a 64-bit Ubuntu version 10.04 with LLVM version 3.1. We repeated each experiment five times and report the mean and standard deviation values.

B. Results and Analysis

In this section, we first summarize the failure path accuracy, and then present the number of explored paths and the results of RDE overhead evaluation. Table II summaries the execution reproduction results for five real software bugs. We observe that RDE can always reproduce a feasible execution path for all five tested bugs and output the exact failure execution path for four out of five cases under missing inputs. We also manually examine those failure executions with different

TABLE II: Summary of RDE failure reproduction results. *Original input* is the original failure-triggering input. *Alternative input* is a different input but of the same type as the original input which does not trigger the failure.

Failure name	Inferred path setting	Inferred path feasibility	RDE		
			Call path difference	Branch difference	Feasible path?
mkdir (5.92)	Original input	Feasible	0	0	Yes
	Alternative input	Infeasible	0	0	Yes
rmdir (4.5.1)	Original input	Feasible	0	0	Yes
	Alternative input	Feasible	0	0	Yes
ln (4.5.1)	Original input	Feasible	0	0	Yes
	Alternative input	Feasible	0	0	Yes
touch (7.6)	Original input	Feasible	0	0	Yes
	Alternative input	Feasible	0	0	Yes
cp (6.10)	Original input	Feasible	0	0	Yes
	Alternative input	Infeasible	2 (8.3%)	34 (17.1%)	Yes

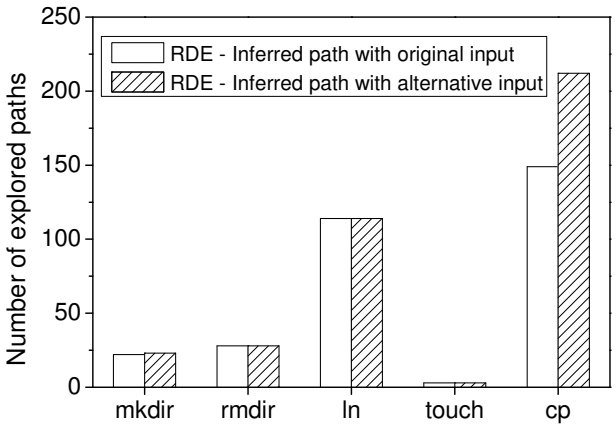


Fig. 5: Number of explored paths.

users and our initial results show that those execution paths reproduced in RDE are useful for debugging as they cover the root cause functions and branches. Furthermore, it shows that RDE is effective even if the input execution path is an infeasible path. RDE could discover a feasible execution path from the infeasible one and synthesize the failure-triggering input. Developers can then take the synthesized input to feed into an interactive development environment for further debugging.

Number of explored paths. We next show the number of execution paths explored by RDE before being able to synthesize the input. Note that Insight only records the branches inside the user source code. It does not record the branches belonging to external libraries (e.g. libc) to avoid mismatches between different library implementations (e.g., libc vs. ulibc, libc-2.10 vs. libc-2.11). Thus, RDE still needs to explore branches inside external libraries even if the path discovered by Insight is the same as the original failure path. Figure 5 shows that RDE can explore hundreds of execution paths within seconds. We will discuss the time overhead in detail later in this section. We also observe that the input provided to RDE also causes impact on the number of explored paths. An infeasible input path makes RDE explore a larger number of execution paths.

RDE overhead. We now evaluate the overhead of RDE. We measure the time taken for RDE to perform path alignment

TABLE III: RDE overhead. *Original input* is the original failure-triggering input. *Alternative input* is a different input but of the same type as the original input which does not trigger the failure.

Failure name	Inferred path setting	Path alignment	Input synthesis
mkdir (5.92)	Original input	0.9 ± 0.1 s	2.3 ± 0.4 s
	Alternative input	0.9 ± 0.2 s	2.4 ± 0.3 s
rmdir (4.5.1)	Original input	0.8 ± 0.1 s	1.8 ± 0.2 s
	Alternative input	0.8 ± 0.1 s	1.8 ± 0.3 s
ln (4.5.1)	Original input	1.0 ± 0.1 s	3.2 ± 0.4 s
	Alternative input	1.0 ± 0.1 s	3.3 ± 0.5 s
touch (7.6)	Original input	1.1 ± 0.1 s	2.1 ± 0.3 s
	Alternative input	1.2 ± 0.2 s	2.2 ± 0.3 s
cp (6.10)	Original input	1.1 ± 0.1 s	3.8 ± 0.4 s
	Alternative input	1.1 ± 0.1 s	3.9 ± 0.3 s

(see Section III for the details) and time taken for RDE to synthesize the input. Table III shows the time overhead of RDE for each failure. As the table shows, RDE takes about 1 second to perform path alignment and less then 4 seconds to synthesize inputs which are significantly faster than performing test generation using symbolic execution. Symbolic execution requires up to 6 hours to explore a program with 1.3 KLOC [9]. This improvement is achieved by RDE because it performs guided symbolic execution to narrow down the search scope.

C. Case Studies

$rmdir$ failure is an example of bug reproduction in RDE given feasible path. This failure is caused by incorrect handling of the special character '/' in the directory path name. Figure 6 shows a subgraph of the call graph for the $rmdir$ failure.

The input execution path to RDE presented by the sequence of line number of $rmdir$ source code file is: 217, 219, 228, 230, ..., 103, 104, 108, 109, 110, ..., 116, 118, ... There are two external interactions with the filesystem via system calls at line 116 and line 217. By solving the path constraints, RDE infers that the interaction result at line 116 is 1 and the interaction result at line 217 is 0.

The failure-triggering input which RDE synthesizes is "\x01\x01/". In fact, only the last character must be '/'. The first two characters only need to be different from the $null$ character ('\x00'). Thus, the STP solver picks the next character in the ASCII table ('\x01') for these two characters.

With this synthesized input, developers can replay the failure in an interactive debugger (e.g., GDB) and place break points at line 116 and line 217. They would see that the dir variable at line 217 is "\x01\x01/" and the $path$ variable at line 116 is "\x01\x01". At this point, they would know that only the last trailing slash is removed in $remove\_parents$, causing the program to try removing the same directory twice. Thus, inserting a duplication check would fix the problem.

$mkdir$ failure is an example of bug reproduction in RDE given an infeasible path. This failure is caused by incorrect handling of a valid directory path input ending with '/'. Figure 7 shows this infeasible path problem in Insight in the

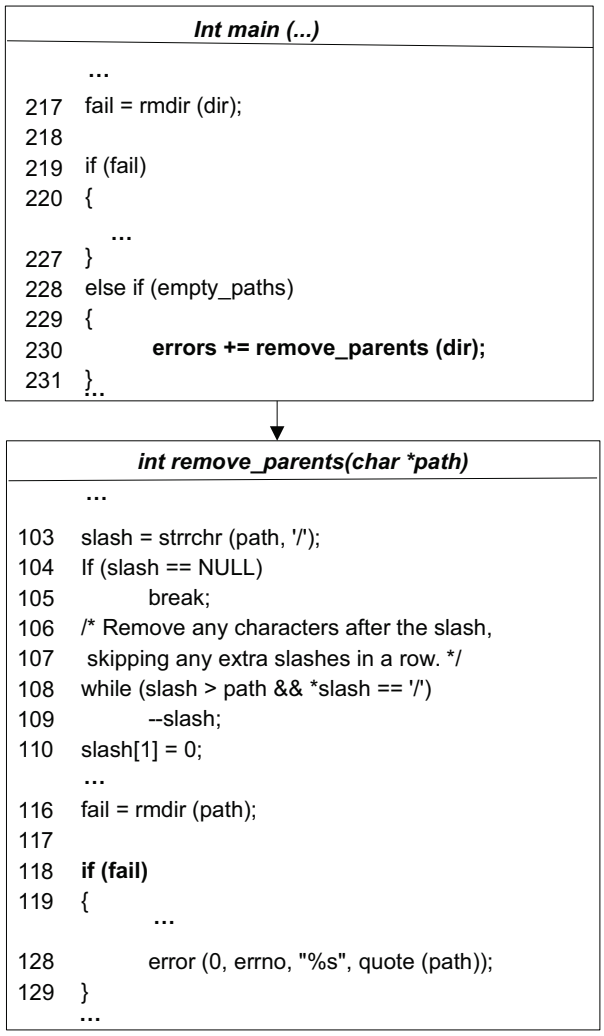


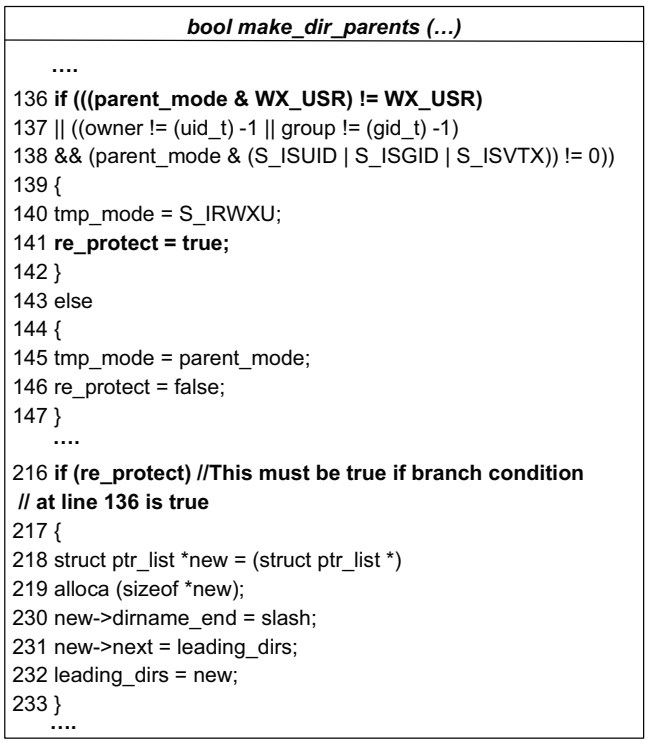
Fig. 6: A subset of the call graph for the $rmdir$ failure. The failure point is highlighted in bold. When an input with a trailing slash at the end is used, only that trailing slash is removed in the function $remove\_parents$, causing $rmdir$ to try removing the same directory twice (at line 217 and line 116).

$mkdir$ failure caused by lack of guidance during the in-situ path inference.

The input execution path to RDE presented by the sequence of line number of $mkdir$ source code file is: 136, 140, 141, ..., 216, 234, ... In other words, Insight outputs the $true$ condition value for the branch at line 136 and the $false$ condition value for the branch at line 216. However, this path is infeasible because if the condition value at line 136 is true, line 141 must be executed and the condition value at line 216 have to be true. RDE detects this and fixes the problem by inferring that the branch condition value at line 216 must be $true$ if the branch condition value at line 136 is $true$. Since this modified execution path is feasible, the following steps are similar to what happened to $rmdir$ program.

V. Related work

Failure input synthesis. ESD [32] presents an approach



```
bool make_dir_parents (...)
    ....
136 if (((parent_mode & WX_USR) != WX_USR)
137 || ((owner != (uid_t) -1 || group != (gid_t) -1)
138 && (parent_mode & (S_ISUID | S_ISGID | S_ISVTX)) != 0))
139 {
140 tmp_mode = S_IRWXU;
141 re_protect = true;
142 }
143 else
144 {
145 tmp_mode = parent_mode;
146 re_protect = false;
147 }
    ....
216 if (re_protect) //This must be true if branch condition
 // at line 136 is true
217 {
218 struct ptr_list *new = (struct ptr_list *)
219 alloca (sizeof *new);
230 new->dirname_end = slash;
231 new->next = leading_dirs;
232 leading_dirs = new;
233 }
    ....
```

Fig. 7: A code snippet for $mkdir$ failure. Insight outputs an infeasible path with the $true$ value for the branch condition at line 136 and the $false$ value for the branch condition at line 216. RDE corrects it by solving the constraints and infers that the branch condition value at line 216 must be $true$ if the branch condition value at line 136 is $true$.

to synthesizing program inputs from a specific failure point (e.g., code crashing statement). It first identifies candidate paths that can reach the failure point through static code analysis, and then symbolically executes the program to synthesize the failure-triggering input. In contrast, RDE does not require the knowledge of failure points, which can synthesize failure-triggering inputs for non-crashing failures. RDE leverages the failure path inferred by production-run failure path inference tool to synthesize the input.

Crameri et al. [11] present an approach of failure input synthesis by combining symbolic execution with partial logging of branches. Their approach labels branches whose condition values depend on program input as symbolic and labels the other branches as concrete. It selectively instruments symbolic program branches and uses both branch labels and instrumentation information to guide the program symbolic execution. Their approach also instruments system calls and guides symbolic execution based on system call logging results. Compared with this approach, RDE relies on the inferred path produced during production-run, which might provide unreliable information such as an infeasible path.

BBR (Better Bug Reporting [7]) uses symbolic execution along the same failure path to synthesize a set of failure inputs that are different from the original ones. It also proposes dynamic flow analysis to remove the constraints that are not related to the failure to further improve the privacy of the generated input. In contrast, RDE does not require any program inputs. RDE can synthesize the failure-triggering inputs from a failure path inferred during production-run which might be infeasible.

BugRedux [19] is a technique that collects four different types of runtime execution data (i.e. point of failure, call stack, function call sequence, and complete program traces.) and synthesizes the execution using symbolic execution. It also compares the efficiency of the failure execution based on different types of runtime recorded information. In contrast, RDE does not require any runtime recorded information for practicality. It only relies on the inferred failure path that is produced by in-situ failure path inference tool.

Record and replay. Replaying the failure executions deterministically from the recording log has been studied extensively, with the focus on using different mechanisms to reduce runtime overheads and log sizes, ranging from the application-level record and replay techniques [14], [15], [17], [27] to VM-level record and replay techniques [10], [20]. These approaches generally have high recording overheads, thus they are not widely used in the production system.

Regarding partial record and replay, a common approach is checkpointing. TRANSPLAY [28] reproduces production-run bugs on a completely different environment using partial recording. It introduces partial checkpointing to record a small but necessary amount of data just before the application encounters a failure. Cheung et al. [8] propose a partial symbolic replay tool which starts replaying from the last checkpoint rather than from the beginning of the system. It collects a log during program execution and discards all previous logs when it reaches a checkpoint to avoid maintaining full logs. In contrast, RDE does not require any runtime data logging or checkpointing.

Guided symbolic execution. There are various usages of symbolic execution. Ramos et al. [26] run symbolic execution for functions rather than the entire program to reduce both the number of and the length of the paths it has to explore. Pathfinder [25] limits the loop iterations and recursions of symbolic execution for Java bytecode. Fitnex [29] suggests a fitness function to measure the distance between a feasible path and a particular target and uses this fitness value to guide path exploration. Instead of directly running every execution path, LATEST [22] does symbolic execution for each function and stitches them together to form a complete program execution.

These papers use different strategies to guide the symbolic execution and alleviate the space explosion problem of the symbolic execution. While these strategies are different from RDE, they are complementary to our approach in mitigating the path explosion problem.

VI. Limitation and Future Work

Although our experiments show that RDE is efficient to reproduce production-run failures with the guidance of runtime inferred path, it is currently focused on non-crashing failures and its application is restricted to small programs. Different from crashing failures which often receive immediate attention, many non-crashing failures go unnoticed. RDE cannot efficiently support large-scale systems due to the following two reasons.

Library function replaying. Our study shows that RDE usually spends a lot of time on the symbolic executions of library functions (e.g., libc). This is because Insight does not record the failure execution path inside the library functions and RDE has no guidance once it enters library functions. This problem becomes more severe since large distributed systems usually contain more library function calls than small programs. We can address the problem by recording the execution path within library calls. However, KLEE currently only supports simplified libraries such as uclic which will cause path alignment issue when we record the path in libc. Moreover, production systems do not use simplified libraries. This limitation can be improved if the symbolic execution could support the libraries used in the production environment.

Multithreading and multiprocessing. Since KLEE could not handle multi-threaded or multi-process programs while most distributed systems are either multi-threaded or multi-process programs, RDE is insufficient in distributed systems at the moment due to the limitation of KLEE [6]. In the future, we plan to run symbolic execution in parallel. Cloud9 [5] is a symbolic execution engine that can explore paths of complex systems in parallel. We will replace KLEE with Cloud9 as the symbolic execution engine in RDE.

VII. Conclusion

In this paper, we have presented RDE, a system for reproducing production-site failures at the development site. RDE synthesizes missing production-site information (e.g., failure-triggering user inputs, missing environment data) from inferred execution path provided by an in-situ failure path inference tool. RDE employs a guided symbolic execution exploration scheme to achieve effective and fast failure reproduction. Our initial prototype implementation shows that RDE is both feasible and efficient. We tested RDE using a set of real software bugs in GNU coreutils. Our experiments show that RDE can successfully reproduce the failure executions and enable interactive debugging without requiring the failure-triggering inputs.

Acknowledgements

We thank the anonymous reviewers for their valuable comments. This work was sponsored in part by NSF CNS1513942 grant, NSF CNS1514256 grant, NSF CAREER Award CNS1149445, IBM Faculty Awards and Google Research Awards. Any opinions expressed in this paper are those of the authors and do not necessarily reflect the views of NSF or U.S. Government.

References

[1] clang: a C language family frontend for LLVM. http:/clang.llvm.org/.
[2] GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb/.
[3] G. Altekar and I. Stoica. ODR: output-deterministic replay for multicore debugging. In *SOSP*, 2009.
[4] P. Boonstoppel, C. Cadar, and D. Engler. Rwset: Attacking path explosion in constraint-based test generation. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.
[5] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel symbolic execution for automated real-world software testing. In *Eurosys*, 2011.
[6] C. Cadar, D. Dunbar, and D. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.
[7] M. Castro, M. Costa, and J.-P. Martin. Better bug reporting with better privacy. In *ASPLOS*, 2008.
[8] A. Cheung, A. Solar-Lezama, and S. Madden. Partial replay of long-running applications. In *FSE*, 2011.
[9] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In *ASPLOS*, 2011.
[10] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX ATC*, 2008.
[11] O. Crameri, R. Bianchini, and W. Zwaenepoel. Striking a new balance between program instrumentation and debugging time. In *Eurosys*, 2011.
[12] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008.
[13] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification*, 2007.
[14] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica. Friday: global comprehension for distributed replay. In *NSDI*, 2007.
[15] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: an application-level kernel for record and replay. In *OSDI*, 2008.
[16] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. In *Software Tools for Technology Transfer*, 2000.
[17] J. Huang, P. Liu, and C. Zhang. LEAP: lightweight deterministic multi-processor replay of concurrent java programs. In *FSE*, 2010.
[18] J. Huang, C. Zhang, and J. Dolby. Clap: recording local executions to reproduce concurrency failures. In *PLDI*, 2013.
[19] W. Jin and A. Orso. Bugredux: reproducing field failures for in-house debugging. In *ICSE*, 2012.
[20] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX ATC*, 2005.
[21] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization*, 2004.
[22] R. Majumdar and K. Sen. Latest: Lazy dynamic test input generation. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2007-36*, 2007.
[23] H. Nguyen, D. Dean, K. Kc, and X. Gu. Insight: in-situ online service failure path inference in production computing infrastructures. In *USENIX ATC*, 2014.
[24] 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 *SOSP*, 2009.
[25] C. S. Păsăreanu and N. Rungta. Symbolic pathfinder: symbolic execution of java bytecode. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 179–180. ACM, 2010.
[26] D. A. Ramos and D. Engler. Under-constrained symbolic execution: correctness checking for real code. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 49–64, 2015.
[27] Y. Saito. Jockey: a user-space library for record-replay debugging. In *Automated analysis-driven debugging*, 2005.

[28] D. Subhraveti and J. Nieh. Record and transplay: partial checkpointing for replay debugging across heterogeneous systems. In *SIGMETRICS*, 2011.

[29] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 359–368. IEEE, 2009.

[30] C. Zamfir, G. Altekar, G. Candea, and I. Stoica. Debug determinism: the sweet spot for replay-based debugging. In *HotOS*, 2011.

[31] C. Zamfir, G. Altekar, and I. Stoica. Automating the debugging of datacenter applications with adda. In *DSN*, 2013.

[32] C. Zamfir and G. Candea. Execution synthesis: a technique for automated software debugging. In *Eurosys*, 2010.