DESCRY: Reproducing System-Level Concurrency Failures

Tingting Yu
University of Kentucky
Lexington, KY 40508
tyu@cs.uky.edu

Tarannum S. Zaman
University of Kentucky
Lexington, KY 40508
tarannum.zaman@uky.edu

Chao Wang
University of Southern California
Los Angeles, CA 90089
wang626@usc.edu

ABSTRACT

Concurrent systems may fail in the field due to various elusive faults such as race conditions. Reproducing such failures is hard because (1) concurrency failures at the system level often involve multiple processes or event handlers (e.g., software signals), which cannot be handled by existing tools for reproducing intra-process (thread-level) failures; (2) detailed field data, such as user input, file content and interleaving schedule, may not be available to developers; and (3) the debugging environment may differ from the deployed environment, which further complicates failure reproduction. To address these problems, we present DESCRY, the first fully automated tool for reproducing system-level concurrency failures based only on default log messages collected from the field. DESCRY uses a combination of *static* and *dynamic analysis* techniques, together with *symbolic execution*, to synthesize both the failure-inducing data input and the interleaving schedule, and leverages them to deterministically replay the failed execution using existing virtual platforms. We have evaluated DESCRY on 22 real-world multi-process Linux applications with a total of 236,875 lines of code to demonstrate both its effectiveness and its efficiency in reproducing failures that no other tool can reproduce.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

Multi-Process Applications, Debugging, Concurrency Failures, Failure Reproduction

ACM Reference format:
Tingting Yu, Tarannum S. Zaman, and Chao Wang. 2017. DESCRY: Reproducing System-Level Concurrency Failures . In *Proceedings of ESEC/FSE'17, Paderborn, Germany, September 04-08, 2017,* 11 pages.
https://doi.org/10.1145/3106237.3106266

1 INTRODUCTION

The ever increasing parallelism in computer systems has made software more prone to concurrency failures, causing problems not only during the development but also after deployment. When concurrency failures occur in a deployed system, developers often have to diagnose them in a different (debugging) environment to identify the root causes. Toward this end, an important step is to reproduce the failure in a timely manner. However, this is challenging due to the limited data generated by production runs. Typically, field data are transferred from customers to developers. However, if they belong to different organizations, customers may not be willing to share their inputs and file contents involved in the failed execution [11]. There are *pre-deployment* debugging techniques, which leverage fine-grained logging for deterministic record-and-replay [2, 6, 7, 14, 17, 21, 26, 27, 30, 42]: although effective in testing, they are ill-suited for deployment because of the often unbearable performance overhead. Thus, for the purpose of reproducing failures in production runs, we have to assume that the only available field data are default log messages generated by the unmodified application.

Under this assumption, we propose DESCRY (**De**bugging **S**ystem-level **C**oncu**r**renc**y** Failures), the first fully automated tool for reproducing failures using only default logs collected from the field. This is a challenging task because, in practice, log messages are often sparsely printed such that there may be thousands of program paths leading to the same message. DESCRY focuses on *inter-process* bugs where multiple operating-system components (e.g., processes, software signals, and interrupts) incorrectly shared resources. They differ from *intra-process* (thread-level) bugs, which are the focus of many prior work on reproducing concurrency failures. The difference is that an intra-process (thread-level) concurrency bug often corrupts only volatile memory within a process, whereas an inter-process (system-level) concurrency bug is more dangerous, since it corrupts the persistent storage and other system-wide resources, thus potentially crashing the entire system. As Laadan et al. [21] noted, more than 73% of the race conditions reported in popular Linux distributions were process-level races, which could not be reproduced by existing (thread-level) debugging tools [21, 38].

Ideally, reproducing a system-level concurrency failure requires the availability of all input data, the entire interleaved execution, and the same execution environment as in the deployed system. However, as we have mentioned earlier, they do not exist in practice. Therefore, DESCRY only assumes the existence of the source code of processes under debugging (PuDs) and default logs generated by the failed execution. Internally, DESCRY leverages a combination of static and dynamic program analysis techniques as well as symbolic execution to compute the failure-inducing data input and interleaving schedule. As such, it shifts the operational cost from the customer side to the developer side, thus avoiding the overhead of fine-grained logging and heavy-weight code instrumentation in production runs – this is the differentiating feature of DESCRY compared to existing techniques.

Figure 1 provides an overview of DESCRY. First, it leverages the logs to identify processes that are relevant to the observed failure (i.e., PuDs). Next, it uses static program analysis to connect the log messages with statements in the source code of the PuDs (i.e., *logging points*) that print these messages. In the third step, DESCRY uses symbolic execution to generate failure-inducing data inputs of the PuDs, which steer the execution through these logging points.

ESEC/FSE'17, September 04-08, 2017, Paderborn, Germany

ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3106266

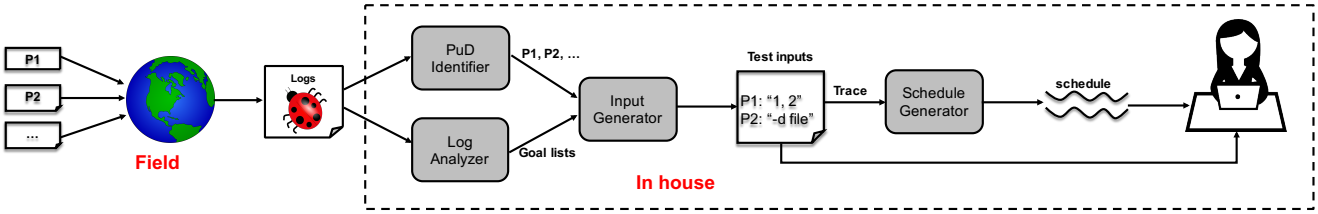

Figure 1: The overview of our DESCRY framework.

To improve scalability, it limits the program state space using summaries of logging points and skipping irrelevant code. DESCRY also constructs a *prediction model* based on the partial order of observed inter-process operations to compute a failure-inducing interleaving schedule. Finally, with the new data inputs and interleaving schedule, DESCRY controls the PuDs to deterministically replay the failure.

DESCRY has been implemented as a software tool using the LLVM compiler front-end [1], the KLEE symbolic virtual machine [5], and the Simics Virtual Platform [8, 38] for deterministic replay. The tool can directly handle multi-process applications written in C/C++. To evaluate DESCRY, we conducted experiments on 22 popular Linux applications with a total of 236,875 lines of code and known concurrency failures. The logs used for our evaluation are real and generated by running the applications with their default (production) settings. Our experimental results show that DESCRY can successfully reproduce 22 of the 24 known failures, which is significant because no other tool can reproduce this many failures. Furthermore, the two remaining failures could have been successfully reproduced by DESCRY if limitations in the implementation of the underlying KLEE [5] were to be removed. Finally, the time taken by DESCRY to compute the failure-inducing data inputs and interleaving schedule is typically a few minutes, indicating that the proposed method is efficient for practical use.

In summary, this paper makes the following contributions:

- We propose DESCRY, the first fully automated method for deterministic reproducing failures of multi-process applications using only default logs generated by these applications.
- We implement DESCRY and conduct an empirical study to demonstrate its effectiveness and efficiency on real-world applications.

The remainder of this paper is organized as follows. First, we use examples to illustrate the main technical challenges and our corresponding solutions in Section 2. Then, we present our detailed algorithms in Sections 3, 4, 5, and 6. Next, we present our experimental evaluation in Section 7. Finally, we review related work in Section 8, and give our conclusions in Section 9.

2 MOTIVATION AND BACKGROUND

In this section, we use examples to illustrate the challenges in reproducing concurrency failures and then formally define our problem.

2.1 Motivating Example

Figure 2 shows a real race condition between two Linux applications `updatedb` and `tail`. `updatedb` is a script that spawns multiple application processes, e.g., by executing "`mv A.txt.backup A.txt`" to recover a file. The race (Bugzilla-438076) occurs when `updatedb` modifies the file name in the database and `tail` displays the file content to standard output. For example, while the command "`tail -F A.txt`" is monitoring the file `A.txt` to output its last 100 lines, a user may execute `updatedb` to spawn "`mv A.txt.backup A.txt`". As a result, the `tail` program finds that `A.txt` is missing.

The reason is that `updatedb` fails to ensure that "`mv A.txt.backup A.txt`" is executed atomically. By atomically, we mean that there should be no point in time when "`A.txt`" does not exist. However, in the actual implementation, `mv` first unlinks the target file (Line 16) if it has multiple hard links (e.g., created by `ln`). If `tail` runs concurrently with `mv`, and accesses the target file (Line 42) after `unlink` (Line 16) but before `rename` (Line 25), it will find the target file missing, thereby triggering the error "`tail: No such file or directory.`".

This bug can be fixed by removing the condition check that triggers the unlink (Line 14 of Figure 2). However, in practice, when reporting the bug to developers, the customer only submitted the default log messages produced by the application, as shown in the second column of Figure 3. Such log messages are fairly common in practice, but not really helpful to developers who try to reproduce the failure in the debugging environment. First, this bug involves a total of 11 processes (e.g., `updatedb`, `mv`, `frcode`) spawned by the `updatedb`, together with the `tail`, despite the fact that only `mv` and `tail` need to be analyzed for diagnosing the failure. Second, there are a total of 141 messages displayed in the default log file, of which 99 messages are from `tail` alone – manually sifting through all these log messages would be time-consuming. More importantly, no information of the failure-inducing data inputs or interleaving schedule is provided.

The first challenge in reproducing such failure is to identify the processes that are responsible for the failure. For instance, among the 11 processes involved in the erroneous execution, only the `mv` and `tail` processes are actually relevant. Therefore, we need a method to quickly weed out the irrelevant processes. Moreover, the failing process might not be the process that contains the bug. For example, although `tail` triggers the failure that contains the bug, the buggy code is actually in `mv`. To decide whether a process is buggy, we need to match the output messages back to the source code that print these messages. However, since many log messages are irrelevant, processing all log messages would be inefficient.

The second challenge is that, most of the time, concurrency failures are triggered by specific combinations of data inputs. Since the total number of possible combinations can be astronomically large, it is extremely difficult for random/stress testing to trigger these failures using randomly generating inputs. For the `mv` and `tail` example in Figure 3, even if the file name `A.txt` is known from the log messages, it may not be part of the failure-inducing inputs unless the file also has hard links to other files.

```
1. main(arc, argv){
2.  if (dest_is_dir) //C1
3.    free (new_dest);
4.  else
5.    ok = movefile (source, dest, x);
6.  return ok;
7. }

8. movefile (char *src_name, char *dst_name, ...){
9.  if (x->backup_type != no_backups && ...){ //C2
10.    if (rename (dst_name, dst_backup) != 0){ //C3
11.       error (0, errno, _("cannot backup %s"),
                 dst_name); //lp1
12.       return false;
       }
13.  }
14. else if(! S_ISDIR (dst_sb.st_mode) && (... // C4
    || (x->preserve_links && 1 < dst_sb.st_nlink))){
15.   printf (_("hard links: %s\n"), dst_name); //lp2:msg1
16.   if (unlink (dst_name) != 0){ //C5
17.     error (0, errno, _("cannot remove %s"),
                dst_name); //lp3
18.     return false;
19.   }
20. }

21. if (!S_ISDIR (src_mode)){ // C6
22.   emit (src_name, dst_name, ...);
23. }

24.if (x->move_mode)  //C7
25.   rename (src_name, dst_name)

26. emit (char *src, char *dst, ...){
27.   printf ("%s -> %s", quote_n (0, src),
              quote_n (1, dst)); //lp4:msg2
28.  if (backup_dst_name)  //C8
29.    printf (_(" (backup: %s)"), backup_dst_name);//lp5
30. }
```

```
31. int main (){
32.  if(forever) /* -F option */  //C9
33.     tail_forever_inotify(wd, f,...)
34. }

35. void tail_forever_inotify(int wd, struct File *f, ...){
36. while(1) {
37.   if (follow_mode == Follow_name)  // C10
38.      recheck (&(f[i]), false);
39.   }
40.}

41.void recheck (f, ...){
42.  int fd = open (f->name, O_RDONLY);
43.  if (fd == -1) //C11
44.     error (0, errno, "%s") //lp6: msg99
45.  else
46.     printf ("output:%s", f->msg);//lp7:msg1, msg2,...
47.}
```

Figure 2: Code snippet showing the race condition caused by interleaved execution of MV (top) and TAIL (bottom).

The third challenge in reproducing system-level concurrency failures is the need to analyze the interleaving schedule across multiple processes. In Figure 2, for example, an inter-process operation is the system call open made by `tail`, which must occur after the system call `unlink` but before the system call `rename` made by `mv`. Enforcing such execution order requires modeling of the system-level happens-before relations and controlling the kernel scheduler, which cannot be accomplished by existing methods focused only on intra-process (thread-level) race conditions.

2.2 Problem Statement

We define the failure production problem as follows. Given the source code of a set of processes under debugging (PuDs) and default logs generated by these PuDs in a failed execution, compute the data inputs for these PuDs and their interleaving schedule such that the failure can be deterministically reproduced.

	Log	Goal List	Path Condition	Data Input
updatedb (frcode, ... mv)	~~frcode: ... [msg1]~~ ... mv: hard links: A.txt **[msg1]** mv: 'A.txt.backup' -> 'A.txt' **[msg2]**	**list1:** <lp2, lp4>	$!C1 \wedge !C2 \wedge !C3 \wedge C4 \wedge !C5 \wedge C6 \wedge C7 \wedge !C8$	./mv A.txt.backup A.txt
tail	tail: output: Hello 1 **[msg1]** tail: output: Hello 2 **[msg2]** tail: ... tail: No such file or directory **[msg99]**	**list1:** <lp7, lp6>	**1st iteration:** C9∧ C10∧ !C11 ∧ C11 **2nd iteration:** C9∧ C10∧ !C11	./tail –F A.txt

Figure 3: Logs, path conditions, and inputs for MV and TAIL.

We assume that a concurrent system consists of a set of processes $\{P_1 \ldots P_m\}$ and a set of software signals $\{S_1 \ldots S_n\}$. Each process may create multiple threads, but for ease of presentation, we focus only on the process-level concurrency in this work while assuming each process has one thread. A *failing process* P_F is a process that behaves erroneously, manifested by the failure messages it prints. A *failure point* is a program statement that prints a failure message.

Logs are sequences of normal or error messages that the application prints to files and consoles. In this context, each entry in the log is a *log message*. To be realistic, we assume log messages are uninterpreted plaintext strings. Each program statement capable of printing a log message is called a *logging point* (LP). There are two types of logging points with respect to the failure: relevant logging points (*RLP*s) and irrelevant logging points (*ILP*s). *RLP*s are the ones that may print log messages relevant to the failure; *ILP*s are the ones that can never print log messages relevant to the failure. Therefore, only *RLP*s need be used to synthesize the failure-inducing data input. For example, in the `mv` program of Figure 2, *lp2* and *lp4* are *RLP*s, since they may print log messages for the failed execution, whereas *lp1*, *lp3*, and *lp5* are *ILP*s, since they can never print log messages relevant to the failed execution.

We assume that logs are generated by applications during production runs, and when a failure occurs, they are transferred from the customer to the developer. This is a realistic assumption, because logging for critical events is a common and important software engineering practice [39–41]. There are quantitative evidence [40] that logging is pervasive during software development and is actively maintained by developers.

A system-level concurrency fault occurs when multiple processes, signals, or interrupts access a system-wide resource (e.g., file, device) without proper synchronization [21]. Such resources are often accessed through system calls. Thus, handling system-level concurrency fault requires the modeling of read/write effects and synchronization operations involving system calls. For example, the `lstat` system call on file f reads the metadata of f. The `clone` system call creates a new process inode under the */proc* directory (write). Synchronization operations control process interactions through kernel process scheduler. Common process-level synchronization primitives include `fork`, `wait`, `exit`, `pipe`, and `signal`.

The second column of Figure 4 shows an example interleaving schedule for running the `mv` and `tail` processes. Each event in the schedule is either a shared resource access ("R" denotes *read* and "W" denotes *write*) or a synchronization operation. Details of shared resource and event modeling can be found in prior work [21, 38]. However, note that system call modeling in DESCRY is different from that in symbolic execution tools (e.g., KLEE [5]). For example, KLEE models the system calls to generate the required program constraints for achieving high coverage, whereas in DESCRY, the systems calls are modeled as concurrency events.

Arbitrary schedule	Events	Permutated schedule
1. mv: stat ("A.txt")	R(A.txt)	1. mv: stat ("A.txt")
2. mv: lstat ("A.txt.backup")	R(A.txt.backup)	2. mv: lstat ("A.txt.backup")
3. mv: lstat ("A.txt")	R(A.txt)	3. mv: lstat ("A.txt")
4. mv: stat ("A.txt")	R(A.txt)	4. mv: stat ("A.txt")
5. mv: access ("A.txt")	R(A.txt)	5. mv: access ("A.txt")
6. mv: unlink ("A.txt")	W(A.txt)	6. mv: unlink ("A.txt")
7. mv: rename ("A.txt.backup", "A.txt")	W(A.txt.backup), W(A.txt)	8. tail: open ("A.txt", R_ONLY)
8. tail: open ("A.txt", R_ONLY)	R(A.txt)	9. mv: rename ("A.txt.backup", "A.txt")

Figure 4: Example execution trace for MV and TAIL

DESCRY Algorithm

1: **Inputs:** P_{all}, $logs$
2: **Outputs:** $\langle T, S\rangle$
3: **begin**
4: $PuDs \leftarrow$ `IdentifyPuD` (P_{all})
5: **for** each $P_i \in PuDs$
6: $LP_i \leftarrow$ `GetLPoints` $(logs, P_i)$
7: $G_i \leftarrow$ `ComputeGoals` (LP_i, P_i)
8: **endfor**
9: **while** $time < TIME_{max}$
10: **for** each $P_i \in PuDs$
11: **if** `ExecuteConcrete`$(T,\ P_i)$ does not lead to $gl \in G_i$
12: $T[i] \leftarrow$ `GuidedSymbolicExecution` (G_i, P_i)
13: **endif**
14: $E \leftarrow$ `GetExecutionTrace` (T)
15: $S \leftarrow$ `ScheduleGeneration` (E)
16: **if** `Replay` (T, S) is successful
17: **return** $\langle T, S\rangle$
18: **endif**
19: **endfor**
20: **endwhile**
21: **end**

Figure 5: The Overall Algorithm of DESCRY.

3 THE DESCRY APPROACH

The overall algorithm of DESCRY is shown in Figure 5. The input of this algorithm includes the set of running processes P_{all} when the failure occurs, as well as the logs. The output is a tuple $\langle T, S\rangle$, where T is the failure-inducing data input and S is the failure-inducing interleaving schedule. In the remainder of this section, we explain each step of the algorithm using the example of Figure 2. Details of the algorithm are described in Sections 4–6.

Given P_{all} and the logs, we first identify the failing process $P_F \in P_{all}$ and the subset $L \subseteq P_{all}$ of processes potentially interacting with P_F (Line 4). Processes in $L \cup \{P_F\}$ are called the *processes under debugging* (PuDs). For example, in Figure 2, among all 11 processes spawned by `updatedb` and `tail`, only `mv` and `tail` are the PuDs.

Next, we invoke `GetLPoints` to obtain the logging points (Line 6). To handle large data and improve the scalability of subsequent procedures for input and schedule generations, we first remove the *repeated* log messages and then map the remaining messages to logging points in the program. Next, we use the subroutine `ComputeGoals` (Line 7) to connect logging points of each PuD to form a set of logging sequences, denoted by G_i, which subsequently will be explored during symbolic execution.

In the example of Figure 3, DESCRY would map all messages to their logging points: msg_1 of `mv` is mapped to *lp2*, msg_2 of `mv` is mapped to *lp4*, msg_1 to msg_{98} of `tail` are mapped to *lp7*, and msg_{99} of `tail` is mapped to *lp6*, which is the failure point. All other logging points in the program are considered to be irrelevant. In the end, `mv` and `tail` each has a goal list: $\langle lp2, lp4\rangle$ for `mv` and $\langle lp7, lp6\rangle$ for `tail`, as shown in Column 2 of Figure 3.

Next, DESCRY computes the failure-inducing data input ($T[i]$) for each individual PuD to exercise at least one of its goal lists in G_i (Line 12). This is accomplished by a customized symbolic execution procedure on each PuD that uses the logging sequences in G_i as guidance. To reduce the computational overhead, we propose three optimization techniques: the seeding of concrete inputs, the pruning of irrelevant states, and the prioritization of program paths. These optimizations are made possible by leveraging results of our static log analysis (see Section 4).

Column 3 of Figure 3 shows the path conditions computed by our symbolic execution procedure. They are used to compute the data input of `mv` (`A.txt` and `A.txt.backup`, where `A.txt` has a hard symbolic link). However, symbolic execution fails to compute the data input of `tail` because the first path connecting *lp7* to *lp6* is infeasible. Therefore, DESCRY relaxes the goals in `tail` by replacing the failure point *lp6* with a *failure predicate* at Line 43, to allow the exploration of more paths. The rationale is that the diverged goal may still be reached through a different path, if the control flow of the PuD is changed under a different event interleaving schedule. When the goal list is changed to $\langle lp6, 43\rangle$, for example, our symbolic execution procedure is able to compute the desired input `-F A.txt`.

Finally, we compute the failure-inducing interleaving schedule S (Lines 14-15). Toward this end, DESCRY first executes all PuDs concretely under their new inputs while following an arbitrary interleaving schedule. If the resulting trace, denoted by E, triggers the failure, we are done. Otherwise, we systematically explore alternative interleavings of the inter-process operations in E in order to trigger the failure. If no such interleaving exists, DESCRY backtracks, and uses symbolic execution to compute a set of new data inputs, until the failure-inducing schedule is successfully reproduced (Lines 10-19).

For example, assume the trace in Figure 4 (the first column) was generated by `mv` and `tail` in Figure 3 following an arbitrary schedule. Since it does not trigger the failure, DESCRY systematically permutes events of the trace to generate a failure-inducing schedule. After swapping the two events $\langle mv$: `rename`, $tail$: `open`$\rangle$, we have found the failure-inducing event interleaving schedule.

Note that the data inputs generated from different PuDs may have different names but need to point to the same shared resources. For example, `tail` may generate an input file called `B.txt`, whose name is different from `A.txt` generated from `mv`. In this case, the two file names must be unified to reproduce the failure. DESCRY records a list of system calls that access the data input for each PuD. If both lists in the pair of PuDs are non-empty, the data input is a shared resource between the PuDs and thus the file names are unified.

4 STATIC ANALYSIS OF LOG MESSAGES

The first step is to identify relevant processes (PuDs) from P_{all}, since there can be tens or hundreds of active processes when the failure occurs in P_F, not all of which are PuDs. We consider a process P as a PuD only in one of the following scenarios:

(1) P accepts input with the same type as the failed process P_F (e.g., `mv` and `tail` both accept files as input);
(2) P_F and P are different instances of the same program (e.g., multiple `bash` processes running concurrently);
(3) P_F and P are different processes spawned by an application (e.g., a `Mutt` mail client process tries to open an `MS Word` attachment process);
(4) P is a software signal within the process P_F.

The last three scenarios are automatically identified by DESCRY. We assume active processes are captured by system built-in tools such as the Linux Auditd Daemon [3, 16]. The first scenario may require user intervention, e.g., to specify the type of input of a process, if it has not yet been specified before. This is the only manual step potentially needed by DESCRY. In practice, there is no technical difficulty in doing this because developers who use DESCRY should know what type of input (files, strings, or numbers) that a process accepts.

For the example in Figure 2, DESCRY would examine the system log directory `/var/log` that contains process names to identify active processes, shown in Column 1 of Figure 3, where `tail` is the failed process. Since the type of input accepted by `mv` is the same as that of `tail` (i.e., file), they are selected as the PuDs. Note that under the default system environment, `/var/log` record system logs that do not involve program inputs from the user space. As such, DESCRY does not assume these inputs are available.

4.1 Identifying the Logging Points

The next step is to identify the logging points (both RLPs and ILPs) that print normal and error messages. This is extremely challenging for real applications because they often use customized logging facilities as opposed to simple `printf` statements. For example, in Figure 2, the `error` call at Line 44 does not contain the string "No such file or directory" observed in the error message, but calls an error handling function named `strerrno` to construct and then print the string. Because of this reason, mapping log messages to logging points alone is a challenging problem, and has been studied by the systems community extensively (e.g., Sherlog [39]).

However, unlike Sherlog [39], which relies on the user to specify log patterns before it can identify the logging points, DESCRY does it automatically by performing an inter-procedural static analysis of each PuD and its libraries (e.g., `strerror.c`) to match log messages against the source code. It first identifies the set of program statements (inside PuDs and libraries) involving the printing APIs (denoted by F_p) such as `printf` and `sprintf`, and records the set Str of format strings passed as parameters to functions in F_p. A format string is determined by extracting the string within the quotes of the API, while excluding the the format specifiers (e.g., %s) and escape characters. For instance, "`hard links`" at Line 15 of Figure 2 is a format string passed to `printf`. For each log message in the logs, if its substring is identical to a string $str \in Str$, the corresponding function in F_p is identified as a logging point. If the failure point $f_p \in F_p$ exists in a library file, DESCRY traces f_p back to the PuD source code through the data and control flow edges of the inter-procedural CFG; the source code location is then identified as a logging point (e.g., Line 44 of Figure 2).

It is possible that the logging points are over-approximate, although it does not occur often. If a log message comes from a dynamically assembled string as opposed to a string constant, it will not be identified statically. The subsequent symbolic execution will have to be an un-guided search.

4.2 Constructing the Goal Lists

We construct the goal lists by connecting relevant logging points of the PuD in a *log hierarchy graph (LHG)*, which describes the partial order of the logging points. A LHG is a directed acyclic graph where nodes correspond to logging points and the hierarchy levels correspond to the order in which they appear in the log: the first log message is at the top level. For each message ID m, the set of logging points associated with m is $\{l_{m,i}\}$, where $l_{m,i}$ is the logging point for m at the i-th program statement.

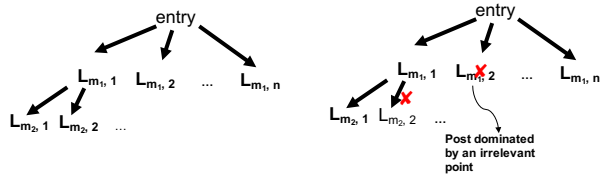

Figure 6: Examples for illustrating the log hierarchy graph (LHG).

Each logging point $l_{m,i}$ indicates an unique program statement. However, a logging point may associate with multiple log messages having different IDs but the same format strings due to loop iterations. In the example of Figure 2, there are 98 messages mapped to $lp7$. DESCRY considers such messages to be the same and thus unifies their message IDs. Specifically, these 98 messages are renamed to msg_a because they have the same format strings. In other words, we mitigate the cost of goal list construction by ignoring loops. Although in theory, it may lead to unsuccessful failure reproduction, e.g., when a failure can only be triggered after going through a loop a fixed number of times, such cases are rare in practice, as shown in our experimental evaluation. Furthermore, we mitigate the problem during our subsequent symbolic execution step by iteratively increasing the number of loop iterations to eliminate unsuccessful failure reproduction should it appear.

A goal list, denoted by $P_p = [l_{m_1,i_{m_1}}, l_{m_2,i_{m_2}}, \ldots, l_{m_k,i_{m_k}}]$, connects the logging points in the LHG. Figure 6 (left) shows an example LHG, where each vertical path corresponds to a goal list. Since enumerating all goal lists takes $O(S^k)$, where S is the number of logging points associated with each log message and k is the length, it can become too expensive in practice. Fortunately, not all goal lists correspond to a feasible execution. In DESCRY, we use a light-weight static program analysis to prune away the infeasible goal lists. Toward this end, we decompose the problem of searching for a complete LHG path into subproblems of searching for segments of the path.

More specifically, DESCRY iterates through all LHG nodes at two neighboring LHG levels i and j, and for each edge $(l_{m_i,i} \rightarrow l_{m_j,j})$, checks if (1) $l_{m_j,j}$ is reachable from $l_{m_i,i}$ and (2) their pre- and post-dominators do not contain irrelevant logging points. If the first condition is not met, we remove $(l_{m_i,i} \rightarrow l_{m_j,j})$ because paths traversing this edge are infeasible. If the second condition is not met, we remove the node $l_{m_j,j}$ and all its associated incoming and outgoing edges, and we do not involve $l_{m_j,j}$ in the next iteration. This is because paths traversing $l_{m_j,j}$ contain irrelevant points and thus must not have been exercised for triggering the failure. Only when both conditions are met, $l_{m_j,j}$ is considered as a new goal after $l_{m_i,i}$.

Applying our new method to the example of Figure 6 (right) will remove the edge $(l_{m_1,1} \rightarrow l_{m_2,2})$ because $l_{m_2,2}$ is not reachable from $l_{m_1,1}$ (the first condition). It will also remove $l_{m_1,2}$ and the edge $(entry \rightarrow l_{m_1,2})$ because $l_{m_1,2}$ dominates an irrelevant log message (the second condition). Thus, the final goal lists will be $[l_{m_1,1}, l_{m_2,1}]$, and $[l_{m_1,n}]$.

5 GUIDED SYMBOLIC EXECUTION

We propose a new symbolic execution procedure to compute failure-inducing data inputs of all PuDs in a multi-process application. This step differs from prior work on testing concurrent software using symbolic execution [12, 13, 24] in that our method is designed for multi-process applications whereas prior works all focus on a single process. Internally, we leverage KLEE [5] to conduct a goal-directed exploration of the PuDs, to traverse program statements specified in the goal list.

As shown in Line 12 of Figure 5, our algorithm takes each PuD P_i and the set of goal lists G_i as input and returns $T[i]$ as output. Here, $T[i]$ is the data input that forces P_i to go through a goal list in $gl \in G_i$. Let $goal \in gl$ be the current goal, and $stateset$ be the set of program states of P_i. At each step of the symbolic execution of P_i, we select a state $s_i \in stateset$ that is more likely to reach $goal$. If no state in $stateset$ can reach $goal$, we check if $goal$ is a program statement in a loop. If $goal$ is in a loop, we increase the number of loop iterations by a fixed number (N=10 in our experiments) and try again until reaching the loop bound L_{max}. This will increase our chance of reaching the goal.

If $goal$ cannot be reached in this way, we backtrack to the previous goal in gl, and search for a path to the new goal. If backtracking is repeated many times, eventually, it may move back to the first goal, indicating that the current goal list cannot be exercised. In this case, we choose another $gl \in G_i$ and try again.

Upon reaching the final goal in gl, we traverse the corresponding program path in P_i to compute the path condition (PC), which is a symbolic expression of the input condition under which this program path will be executed. We compute the data input $T[i]$ by solving the path condition using an SMT solver.

The main problem in this log-guided symbolic execution is to make the computation efficient by exploring the more "promising" program paths. Toward this end, we propose several techniques. First, we statically analyze the source code of each PuD to prune away basic blocks that do not lead to the goals – they correspond to not only the irrelevant logging points (Section 4.2) but also the related non-logging program statements. Second, we skip computationally expensive constraint solver calls unless the program path traverses some previously unexplored system calls. In addition to these optimizations, we prioritize the path exploration based on the estimated distance between current program state and the next goal to increase the likelihood of reaching it sooner. Finally, for efficiency reasons, if the program path does not traverse any previously unexplored system call, DESCRY heuristically avoids generating the data input because it will be less useful for failure reproduction.

We next explain how the next state is selected at each step, how concrete configuration options are leveraged to avoid generating a large number of invalid inputs, and how concrete data inputs are computed for other PuDs to further avoid expensive constraint solving.

5.1 Selecting the Next State

At each step of the symbolic execution of process P_i, we need to prioritize the exploration by selecting the most promising next state. Internally, DESCRY estimates the distance between each state s_i and $goal$ before selecting the most promising one. The distance is defined as the number of instructions to be executed from s_i to $goal$ and is computed by statically traversing the control flow graph of the PuD. If multiple states have the same distance to $goal$, DESCRY would favor the one through which some previously unexplored system calls can be invoked. As such , the search strategy significantly differs from prior state prioritization techniques [5, 20] which do not attempt to maximize the exploration of logging points or previously unexplored system calls.

Furthermore, to avoid computing the distance more than once, we cache the result into a map so it can be queried in subsequent symbolic execution steps. This is relevant because the distance between two nodes was often queried multiple times (e.g., due to backtracking) and without caching, there would be re-computations. Our search strategy does not require the distance computation to be precise: a less accurate estimation will not affect the correctness of the algorithm. For example, we use approximation to handle external libraries with missing source code. The potential imprecision caused by such approximation will be eliminated during the subsequent dynamic analysis step, which concretely executes the PuDs for interleaving schedule generation and replay.

5.2 Seeding Concrete Configurations and Inputs

During symbolic execution, the symbolic variables of each P_i can be either user inputs or configuration options. For many Linux applications, configuration options (or configuration parameters) are treated as a special type of inputs. These options are often specified in a configuration file read by the program. Treating configuration options (or configuration files) as symbolic inputs, which is the standard approach in symbolic execution, can be inefficient because it may take a large amount of time and memory to explore all possible configuration options, most of which are invalid (e.g., modules.server = "?#$") [37]. Using concrete options can simplify the constraint solving and thus improve the scalability of symbolic execution. Therefore, we propose a heuristic method for identifying configuration options that are more relevant to the logging points.

Specifically, we employ a technique called *static program chopping*, which computes the intersection of forward and backward slicing. In program chopping, two points of interest – source (s) and target (t) – are chosen, and the chop [36] is defined as all statements that could transmit effect of executing s to t. As such, chopping reveals the ways in which one program point may affect another program point.

DESCRY takes the process P and the goal list gl as input. It considers the read points of each configuration option $c \in C$ as chopping sources (S_c) and logging points in the goal list ($gl \in gl$) are chopping targets (T_{gl}). For each c and gl, it applies the static chopping algorithm to compute the chop set CS_c with respect to c and gl. If CS_c is not empty, the configuration option c is potentially relevant to gl, in which case DESCRY adds c to the relevant configuration option set C_{gl}. When exploring a goal list, `GuidedSymbolicExecution` directly takes the concrete configuration options in C_{gl} as input, as opposed to treating them as symbolic values.

Given N PuDs, where $N > 1$, DESCRY in general may need to invoke the `GuidedSymbolicExecution` routine N times. To reduce the computational cost, it heuristically seeds the concrete input data generated by symbolic execution from one PuD P to another PuD P', if P' accepts the same type of input data. In this case, rather than invoking `GuidedSymbolicExecution` on P', we check if we can use the input data D from P as the concrete input to P'. If D allows P' to reach its goal list, then no expensive constraint solving is needed.

6 INTERLEAVING SCHEDULE GENERATION

We propose a predictive dynamic analysis method to generate the failure-inducing interleaving schedule when given a set of PuDs and their corresponding data inputs. At the high level, we first execute the application (consisting of all PuDs) under an arbitrary schedule to generate the initial execution. Since the schedule is defined by the order of inter-process events (system calls), it is represented by the sequence of these events. If the initial execution does not trigger the failure, we systematically generate alternative interleavings of these events. Our method for generating alternative interleavings relies on a *predictive (constraint) model* constructed from the initial execution trace.

6.1 Constraint Model

The constraint model captures the partial order relation of the system call events appeared in the initial execution. For any two events e_i and e_j, we say that $e_i \rightarrow e_j$ if e_i must happen before e_j. If two events do not follow any must-happen-before order, they can be flipped to create a new schedule. Specifically, we employ the following order relations:

- **Program order:** $e_i \rightarrow e_j$ when e_i occurs before e_j in the same process/thread.
- **Fork-return order:** $e_i \rightarrow e_j$ when e_i is the `fork` that starts the child process P_j, and e_j is the return of P_j.
- **Wait-exit order:** $e_i \rightarrow e_j$ when e_i is the `wait` that blocks a parent process, and e_j is the `exit` that terminates the child process.
- **Pipe-read order:** $e_i \rightarrow e_j$ when e_i is a stream write to a pipe, and e_j is the corresponding stream read.
- **Signal order:** $e_i \rightarrow e_j$ when e_i is an event of the process before it enables a software signal S, and e_j is in S.

We represent the constraint model as a partial order graph (POG) denoted by (V, E), where V is the set of nodes corresponding to the inter-process events and E is the set of edges between the nodes. Each edge $(e_i, e_j) \in E$ represents a must-happen-before relation between e_i and e_j.

To generate the failure-inducing schedule, we systematically permute events in the initial execution, by flipping the order of system calls involving shared resource accesses while respecting the order relations. Specifically, each time, we pick an event pair and flip their order to generate a new interleaving schedule offline, and then check if the schedule is feasible by replaying the PuDs under the new schedule.

6.2 Generating New Schedules

Our new schedule generation algorithm takes the partial-order graph as input and returns a set of interleaving schedules as output. Internally, DESCRY analyzes two PuDs at a time, e.g., the failing process P_F and a PuD P_i chosen by DESCRY to pair with P_F. This is an approximation that is sufficient for handling all bugs encountered in our experiments (where all the bugs involve only two processes). It is consistent with a prior study [23], which found most real-world bugs involve two threads or processes. There are two challenging problems in generating new schedules: (1) which event pair to pick and flip, as there can be many event pairs; and (2) how to ensure the new schedule is not only feasible but also more likely to trigger the failure.

To solve the first problem, DESCRY prioritizes event pairs where at least one of the two events is a write system call and is closer

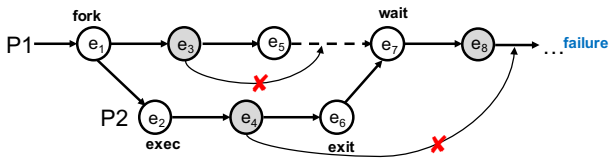

Figure 7: An example of schedule generation.

to the failure point, based on our observation that events closer to the failure point are more likely to be relevant. DESCRY starts from events in P_F and selects the event e_f closest to the failure point. Next, it selects an event e_i that is close to but does not have order relations with e_f. The event pair (e_i, e_f) is called a suspicious event pair. For each suspicious event pair, DESCRY generates a new interleaving schedule by flipping the order of the two events. To avoid obviously-infeasible schedules, which is the second problem mentioned above, DESCRY ensures that the new schedule is consistent with the order relations captured by the POG.

Figure 7 shows an example trace, where (e_4, e_8) and (e_3, e_4) are two event pairs in the two processes $P1$ and $P2$, respectively. The solid edges represent the order relations. Here, DESCRY would consider flipping (e_4, e_8) first because it has a higher priority (e_8 is closer to the failure point). However, the two events have a must-happen-before relation because of *fork-return*, and therefore cannot be flipped. In contrast, the suspicious event pair (e_3, e_4) can be flipped and thus DESCRY would try to schedule e_3 after e_4.

However, there are multiple ways of executing e_3 after e_4. If we simply execute e_3 immediately after e_4, it would violate the partial order relation of e_3 and e_5. Therefore, during the reordering, our algorithm moves not only the candidate event but also events depending on the candidate event. In Figure 7, this corresponds to moving both e_3 and e_5 after e_4, which leads to the new interleaving schedule $(e_1, e_2, e_4, e_3, e_5, e_6, e_7, e_8)$.

In the replay phase, DESCRY executes the PuDs with the newly generated data inputs while controlling the system call events to follow the newly generated interleaving schedule. If the execution does not match the permuted events, which means the execution has taken a different branch from what is expected, DESCRY skips the execution (since it is an infeasible schedule) and proceeds to generate another interleaving schedule; this amounts to identifying and flipping another suspicious event pair. If DESCRY cannot find any failure-inducing interleaving schedule after permuting all suspicious event pairs, it will backtrack and invoke the symbolic execution procedure again to generate another set of data inputs.

7 EXPERIMENTS

We have implemented DESCRY in a software tool built upon a number of open-source platforms. Specifically, our static program analysis for mapping log messages to program statements was implemented in LLVM [1], our log-guided symbolic execution was implemented using KLEE [5], and our interleaving schedule generator was implemented using the Simics Virtual Platform [8].

To evaluate DESCRY, we consider two research questions:

RQ1: How effective is DESCRY in reproducing real-world concurrency failures in multi-process applications based only on the default logs generated by these applications?

RQ2: How efficient is DESCRY in computing the failure-inducing data inputs as well as the event interleaving schedules?

Table 1: Benchmarks and Descriptions of the Failures

Prog.	*LoC*	*Bug ID*	*# Proc*	*Observed Failure*
mv	7,002	Bugzilla-438076	19	another process terminates ("file is missing")
rm	5,525	Bugzilla-1211300	12	rm terminates ("directory not empty")
pxz	370	Bugzilla-1182024	21	file permission mode is modified
chmod	3,983	GNU-11108	13	file permission mode is modified
ln	3,890	Debian-357140	18	ln terminates ("file doesn't exist")
mkdir	4,213	Debian-304556	10	file permission mode is modified
mknod	3,840	Debian-304556	21	file permission mode is modified
mkfifo	3,959	Debian-304556	12	file permission mode is modified
tail1	4,492	Changelog	22	output not updated after attached process exits
tail2	4,317	Changelog	22	incorrect output lines after a delivery of signal
cp1	4,010	Changelog	18	file permission mode is modified
cp2	4,132	Changelog	18	directory create fails ("directory exists")
sort (sig)	3,862	Changelog	21	program terminates ("unlink failed")
ps	4,695	Bugzilla-55057	25	error message (print "grep -n aaa" itself)
strace1	25,192	Bugzilla-558471	17	the process been tracked is not detatched
strace2 (sig)	25,192	Bugzilla-548363	17	hang
logrotate	2,393	Debian-400198	28	file access permission denied
bash	39,102	Debian-283702	31	corrupted history file
tcsh	47,167	Debian-632892	31	corrupted history file
bzip2	9,263	Debian-303300	22	file permission mode is modified
gzip	7,252	Debian-303927	19	file permission mode is modified
lighttpd-1	37,919	Lighttpd-2217	29	http timeout
lighttpd-2	41,292	Lighttpd-2542	29	error (200 response with an empty body)
apache	195,005	Apache-43696	31	server shutdown command is ignored

7.1 Benchmarks and Evaluation Metrics

All our benchmarks are real Linux applications with known concurrency failures due to incorrectly shared resources between processes and/or signal handlers. They are identified by searches of open-source repositories such as GNU, Bugzilla, and Debian. There are 22 program versions from 20 unique applications, among which 11 applicaitons were from Linux coreutils. Searches of these open-source repositories were conducted by research assistants (students) who are not involved in the DESCRY project to minimize bias. Furthermore, the root causes of these failures were unknown to us until we finished running and analyzing the results of DESCRY.

Table 1 shows the statistics of each benchmark, including the name, the number of non-comment lines of code, the bug ID, the total number of processes, and a short description of the symptom. In this table, the benchmarks are divided into two categories, separated by the double horizontal lines. Benchmarks in the first category are from the GNU Coreutils. Benchmarks in the second category are from other popular Linux applications.

Evaluation metrics. We measure both the *effectiveness* and the *efficiency*. To measure the effectiveness, we check whether a known failure can be successfully reproduced within the time limit. To measure the efficiency of DESCRY, we analyze the failure-reproduction time, by measuring the time spent on PuD identification, log analysis, input generation, and schedule generation, respectively.

Our experiments were conducted on a computer with an Intel Core i5-2400 3.10 GHz CPU, 8 GB RAM and Ubuntu 14.10 Linux. We used the most recent version of KLEE built from LLVM 3.4. We set a two-hour time limit for all four techniques. To control for variance due to randomization, we ran each of the three DESCRY techniques five times to compute the average.

7.2 Experimental Results and Analysis

Table 2 summarizes the results of applying DESCRY to the benchmarks. Column 1 shows the benchmark name. Column 2 shows the number of log messages. Column 3 shows the number of relevant PuDs. Columns 4-6 show the result of our log analysis, including the number of relevant logging points (#RLP), the number of irrelevant logging points (#ILP), and the number of goal lists (#GL).

Column 7 shows the failure-inducing data input. Columns 8-10 show the result of our schedule generation, including the total number of explored happens-before relations (#HB), system calls (#SC), and system-level resources (#SV). In the latter two columns, numbers in the parenthesis correspond to the events of interest (i.e., relevant system calls and shared resources). Column 11 shows the total number of loop iterations in the main algorithm; it is the summation of the number of data inputs generated and the number of times that goal relaxation occurred (numbers marked with ⋆ indicates that goal relaxation occurred). Column 12 shows the total execution time of DESCRY. Column 13 shows the result of failure reproduction, where "Y" means it succeeded and "No" means it failed. Finally, Column 14 shows the root cause of the failure, where events are system calls in the current process, whereas system calls marked with ⋆ are from other processes. For example, in `mv` (with bug ID 438076), the buggy process causes another process to terminate early due to a missing file when the atomicity of `unlink` and `rename` is broken by the write operation of another process.

RQ1: Effectiveness in reproducing failures. DESCRY succeeded in reproducing 22 of the 24 failures. We repeated each experiment five times, and found that DESCRY consistently succeeded in generating the failure-inducing data inputs and interleaving schedules to reach all failure points of these 22 failures. Therefore, the results indicate that DESCRY is effective in reproducing system-level concurrency failures.

For the two cases where DESCRY failed, the failures were all due to limitations of KLEE in modeling the file system, not limitations of the algorithms in DESCRY. Specifically, the failures require the program inputs to be hierarchical directories, but KLEE models the file system as a flattened system, where symbolic files have pathnames such as "A","B", and "C" without any hierarchy [5]. Fixing this problem in KLEE would have allowed DESCRY to handle these two failures without any modification.

DESCRY is the only tool that can automatically reproduce the 22 known failures. Existing tools, such as RacePro [21] and SimRacer [29, 38], may appear to be similar but cannot really solve the same problem, due to the following limitations. First, they require the user to provide concrete data inputs, which cannot be satisfied in practice. Second, their search for erroneous interleaving schedules is not guided by logs. Since prior work has shown that SimRacer outperformed RacePro (due to limitations of RacePro such as replay divergence [38]), in this study, we compare DESCRY only to SimRacer. SimRacer does not have the capability of generating new data inputs, so we had to feed random inputs to SimRacer. As shown in Table 3, SimRacer reproduced only eight out of the 22 failures reproduced by DESCRY.

Within DESCRY, we also evaluated different search strategies. By default, DESCRY uses our log-directed symbolic execution to compute data inputs. Another option is to use the search strategy provided by KLEE, denoted DESCRY$_{DFS}$, where the symbolic execution is not guided by the logging points. This controlled experiment allows us to evaluate the performance of our new input generation algorithm. The third option is DESCRY$_{AS}$, which uses the logging points to guide symbolic execution, but does not use our new schedule generation algorithm. Instead, DESCRY$_{AS}$ relies on active testing techniques [31, 38]. This controlled experiment allows us to evaluate the performance of our new schedule generation algorithm.

As shown in Table 3, DESCRY$_{DFS}$ reproduced only 6 of the 22 failures reproduced by DESCRY, indicating that our new logging points-guided symbolic execution procedure in DESCRY is significantly more effective than existing techniques. DESCRY$_{AS}$

Table 2: Results of applying DESCRY to all benchmark applications.

Prog.	*#MSG*	*#PuDs*	*Log analysis*			*Data input generated*	*Schedule Generation*			*Total Iterations*	*Time (min)*	*Replay success*	*Bug Description*
			#RLP	*#ILP*	*#GL*		*#HB*	*#SC*	*#SV*				
mv	2	3	5	62	6	`A B.lnk`	0	46 (27)	15 (5)	20*	8	Y	(unlink, rename, ..., stat*) → (unlink, ..., stat*, rename)
rm	102	3	2	29	2	?	0	38 (18)	5 (14)	1138	>	No	(openat, fstat, ..., unlink*) → (openat, ..., unlink*, fstat)
pxz	2	5	2	33	1	`-z -t A`	0	1494 (959)	61 (5)	1	5	Y	(umask, chmod, ..., symlink*) → (umask, ..., symlink*, chmod)
chmod	1	3	1	21	1	?	0	36 (17)	15 (5)	1296	>	No	(stat, fchmodat, ..., symlink*) → (stat, ..., symlink*, fchmodat)
ln	1	2	2	2	1	`-s -f A B`	1	36 (16)	13 (6)	10*	4	Y	(stat, unlink, ..., unlink*) → (stat, ..., unlink*, unlink)
mkdir	98	3	3	6	2	`-m 400 A`	0	36 (18)	13 (6)	28	21	Y	(mkdir, chmod, ..., symlink*) → (mkdir, ..., symlink*, chmod)
mknod	70	3	3	20	2	`-m 400 A`	0	39 (19)	14 (6)	22	20	Y	(mkdir, chmod, ..., symlink*) → (mknod, ..., symlink*, chmod)
mkfifo	3	3	3	7	2	`-m 400 A`	0	34 (16)	13 (6)	26	20	Y	(mkdir, chmod, ..., symlink*) → (mkdir, ..., symlink*, chmod)
tail1	201	2	11	68	6	`-s -f A --pid=101`	2	52 (32)	14 (6)	4	5	Y	(write, ..., read*, exit) → (read*, write, ..., exit)
tail2	112	2	5	65	3	`-f A`	2	88 (37)	15 (6)	1	3	Y	(read, stat, ..., write*) → (read, ..., write*, stat)
cp1	149 (3)	4	5	126	3	`-p 600 dir1 dir2`	0	72 (48)	18 (8)	8	6	Y	(mkdir, stat, ..., fchmod*) → (mkdir, ..., fchmod*, stat)
cp2	68	3	4	131	2	`-R dir1 dir2`	0	44 (23)	16 (6)	11*	8	Y	(stat, mkdir, ..., mkdir*) → (stat, ..., mkdir*, mkdir)
sort (sig)	71	2	7	54	3	`-n -r A`	3	53 (29)	21 (9)	1	5	Y	(read, unlink, ..., unlink*) → (read, ..., unlink*, unlink)
ps	447	2	2	419	1	`-u`	4	2636 (2512)	65 (8)	2	14	Y	(read*, execve) → (execve, read*)
strace1	65	2	3	2655	1	`-f A`	5	1138 (331)	24 (7)	1	5	Y	(stat, write*, fork) → (fork, stat*, write)
strace2 (sig)	65	2	4	2655	2	`A.fork`	4	1492 (382)	32 (9)	1	8	Y	(wait*, execve) → (execve, wait*)
logrotate	28	4	4	224	2	`-f A`	3	96 (31)	23 (7)	48*	4	Y	(open, chown, ..., fchmod*) → (open, ..., fchmod*, chown)
bash	2	2	2	836	1	`-c "aa;history -w"`	4	619 (208)	64 (41)	102	48	Y	(write, ..., write) → (..., write, write*)
tcsh	2	2	5	1045	2	`-c "aa"`	3	358 (172)	64 (23)	138	18	Y	(write, ..., write) → (..., write, write*)
bzip2	2	2	2	209	1	`-d A`	0	63 (29)	19 (5)	3	9	Y	(close, chmod, ..., write*) → (close, ..., write*, chmod)
gzip	2	3	3	211	1	`-d A`	0	49 (24)	14 (5)	5	10	Y	(close, chmod, ..., symlink*) → (close, ..., symlink*, chmod)
lighttpd-1	503	4	15	1268	19	`mod.cgi`	26	2058 (1064)	1962 (581)	199*	21	Y	(waitpid*, exit) → (exit, waitpid*)
lighttpd-2	522	4	17	1356	20	`mod.cgi`	29	2132 (948)	1864 (572)	251*	19	Y	(waitpid, close, ..., waitpid*) → (close, ..., waitpid*, waitpid)
apache	698	5	24	3505	18	`-k start`	41	3946 (1588)	2433 (962)	432*	29	Y	(signal*, sigpromask) → (sigpromask, signal*)

Table 3: Comparing the success rate of different methods.

Prog.	*DESCRY success*	*SimRacer [38] success*	*DESCRY$_{DFS}$ success*	*DESCRY$_{AS}$ success*
mv	Y	No	No	Y
rm	No	No	No	No
pxz	Y	No	No	Y
chmod	No	No	No	No
ln	Y	No	No	Y
mkdir	Y	No	No	Y
mknod	Y	No	No	Y
mkfifo	Y	No	No	Y
tail1	Y	No	No	Y
tail2	Y	No	No	Y
cp1	Y	No	No	Y
cp2	Y	Y	Y	Y
sort (sig)	Y	No	No	Y
ps	Y	Y	Y	Y
strace1	Y	Y	No	Y
strace2 (sig)	Y	No	No	Y
logrotate	Y	Y	No	Y
bash	Y	No	No	Y
tcsh	Y	No	No	Y
bzip2	Y	Y	Y	Y
gzip	Y	Y	Y	Y
lighttpd-1	Y	Y	No	Y
lighttpd-2	Y	Y	No	Y
apache	Y	No	No	Y
total	**22**	**8**	**4**	**22**

reproduced all 22 failures. However, as will be discussed in the results of RQ2, it took significantly longer time than DESCRY.

RQ2: Efficiency in reproducing failures. For all 22 successful cases, DESCRY reproduced the failures in less than 40 minutes. On average, each benchmark took only a couple of minutes, indicating that DESCRY is efficient for practical use. Furthermore, on average 81.7% of the time was spent on generating the failure-inducing data inputs, while log analysis and schedule generation took only 8.1% and 10.2% of the time, respectively.

Figure 8 compares the total time (in minutes) taken by SimRacer, DESCRY$_{DFS}$, DESCRY$_{AS}$, and DESCRY. When a bar reaches the top of the vertical axis, it means the corresponding method failed to reproduce the failure in the two-hour time limit. Among the eight failures successfully reproduced by SimRacer, SimRacer was 3 times faster than DESCRY on average. Unfortunately, there are 14 other failures that SimRacer could not reproduce but DESCRY could. Compared to DESCRY, the method DESCRY$_{DFS}$ was 2.1 to 11 times slower on the individual benchmarks both can handle (and 7.8 times slower on average), and the method DESCRY$_{AS}$ was 1.5 to 9 times slower on the individual benchmarks both can handle (and 2 times slower on average). These results demonstrate the effectiveness of

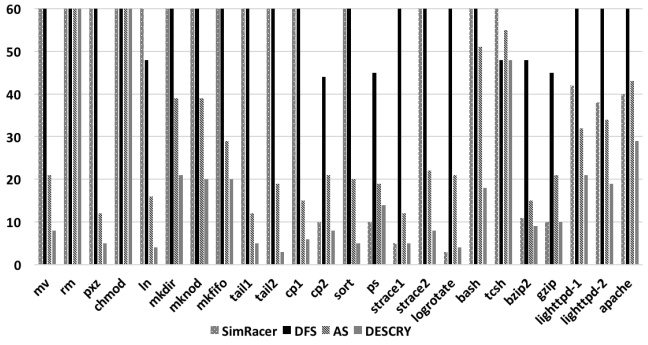

Figure 8: Comparing the time taken by different methods.

our new symbolic execution and predictive schedule generation algorithms.

7.3 Case Study

We present two representative examples where DESCRY succeeded in reproducing the system-level concurrency failures, to demonstrate why it is helpful to developers.

Bash – Bash. When executing multiple bash shells concurrently, the shell history file may be corrupted, e.g., when one bash process *P*1 opens *.bash_history* using *open(fd, O_WRONLY)* before writing to it, and another bash process *P*2 also opens the file for writing. The failed execution is `P1:open` → `P2:open` → `P1:write` → `P2:write` → `P1:close` → `P2:close`, which produced only the log message "writing to the history file". Therefore, manually inspecting the source code of *bash* to figure out the root cause of the failure would be extremely difficult. In contrast, DESCRY was able to automatically reproduce the failure using the log message. Specifically, it produced a program input containing the command-line option "-c" (i.e., to execute a new command) and the bash command "aa; history -w", as well as the failure-inducing event interleaving schedule.

Lighttpd – CGI. When the CGI program writes an HTTP response to *stdout*, the server cannot complete the transaction. Under normal

execution, the `lighttpd` process calls *waitpid* to wait for the CGI process to terminate. Once the CGI terminates, the `lighttpd` receives the *FDEVENT_HUP* signal and the *http* transaction successfully terminates. However, when the *waitpid* in the `lighttpd` process happens after the CGI process terminates (*exit*), `lighttpd` does the cleanup work and removes the pipe's *fd*. In this case, `lighttpd` never gets the *FDEVENT_HUP* event, and thus the connection cannot be closed, causing a timeout of the *http* transaction. This race is caused by the *waitpid* and *exit* calls on the shared resource `/proc/pid`, which corresponds to the following log messages:

```
Benchmarking localhost
Completed 100 requests
...
Completed 599 requests
Completed 600 requests
apr_poll: The timeout specified has expired (70007)
Total of 600 requests completed
```

For `lighttpd`, DESCRY first identified the repeated messages (i.e., lines 2-5: a list of items printed by the same logging point in `lighttpd`) and located the relevant logging points. Then, it used static chopping to identify that the configuration option `mod_cgi` is relevant to the logging points. This option is specified in the configuration file. Next, it generated an input (`start`) using guided symbolic execution. Finally, given the configuration file and a regular CGI process (`http://cgi-bin/hello.cgi`), DESCRY generated a failure-inducing interleaving schedule by swapping the order of `waitpid` and `exit`.

7.4 Discussion

Symbolic execution. One limitation of DESCRY is that it relies on symbolic execution, which is known to only scale up to medium-sized applications, as in KLEE, although if suitably defined, well tuned and well engineered, symbolic execution can scale up to larger applications [10].

Quality of logs. The effectiveness of DESCRY depends on the quality of logs – in general, it performs better when given more detailed logs. Fortunately, research has shown that logging is pervasive in software development practice [39–41]. During our experiments, the size of the logs ranges from one message to hundreds of messages (Column 2 of Table 2), indicating its effectiveness even in the absence of detailed logs. (Server and storage applications tend to produce significantly more detailed logs [40].) There is only one case (*tcsh*) where our log-guided input generation algorithm did not outperform the DFS-based algorithm of KLEE. Upon further examination, we found it is because the logging points were not under control flow points that lead to the observed failure. In other words, these log messages are too general to provide any useful information of the erroneous execution.

It is worth noting that logs used by DESCRY were generated by the applications under their default production setting (i.e, verbosity level). In such case, the logging overhead is less than 1% in the eleven applications from the Linux coreutils, and 1.3% – 2.8% in other nine applications.

8 RELATED WORK

Fault detection. The focus of our work is reproducing failures using only default logs generated by the failed applications. However, there is related work on detecting faults in concurrent systems, such as the *time of check to time of use* (TOCTTOU) bugs [28, 34], signal races [33], and order violations [22]. These techniques focus only on fault detection using existing data inputs, e.g., by executing target programs in specialized environments. None of them can automatically generate new data inputs to reproduce system-level failures. Similarly, RacePro [21] is a tool for detecting process-level races but it assumes the failure-inducing data input already exists. In contrast, DESCRY relies only on default logs generated in the deployment environment.

Log and core dump analysis. There are tools for analyzing failure information in core dumps and heap data [35, 42]. Core dumps were also used to guide the search for failure-inducing program inputs [30]. Jin et al. [18] use various runtime information of the failed execution to guide the generation of program inputs. SherLog [40] analyzes the program's source code by leveraging coarse-grained logs generated by the program to infer paths that may lead to the failure. However, these techniques focus on sequential programs only. Furthermore, since they rely on core dumps, they cannot handle non-crashing failures that do not lead to core dumps.

Symbolic execution. Symbolic execution has been widely used in software testing to help increase code coverage and reveals bugs for both sequential [4, 5, 24, 25] and concurrent [9, 32] software. Zamfir et al. [42] leverage symbolic execution to generate failure-inducing execution paths and the corresponding interleaving schedules. However, their method only handles thread-level concurrency failures. Symbolic execution has also been combined with other techniques for testing and debugging multithreaded software [13, 15, 19]. For example, Guo et al. [13] use static analysis to identify program paths that do not lead to any failure and prune them away during symbolic execution. Huang et al. [15] compute feasible thread schedules using a constraint solver by combining constraints from thread paths and constraints from the memory model. Again, none of these techniques handle multi-process systems.

Fault localization. Existing methods for localizing concurrency bugs often rely on analyzing a large set of (passed and failed) test runs to narrow down the root cause [6, 26, 35]. For example, Park et al. [26] localize anomalous data-access patterns associated with a program's pass/fail results based on statistical analysis. Weeratunge et al. [35] diagnose Heisenbugs by comparing core dumps of failing and passing runs. However, these methods all assume that developers already have the failure-inducing data inputs together with a large set of passing and failing execution traces. In contrast, our method does not make such assumptions. Instead, it relies only on the default logs of failed applications.

9 CONCLUSIONS

We have presented DESCRY, the first fully automated software tool for reproducing concurrency failures of multi-process applications using only default logs from these applications. DESCRY leverages new static program analysis to identify the logging points, new symbolic execution strategy to generate data inputs, and new predictive dynamic analysis to generate failure-inducing interleaving schedules. We have evaluated DESCRY on a large number of widely-used Linux applications and showed that it successfully reproduced 22 real failures that could no be reproduced by any other tool. Therefore, it is a useful addition to the application developer's toolbox for debugging multi-process concurrent software systems.

ACKNOWLEDGMENTS

This work was supported in part by NSF grants CCF-1464032, CNS-1405697, and CCF-1722710.

REFERENCES

[1] Vikram Adve, Chris Lattner, Michael Brukman, Anand Shukla, and Brian Gaeke. LLVM: A low-level virtual instruction set architecture. In *IEEE/ACM International Symposium on Microarchitecture*, San Diego, California, Dec 2003.

[2] Gautam Altekar and Ion Stoica. ODR: Output-deterministic replay for multicore debugging. In *Proceedings of the symposium on Operating systems principles*, pages 193–206, 2009.

[3] 2016. http://linux.die.net/man/8/auditd.

[4] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing symbolic execution with veritesting. In *International Conference on Software Engineering*, pages 1083–1094, 2014.

[5] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 209–224, 2008.

[6] Jong-Deok Choi and Andreas Zeller. Isolating failure-inducing thread schedules. In *International Symposium on Software Testing and Analysis*, pages 210–220, 2002.

[7] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *International Conference on Software Engineering*, pages 342–351, 2005.

[8] Jakob Engblom, Daniel Aarno, and Bengt Werner. *Full-System Simulation from Embedded to High-Performance Systems*. 2010.

[9] Azadeh Farzan, Andreas Holzer, Niloofar Razavi, and Helmut Veith. Con2colic testing. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 37–47, 2013.

[10] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[11] Patrice Godefroid and Nachiappan Nagappan. Concurrency at Microsoft – an exploratory survey. Technical report, Microsoft Research, 2008.

[12] Shengjian Guo, Markus Kusano, and Chao Wang. Conc-iSE: Incremental symbolic execution of concurrent software. In *IEEE/ACM International Conference On Automated Software Engineering*, pages 531–542, 2016.

[13] Shengjian Guo, Markus Kusano, Chao Wang, Zijiang Yang, and Aarti Gupta. Assertion guided symbolic execution of multithreaded programs. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 854–865, 2015.

[14] Jeff Huang, Peng Liu, and Charles Zhang. Leap: Lightweight deterministic multi-processor replay of concurrent java programs. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 207–216, 2010.

[15] Jeff Huang, Charles Zhang, and Julian Dolby. Clap: Recording local executions to reproduce concurrency failures. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 141–152, 2013.

[16] Koral Ilgun. Ustat: A real-time intrusion detection system for unix. In *IEEE Computer Society Symposium on Research in Security and Privacy*, pages 16–28, 1993.

[17] Nicholas Jalbert and Koushik Sen. A trace simplification technique for effective debugging of concurrent programs. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 57–66, 2010.

[18] Wei Jin and Alessandro Orso. Bugredux: Reproducing field failures for in-house debugging. In *International Conference on Software Engineering*, pages 474–484, 2012.

[19] Sepideh Khoshnood, Markus Kusano, and Chao Wang. ConcBugAssist: Constraint solving for diagnosis and repair of concurrency bugs. In *International Symposium on Software Testing and Analysis*, pages 165–176, 2015.

[20] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 193–204, 2012.

[21] Oren Laadan, Nicolas Viennot, Chia-Che Tsai, Chris Blinn, Junfeng Yang, and Jason Nieh. Pervasive detection of process races in deployed systems. In *ACM symposium on Operating Systems Principles*, pages 353–367, 2011.

[22] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. Samc: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 399–414, 2014.

[23] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 329–339, 2008.

[24] Paul Dan Marinescu and Cristian Cadar. Make test-zesti: A symbolic execution solution for improving regression testing. In *International Conference on Software Engineering*, pages 716–726, 2012.

[25] Paul Dan Marinescu and Cristian Cadar. KATCH: High-coverage testing of software patches. In *Proceedings of the Foundations of Software Engineering*, pages 235–245, 2013.

[26] Sangmin Park, Richard W. Vuduc, and Mary Jean Harrold. Falcon: Fault localization in concurrent programs. In *International Conference on Software Engineering*, pages 245–254, 2010.

[27] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H Lee, and Shan Lu. PRES: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the symposium on Operating systems principles*, pages 177–192, 2009.

[28] Mathias Payer and Thomas R. Gross. Protecting applications against TOCTTOU races by user-space caching of file metadata. pages 215–226, 2012.

[29] Supat Rattanasuksun, Tingting Yu, Witawas Srisa-An, and Gregg Rothermel. Rrf: A race reproduction framework for use in debugging process-level races. pages 162–172, 2016.

[30] Jeremias. Robler, Andreas Zeller, Gordon Fraser, Cristian. Zamfir, and George Candea. Reconstructing core dumps. In *IEEE International Conference on Software Testing, Verification and Validation*, pages 114–123, 2013.

[31] Koushik Sen. Race directed random testing of concurrent programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 11–21, 2008.

[32] Koushik Sen and Gul Agha. Concolic testing of multithreaded programs and its application to testing security protocols. Technical Report UIUCDCS-R-2006-2676, University of Illinois at Urbana Champaign, 2006.

[33] Takamitsu Tahara, Katsuhiko Gondow, and Seiya Ohsuga. DRACULA: Detector of data races in signals handlers. In *Proceedings of the Asia-Pacific Software Engineering Conference*, 2008.

[34] Dan Tsafrir, Tomer Hertz, David Wagner, and Dilma Da Silva. Portably solving file TOCTTOU races with hardness amplification. In *Proceedings of the USENIX Conference on File Storage Technologies*, 2008.

[35] Dasarath Weeratunge, Xiangyu Zhang, and Suresh Jagannathan. Analyzing multicore dumps to facilitate concurrency bug reproduction. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 155–166, 2010.

[36] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449, 1981.

[37] Edmund Wong, Lei Zhang, Song Wang, Taiyue Liu, and Lin Tan. Dase: Document-assisted symbolic execution for improving automated software testing. In *IEEE International Conference on Software Engineering*, pages 620–631, 2015.

[38] Tingting Yu, Witawas Srisa-an, and Gregg Rothermel. SimRacer: An automated framework to support testing for process-level races. In *International Symposium on Software Testing and Analysis*, pages 167–177, 2013.

[39] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. Sherlog: Error diagnosis by connecting clues from run-time logs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 143–154, 2010.

[40] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. Characterizing logging practices in open-source software. In *International Conference on Software Engineering*, pages 102–112, 2012.

[41] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving software diagnosability via log enhancement. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–14, 2011.

[42] Cristian Zamfir and George Candea. Execution synthesis: A technique for automated software debugging. In *European Conference on Computer Systems*, pages 321–334, 2010.