

Locating Failure-Inducing Environment Changes

Dawei Qi, Minh Ngoc Ngo, Tao Sun, Abhik Roychoudhury
School of Computing, National University of Singapore
{dawei,cngo,sunt,abhik}@comp.nus.edu.sg

ABSTRACT

Traditionally, debugging refers to the process of locating the program portions which are responsible for a program failure. However, a program also fails when the execution environment does not meet the requirement/assumption of the program. Unfortunately, few existing debugging techniques addresses the problem of changing operating system environment. In this paper, we propose an effective record-replay technique called *Semi-replay* to solve this problem. Semi-replay records all the essential interactions between an application and its underlying operating system environment where it successfully executed. Semi-replay then allows the recorded interactions to be partially replayed and partially executed in another operating system to identify those interactions which contribute to the root cause of the application failure induced by the environment changes. We have conducted three case studies on real-life programs which show the significance and efficiency of the Semi-replay technique in locating failure-inducing environment changes. We have also implemented a tool for the Linux kernel to demonstrate the feasibility of the proposed approach.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging—*Debugging aids*

General Terms

Experimentation, Reliability

Keywords

Configuration Error, Environment Change, Semi-replay

1. INTRODUCTION

In application software, many bugs are related to the *execution environment*. According to a study by Chandra et al. [3], around 56% of faults in Apache depend on execution environment. Browsing through the Ubuntu bug list [23] reveals that many bugs are also environmental-related. More specifically, these bugs occur due to

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PASTE'11, September 5, 2011, Szeged, Hungary.

Copyright 2011 ACM 978-1-4503-0849-6/11/09 ...\$10.00.

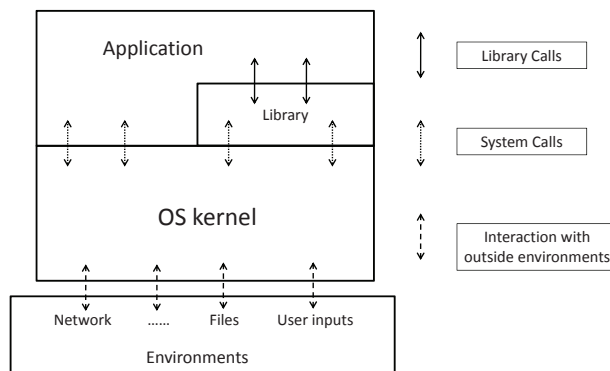


Figure 1: Relations between application, library, OS kernel and outside environments

some changes in the Operating System (OS) environments. It is quite common that an application works perfectly in one OS environment but fails in another OS environment. As operating systems become increasingly complicated, debugging those operating system induced errors is a challenging task.

Figure 1 shows the relationship between applications and the operating system. An application has access to services provided by the OS kernel through system calls. System calls are defined in the Application Binary Interface (ABI) of an OS. The system calls allow the application to invoke kernel services to perform privileged tasks on behalf of the application, such as read or write a file, issue a control command to a device, create a new process, allocate memory and so on. Generally, systems also provide libraries that sit between the OS kernel and a normal application to increase portability, for example the `glibc` library. In this way, the application is less dependent on the OS kernel. The application communicates with the library functions through library calls. Figure 1 also shows that the OS kernel acts as an interface between an application and the outside environment to enable an application to interact with the user (or the network or the file system) to perform its intended function. As such, the operating system plays an important part in defining an application's behavior. Even when the input and the program is fixed, the behavior of a program P with an input t can still be affected by a lot of factors in the operating system. For example, dynamic libraries can be implemented differently which leads to implicit semantic difference. A more common case is that the content of some configuration file might be different from one environment to another.

In this paper, we assume that the application is statically linked. The problem we tackle can be formalized as follows.

Problem Statement: Suppose we have an application program P , an input t , and two OS environments E and E' . The execution of P with input t succeeds in E but fails in E' . In this paper, we are trying to explain the failure of P in E' by discovering a subset δ of the changes in $\Delta E = E' \setminus E$. Of course, we try to minimize δ to make our result precise and meaningful.

Building on the assumptions that P does not change, only the OS environment changes, the different behaviors of P in E and E' are completely determined by communications between the program and the underlying OS environment through system call interface. One intuitive solution to the above problem would be to compare the interface communication through the system calls. For each executed system call, we can record the return value and the side effect of the system calls. For each system call $syscall$, we record it as $syscall = \langle num, paras, ret, side_effect \rangle$. The num is the system call number. Parameters and the return values are recorded in $paras$ and ret respectively. The side-effect of the system call (if any) is recorded in $side_effect$. In this paper, side-effect of a system call refers to the side-effect in the user space unless otherwise specified. The communications through system calls can be represented using a sequence $seq \langle syscall_1, syscall_2, \dots, syscall_n \rangle$. Two sequences of system calls can be compared. There are several difficulties in employing this approach. First, there can be a large number of differences between the two sequences (of system calls) if the two environments are very much different. However, only a very small subset of the differences could be the root cause of the failure. In this case, finding the root cause is a tedious task. Secondly, aligning the two sequences is another error-prone task. Moreover, as the execution environments are different, there might be some system calls in one sequence with no matching system calls in the other. This makes comparing the two sequences more complicated.

We propose a record-replay technique to solve the aforementioned problem. There have been a lot of existing record-replay research in the literature. However, none of the existing techniques is suitable for our task. Most of the existing record-replay techniques replay the execution using the entire recorded data in the same OS environment. On the other hand, we are trying to identify a small subset of the data that constitutes the reason of the failure. Hence, we want to selectively change part of a failing environment and test whether the program succeed. More specifically, we need a technique which enables us to partially replay (for successful environment) and partially execute (for the failing environment) the application. When a system call is replayed, the effect of the system call on the running application is as if the system call is executed in the environment where the system call is recorded. Unfortunately, none of the existing techniques handles such partial replay with recorded data from a different environment. Moreover, there are a few challenges in such a partial replay. One of these arises from the dependency between system calls. Figure 2 gives a concrete example: the `read` system call is dependent on the file descriptor returned by the `open` system call. Consequently, in selective replay they have to be replayed or executed together. If only the `open` is replayed and the `read` is executed, the reading from undefined file descriptor will result in errors. Another problem is caused by the complex trace of system calls which makes it difficult to locate the root cause. Therefore, we need an efficient selective replay strategy to enable fast localization of a system call which contributes to the root of the failure. Our selective replay approach is designed to overcome these challenges. The proposed record-replay technique tracks system call dependencies to avoid inconsistent system state. Our technique uses binary-search to fast localize the failure-inducing environment change in a complex system call sequence.

```

1  #include <unistd.h>
2  int main(int argc, char *argv[]){
3      int fd;
4      char[128] data;
5      char* config_file =
6          "/path/to/config_file";
7      fd = open(config_file, O_RDONLY);
8      read(fd, data, 128);
9      if(check_format(data)){
10         close(fd);
11         exit(1); //error
12     }else{
13         close(fd);
14         exit(0);
15     }

```

Figure 2: One example

The contribution of this paper are as follows:

- We propose a *Semi-replay* technique which allows partial replay and partial executing an application. The proposed technique enables efficient fault localization in the context of changing OS environment.
- We implemented our technique for Linux based on Valgrind.
- We conducted case studies on three real life bugs to evaluate the effectiveness of the proposed technique. In all three cases, our technique is able to locate the change in OS environment that causes the bug.

2. OVERVIEW

In this section, we give an overview of our approach through a motivating example. Let consider the program P in Figure 2. This program opens a configuration file (line 6), reads data from the file (line 7) and checks the format of the data (line 8). The `check_format()` function in P checks whether the data read from the file satisfies some pre-defined format. Suppose the program in Figure 2 succeeds in environment E but fails in environment E' . We assume that the failed execution of P in E' is caused by the configuration file in E' which does not follow the pre-defined format. Our debugging method works as follows.

We first record the system call sequences of P in E and use it to identify the problematic system call of P in E' . Suppose the sequence $seq = \langle open_E, read_E \rangle$ has been recorded when executing P in E , where $open_E$ and $read_E$ denote the system calls `open` and `read` in execution environment E respectively.

To identify the root cause of the failed execution of P in E' we “selectively” execute seq in E' . First we replay the first half of seq in E' and execute the remaining half in E' , which means the system call $open_E$ will be replayed in E' and the system call $read$ will be executed in E' . To replay a system call, we instrument the program executable file to intercept all the system calls and their return values and resulting side effects. Therefore, when executing P in E' , whenever the system call `open` is invoked, we will replace its return values and side effects with the one recorded for $open_E$. However, when the system call $open_E$ is replayed and the system call $read$ is executed in E' , the program execution would become inconsistent because of the dependency between $open_E$ and $read_E$. More specifically, the `open` in line 6 is not executed since it is only replayed in E' . As a result, the `read` in line 7 would

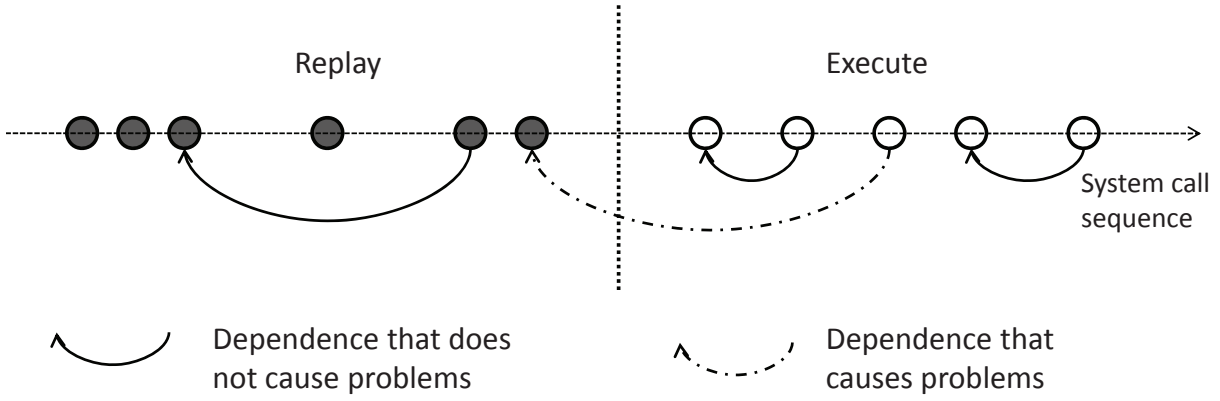


Figure 3: Dependencies among system calls. Each circle represents one system call. Arrows denote dependencies among system calls.

fail when being executed in E' because the file descriptor `fd` is not initialized. This program failure is caused by the dependency between the `read` in line 7 and the `open` in line 6.

To avoid inconsistent program state caused by system call dependencies, we track the effects of system calls on the kernel state, such as initializing file descriptor in an `open` system call. By doing this, we are able to detect that when we replay the `open` in E' , we also need to execute it in E' so as to generate the desired kernel state and more importantly, to enable the execution of `read` in E' . However, when executing P in E' , the recorded value for `openE` is still used as the return value for `open` in line 6.

Replaying `openE`, and executing `read` for the execution of P in E' , the execution fails. Consequently, we are able to deduce that the root cause of the failed execution lies in the second half of `seq` which is being executed in E' . As the second half contains only `readE`, we are able to locate `readE` as the system call which causes the failed execution of P in E' . We then further analyze from here. A simple comparison of the return values and side-effects recorded between the two executions reveals that when the system call `read` is executed, the contents of the file are different in two executions. This comparison enables us to conclude that the root cause of the failed execution of P in E' is caused by the wrong file format of the file provided in E' .

3. OUR APPROACH

Given a single-threaded deterministic program, the expected behavior of the program execution can be determined by three factors: (i) the program input (ii) the executed code and (iii) the environment. In this paper, we assume that the program is compiled statically, which means all the libraries must be included during compile-time. This guarantees that the entire executable is fixed. Therefore, the behavior of the program is only determined by (i) the program input and (ii) the interface communications through system calls. We also assume that same program input is used in the two executions. If for the two executions, one fails and one passes, then the problem lies in the interface communication with the underlying OS. Based on this, we devise a selective replay technique called *Semi-replay* to locate the changes in the underlying OS which causes the failed execution.

We are able to monitor almost all the interactions between a program and the execution environment by monitoring the system calls made by a program. For example, all file access, network access and even access to system time are done through system calls. Building our technique on system call, not only can we detect

semantic changes in system call implementations, but also detect other environment changes reflected by system calls.

Given a program P and an input t , let E' be an environment where P fails to execute with t . Let the executed system call sequence of P in E' be

$$\langle \text{syscall}'_1, \text{syscall}'_2, \dots, \text{syscall}'_{n-1}, \text{syscall}'_n \rangle$$

Algorithm 1 returns a number r such that `syscall'r` causes the failure of P in E' . The following properties are satisfied by `syscall'r`:

1. Program P fails when $\langle \text{syscall}'_1, \text{syscall}'_2, \dots, \text{syscall}'_{r-1} \rangle$ is replayed in *Semi-replay*.
2. Program P passes when $\langle \text{syscall}'_1, \text{syscall}'_2, \dots, \text{syscall}'_r \rangle$ is replayed in *Semi-replay*.

In *Semi-replay*, system call dependencies cause unexpected program failure if they are not handled properly. In particular, suppose an executed system call `syscallj` is dependent on a replayed system call `syscalli`. System call `syscallj` definitely fails if `syscalli` is only replayed but not executed. In this case, we need to handle the replay of `syscalli` differently. The following sub-sections defines and illustrates system call dependencies and then presents the *Semi-replay* technique through the *Locate_cause* algorithm in Algorithm 1.

3.1 System call dependencies

We say that a system call `syscallj` is dynamically dependent on `syscalli` when information flow from `syscalli` to `syscallj` and denote as `syscallj → syscalli`. More formally, `syscallj → syscalli` iff. `syscallj` is transitively dependent on `syscalli` via a chain of dynamic data dependencies. For example `read → open`, which means the `read` system call is dependent on the `open` system call as the file descriptor returned by the `read` system call is used by the `open` system call.

System call dependencies lead to unexpected program behavior during *Semi-replay* if dependent system calls are not being replayed or executed together. Figure 3 illustrates dependencies among system calls. Suppose we want to replay the first k system calls and execute the rest in E' . Obviously, dependent system calls that all belong to the “replay” or “execute” half do not cause any unexpected program behavior. However, if there is a system call `syscallj` in the “execute” half which is dependent on a system call `syscalli` in the “replay” half, then `syscallj` will fail because `syscalli` is only replayed but not executed. The problem is caused by the inconsistent assumption on system state by `syscallj`. The system call

$syscall_j$ assumes the existence of some kernel state modified by $syscall_i$. However, the modification of kernel state by $syscall_i$ is not regenerated when $syscall_i$ is replayed.

Algorithm 1 *Locate_cause*

```

1: INPUT:
2:  $R$  // recorded syscall sequence in a successful run
3:  $P$  // program being debugged
4:  $E'$  // an environment that  $P$  fails in
5:
6:  $start = 0$ 
7:  $end = n$ 
8: while  $end - start > 1$  do
9:    $ret = Semi-replay(P, E', R, (end + start)/2)$ 
10:  if  $ret$  then
11:     $end = (end + start)/2$ 
12:  else
13:     $start = (end + start)/2$ 
14:  end if
15: end while
16: return  $end$ 
17:
18: procedure  $Semi-replay(P, E', R, k)$ 
19:  let  $R$  be  $\langle syscall_1, syscall_2, \dots, syscall_n \rangle$ 
20:   $i = 0$ 
21:  execute  $P$  in  $E'$ 
22:  while a syscall is encountered in the execution do
23:     $i = i + 1$ 
24:    if  $i \leq k$  then
25:      if  $check\_dep(i, k, R)$  then
26:        execute the syscall
27:        overwrite syscall results using  $syscall_i$ 
28:      else
29:        replay the syscall using  $syscall_i$ 
30:      end if
31:    else
32:      execute the  $syscall$  in  $E'$ 
33:    end if
34:  end while
35:  if execution passes then
36:    return  $true$ 
37:  else
38:    return  $false$ 
39:  end if
40: end procedure
41:
42: procedure  $check\_dep(i, k, R)$ 
43:  let  $R$  be  $\langle syscall_1, syscall_2, \dots, syscall_n \rangle$ 
44:  for all  $j$  from  $k + 1$  to  $n$  do
45:    if  $syscall_j$  is dependent on  $syscall_i$  then
46:      return  $true$ 
47:    end if
48:  end for
49:  return  $false$ 
50: end procedure

```

3.2 Algorithms

In this section, we explain in details the *Locate_cause* algorithm. Given a recorded system call sequence generated by the successful execution of P in E , the algorithm returns the index of a system call in the sequence which is the root cause of the failed execution of P in E' . The *Locate_cause* algorithm resembles

a binary search algorithm. The main part of the algorithm is the *Semi-replay* procedure. The basic idea of the algorithm is as follows. The algorithm first takes the mid-point of the sequence and tries to replay the first half of the system call sequence (line 25-30) and execute the remaining half (line 32). We refer to the first half as the *R-half* and the second half as the *E-half* and the index defining these two halves as k . There are two possibilities:

1. *The execution passes* (line 36): The *Semi-replay* procedure returns *true*, which means that the root cause of the failure of P in E' must be located in the *R-half*. This is because executing the *E-half* does not cause the failure. In this case, the *Locate_cause* algorithm minimizes the the search for the problematic system call in the *R-half* by dividing this half further into two halves, moving the index k upwards to the mid-point of the *R-half* and iteratively calling the *Semi-replay* procedure. As k is now moved to the mid-point of the *R-half*, the *Semi-replay* procedure will replay only the first half of *R-half* and execute the second half of *R-half* plus *E-half*.
2. *The execution fails* (line 38): The *Semi-replay* procedure returns *false* which means that the root cause of the failure of P in E' must be located in the *E-half*. This is because when we execute the *E-half* of system calls in E' , the execution fails. In this case, the *Locate_cause* algorithm minimizes the search for the problematic system call in the *E-half* by dividing the *E-half* further into two halves, moving the index k downwards to the mid-point of the *E-half* and then iteratively calling the *Semi-replay* procedure. As k is now moved to the mid-point of the *E-half*, the *Semi-replay* procedure will replay the *R-half* plus the first half of *E-half* and execute the second half of *E-half*.

This process continues until a single system call is located which causes a failed execution. One key procedure in this algorithm is $check_dep(i, k, R)$. This is very important as executing a sequence of system calls involves handling dependency between system calls. For each system call $syscall_i$ being replayed, which means in the *R-half*, this procedure searches in the *E-half* for all the system calls that are dependent on the $syscall_i$. If there are some system calls that are dependent on $syscall_i$, then $syscall_i$ need to be executed as well to regenerate the side-effects and returned values to be used later by the dependent system calls.

Another important feature of the *Semi-replay* algorithm is that a prefix of system call sequence is always replayed and the suffix is executed. This feature is based on the fact that it is almost impossible to execute a prefix of a system call sequence and replay the suffix. This is because executing a prefix of a system call sequence could possibly drive the program execution to a totally different path from the recorded execution. Consequently, the recorded suffix does not match the executed prefix; thus making replaying of the suffix impossible.

4. APPLICATION OF OUR APPROACH

We design our technique in such a way that we only need to record one successful execution for one program input. This design choice is critical for some application scenarios of our technique. As we noted that, when a program fails to run in one OS environment, sometimes we do not have a reference environment where the program runs successfully. Our technique can be used in any of the following scenarios:

- A system administrator deployed a software on large number of machines, the software fails to execute on some machines.

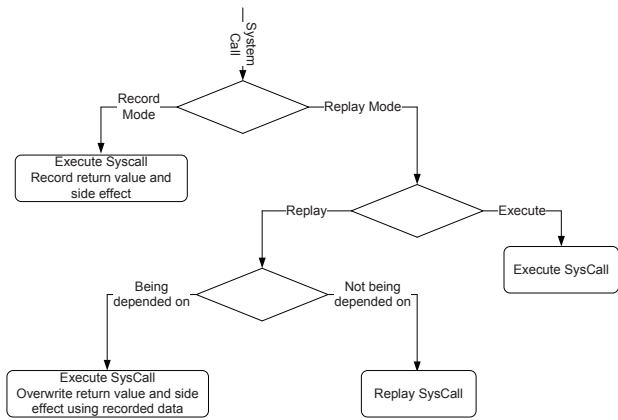


Figure 4: Flow of handling system calls in our implementation

- A software works in an old legacy system, but fails to work after system upgrade.
- An end-user fails to execute a software obtained from a software distribution (possibly from the Internet).

In the last case, software manufacturers could also distribute the recorded system call data from successful executions together with the distribution of the software. This allows the end-user to quickly locate and solve the problem when the software fails to execute. Note that our technique compares the system call sequences for two execution environments and these system call sequences are collected for the same program input. Hence if the software manufacturer distributes some sample inputs as well as the recorded system call sequences for these sample inputs, the end-user can compare the system call sequences obtained at his end for the same inputs.

5. IMPLEMENTATION

The proposed approach is implemented as a plug-in of Valgrind [24]. Valgrind [24] is a widely used dynamic binary analysis tool. To capture the communication between an application and the underlying execution environment, we leverage the Valgrind API to instrument the application’s binaries. The instrumentation is designed to capture all system calls between an application and the OS. For each system call, the return values and the side-effects are recorded.

The process of handling systems call is shown in Figure 4. Our plug-in provides two modes namely record mode and replay mode. In record mode, whenever a system call is encountered, the system call is executed directly by the OS kernel. When the execution of the system is completed, we record $\langle num, retval, side_effect \rangle$ into our record file. The num and $retval$ are the system call number and the return value of this system call respectively. The $side_effect$ contains other changes to the memory space of the executed program. For example, a `read` stores the read data into the buffer pointed to by the parameter of the system call. In this case, the content in the buffer is recorded after the system call’s execution completes.

In the the replay mode, the plug-in takes as input a sequence of recorded system calls and only replays a prefix of the system call sequence as mentioned in Algorithm 1. When a system call $syscall_i$ is replayed, we first check whether $syscall_i$ is being dependent on by any other system calls that are to be *executed*. If yes,

then the system call $syscall_i$ is both executed and replayed; that is we first execute the system to make the kernel state consistent and then use the recorded data to overwrite the return values and side-effects of the system call. If the system call $syscall_i$ is not being dependent on by any system calls that are to be *executed*, then the system call $syscall_i$ is only replayed with the recorded data without getting executed in the kernel.

There are some system calls that we did not fully implement. For these un-handled system calls, we simply execute them in both record mode and replay mode. Our techniques cannot deal with failures caused by these un-handled system calls.

6. EXPERIMENTS

In this section, we report our experience in using the proposed technique to locate failure-inducing environment changes in real-life case studies. We present the results of our experiments in evaluating the effectiveness of our method.

Given a recorded system call sequence with length N , our technique only takes $\log(N)$ executions to finish. In each following case study, our technique takes less than one minute to find the problematic system call. The record files are less than 10MB in all three cases.

6.1 Experience with MPD

Music Player Daemon (MPD) [13] is a server-side application which allows remote access for playing music. MPD comes with a client program, which is a console based jukebox commander. Clients may communicate with the server remotely over an intranet or over the Internet. To start MPD, a folder named “mpd” must be created under it `/var/run/`. In this folder, files including the `pid` file will be stored.

We run MPD in two different OS environments: (E) Ubuntu 9.04 with X service and (E') Ubuntu 8.04 in shell mode without X service. When running MPD in E , we were able to start and use the application. However, when running in E' , we were unable to start the application and encountered the following error message:

```
Starting Music Player Daemon:
could not open pid_file "/var/run/mpd/pid"
for writing:
No such file or directory failed.
```

This program failure can be manually fixed by creating the folder and setting proper permission so that the `pid_file` can be created and MPD can be started. However, this does not fix the problem permanently because the bug occurs everytime the system is rebooted and it is unclear what is the root cause of this bug. Without using any debugging tool, it is difficult for the user to diagnose the root cause of this problem.

We applied the *Locate_cause* algorithm in Section 3.2 to locate the root cause of this problem. We collected the system call sequence when executing MPD in E . The sequence consists of 1038 system calls. We then run *Locate_cause* on the sequence of system calls, it returns the 7th system call as the root cause of this problem, which is: `open`. This system call suggests that MPD assumes the existence of the path `/var/run/` so the application straight away creates a `pid` file under the directory `/var/run/mpd/` everytime the system is rebooted. However, under the execution environment E' , the path `/var/run/` is not automatically mounted which caused the failed execution of MPD.

6.2 Experience with VSFTPD

VSFTPD is an FTP server daemon that runs on most current Unix-based operating systems. To allow anonymous user access in

VSFTPD, one needs to create an anonymous FTP user and set appropriate permissions. According to the VSFTPD document, there are two ways to enable anonymous user access in the VSFTPD configuration file `vsftpd.conf`:

1. set `anonymous_enable=YES` in `vsftpd.conf`
2. create an user list file named `vsftpd.user_list` containing the “anonymous”, set `user_list_enable=YES` in `vsftpd.conf`

However, to use the second way, the `local_enable` has to be set to `YES` in `vsftpd.conf`.

In both environments E and E' , Ubuntu version 8.04 is used. In the reference environment E , `anonymous_enable=YES` is set in `vsftpd.conf` to allow anonymous users. In the environment E' where VSFTPD fails, `anonymous_enable` is set to `NO` and `user_list_enable` is set to `YES` in `vsftpd.conf`. However, since `local_enable` is not set to `YES` in `vsftpd.conf`, anonymous users are not allowed in environment E' . An anonymous access attempt in E' gets the following error message:

```
530 Permission denied. Login failed
```

This error message is too general and does not help much in figuring out the real cause of this problem.

We use our technique to locate the cause of this denied anonymous user access in E' . In this case study, the OS versions are the same in the two environments. The environment changes lie in the configuration used to enable anonymous user. We run the application in the successful environment E and used our tool to record the executed sequence of system calls. There were 26 system calls altogether that have been recorded. We then applied the *Locate_cause* algorithm in Section 3.2 and located the 2nd system call `read`, which caused the failed login of anonymous users. We further investigated from this system call. We found that it was the content of the file read by the system call `read` that caused the failed execution of the program. We checked the file read by the system call and found out that it was the configuration file `vsftpd.conf` which was read. By comparing the content of the two configuration files, one in E and one in E' , we were able to figure out the difference in the configuration file that caused of the failed login with the anonymous user.

6.3 Experience with Miniweb

Miniweb [12] is an efficient light-weight web sever. Miniweb listens to a certain port of the OS for incoming HTTP requests. Either the port is specified by a command line option to Miniweb or the default port 80 is used. When the port used by Miniweb is already occupied, Miniweb fails to start with the following error message, which does not provide sufficient clue to locate the root cause of this problem:

```
Error starting instance #0
Failed to launch miniweb
Shutting down instance 0
```

In our experiment, we run Miniweb in two different environments: (E) Ubuntu 8.04 with the port used by Miniweb not occupied. (E') Ubuntu 8.04 with the port used by Miniweb occupied.

Using our technique, we record the system call sequences when Miniweb is executed in E . The successful execution of Miniweb in E returns a sequence of 16 system calls. Note that as Miniweb is a web-server, it will keep on running and generating a lot more system calls. Therefore, to successfully conduct this case study, we have to manually kill the process after it has started successfully.

We then used this recorded sequence to locate the cause of failing to start Miniweb in E' . To tell whether Miniweb runs successfully, we check whether Miniweb is still running after it is started for a certain time (say 5 seconds). If Miniweb is still running after 5 seconds, we terminate Miniweb and deem the execution to be successful. Otherwise, Miniweb must have failed to start. Applying the Algorithm 1 in Section 3.2, we located a system call `socketcall` as the cause of the failure of Miniweb in E' . This `socketcall` system call is at the 7th place in the recorded sequence. The format of the `socketcall` system call is as follows:

```
int socketcall(int call, unsigned long *args);
```

where `call` determines which socket function to invoke, `args` is a pointer pointing to a block containing the actual arguments which are passed through to the appropriate `call`. For the `socketcall` we located, parameter `call` indicated that it is a *bind* operation. The `args` contained information about which port to bind and some other information. In the recorded (successful) execution, the `socketcall` system call returns 0 indicating that the operation is successful. However, in the failed execution, `socketcall` returned the error code 98, which corresponds to `EADDRINUSE` in `error.h`. `EADDRINUSE` means that the “address is already in used”, which is root cause of the failed execution of Miniweb in environment E' .

7. LIMITATIONS

In this paper, we assume that the program only use static libraries. The applicability of our technique is greatly reduced by this restriction. Allowing dynamic libraries give rise to a lot of challenges especially when different implementations of a dynamic libraries are used in different environments. We plan to look into these challenges in future.

We only focused on single-thread deterministic program in this paper. Therefore, if a bug is caused by non-deterministic signal and interrupt, our technique is not able to handle it.

The result of our technique is dependent on the closeness between the faulty environment and the reference environment. If the faulty environment is intended to be configured differently from the reference environment, the result from our technique may not be very useful.

Our debugging technique works at the system call interface layer. Therefore, we can only provide some suspicious system calls as the result of our technique. In some situations, the located system call does not provide enough detailed information to help debugging. For example, if a program uses one `read` system call to read a large faulty configuration file, our technique can only pinpoint this `read` system call. In this case, the user needs to look into the large configuration file to figure out the root cause of the program failure.

8. RELATED WORK

There have been a large collection of research on record-replay techniques. However, most of the existing record-replay techniques are not suitable for our debugging task. Most of the techniques replay the complete trace or part of the trace in the same environment [4, 6, 7, 10, 14, 16, 18–20].

Orso et al. [16] proposed a selective record-replay technique, which specifically target components’ interactions. A subsystem of interest must be manually selected. The interactions between the selected subsystems and the rest of the application will be captured and replayed in isolation. This approach only captures essential information which is relevant to the execution thus makes it more efficient.

Burger and Zeller [2] have developed the Jinsi tool which is capable of capturing and replaying the inter/intra-components interactions in a system. More importantly, Jinsi is able to isolate a failure-inducing sequence of calls. This work is an extension of an earlier work [15] by Orso et al. The work from Orso et al. [15] isolates relevant interactions between the observed component and other layers of a complex application. The work combines record and replay technique with delta debugging [26] to isolate relevant events.

Kwon and Su [9] realize the problem of unsafe component loadings and propose a dynamic analysis technique to solve this problem. This approach consists of two phases: (i) dynamic binary instrumentation to capture a program sequence of events related to component loading and (ii) offline profile analysis to extract each component loading from the captured information and detect defects in the resolution of a target component and its dependency. We capture a sequence of system calls which specifically describe the interactions of a program and the underlying environment.

Recently, Clause and Orso [5] propose a record and replay technique which enables recording in one execution environment and replaying in another execution environment. When replaying a program, the proposed technique ensures that the program fails in the same way as in the recording environment. The technique intercepts all the software interactions with the environment through the OS and produces an execution recording that consists of an event log and a set of environment data. As such, the execution can be replayed without the need for the original execution environment. So far, no technique handles the replay and partially execute in different environments which takes into consideration the combination of two different environments.

From a high level view point, a system can be viewed as a coherent set of components interacting with one another. In this sense, our work is related to existing works which are based on component interaction analysis. Component-based development has become an important approach to building more flexible and reusable application. At the same time, Component-based development also presents new challenges such as component's compatibility. Behavior capture and test (BCT) has been proposed by Mariani et al [11] to automatically detect COTS component incompatibility through dynamic analysis. BCT builds for each component (i) an I/O model describing the relations between the values the components exchange and (ii) an interaction models representing the sequences of interaction triggered by invoking the component's services. The models describe the component's behaviors in different contexts. Subsequently, when a component is reused in new contexts, their behaviors will be compared with the behavioral models built from former executions to identify new behaviors. If we look at applications as components and execution environments as contexts, our work is quite close to the BCT approach. However, our work does not "compare" but selectively execute some interactions in the new environment to identify failure-inducing interactions. Moreover, our work aims to locate failure-inducing changes in the environment while BCT tries to identify new behaviors.

Debugging is a tedious and time-consuming task. Most of the existing debugging techniques target bugs in the program itself. More specifically, given a program and an observable error for a given failing program input, these techniques try to locate the root cause of the observable error in the program source code [17, 25, 26]. However, Holmes et al. [8] have pointed out that a developer can evolve an program's behaviors not only by changing the program's source code but also by altering the execution environment. The authors have also introduced a program partitioning approach for categorizing program changes with regards to the program's be-

haviors. As such, the approach is able to tell the developers which groups of changes deserve deeper developer attention. The approach, however, does not focus on locating changes in the environment that are related to an observable error. Attariyan et al [1] proposed a technique based on information flow analysis to locate the root cause of configuration errors, which is only one type of OS induced errors. In term of environment induced errors, this work is quite close to ours. The approach instruments application binaries to monitor the information flow at runtime and uses this information to detect the relationship between erroneous behavior and configuration files. The work from Su et al. [21, 22] also targets at misconfiguration problem. They use speculative execution to examine the effect of configurations and roll-back to earlier configurations when necessary. Different from their work, we leverage a reference execution environment to located the cause of program failure.

Our paper considers the problem of debugging OS induced error in which the program source code does not change but the OS execution environment does. Locating root cause in the context of environment changes is challenging due to the complexity of the execution environment. We have presented a debugging technique that works at the system call interface between an application and the underlying execution environment. By using the record and replay technique, we record the execution of the program under the successful environment and semi-replay it under the failing environment. By *semi-replaying*, which means partially executing, our technique can efficiently locating the system call which causes the failed execution.

9. CONCLUSION

In this paper, we have presented the *Semi-replay* method for locating the failure-inducing environment changes. Our approach takes in a program and two different OS environments where the execution of the program fails in one environment and passes in another. The proposed approach then locates a system call which is able to explain the failed execution of the program. Our approach captures the system call interface between an application and the underlying OS environment generated during the successful execution. The recorded system call sequence is then used to debug the failed execution of the application under another faulty OS environment. We have applied the proposed approach in three real-life case studies which give evidences to show the utility of our technique in debugging real bugs. The system call located by our approach can be used to easily identify the root cause of an error.

Acknowledgements. This work was partially supported by a Ministry of Education research grant MOE2010-T2-2-073 (R-252-000-456-112 and R-252-100-456-112), as well as a Defence Innovative Research Programme (DIRP) grant (R-252-000-393-422) from Defence Research and Technology Office (DRTech).

10. REFERENCES

- [1] Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation, OSDI'10*, pages 1–11, Berkeley, CA, USA, 2010. USENIX Association.
- [2] Martin Burger and Andreas Zeller. Replaying and isolating failing multi-object interactions. In *Proceedings of the 2008 international workshop on dynamic analysis: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*, WODA '08, pages 71–77, New York, NY, USA, 2008. ACM.

- [3] Subhachandra Chandra and Peter M. Chen. Whither generic recovery from application faults? a fault study using open-source software. *Dependable Systems and Networks, International Conference on*, 0:97, 2000.
- [4] Jong-Deok Choi and Harini Srinivasan. Deterministic replay of java multithreaded applications. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, SPDT '98, pages 48–59, New York, NY, USA, 1998. ACM.
- [5] James Clause and Alessandro Orso. A technique for enabling and supporting debugging of field failures. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 261–270, Washington, DC, USA, 2007. IEEE Computer Society.
- [6] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *In Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, pages 211–224, 2002.
- [7] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans, and Kaashoek Zheng Zhang. R2: An application-level kernel for record and replay. In *OSDI*, 2008.
- [8] Reid Holmes and David Notkin. Identifying program, test, and environmental changes that affect behaviour. In *ICSE*, Honolulu, Hawaii, May 2011.
- [9] Taeho Kwon and Zhendong Su. Automatic detection of unsafe component loadings. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 107–118, New York, NY, USA, 2010. ACM.
- [10] Oren Laadan, Nicolas Viennot, and Jason Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '10, pages 155–166, New York, NY, USA, 2010. ACM.
- [11] L. Mariani and M. Pezze. Dynamic detection of cots component incompatibility. *IEEE Software*, 24(5):76–85, 2007.
- [12] Miniweb. Mini web server. <http://miniweb.sourceforge.net/>, March 2011.
- [13] MPD. Music player daemon. http://mpd.wikia.com/wiki/Music_Player_Daemon_Wiki, March 2011.
- [14] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: continuously recording program execution for deterministic replay debugging. In *Proc. 32nd Int. Symp. Computer Architecture ISCA '05*, pages 284–295, 2005.
- [15] Alessandro Orso, Shrinivas Joshi, Martin Burger, and Andreas Zeller. Isolating relevant component interactions with jinsi. In *Proceedings of the 2006 international workshop on Dynamic systems analysis*, WODA '06, pages 3–10, New York, NY, USA, 2006. ACM.
- [16] Alessandro Orso and Bryan Kennedy. Selective capture and replay of program executions. In *Proceedings of the third international workshop on Dynamic analysis*, WODA '05, pages 1–7, New York, NY, USA, 2005. ACM.
- [17] Dawei Qi, Abhik Roychoudhury, Zhenkai Liang, and Kapil Vaswani. Darwin: an approach for debugging evolving programs. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 33–42, New York, NY, USA, 2009. ACM.
- [18] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: treating bugs as allergies—a safe method to survive software failures. *SIGOPS Oper. Syst. Rev.*, 39:235–248, October 2005.
- [19] Yasushi Saito. Jockey: a user-space library for record-replay debugging. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, AADEBUG'05, pages 69–76, New York, NY, USA, 2005. ACM.
- [20] Sudarshan M. Srinivasan, Srikanth Kandula, Srikanth K, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *In USENIX Annual Technical Conference, General Track*, pages 29–44, 2004.
- [21] Y.Y. Su, M. Attariyan, and J. Flinn. Autobash: improving configuration management with operating system causality analysis. *ACM SIGOPS Operating Systems Review*, 41(6):237–250, 2007.
- [22] Y.Y. Su and J. Flinn. Automatically generating predicates and solutions for configuration troubleshooting. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, pages 17–17. USENIX Association, 2009.
- [23] Ubuntu. Ubuntu bugs. <https://bugs.launchpad.net/ubuntu>, February 2011.
- [24] Valgrind. Valgrind. <http://valgrind.org/>, November 2010.
- [25] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In *ESEC/FSE'99*, volume 1687 of *Lecture Notes in Computer Science*, pages 253–267. Springer Berlin / Heidelberg, 1999. 10.1007/3-540-48166-4_16.
- [26] Andreas Zeller. Isolating cause-effect chains from computer programs. *SIGSOFT Softw. Eng. Notes*, 27:1–10, November 2002.