A Technique for Enabling and Supporting Debugging of Field Failures

James Clause and Alessandro Orso
College of Computing
Georgia Institute of Technology
{clause, orso}@cc.gatech.edu

Abstract

It is difficult to fully assess the quality of software in-house, outside the actual time and context in which it will execute after deployment. As a result, it is common for software to manifest field failures, failures that occur on user machines due to untested behavior. Field failures are typically difficult to recreate and investigate on developer platforms, and existing techniques based on crash reporting provide only limited support for this task. In this paper, we present a technique for recording, reproducing, and minimizing failing executions that enables and supports in-house debugging of field failures. We also present a tool that implements our technique and an empirical study that evaluates the technique on a widely used e-mail client.

1. Introduction

Quality-assurance activities, such as software testing and analysis, are notoriously difficult, expensive, and time-consuming. As a result, software products are often released with faults or missing functionality. In fact, real-world examples of field failures experienced by users because of untested behaviors (*e.g.*, due to unforeseen usages), are countless. When field failures occur, it is important for developers to be able to recreate and investigate them in-house. This pressing need is demonstrated by the emergence of several crash-reporting systems, such as Microsoft's error reporting systems [13] and Apple's Crash Reporter [1]. Although these techniques represent a first important step in addressing the limitations of purely in-house approaches to quality assurance, they work on limited data (typically, a snapshot of the execution state) and can at best identify correlations between a crash report and data on other known failures.

In this paper, we present a novel technique for reproducing and investigating field failures that addresses the limitations of existing approaches. Our technique works in three phases, intuitively illustrated by the scenario in Figure 1. In the *recording phase*, while users run the software, the technique intercepts and logs the interactions between application and environment and records portions of the environment that are relevant to these interactions. If the execution terminates with a failure, the produced execution recording is stored for later investigation. In the *minimization phase*, using free cycles on the user machines, the technique replays the recorded failing executions with the goal of automatically eliminating parts of the executions that are not relevant to the failure. In the *replay and debugging phase*, developers can use the technique to replay the minimized failing executions and investigate the cause of the failures (*e.g.*, within a debugger). Being able to replay and debug real field failures can give developers unprecedented insight into the behavior of their software after deployment and opportunities to improve the quality of their software in ways that were not possible before.

To evaluate our technique, we implemented it in a prototype tool, called ADDA (Automated Debugging of Deployed Applications), and used the tool to perform an empirical study. The study was performed on PINE [19], a widely-used e-mail client, and involved the investigation of failures caused by two real faults in PINE. The results of the study are promising. Our technique was able to (1) record all executions of PINE (and two other subjects) with a low time and space overhead, (2) completely replay all recorded executions, and (3) perform automated minimization of failing executions and obtain shorter executions that manifested the same failures as the original executions. Moreover, we were able to replay the minimized executions within a debugger, which shows that they could have actually been used to investigate the failures.

The contributions of this paper are:

- A novel technique for recording and later replaying executions of deployed programs.
- An approach for minimizing failing executions and generating shorter executions that fail for the same reasons.
- A prototype tool that implements our technique.
- An empirical study that shows the feasibility and effectiveness of the approach.

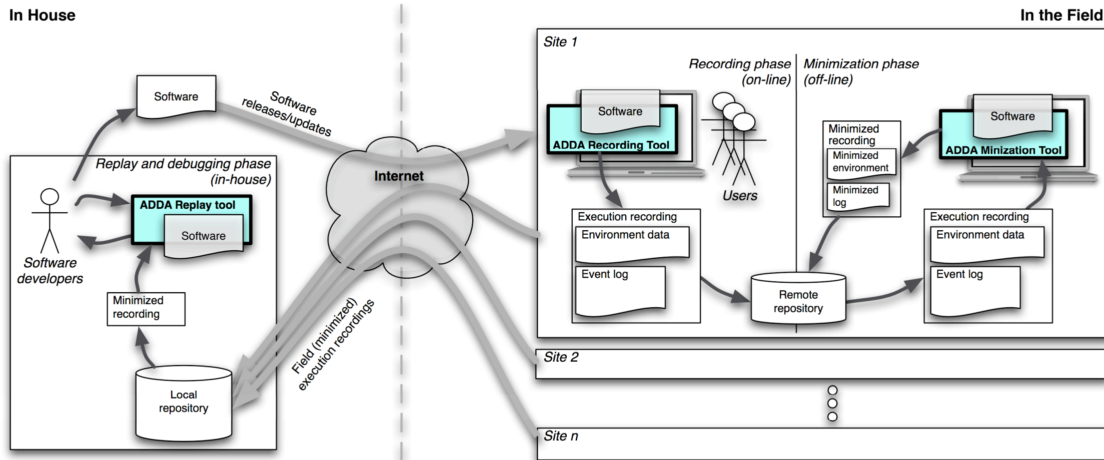

Figure 1. An intuitive scenario of usage of our technique.

2. Related Work

This work encompasses several areas. We present the most related efforts, organized into categories.

Record and replay. The techniques most closely related to our approach are those that record and replay executions for testing or debugging. Some of these techniques perform deterministic replay debugging, that is, replay of executions that led to a crash (*e.g.*, [3, 9, 14, 15, 21]). Various commercial and research tools record and replay user interactions with a software product for regression testing (*e.g.*, [11, 22, 23]). Unlike our approach, most of these techniques are designed to be used during in-house testing or debugging. The overhead they impose (on space, time, or infrastructure required for recording) is reasonable for their intended use, but would make them impractical for use on deployed software. The few record and replay techniques that are either defined to operate in the field (*e.g.*, BugNet [14]) or may be efficient enough to be used in the field (*e.g.*, [9, 21]) require a specialized operating-system or hardware support, which considerably limits their applicability in the short term.

More recently, several researchers presented techniques for recording executions of Java subsystems [5, 16, 17, 20]. These approaches are heavily based on Java's characteristics and target the recording of subsystems only. It would be difficult to adapt them to work in a more general context. Moreover, most of these approaches are defined to be used in-house, for regression testing, and impose an overhead that would prevent their use on deployed software.

Remote data collection. Pavlopoulou and Young propose residual testing [18], which continuously monitors test obligations not fulfilled in the development environment. The Expectation-Driven Event Monitoring project [4, 7] collects program-usage data to compare against an interaction model. Elbaum and Diep [6] investigate ways to efficiently collect field data (specifically, coverage data) that could be used for improving the representativeness of test suites. The Cooperative Bug Isolation project [2, 10] uses statistical techniques to sample and analyze execution data collected from real users and perform fault localization. These efforts are all related to our research, in that they collect information on-line and use it to augment in-house techniques. However, none of these projects tries to record and reproduce field executions for debugging.

Debugging of deployed software. When a deployed program fails, most modern operating systems generate a crash report that the user can send to the developers (*e.g.*, [1, 13]). Crash reports typically contain a snapshot of the execution's final state, which can be used to investigate the causes of the crash. These systems address some of the limitations of purely in-house approaches, but they are still limited. When developers receive a report, they have little information about the failure and can at best look for correlations with other known failures. Existing techniques that leverage field data to do fault localization for deployed software (*e.g.*, [8, 10]) can improve the situation by identifying parts of the code that are likely to be faulty, but still do not let developers reproduce and observe failing executions.

3. Technique and Tool

Our overall goal is to record, minimize, and replay (failing) executions of deployed software to support debugging of field failures. Existing approaches for record and replay have three main limitations with respect to this goal. The first two, efficiency problems and need for specialized hardware, were discussed in Section 2. The third limitation is that none of these approaches is amenable to minimization of the recorded executions. Most existing techniques record executions in terms of low-level events, such as memory accesses, and then reproduce these events exactly during replay. Recording executions at a low-level of detail allows

for capturing all sources of non-determinism without having to worry about types and sources of inputs, and works well for replaying complete executions. When minimizing executions, however, the greatest challenge is to maintain consistency while eliminating parts of the execution. Low-level event logs contain events that are highly dependent on one another. Due to these dependencies, removing even a single event from a log is often enough to make the rest of the execution inconsistent and, thus, unusable. We and other researchers experienced this issue while trying to minimize recorded executions (*e.g.*, [16, 20]).

In the next sections, we discuss how we addressed the above issues to develop a record and replay technique that

- is efficient enough to be used on deployed programs,
- does not require any specialized hardware,
- can replay executions in a sandbox, and
- is flexible enough to support minimization of executions.

3.1. General Approach

This section provides a high-level description of our technique. Low-level aspects of the approach and implementation details are presented in Section 3.2. One novelty of our approach is that it steps away from replay techniques that drive a program (almost) statement by statement based on a recorded event log. We use an execution log when replaying entire executions, but discard it almost completely otherwise. Intuitively, our technique treats executions as sequences of interactions between software and environment.

While the software interacts with the environment through the Operating System (OS), by accessing different I/O streams (*e.g.*, keyboard input, network, and files), our technique intercepts such interactions and produces an *execution recording* that consists of an event log and a set of environment data. The *event log* records relevant information on the observed interactions. The *environment data* consist of a dump of the input streams used by the software (*stream dumps*) and a copy of the files accessed by the software (*environment files*). Stream dumps also contain metadata that provide grouping and timing information for the data in the streams, used during minimization. Using the information in event logs and environment data, our technique can replay recorded executions in two ways. The first way is to perform a *complete replay*. In this case, our technique replays the execution as it was recorded, by enforcing the sequence of events in the log and providing the program with data from the stream dumps at the appropriate times. The second way is to perform a *minimized replay* by trying to eliminate as much data as possible from the stream dumps, discarding the event log, and letting the program run on this set of reduced inputs. In the case of a failing user execution, a minimized replay that still fails in the same way as the original execution would provide developers with a way to not only debug the field failure, but also to do it on an execution in which some of the parts that are irrelevant for the failure have been eliminated.

We are aware of the issues of privacy involved with the proposed approach. At this stage of the research, however, we think it is more important to focus on the technical aspects of the problem and show the feasibility of our approach. Moreover, there are several ways in which these issues could be addressed. One possibility is to show users the collected data and let them decide whether to send the minimized executions. Other possibilities include data-sanitization techniques and incentive mechanisms (*e.g.*, free updates for participating users).

3.1.1. Recording Executions

When recording executions, our technique intercepts three kinds of actions performed by the software: POLL, FILE, and PULL. Each action refers to a specific input stream (*current stream*, hereafter) or file. For each type of action, the technique updates the event log and the environment data, by either adding files to the set of environment files or modifying the stream dump for the current stream.

POLL actions are performed by the software to check for availability of data on an input stream. For each POLL action, our technique adds to the event log a *POLL event* with the following information: (1) a unique id for the current stream; (2) the outcome of the POLL action (OK if data were available on the specified stream, or NOK otherwise); and (3) the amount of time elapsed before the software obtained a response to the POLL action. If the outcome of the POLL is positive, the technique also adds the amount of time elapsed, in the form of metadata, to the stream dump associated with the current stream.

FILE actions are interactions between the software and a filesystem. For each FILE action performed by the software, our technique checks to see if the action is the first reference to a specific file or directory.[1] If so, it copies the file into the set of environment data and stores the file's properties (*e.g.*, owner, permissions, time stamps). The technique then increments a FILE-action sequence number and adds to the event log a *FILE event* that specifies (1) the current sequence number and (2) the name of the file being accessed. For subsequent accesses to the same file, the technique checks whether the file has been externally modified (by checking its modification time against the file's stored properties). If not, the technique increments the FILE-action sequence number, updates the file properties if needed, and does not log any event. Otherwise, a new version of the file is stored in the environment data and a new entry for the file is created in the event log. A numeric suffix distinguishes different versions of a file.

[1] For simplicity, in the rest of the paper, we use the term file to refer to both files and directories, unless otherwise indicated.

```
------------ Event log ------------
PARAMS -I i
FILE 01 /etc/nsswitch.conf.1
FILE 02 /home/clause/.pinerc.1
...
POLL STREAM01 NOK 8000
POLL STREAM01 OK 5680
PULL STREAM01 1
POLL STREAM01 OK 1986
PULL STREAM01 3
FILE 41 /var/spool/mail/clause.2
FILE 63 /var/spool/mail/clause.3

----- Environment data:  Files -----
/etc/nsswitch.conf.1
/home/clause/.pinerc.1
/var/spool/mail/clause.1
/var/spool/mail/clause.2
/var/spool/mail/clause.3

---- Environment data:  STREAM01 ---
{time:5680}|m|{time:1986}|sko|{separator}...
```

Figure 2. Example of recorded execution.

PULL actions are atomic reads of some amount of data from a stream. For each PULL action performed by the software, our technique first adds to the event log a *PULL event* that contains (1) a unique id for the current stream and (2) the amount of data being read in bytes. Then, the technique groups the data being read and appends the group to the appropriate stream dump, creating a new dump if one does not already exist.

Finally, if the last observed action accessed stream s1, and the new action either refers to a different stream or is a FILE action, the technique adds, as metadata, a separator marker to the stream dump for s1, in its current position.

Figure 2 shows an example of execution log and environment data generated by our recording technique. The first log entry contains the command-line parameters used for the software. The following two entries are FILE events that indicate a first access to Version 1 of two different files. For each entry, there is a corresponding file in the set of environment files (restored during replay). The first POLL event in the log is an unsuccessful polling of the input stream with id STREAM01. The third attribute of the event indicates that the POLL waited for eight second before receiving a negative response. Conversely, the next POLL event received a positive response after 5,680 milliseconds, as shown both in the event log and in the metadata associated with the stream dump (shown within brackets and in boldface). After that, the program read one byte from stream STREAM01 (character “m” in the corresponding stream dump). The situation is analogous for the next POLL and PULL entries, except that three characters are read in this case, and the three characters appear thus as a group in the stream dump. Because the program then switched from an access to STREAM01 to a FILE action, the metadata for STREAM01’s dump contain a separation marker in the current position. The two FILE events correspond to the first accesses to Versions 2 and 3 of the user’s mailbox, respectively. The difference in the value of the sequence numbers for the two events indicates that there have been 21 additional FILE actions between the two, but none of them was a first access to a new or externally-modified file.

3.1.2. Replaying Executions

Our technique replays executions in a sandbox, which has two main advantages. First, it does not need the original user environment, which allows for replaying an execution in a different environment than the one where it was recorded. In particular, it allows for replaying in-house executions captured in the field. Second, it eliminates side effects of the replayed executions, which prevents potentially harmful actions performed by the replayed software from occurring (*e.g.*, deletion of files, modification of databases, transmission of files over the network).

In this section, we describe how our technique performs complete replay. Minimized replay is described in the next section. To completely replay an execution, our technique sets the current position in the event log and in the stream dumps to their first entry, initializes the sandbox that will contain the filesystem for the replay, and executes the software using the command-line parameters stored in the event log. Then, it lets the software run while intercepting the same software actions it intercepted during recording (plus output actions that need to be prevented from executing).

When the software performs a **POLL action**, the technique retrieves the outcome of the action and the response time from the current log entry. It then returns the retrieved outcome to the software after the specified response time. For the first POLL event in our example log, for instance, the technique would wait for eight seconds and then return a negative result to the software.

When the software performs a **FILE action**, the technique increments a FILE-action sequence number and then checks whether the current event in the log is a FILE event with a matching sequence number. If so, the technique retrieves the corresponding file version from the environment data and restores it, with the correct attributes, to the sandbox. Otherwise, no file is restored. If the FILE action refers to the file by name (*e.g.*, in a file open), the technique manipulates that parameter of the call so that it refers to the file in the sandbox. For the last FILE action in our example, the technique would copy /var/spool/mail/clause.3 from the environment data, together with its attributes, to {sandbox dir}/var/spool/mail/clause.

When the software performs a **PULL action**, the technique reads from the event log the id of the stream being accessed and the amount of data to be read. It then reads that amount of data from the stream dump corresponding to the stream id and returns the data to the program. For the first and second PULL actions in our example, the technique would return one and three bytes, respectively, from the stream dump corresponding to id STREAM01.

3.1.3. Minimizing Executions

The goal of execution minimization is to transform failing executions into shorter executions that can be used to

efficiently investigate the failure. The way to check whether two executions, a minimized and an original one, fail in the same way may depend on the failure considered and on the context. Section 4.4 discusses how we performed this check in our experiments. Our technique minimizes along two dimensions, time and environment, in four steps.

In *Step 1*, the technique performs *partial fast forward* of executions by eliminating idle times due to unsuccessful POLL actions (*i.e.*, polling calls that returned after a timeout because no data were available). To do this, the technique returns a negative response immediately to all such actions. The result is an execution in which reading data and processing inputs are replayed at their original speed, whereas the time during which the software was idle waiting for input is eliminated. If partial fast forward produces an execution that still manifests the original failure, our technique moves to Step 2. Otherwise, it stops the minimization process and stores the original execution.

In *Step 2*, the technique pushes the time compression further and performs *complete fast forward* by eliminating `all` idle times in the replayed executions. Besides immediately returning a negative response to all unsuccessful POLL actions, the technique immediately returns a positive response to all successful POLL actions as well. With complete fast forward, executions have virtually no idle time. If the time compression changed the outcome of the execution, our technique notifies Step 3 to operate in partial fast forward mode, that is, to use the timing information in the stream dumps instead of disregarding it. Otherwise, the technique performs Step 3 in complete fast forward mode.

Steps 1 and 2 are concerned with minimization along the time dimension and can considerably reduce the duration of a failing execution. Steps 3 and 4, conversely, perform minimization of the environment by trying to minimize files and stream dumps, respectively.

To minimize files, *Step 3* replaces all environment files with files of size zero and then replays the execution. If the execution still manifests the original failure, it means that the content of the files is irrelevant for the failure and can be removed. If the execution does not fail, or fails for different reasons, the technique tries again by zeroing only one half of the files. The process continues until the technique identifies which files, if any, can be eliminated without modifying the outcome of the execution. In our empirical evaluation, this process was able to eliminate a large percentage of files (around 86% on average). For the files left, the technique performs minimization at a finer level of granularity. It tries to eliminate parts of the files, checks whether the original failure still occurs, and backtracks and tries a different alternative otherwise. This second part of Step 3 is meant to be flexible and allow for using a range of minimization algorithms. The rationale for this decision is to enable the use of developer-provided minimization algorithms, which can take into account the structure of the files used by the software and be more effective in reducing their size. If no customized algorithm is provided, we can default to the use of a standard input-minimization algorithm, such as the widely-used delta debugging [24].

The final step of the minimization, *Step 4*, operates similarly to Step 3, but on stream dumps. The minimization is also performed hierarchically. Initially, the technique tries to eliminate parts of the stream dumps that correspond to contiguous actions referring to the same stream. To identify these groups, we use the separator marks added to the stream dumps during recording. Based on our initial experience, it is common to observe large sequences of consecutive accesses to the same source. Our intuition, confirmed by our empirical results, is that these groups of events map to specific high-level operations in the software (or even phases) and can sometimes be eliminated without affecting the overall outcome of the execution.

After performing this high-level minimization, our technique tries to further minimize the stream dumps at the data-group level (*i.e.*, by considering as a unit groups of data that were read atomically by the software in the original execution). This step also allows for using externally-provided minimization algorithms and defaults to a standard input-minimization algorithm if no custom algorithm is specified.

Steps 1 through 4 can be performed several times, until no additional minimization is achieved or until a given time threshold is reached. Additional iterations might expose possible synergies between different steps (*e.g.*, eliminating parts of a stream dump may allow for eliminating a file that was previously necessary).

Replaying Minimized Executions. In the discussion so far, we purposely glossed over technical issues related to the replay of minimized executions. Whereas time-based minimization is still driven by the event log, and thus ensures some degree of control on the execution of the software, input-based minimizations do not rely on the event log. The reason for this difference is that input-based minimizations reduce or even eliminate inputs (files and streams), which can result in a potentially large number of log entries being inconsistent with the actual replayed execution. For example, consider the effect of removing all data from stream `STREAM01` in the log of Figure 2. All successful POLL events for that stream would be inconsistent with the environment, and all PULL events for the stream could not be replayed because there would be no data to read.

This simple example is representative of what would happen with traditional record and replay techniques that are completely based on recording and replaying a log. Our technique uses a log when performing complete (and fast-forward) replay, but can also replay without having to enforce a specific sequence of events. Thanks to its notion of environment and to the way it records files and streams,

our technique can replay in the presence of a minimized environment by (1) still intercepting the usual software actions, and (2) responding to these actions based on the state of the minimized environment instead of the information in the log, as follows.

If the software performs a **POLL action**, the technique returns a negative result if there is no data in the corresponding stream dump. Otherwise, it returns a positive response either right away (complete fast forward mode) or after the timeout specified at the current position in the stream dump (partial fast forward mode). If the software performs a **PULL action**, our technique reads and returns the next group of data in a stream, if any. The situation for a **FILE action** is slightly more complex in the presence of files with multiple versions. In such a situation, without relying on sequence numbers, our technique cannot distinguish between actions that should access a new version of a file and actions that should access the current version. To address this issue, our technique uses a single version of a file throughout a (minimized) execution. In particular, it tries in turn the last, the first, and then the intermediate versions.

Obviously, there are many ways in which an execution could take a wrong path and either not fail or manifest a different failure. However, as our empirical results show, the flexibility of our approach allows for identifying many executions that maintain their behavior of interest (the failure) even when run on a different set of environment data.

3.2. The ADDA Tool

To experiment with our technique, we implemented it in a tool called ADDA (Automated Debugging of Deployed Applications). The first implementation challenge we faced with ADDA was how to intercept interactions between software and environment. As a first attempt at this task, we decided to focus on interactions with the OS and not consider interactions occurring through a windowing system. Also, we decided not to record and replay concurrency-related events, which may prevent ADDA for performing deterministic replay in some cases (but was not a problem for our studies). Because interactions with the OS occur through system-library calls, we considered two alternative approaches: operating at the system-call level or at the C-library level. For our current implementation, we decided to work at the C-library level because instrumenting at the system-call level has several drawbacks. For example, at the system-call level there is no explicit notion of parameter types and number (for function calls). Also, it is difficult to limit the amount of instrumentation code executed because it is not always possible to distinguish relevant from non-relevant calls based on the context. Instrumenting at the C-library level eliminates these and other drawbacks, but limits the applicability of the technique to software written in C or C++. Given the high percentage of applications written using these languages, we believe this to be a minor limitation.

A second challenge was the mapping of C-library functions to POLL, FILE, and PULL actions. (Note that ADDA also intercepts additional function calls, such as `rand`, to account for some sources of non-determinism in complete replay. During minimized replay, these calls are not intercepted.) For many functions, the mapping is trivial. For example, functions such as `open` and `stat` map naturally to FILE actions, and functions such as `select` and `poll` map naturally to POLL actions. For other functions, however, the mapping is more complex. For example, function `read` can map to both POLL and PULL actions, depending on the context. Luckily, the number of C-library calls to be considered for a given platform is small enough to allow for an exhaustive study of their characteristics. We studied, classified, and mapped the functions in the C library as a preliminary step towards our implementation of ADDA. ADDA uses this information to redirects to wrapping functions all relevant calls from the software to the C library, using binary instrumentation. The wrapping functions invoke the original functions while recording the information needed to update the event log and the environment data (*e.g.*, data read from streams and timing information).

The current implementation of ADDA is based on the Pin binary instrumentation framework [12]. We chose Pin for several reasons. First, it allows for instrumenting binaries on the fly, so we can perform execution recording and replay of a software without any need to modify the software beforehand. Second, Pin can be very efficient, which is a requirement for our technique to work on real deployed software. Third, Pin works on a large number of architectures that include IA32 (Mac OS X, Windows, and Linux), IA32e (Linux), Arm (Linux), and Itanium (Linux).

The minimization component of the ADDA tool is implemented as an extensible set of Python scripts that invoke the ADDA replay tool. As discussed in Section 3.1.3, we want the minimization module to allow for plugging different minimization algorithms for different types of files and stream dumps. To avoid problems of bias, for our experiments we have used the default minimization algorithm based on delta debugging [24].

4. Empirical Evaluation

To assess the feasibility and effectiveness of our approach, we performed an empirical evaluation using ADDA and investigated the following research questions:

RQ1: Feasibility – Can ADDA record and replay real executions in an accurate way?

RQ2: Efficiency – How much time and space overhead does ADDA impose while recording executions?

RQ3: Effectiveness – Can ADDA automatically reduce

the size of recorded failing executions and generate shorter test cases that manifest the same failures as the original executions? Can these executions still be used to debug the observed failure?
The following sections explain our experimental setup and discuss the results of our empirical evaluation.

4.1. Subject and Data

In selecting a subject for our study, our goal was to create a realistic instance of the scenario shown in Figure 1. To this end, we selected PINE [19], a widely used e-mail and news client developed at the University of Washington. We chose PINE because it is a non-trivial program, with a large user base and, moreover, PINE's developers provide source code and change logs for many previous versions of the software (the earliest ones dating back to 1993).

After studying several versions of PINE, we selected version 4.63, which contains two faults that were fixed in the subsequent version. The fact that the faults were documented and fixed is a good indication that they were (1) discovered in the field and (2) considered relevant enough to be worth fixing. Because knowledge about these faults and their manifestation is important for a correct understanding of our study, we describe them in detail in the next section.

4.2. Faults Considered

Header-color fault. This fault causes PINE to crash if (1) color is enabled in PINE's configuration, (2) a user adds one or more header colors, and (3) a user removes all header colors. This fault is ideal for our study because it was not discovered during in-house testing and resulted in field failures, which is the scenario we are targeting.

There are several ways in which a user can expose this fault and trigger the corresponding failure. Figure 3 shows three possible sequences of actions (and environment conditions) that we considered in our study. An edge between two actions A and B indicates that A must occur before B, but there can be any number of different interleaving actions between the two. For example, users could read and write several email messages between the time they save the configuration file and the time they go back to the configuration to delete header colors.

Address book fault. This fault causes PINE to crash if (1) the address book contains two entries with the same nickname and (2) the user edits the first of these entries changing it from a single address to a list of addresses. One unique characteristic of this fault, which makes it especially interesting for our evaluation, is that PINE does not let users create two entries with the same nickname. For the fault to manifest, it is thus necessary to perform an external edit of the address book. For this fault, simply recreating user actions is not enough to reproduce the failure because the environment also plays a fundamental role in the outcome.

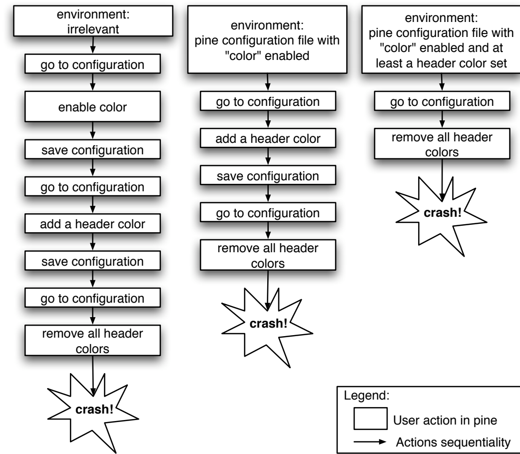

Figure 3. Possible sequences of actions that trigger the header-color fault.

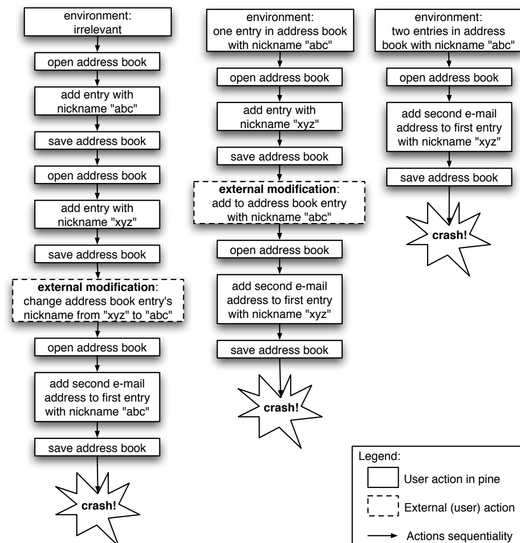

Figure 4. Possible sequences of actions that trigger the address-book fault.

There are several ways in which a user can expose this fault. Figure 4 shows three possible alternatives that we considered. The figure uses the same notation as Figure 3 but also shows *external user actions*—actions performed outside of PINE that affect PINE's behavior.

4.3. Collecting PINE's Execution Recordings

To assess replay and minimization capabilities of ADDA we need a set of executions that expose the failures considered. Moreover, these executions should be rich and varied. They should contain both actions that contribute to the failure and actions that do not contribute to the failure. Because executions captured from a sample of user sessions would be unlikely to trigger the failing behavior, and simply creating the executions ourselves could create problems of bias, we decided to generate executions in a random fashion.

Using PINE's manual, we compiled a list of high-level user actions. We then associated weights to actions to indicate the likelihood of users performing a given action. For example, we expect users to read and send email more often than they update PINE's configuration. We then wrote a simple program that produces random execution scripts of a given length. Each execution script contains, interleaved with other actions, one of the core sequences of actions shown in Figures 3 and 4.

Overall, we generated 20 execution scripts (10 for each of the two faults considered), containing between 10 and 100 high-level actions each. For each execution script, we first executed the script on PINE, run normally, and verified that the actions in the script produced the expected failure. We then executed the script again while recording the execution. At the end of this process ADDA had produced 20 execution recordings, one per script.

The next sections describe our studies and results. All studies were performed on a dual-processor Pentium D, 3.0Ghz, with 4GB of memory, running GNU/Linux 2.6.

4.4. Studies, Results, and Discussion

RQ1: Can ADDA record and replay real executions in an accurate way? To address RQ1, we replayed each execution recording and checked whether the execution failed as expected. To do this, we checked that (1) the replayed execution crashed due to the same process signal as the original execution, and (2) the value of the instruction pointer was the same in the two cases. For all 20 executions, ADDA was able to correctly reproduce PINE's failing behavior.

To reduce the threats to external validity of this first study, we recorded and fully replayed executions of two additional subjects: BC, a numeric processing language with arbitrary precision, and GCC, a widely used C compiler. We used Version 1.06 of BC (17 KLOCs) and Version 4.0.3 of GCC (around 1,000 KLOCs). As inputs for BC, we used several programs from the set of BC number theory programs (`http://www.numbertheory.org/gnubc/`) run with random numeric arguments. To verify whether the record and replay worked successfully, we simply checked the final results of the original and replayed executions. For GCC, we recorded the compilation of a set of student projects and used the correct termination of the compilation as the indicator of a successful replay. For both subjects, all executions were recorded and replayed correctly to the extent that we could verify.

These results show the feasibility of our approach. In the cases considered in the study, ADDA was able to record and replay complete executions.

RQ2: How much time and space overhead does ADDA impose while recording executions? We expect our technique to impose low time overhead during recording. Interactions between software and environment typically involve I/O operations that are relatively expensive and are likely to dominate the cost of our instrumentation.

It would be difficult to measure the overhead imposed by our technique on PINE, due to its interactive nature. Although we did not experience any noticeable overhead while recording the execution scripts, ours is just a qualitative and subjective assessment. To obtain a quantitative and more objective assessment of the cost of the approach, we measured the overhead imposed by ADDA on the additional two subjects used in the previous study: BC and GCC. For all executions, the difference between the execution times with and without ADDA was too small to be measured using UNIX's `time` utility. Therefore, at least for these subjects and executions, the overhead was negligible.

Despite these results, there may be cases in which our approach imposes a non-negligible overhead. In particular, if an application accesses very large files that are often changed externally, ADDA would make several copies of these files, which may involve a large time and space overhead. However, we do not expect situations of this kind to be very common. If we do encounter cases where our approach is too costly, there are ways to improve its efficiency. For example, for the issue mentioned above, we may modify our technique so that it uses a differencing-based approach to save only deltas instead of complete file versions.

RQ3: Can ADDA automatically reduce the size of recorded failing executions and generate shorter test cases that manifest the same failures as the original executions? Can these executions still be used to debug the observed failure? To address RQ3, we fed the 20 execution recordings for PINE to the ADDA minimization tool, which produced 20 corresponding minimized recordings using the four steps discussed in Section 3.1.3. The minimization step took at most 75 minutes per execution. As we did for RQ1, we verified that the minimized executions exhibited the same failing behavior as the original executions by checking process signal and value of the instruction pointer at the time of the crash. Figure 5 shows a bar chart with the results for the two sets of executions.

For each set, the bar chart shows the average percentage reduction achieved by ADDA for four measurements: total size of the stream dumps (streams size), total size and number of environment files (files size and # files, respectively), and number of unique instructions executed during replay (# unique instructions).

As the figure shows, ADDA was able to minimize executions along all four measurements, and for three of them it achieved a considerable reduction: on average, it was able to (1) reduce the streams size by almost 95% (from about 300 to 16 characters), (2) eliminate more than 85%

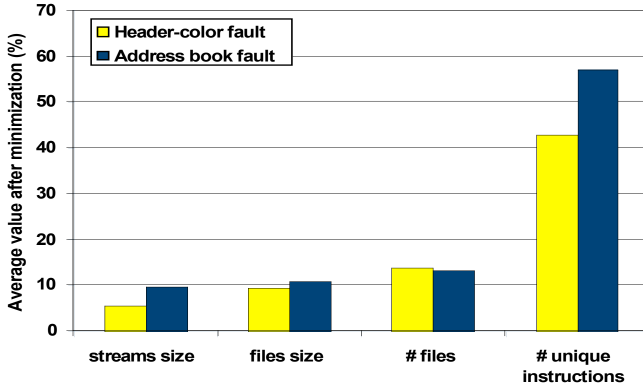

Figure 5. Results of the minimization performed by ADDA.

of the environment files (from 32 to three), and (3) reduce the total size of the environment by more than 90% (from approximately 800KB to 72KB). Whereas these first three measurements considered are indicative of ADDA's effectiveness in reducing execution recordings' size, the fourth measurement (number of unique instructions executed) is an indicator of how much the minimization can help debugging: having less instructions to inspect with respect to the original failing executions has the potential to speed up debugging. As our results show, ADDA was able to generate minimized executions that exercised 42% (for the header-color fault) and 57% (for the address-book fault) of the instructions exercised by the original executions. Although there may be no direct connection between these results and the reduction in debugging effort, the results are encouraging. Moreover, we stress that the possibility of replaying and debugging (in-house) failing field executions, even in the case of little of no reduction, would be a considerable improvement over the state of the art.

Note that we chose to show the reduction in the number of executed instructions instead of the reduction in execution time for fairness. PINE is driven by user actions, so its running time could increase simply due to idleness that would be eliminated during minimization and unnaturally inflate the savings. Note also that, when recording PINE's executions, we generated just the minimal amount of inputs needed to perform the actions in the script. For example, for all e-mail-sending actions, we wrote one-liner e-mails with one-word subjects. We chose this approach again to avoid an artificial inflation of the savings achieved by our technique (*e.g.*, minimizing a 10,000-character email would certainly result in a larger reduction than eliminating a 10-character email). Because we followed the same principle wherever applicable, using ADDA on executions recorded from actual users is likely to result in even larger reductions than the ones we obtained in this study.

Looking at the different results, we were especially pleased by the amount of reduction achieved by ADDA on the environment data. In almost all cases, the tool was able to completely eliminate a large percentage of files and also considerably reduce the size of the remaining files and streams. For example, there is a common group of three files that are necessary for both faults: PINE's configuration file, `/etc/passwd`, and `/lib/terminfo/x/xterm`. Although the technique was not able to minimize the `xterm` terminfo file, probably because of its binary content, it reduced `/etc/passwd` to a single line, the one with the entry for the email user, and PINE's configuration file to one or two lines, depending on the execution. Interestingly, one of these two lines is the one that records whether PINE has been run at least once already, to avoid opening PINE's greeting message multiple times. When that line was removed, no execution failed because PINE was stuck on the greetings page, waiting for an acknowledgment from the user (and none of the recorded keyboard streams contained the right key presses).

To gain some confidence that the executions recorded and minimized by ADDA can actually be used for debugging, we also performed a preliminary study on four of the 20 execution recordings. (We simply picked the first two executions generated for each fault.) For the study, we attached a well-known debugging tool, `gdb` (`http://www.gnu.org/software/gdb/`), to the replayed executions and used the tool to debug PINE while it was being replayed by ADDA. The result of the study was successful, and we were able to perform typical debugging actions, such as stopping at a breakpoint or watching the value of a variable, on the replayed executions.

Because we performed most of our studies on a single subject, they may not generalize. However, the subject we used is real and widely used, the faults we considered are real faults that were not discovered during testing, and we also performed a subset of the studies on two additional subjects. Therefore, although more studies are needed, our initial results with ADDA are promising and motivate additional research in this area.

5. Conclusion and Future Work

We presented a technique for debugging failures of deployed software—failures that occur while the software runs on user platforms. Our technique allows for recording, replaying, and minimizing user executions. The resulting minimized execution can then be used to debug the problem(s) leading to the observed failure.

We also presented ADDA—a prototype tool that implements our technique, is based on binary dynamic instrumentation, and works on x86 binaries—and an empirical evaluation of the approach performed using ADDA. The results of our evaluation, although still preliminary, show that our approach is effective and practical, at least for the subject and executions considered. Overall, ADDA was able to efficiently record many failing executions of a real program

and considerably reduce the size of the recorded executions while preserving their failing behavior. Moreover, we were able to replay and examine the minimized failing executions from within a debugger, which provides evidence that they could be used to investigate the original failure and support the scenario we sketched in Figure 1.

There are several directions for future work. *First*, we will perform additional studies with real users in our lab. *Second*, we will investigate how to extend our technique to handle interactions with a windowing systems. One possibility is to treat the windowing system as part of the software when intercepting interactions with the environment. Alternatively, we may be able to add (part of) the API of the windowing system to the set of calls that our approach intercepts and consider GUI events as an additional class of input streams. *Third*, if further studies show that concurrency creates problems for ADDA, we will also investigate ways to add concurrency-related events to our approach (leveraging Pin's support for concurrency [12]). *Fourth*, we will investigate ad-hoc minimization algorithms. Although our initial experience shows that our current minimization approach can reduce the size of the recorded executions, we believe that more sophisticated techniques can produce even better results. *Finally*, we will study ways to apply anonymization techniques to the recorded (and minimized) executions.

Acknowledgments

This work was supported in part by NSF awards CCF-0541080 and CCR-0205422 to Georgia Tech.

References

[1] Apple Crash Reporter, 2006. `http://developer.apple.com/technotes/tn2004/tn2123.html`.

[2] The Cooperative Bug Isolation Project, 2006. `http://www.cs.wisc.edu/cbi/`.

[3] S.-K. Chen, W. K. Fuchs, and J.-Y. Chung. Reversible Debugging Using Program Instrumentation. *IEEE Trans. on Software Engineering*, 27((8)):715–727, August 2001.

[4] Expectation-Driven Event Monitoring (EDEM), 2005. `http://www.ics.uci.edu/~dhilbert/edem/`.

[5] S. Elbaum, H. N. Chin, M. Dwyer, and J. Dokulil. Carving Differential Unit Test Cases from System Test Cases. In *Proc. of the 14th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 2006)*, November 2006.

[6] S. Elbaum and M. Diep. Profiling Deployed Software: Assessing Strategies and Testing Opportunities. *IEEE Trans. on Software Engineering*, 31(4):312–327, 2005.

[7] D. M. Hilbert and D. F. Redmiles. Extracting Usability Information from User Interface Events. *ACM Computing Surveys*, 32(4):384–421, Dec 2000.

[8] J. A. Jones, A. Orso, and M. J. Harrold. Gammatella: Visualizing Program-execution Data for Deployed Software. *Information Visualization*, 3(3):173–188, 2004.

[9] S. King, G. Dunlap, and P. Chen. Debugging Operating Systems with Time-traveling Virtual Machines. In *Proc. of the Usenix Annual Technical Conf.*, pages 1–15, April 2005.

[10] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable Statistical Bug Isolation. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI 2005)*, June 2005.

[11] Mercury LoadRunner, 2006. `http://www.mercury.com/us/products/performance-center/loadrunner/`.

[12] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools With Dynamic Instrumentation. In *Proc. of the 2005 ACM SIGPLAN conf. on Programming Language Design and Implementation (PLDI 2005)*, pages 190–200, 2005.

[13] Microsoft Online Crash Analysis, 2006. `http://oca.microsoft.com`.

[14] S. Narayanasamy, G.Pokam, and B.Calder. Bugnet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *Proc. of the 32th Annual International Symposium on Computer Architecture (ISCA-32)*, June 2005.

[15] R. H. B. Netzer and M. H. Weaver. Optimal Tracing and Incremental Reexecution for Debugging Long-Running Programs. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI 1994)*, June 1994.

[16] A. Orso, S. Joshi, M. Burger, and A. Zeller. Isolating relevant Component Interactions with JINSI. In *Proc. of the Fourth International ICSE Workshop on Dynamic Analysis (WODA 2006)*, pages 3–9, Shanghai, China, May 2006.

[17] A. Orso and B. Kennedy. Selective Capture and Replay of Program Executions. In *Proc. of the Third International ICSE Workshop on Dynamic Analysis (WODA 2005)*, pages 29–35, May 2005.

[18] C. Pavlopoulou and M. Young. Residual Test Coverage Monitoring. In *Proc. of the 21st International Conf. on Software Engineering (ICSE 99)*, pages 277–284, May 1999.

[19] Pine© – A Program for Internet News & Email, 2006. `http://www.washington.edu/pine/`.

[20] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic Test factoring for Java. In *Proc. of the 20th Annual International Conf. on Automated Software Engineering (ASE 2005)*, pages 114–123, November 2005.

[21] S. Srinivasan, S. Kandula, C. Andrews, and Y. Zhou. Flashback: A Light-weight Rollback and Deterministic Replay Extension for Software Debugging. In *Proc. of the 2004 USENIX Technical Conf.*, June 2004.

[22] J. Steven, P. Chandra, B. Fleck, and A. Podgurski. JRapture: A Capture/replay Tool for Observation-based Testing. In *Proc. of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2000)*, pages 158–167, 2000.

[23] Mercury WinRunner, 2006. `http://www.mercury.com/us/products/quality-center/functional-testing/winrunner/`.

[24] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. on Software Engineering*, 28(2):183–200, 2002.