Autom Softw Eng (2010) 17: 347–372
DOI 10.1007/s10515-010-0068-6

AFID: an automated approach to collecting software faults

Alex Edwards · Sean Tucker · Brian Demsky

Received: 16 September 2009 / Accepted: 5 May 2010 / Published online: 25 May 2010

Abstract We present a new approach for creating repositories of real software faults. We have developed a tool, the Automatic Fault IDentification Tool (AFID), that implements this approach. AFID records both a fault revealing test case and a faulty version of the source code for any crashing faults that the developer discovers and a fault correcting source code change for any crashing faults that the developer corrects. The test cases are a significant contribution, because they enable new research that explores the dynamic behaviors of the software faults. AFID uses an operating system level monitoring mechanism to monitor both the compilation and execution of the application. This technique makes it straightforward for AFID to support a wide range of programming languages and compilers.

We present our experience using AFID in a controlled case study and in a real development environment to collect software faults in the internal development of our group's compiler. The case studies collected several real software faults and validated the basic approach. The longer term internal study revealed weaknesses in using the original version of AFID for real development. This experience led to a number of refinements to the tool for use in real software development. We have collected over 20 real software faults in large programs and continue to collect software faults.

Keywords Fault collection

A. Edwards
Computer Science Department, University of California, Los Angeles 90095, CA, USA

S. Tucker
Western Digital, Michelson Drive, Irvine 92612, CA, USA

B. Demsky (✉)
Department of Electrical Engineering and Computer Science, University of California, Irvine 92697, CA, USA
e-mail: bdemsky@uci.edu

1 Introduction

The software engineering and programming languages research communities have traditionally relied upon anecdotes and intuition about the relative importance of various types of software faults to guide our efforts. Numerous papers evaluate prototype tools on a few hand-selected software faults or even synthetically-injected faults. In the rare cases when researchers do use their tools to detect new faults in existing systems, they must manually verify that the software faults they detect are both real and important. Moreover, in such studies the research often does not provide empirical evidence that their tool catches a significant percentage of important faults of the given type because fault sets to perform such tests are often unavailable.

Researchers have recently begun to perform empirical studies of large data sets of software faults. These studies typically mine fault data from CVS archives that have become available in recent years due to the creation of large, open software systems by the open-source community. Unfortunately, these archives often lack the information necessary to easily reproduce the software faults.

Collections of real software faults and the test cases to reproduce the faults have the potential to provide a powerful new tool for software researchers. We can use the test cases to automatically classify faults based on the error they introduce in the program's execution. A researcher could, for example, use such a classification to determine whether null pointer exceptions represent an important real world problem. Later on, the same data sets would enable researchers to more easily and more rigorously evaluate fault finding tools. The fault data set would provide real software faults that researchers could use to evaluate their tools in an automated fashion.

One problem with most existing data sets is that they lack test cases that reveal software faults. In an early attempt to remedy this situation, we tried to manually collect real software faults. Our approach was to ask graduate students to record the faults that they corrected while developing software for their research. For each fault, we asked the students to record: (1) the test case that revealed the fault, (2) a copy of the source code that contained the fault, and (3) the source code change that removed the fault. They found recording this information to be tedious, and instead they often focused on the development task at hand and forgot to record any information. The lesson from this experience is that the successful collection of software faults must be automated.

1.1 Basic approach

In this paper we introduce a novel approach that monitors the software development process to automatically record software fault data. For each fault, our approach records: (1) a test case that reveals the fault, (2) a version of the source code that contains the fault, and (3) a change to the source code that corrects the fault.

We have implemented this approach in the Automated Fault IDentification Tool (AFID). AFID automatically records software faults by monitoring the compilation and execution steps of the software development process. The underlying design principle for AFID is to record as much software fault data as possible while imposing minimal runtime overheads and requiring minimal assistance from the developer. The

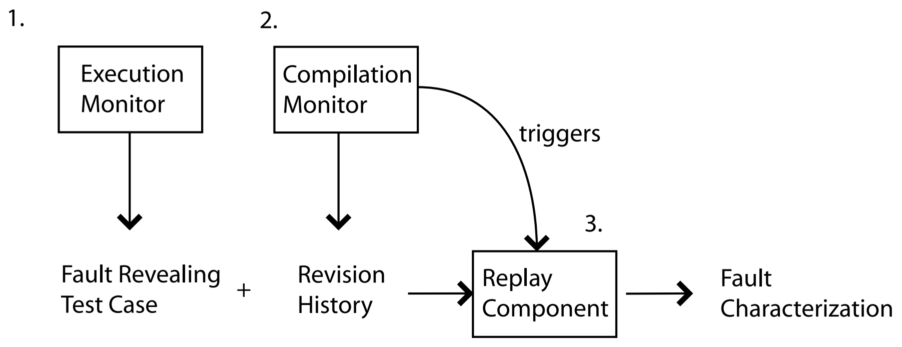


Fig. 1 Overview of fault characterization

final goal of the AFID project is to collect fault data from a wide range of software developers working on real projects. Therefore, requiring the developer to actively participate in recording faults would potentially make finding developers to use AFID much more difficult. According to this principle, AFID has been designed to only record faults that actually cause crashes. AFID does not recognize more subtle correctness faults because that would burden the developer with describing the desired behavior of an application. We expect that we can learn much interesting information from crashing faults alone.

Figure 1 presents an overview of our approach. Our approach is architected with the following three primary components:

1. *Execution monitor*: The execution monitor traces executions of the application under development. The execution monitor records the inputs to the application. If the application crashes, the execution monitor uses the recorded inputs to create a test case that reproduces the observed failed execution. At this point, AFID records (1) a test case that contains the application inputs that reveal the fault and (2) the source code version in which the fault was discovered. AFID places this new test case in its repository of unresolved test cases and stores a reference to the current version of the subversion source repository in the test case.
2. *Compilation monitor*: The compilation monitor traces executions of the compiler to automatically discover which source files comprise the application under development. Whenever the application is recompiled, the compilation monitor records both a list of any new source files it discovers and a list of all source files that have changed since the last compilation. The compilation monitor then updates its internal subversion repository with any changes that have been made to the application. Finally, the compilation monitor invokes the replay component to check if the recent changes correct any known software faults.
3. *Replay component*: The replay component executes the newly compiled version of the application on all of the unresolved fault revealing test cases. If the application executes a fault revealing test case without crashing, the replay component assumes that the most recent code change corrected the underlying fault. The replay component records the current version identifier as the fault correcting code change. The replay component then marks the test case as resolved. Researchers

have developed many replay systems for debugging applications (Choi and Srinivasan 1998; Steven et al. 2000; LeBlanc and Mellor-Crummey 1987). These other systems replay the exact execution, while AFID generates test cases from the application inputs with the goal of running different versions of the application on the same test case. The exact executions of these new versions can potentially differ from the version in which the test case was first recorded.

1.2 Contributions

This paper makes the following contributions:

– *Automated fault collection strategy*: It presents heuristics that monitor the development process to automatically record fault revealing test cases and automatically detect which code changes correct these software faults.
– *Process monitoring technique*: It presents a language and tool chain independent technique to monitor both the executions of the application under development and the evolution of its source code.
– *Automated recording of test cases*: It presents a technique to automatically record test cases from failed executions. These test cases can potentially be incorporated into the application's regression test suite.
– *Monitoring overhead measurement*: It presents measurements of the runtime overhead of AFID's monitoring for both a computationally bound benchmark and an I/O bound benchmark.
– *Case study*: It presents our experience using the tool to collect software faults in a case study.
– *Real world experience*: It presents our experience using AFID to monitor the development of our research groups' compiler infrastructure.

The remainder of the paper is structured as follows. Section 2 presents an example to illustrate how the approach works. Section 3 presents the automatic fault collection tool AFID. Section 4 discusses possible privacy concerns. Section 5 presents overhead measurements and the results of our initial case study. Section 6 presents our real world experience using AFID to collect software faults. Section 7 presents related work; we conclude in Sect. 8.

2 Example

We next use an example to illustrate our approach. Let's suppose that the developer uses a text editor to write the program shown in Fig. 2. This program takes a command parameter that specifies its input file. The program then opens this file and reads a series of commands from it. These commands instruct the program to either write a digit to an array element, print an array element, or prompt the user whether to continue. Note that line 19 is missing a `break` statement, which would cause the execution of the prompt command to erroneously continue into the code for the read command.

Fig. 2 Faulty example program

```java
public class Example {
  public static void main(String[] arg)
   throws IOException {
    int array[]=new int[10];
    FileReader fr=new FileReader(arg[0]);
    while(true)
      switch(fr.read()) {
      /* Write to array element. */
      case 'W':
        int woff=fr.read()-'0';
        int val=fr.read()-'0';
        array[woff]=val;
        break;
      /* Prompt user whether to continue. */
      case 'P':
        System.out.println("Continue (y/n)?");
        if(System.in.read()=='n')
          return;
      /* This line is missing a break. */
      /* Print array element. */
      case 'R':
        int roff=fr.read()-'0';
        System.out.println(array[roff]);
        break;
      case -1:
        return;
      }
  }
}
```

2.1 Monitoring compilation

After a developer finishes writing the program, he/she would typically compile the program using one of many Java compilers. AFID tracks the evolution of the program's code by monitoring the execution of the compiler. When the compiler compiles the example program, it would make a system call to the operating system to open `Example.java` for read access. AFID intercepts the `open` system calls made by the compiler to record when the developer adds new source files to the application. AFID then examines the file's extension to determine that this file contains source code for the application. The primary benefit of this approach is that it enables AFID to support most compilers while not requiring the developer to manually identify the source files that comprise the application's source code.

2.2 Monitoring program execution

In the normal development process, we expect that the developer would next execute the example program on an input file. Figure 3 presents an input file for the example

Fig. 3 Fault revealing input file `input.txt`

```
W23PR2
```

program. The input file contains a sequence of three commands: `W23` instructs the program to write the value `3` to array element `2`, `P` instructs the program to prompt the user whether to continue, and `R2` instructs the program to print the second array element. Note that this input file invokes the prompt functionality and if the user chooses to continue it would reveal the fault in the prompt functionality of the example program.

Typically, the developer would next execute the example program on this input file by typing `java Example input.txt`. AFID's execution monitor then records the command line used to execute the program. The program's execution opens the file `input.txt` for read access using the `open` system call. AFID's process monitor intercepts this call and records that the execution reads from the file `input.txt`. Then the process prints the string `Continue (y/n)?` to the screen. Let's suppose the developer types "y" which reveals the fault in the prompt code that causes the program to continue into the array element printing code. The program then uses the byte intended to specify the read command as an index. This causes the program to exit due to an array out of bounds exception. AFID inspects the execution's exit value to determine that the program crashed.

The goal is to create a test case that can reproduce the crash. AFID records the command line that was used to invoke the fault revealing execution, makes copies of all the input files that the program opened, stores a trace of the console user interactions, and stores the mapping from the pathnames of the files that the program opened to the copies made by AFID.

2.3 Recording fault corrections

We expect that the developer will eventually correct any important software faults. In this case, we assume that the developer has corrected the fault in this program by changing line 19 to a `break` statement. When the developer compiles the corrected program, AFID would then record that line 19 of the `Example.java` file has been changed.

AFID then invokes its replay component to replay the fault revealing test cases on the new version of the example program. The replay component executes the example program using the recorded command line. When the example program executes, it makes a system call to open the `input.txt` file. AFID intercepts this system call before the operating system processes it and changes the filename to the name of the copy in the test case. When the program prompts for user input, AFID recognizes the prompt and responds with the recorded input `y`. Because the developer corrected the underlying software fault, the program executes correctly on the test case. AFID inspects the program's return value to determine that the underlying fault was corrected.

At this point, AFID has identified that the most recent source code change corrects the underlying software fault. AFID has recorded the following information for the example fault: (1) the buggy version of the example program from Fig. 2, (2) the test case that reveals a fault in the buggy version from Fig. 3, and (3) a diff that gives the source code change that corrects the fault (for this example, replacing line 19 with `break;`). AFID records all of this information in its record for this fault. It then (optionally) uploads this fault information to a centralized fault repository.

3 Automated fault identification

We have architected AFID as three basic components: (1) the execution monitor, which records crashes and creates fault revealing test cases to reproduce these crashes, (2) the compilation monitor, which identifies new source files and tracks changes to the source code, and (3) the replay component, which records when a source code change corrects a fault. Each component of AFID uses the same basic monitoring strategy—they intercept the system calls that the application or compiler uses to communicate with the underlying operating system. This approach enables AFID to easily support many different compilers, virtual machines, and programming languages with only small configuration changes.

The goal of AFID is to collect complete information for software faults. AFID collects the following information for each fault:

– *Fault revealing test case*: For each reported fault, AFID records the test case that reveals this fault.
– *A version of the application with the fault*: For each reported fault, AFID records a copy of the source code of the application version that contains the fault. For space efficiency, this is stored as a version identifier to a version control system repository.
– *Fault correction*: For each reported fault, AFID records the source code change that corrected the fault. For space efficiency, this is stored as a version identifier to the version control system update that stores the correction.
– *Revision history of the application*: AFID records a fine-grained revision history of changes to the application's source code.

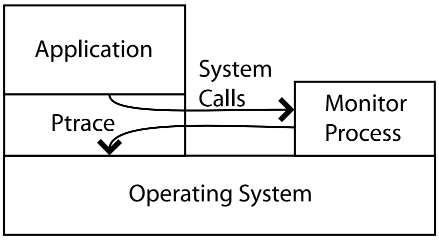
3.1 Recording test cases

AFID's execution monitor traces the executions of the application under development to generate fault revealing test cases. The execution monitor records the inputs to the application's execution by intercepting the system calls from the application to the underlying operating system.

The execution monitor uses the `ptrace` system call to monitor executions of the application under development (Haardt and Coleman 1999). Figure 4 presents an overview of the approach. The `ptrace` interface allows the execution monitor to intercept system calls made by the application under development before the operating system processes the call. We next describe our `ptrace`-based approach in more detail.

The execution monitor begins by forking a new child process, the child process calls `ptrace` with the `PTRACE_TRACEME` option to request tracing, and then the

Fig. 4 Ptrace interface



child process calls the exec system call to execute the application under development. When the child calls the exec system call, the previous invocation of ptrace with the PTRACE_TRACEME option causes the child process to stop before executing the new application image.

The monitoring process then calls the ptrace system call with the PTRACE_SYSCALL option and then calls wait. The next time the child process makes a system call, the operating system suspends the child process and wakes up the monitoring process. When the execution monitor is awoken, it uses ptrace's PTRACE_GETREGS option to read the system call parameters to determine the type of the system call. If the child process performs an open system call, the execution monitor reads the system call parameters to obtain a pointer to the filename and the file access mode. The execution monitor then uses ptrace's PTRACE_PEEKDATA option to read the filename from the monitored process's memory space using the pointer passed into the open system call. AFID records the absolute pathname of the file that was opened.

If the monitored application has requested to open the file for write access, the execution monitor must immediately make a copy of that file. If AFID delays copying the file until the monitored application actually crashes, the monitored application would likely have already changed the contents of the file. If the monitored application has requested to open the file for read access, the execution monitor uses a lazy copy strategy. It delays the overhead of copying the file until the monitored application actually crashes.

When the monitored application exits, the execution monitor inspects its return value to determine whether it crashed. If the monitored application has crashed, the execution monitor makes copies of all of the files that the monitored application read. It then stores the mapping between the pathnames that the monitored application used to access the files and the files' copies in a text file in the test case. Otherwise, if the application successfully exits, AFID discards the files.

3.1.1 Recording user interactions

We next describe how AFID records user interactions. AFID uses the same ptrace-based mechanism to record a trace of read events from standard input and write events to standard output. One potential issue with simply replaying the exact user interaction is that changes in the program (or even the time) may change the text that the program outputs. For example, consider the user interaction shown below:

```
Display:    <STARTING>
Display:    [Tuesday, April 20, 2010]>
Response:   ls
```

If we require that the output match exactly, the test case will have significant problems generalizing to future executions of the program. Instead, for each input event AFID computes the shortest suffix of the program output since the last input event that uniquely identifies when the input occurred. For the example, this suffix is just the last two character '`]>`' in the prompt. This fuzzy matching approach allows the recorded test case to generalize over small changes to the program's output.

3.1.2 Duplicate test cases

One potential issue is that the developer may rerun the same test case multiple times. To avoid storing multiple copies of the same test case, the monitor computes a hashcode for each test case. The monitor then compares this hashcode to a list of hashcodes for the other test cases. If AFID records a hashcode match, it deletes the new test cases. AFID makes the assumption that the hash values do not collide. In the unlikely event that two different test cases have the same hash value, AFID only stores the first test case.

3.1.3 Filtering inputs

The monitored application's execution typically reads many files that would not be considered inputs to the application. For example, the dynamic linker may load library files or a virtual machine may load class files, virtual machine components, virtual machine configuration files, and various system files. These extraneous input files would make the test cases very large. Moreover, recording input files from dynamic libraries or virtual machine internals could make the test case specific to the exact execution environment.

AFID employs a filtering mechanism to remove these extraneous files. The filter mechanism uses a configuration file that contains a list of regular expressions that match the filenames to exclude from the test cases. AFID can automatically generate this configuration file for Java applications by monitoring the execution of a dummy Java application and then generating a list of files that are loaded by the JVM. AFID then adds some default expressions that exclude class files and other known extraneous files.

3.2 Monitoring compilation

AFID stores a copy of the source code each time the developer compiles the application. To efficiently store multiple versions of the application's source code, AFID maintains an internal subversion repository. Subversion is an open-source version control system with support for atomic commits (Collins-Sussman 2002). AFID interacts with subversion by calling the standard command line Subversion client. Modern decentralized version control systems such as GIT could alternatively be used to possibly support merging the AFID repository into the main branch, but would require a tighter coupling with the development repository (Chacon 2010). Each time the developer compiles the application, the compilation monitor component of AFID monitors the compiler to determine which files contain the application's source code. The compilation monitor uses the `ptrace`-based monitoring technique described in Sect. 3.1 to record application source files.

When the compilation monitor discovers a new source file, it adds the file to its internal subversion repository. Then the compilation monitor commits all of the source code changes since the last compile to its internal subversion repository. Finally, the compilation monitor calls the replay component to replay all of the unresolved fault-revealing test cases on the new version of the application.

One challenge is that AFID's internal subversion repository may conflict with development projects that make use of subversion. To maintain compatibility with subversion, the compilation monitor makes its own copy of the source code tree to use for its internal subversion repository. To avoid the overhead of copying large files, the compilation monitor makes hardlinks from the filename in its internal copy of the source code tree to the original in the developer's source code tree. The compilation monitor then uses the copy of the source code tree to build its internal repository.

3.3 Replaying test cases

The replay component checks whether the most recent source code changes correct any of the faults AFID has recorded. The basic strategy is to execute the new version of the application on each of the unresolved fault revealing test cases. If the application executes successfully, the replay component has determined that the most recent code change corrects the fault revealed by that test case. The replay component then stores the subversion version identifier of the source code version that corrects the fault in the test case and marks the test case as resolved.

3.3.1 Sandboxing replay

A naive replay implementation would simply copy the files in the test case back to their original locations and then execute the application. However, this strategy has serious potential consequences—the replay component could potentially overwrite important files when copying the test case files or the execution of the application could overwrite important files. AFID prevents the replay of applications from overwriting important data by using the same `ptrace`-based technique to partially sandbox the application. This sandbox is not intended to isolate a hostile application—it is intended to prevent the replay of normal applications from accidentally overwriting important files.

The replay component implements the sandbox by intercepting file open requests. If the application makes a file open request for one of the test case files, the replay component will redirect the request to the file in the test case. If the application makes a request for an excluded file, the replay component will pass the open request unmodified to the operating system. Note that if the application is modified or the fault is corrected, the application can open files that were neither present in the test case nor filtered by the filter expressions. It is straightforward to modify the replay component to make a copy of that file and redirect the request to the copy. This sandbox provides the application with the illusion that the test case files are in the same location as the files in the original execution—a secondary benefit of this approach is that it enables the test case to reproduce software faults that depend on the exact location of the input files.

We next discuss how we implement the sandbox using the `ptrace` system call. The replay component begins by making a copy of the test case. It then starts the monitored application's execution inside the partial sandbox. The basic idea is to use the technique described in Sect. 3.1 to intercept `open` system calls. When the replay component intercepts an `open` system call, it retrieves the requested filename. If the filename is contained in the test case, the replay component will modify the system call's parameters to open the copy in the test case. The replay component changes the `open` system call's filename by using `ptrace`'s `PTRACE_SETREGS` option to modify the register that stores the pointer to the filename to point to a new memory location. Then the replay component uses `ptrace`'s `PTRACE_ POKEDATA` command to write the filename of the copy to this new memory location. The replay component then restarts the application to allow the operating system to service the system call.

Note that the replay tool must obtain memory in the other application's memory space to store the filenames of the copies. The replay system obtains this memory by intercepting the first system call that the application performs. The replay system rewrites this system call's parameters to change it into a `brk`[1] system call to obtain the initial bottom of the heap. The replay system restarts the application and then the operating system executes the injected `brk` call. The application is halted after the system call is performed and control is returned to the replay tool. The replay tool then modifies the program counter to cause the application to re-execute the same system call. The replay tool then repeats the same system call injection strategy to inject a second `brk` system call that sets the new bottom of the heap. The replay system has now allocated its own space in the application's memory space. The replay system then resets the program counter another time to perform the initial system call. If the application later uses the `exec` system call to load a new binary, the replay system repeats the same procedure to obtain space in the newly loaded application's memory space.

If the application's execution is successful, the replay component has discovered that the most recent source code change corrects the fault. Note that the test case may not contain some files that were present on the local disk. In this case, it is straightforward for the replay component to add copies of these files to the test case.

3.3.2 Termination

It is possible that the developer may make a source code change that causes the application to loop on an unresolved test case. To address this issue, AFID records the elapsed time for each execution of the application. The replay component then uses

[1]The `brk` system call is used to read and set the bottom of the heap. This system call is the primitive that underlies library-based memory allocation functions such as `malloc`.

this record of execution times to estimate an upper bound on the application's execution. When the application executes for longer than this bound, AFID assumes that the application is looping. This prevents the replay component from waiting indefinitely for a non-terminating computation. Note that in the worst case, when a timeout is used to incorrectly identify an execution as looping, the effect is only to prevent AFID from recognizing a fault correction.

3.4 AFID server

AFID uses a web-based server application that aggregates the faults discovered by the AFID client. AFID supports two update modes: automated and manual. The automated mode automatically uploads a test case once the client has discovered the fault correcting code change. The manual mode allows the developer to manually control the uploading process. We developed the manual mode in anticipation that some developers will wish to maintain control over when uploads are performed. The client uploads the fault revealing test case, the version identifier for the source code version whose execution generated the fault revealing test case, the version identifier for the code change that corrects the fault revealing test case, and the latest version of AFID's internal subversion repository for the application.

3.5 Interpreted languages

The current implementation of AFID is designed for language environments in which there is a separate compilation and execution phase. Therefore it does not adequately address interpreted languages. We note that the basic techniques developed in this paper can be used in this environment if the execution monitor and compilation monitor are combined into a single tool.

The basic idea is to record for execution (1) the source files that the interpreter opens and (2) the files that the program reads. The two types of files can be distinguished by their extensions. The combined monitor would then perform a repository checking for the source files in the same manner as the compilation monitor. If the program crashed, the combined monitor would generate a test case in the same manner as the execution monitor.

3.6 Recording regression tests

The design of AFID is focused on recording fault data for research. However, we expect that practitioners may also find AFID beneficial for recording regression tests. In particular, AFID's fault data set includes test cases for each fault that the developer has discovered and corrected. We expect that this library of test cases may be a useful addition to the application's regression test suite. AFID's execution monitor provides the functionality to cleanly bundle the component files into test cases. AFID's replay component allows the test cases to be easily replayed on future versions of the application. Practitioners may find AFID particularly useful for test cases that contain files that are scattered throughout the directory structure or that involve the modification of common configuration files or any other files that are shared with other applications. An AFID test case consists of a collection of input files that comprise the inputs for the test case, a transcript that records the user interactions, and a text file that list the original pathnames for each of the input files. For test cases whose input files are isolated into a single directory, it is straightforward to convert an AFID test case into a standard test case that is usable in regression testing frameworks.

3.7 Limitations

The primary design goal of AFID, to minimize developer burden, places significant limitations on its scope. For example, AFID relies on the error code returned by a program to detect crashes instead of using test cases to validate correct behavior. This approach works well to detect uncaught exceptions in Java applications. However, many programs return error codes in their normal execution to indicate an error in their input. For example, our compiler returns a negative value if the input source code contains syntax errors or semantic errors. This causes a difficulty—AFID cannot tell the difference between an error in the compiler and an error in the input. Note that errors in the input will never generate a false report as the given input will always cause the compiler to exit with a negative return value. But over time these test cases can build up, and cause the replay process take increasing amounts of time. A second concern is that bugs in error handling code will often never be recorded, because even after they are corrected the program will still return a negative value.

AFID implicitly assumes that bugs are deterministic. Non-deterministic bugs can cause AFID to report the wrong code change as a bug fix. In our internal use of AFID, we have occasionally observed this problem. We have found that asking the developer to confirm bug fixes helps filter these cases. We have also occasionally filtered such bug reports on the server side. Retrying test cases multiple times can be used to automatically exclude non-deterministic bugs. If capturing non-deterministic bugs is desirable, statistical approaches applied across many versions of the code could potentially be used to detect which source code change was likely to have corrected the bug.

AFID is currently limited to console programs. AFID could be extended to support applications that interact with users through the graphical user interface. The idea is to extend the GUI library to export a trace of both user inputs and program events and an interface that allows AFID to inject user inputs. AFID could then use a similar approach to its approach for console I/O to the GUI.

4 Privacy concerns

Privacy may be a concern when using AFID for software fault user studies. Because AFID records all source code changes along with the application inputs, it may be possible to discover the actual identity of a study participant from the comments, coding style, project, and test cases. We expect that user studies will not use AFID to monitor the development of applications that contain sensitive source code or that may process sensitive inputs. Because a developer may accidentally input private information into the application under development, AFID supports a manual test case transfer mode that allows the developer to maintain complete control over whether to include test cases in a data set.

5 Evaluation

We next discuss our experience using the AFID implementation. The AFID implementation consists of approximately 5,000 lines of C code and shell scripts. The implementation is available for download at http://demsky.eecs.uci.edu/afid/ and we encourage readers to download AFID and contribute bugs to the repository. In this section, we report our measurements of AFID's monitoring overhead on two applications and then discuss our experiences using AFID to monitor software developers.

We measured AFID's overheads on a workstation with a 2.2 GHz Core 2 Duo processor, 2 GB of RAM, and Debian Linux running kernel version 2.6.25. We used version 1.5.0_14 of Sun's HotSpot JDK.

We used two different benchmarks: the Jasmin byte code assembler and the Inyo ray tracer. We used version 2.3 of the Jasmin bytecode assembler. It contains 11,450 lines of code and is available for download at http://jasmin.sourceforge.net/. We selected Jasmin because assembling bytecode involves a relatively large amount of I/O and therefore is likely to incur a significant monitoring overhead under AFID. The Inyo ray tracer contains 5,843 lines of code and is available for download at http://inyo.sourceforge.net/. We selected Inyo to give results for a longer-running, computational-bound benchmark.

5.1 Compilation overhead

Table 1 presents the compilation overhead measurements. All of these measurements were taking with no outstanding test cases and no code changes. Without monitoring, we measured the time to compile Jasmin as 2.54 seconds, the time to compile Inyo as 1.34 seconds, and the time to compiler our group's compiler as 8.32 seconds. With monitoring and updating AFID's internal SVN repository, we measured the time to compile Jasmin as 6.53 seconds, Inyo as 4.67 seconds, and our group's compiler as 13.98 seconds. We then measured the time to compile with monitoring but without updating the internal SVN repository for Jasmin as 3.79 seconds, for Inyo as 1.57 seconds, and for our group's compiler as 11.32 seconds. In our initial conference publication, we initially expected that these numbers were acceptable. After using AFID to monitor our own internal development, we have since discovered that even these delays were annoying.

One of the largest issues we discussed above was the extra time AFID adds to compilation. We have addressed this concern by extending AFID to support background commits to the subversion repository and to replay test cases in the background. The combination of these two changes means that AFID's monitoring causes a negligible delay in the compilation time. The primary challenge with supporting background

Table 1 Monitoring overhead

	Jasmin	Inyo	Compiler
Normal compile	2.54 s	1.34 s	8.32 s
Monitored compile with svn	6.53 s	4.67 s	13.98 s
Monitored compile without svn	3.79 s	1.57 s	11.32 s
Background compiler	2.95 s	1.43 s	8.71 s

Table 2 Execution overhead

	Jasmin	Inyo
Normal execution	0.21 s	30.73 s
Monitored execution	0.43 s	31.99 s

processing is ensuring that multiple background instances of the compilation monitor cannot simultaneously update and therefore potentially corrupt AFID's internal data structures. We have modified AFID to use locking to ensure that only a single background instance of the compilation monitor can update the internal data structures at once. We have modified the execution monitor to atomically add new test cases to the repository of unresolved test cases using the standard directory renaming technique to prevent possible races with the replay component.

Table 1 also presents overhead measurements for the compilation monitor that compares foreground processing to the new background processing mode. We can see that background processing significantly lowers the overhead of monitoring compilation. With background processing, the compilation monitoring overhead is only 9% on average. Note that the relative benefits of background processing increase as the system builds a collection of test cases as they are also tested in the background.

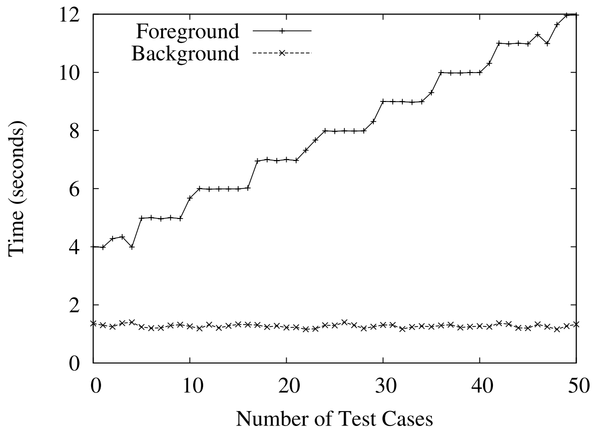
5.2 Execution overhead

Table 2 presents the execution overhead measurements. Our workload for Jasmin consisted of all of the examples contained in the Jasmin distribution. Without monitoring, Jasmin took 0.21 seconds to execute on this workload. With monitoring, Jasmin took 0.43 seconds to execute on this workload. Our workload for Inyo consisted of the model file included with the Inyo distribution. Without monitoring, Inyo took 30.73 seconds to execute on this workload. With monitoring, Inyo took 31.99 seconds to execute on this workload. We expect that Jasmin's monitoring overhead of 104% represents a worst case as Jasmin performs a large number of system calls, which incur extra overheads under AFID, and relatively little computation. We expect that Inyo's monitoring overhead of 4% represents the best case as Inyo performs relatively few system calls and a large amount of computation. We expect that this range of overhead will be acceptable in most development environments.

5.3 Scalability

We performed a set of experiments to explore how AFID's execution time varies as the number of test cases increases. We performed these experiments on a 2.26 GHz Core 2 Duo with 2 GB of RAM running Linux version 2.6.30 and JDK version 1.5.0_19-b02. We generated a set of 50 inputs that cause the Inyo ray tracer to exit with an error code. We then measured how long it took to compile the Inyo ray tracer under AFID as we increased the number of outstanding test cases. Figure 5 presents the results of this experiment. For the foreground mode of AFID, we see that the compilation time increases linearly in this experiment with the number of test cases. We note that test cases that take longer to execute would result in longer compilation

Fig. 5 Scalability of AFID



times. For the background mode of AFID, we see that the compilation time does not change as we increase the number of test cases. As modern processors typically include 4 or more cores, we expect that the background processing performed by AFID will have a negligible effect on the usability of the machine.

5.4 Case study

Our case study attempts to explore the most basic question one can ask about the AFID tool: Does it effectively record real software faults? To answer this question, we recruited a population of software developers and had each developer complete a programming problem while being monitored by AFID.

5.4.1 Developer population

One goal of this case study is to verify that AFID's fault identification heuristics work with the wide range of debugging approaches used by developers. We attempted to represent this wide range in our study population by recruiting 8 students with diverse backgrounds: the study participants had widely varying educational backgrounds, industrial experience, years of programming experience, and countries of education. Their educational backgrounds ranged from current undergraduate students to doctorates. Several participants had industrial experience while other participants had only academic experience. The study participants were educated in the United States, China, and India.

5.4.2 Methodology

We installed the AFID tool in each developer's account and instructed the developer in the use of the AFID tool. We then asked each developer to complete a programming problem in Java while using the AFID monitoring tool. We selected the programming problems from practice programming contest problems and basic data structure implementation problems.

Table 3 Fault breakdown

Fault Type	Count
Parsing logic error	3
Null pointer dereference error	3
Initialization error	2
Missing condition check	1
Loop bound error	1
Shadowed field	1
Incorrect comparison	1

5.4.3 Fault breakdown

After a developer completed the problem, we asked the developer to go through the fault reports that AFID had collected, verify that the recorded corrections were correct, and if so, to describe the underlying programming error. We then examined their responses and attempted to classify the faults by their underlying programming errors. Table 3 presents a breakdown of the recorded faults by the type of the underlying programming error. The two largest categories were errors in the logic for parsing the input and null pointer dereference errors. The parsing errors typically involved errors in reading the specification of the input format. The null pointer dereference errors were not simply omitted null pointer checks, but instead a wide range of logic errors that caused the programs to dereference null pointers.

We observed that even though AFID can only record failures that cause the application to throw an exception, in our case study, AFID recorded a rich set of software faults. Even in this small case study, AFID recorded high-level faults including errors caused by misunderstandings of the exact format of the input file.

5.4.4 Fault recording errors

We next discuss how often AFID recorded the correct fault-correcting source code change. For each recorded fault, we asked the participant to verify whether AFID had correctly identified this change as fault correcting. We report the results in Table 4. The table contains a row for each participant in the study. The first column gives designators for each participant, the second column reports the number of faults AFID recorded for that participant, and the third column reports how many of these faults contained the correct fault correcting source code change.

We note from the table that AFID has recorded fault data entries that contain the wrong fault correcting code change for two of the study participants. We then examined the incorrect fault correcting source code changes to better understand the problem. We found a surprise—these two study participants employed an experimental approach to correcting software faults. They made changes to the code to improve their understanding of why the application threw an exception. For example, in two cases the participant commented out the line of code that was throwing the exception. AFID then recorded that this source code change cause the program to no longer crash and record the experimental code change as the fault correcting

Table 4 Fault counts by participant

Participant	Number of recorded faults	Number of verified corrections
A	2	2
B	1	1
C	4	2
D	8	5
E	1	1
F	1	1
G	0	0
H	0	0

code change. In the other three such cases, the participants commented out incorrect debugging code that caused the program to throw an exception. The programming problems were relatively simple and participants G and H solved the problems without making any errors.

In response to this case study, we have extended AFID to verify suspected fault correcting source code changes with the developer before adding them to the repository. We made use of this functionality in the internal deployment of AFID described in Sect. 6.

5.4.5 Multiple corrections

When we manually reviewed the fault correcting source code changes, we noticed one source code change that contained corrections for many different faults. In this case, what happened was when the developer discovered the first fault, he realized he had made the same mistake two more times in the same method and corrected all instances of this mistake. We observed only a single instance of a source code change that corrected multiple faults. We foresee that future versions of AFID will allow a developer to note when the developer believes that a source code change corrects multiple fault instances.

5.4.6 Developer feedback

The user experience for AFID users is a concern for large user studies. After the user study, we asked the participants to provide feedback about their experience using the AFID monitoring tool. One participant commented that using the tool was unnoticeable as the user just used the regular javac and java commands. The participant thought the general experience was very good. One participant was "amazed … at how accurately AFID caught my critical bugs". Several participants noticed a slight delay when compiling programs. We plan to address this delay by performing both the repository updating and test case replaying in the background.

6 Real world experience

During the past several months we have used AFID internally in our group. Group members have used AFID to monitor the development of our research compiler and some of their course assignments. In this effort we have collected information about several real software faults. This effort exposed several usability issues with our original implementation and we have adapted the implementation to address these issues.

6.1 Faults recorded

Table 5 presents the faults we have recorded using AFID. We have recorded faults during the development of our group's research compiler (C1 through C15) and two class projects (P1 and H1 through H6). Our group's research compiler currently contains over 68,000 lines of Java and C code.

An examination of the faults reveals that AFID has recorded a rich set of faults. The faults include examples of common programming errors including negation of the condition in an if statement and errors in code to handle null pointers. The faults also include more complex algorithm specific errors in parsing code, type checking logic, and pointer analysis logic.

One potential concern with AFID's crash recording approach is whether it can record programming faults beyond simple bugs such as division by zero errors and missing null pointer checks. A quick review of the recorded faults reveals that AFID

Table 5 Faults collected

P1. Missing condition in if statement and error in array index
H1. Use of wrong variable
H2. Use of == instead of != in if statement
H3. Missing table lookup
H4. Missing bit shift
H5. Extra bit shift
H6. High level changes in use of array
C1. Missing null pointer check in printing code
C2. Parse tree traversal bug when generating AST
C3. Logic bug about which allocation site to analyze
C4. Parse tree traversal bug when generating AST
C5. Null pointer check when flattening AST
C6. Null pointer check when flattening AST
C7. Null pointer check when comparing specificity of methods
C8. Negated condition in if statement
C9. Missing if condition in array type checking code
C10. Error in handling null in testing equivalence
C11. Error in ordering of operations when mutating graph
C12. Cast to the wrong class
C13. Omission of adding node to set to visit
C14. Missing null pointer check
C15. Large logic change

recorded a rich set of software faults. The intuition why AFID can record complex faults is that AFID can record faults that break subtle program invariants because these program invariants are implicitly checked by other parts of the program. When an invariant is violated, these implicit checks cause the program to crash, and this crash is detected by AFID.

Examples of rich faults that AFID has recorded include bugs in the compiler code that generates the abstract syntax tree from the parse tree. Our compiler contained two bugs that improperly traversed the parse tree and resulted in errors. AFID was able to record both of these bugs.

AFID also recorded a fault in the compiler's loop optimization pass. This fault performs transforms to control flow graph that spliced in a loop header and added edges in the wrong order.

In monitoring the development of our research compiler, AFID has recorded both recently introduced faults introduced and long lived faults. While many of the recorded faults were recently, AFID did record long lived faults. In particular, fault C9 in the type checking code had existed for over two years.

An examination of the recorded faults reveals them to be significantly richer than those generated by automated fault injection strategies. While we did observe simple faults such as negating conditions, missing statements, and missing null pointer checks in our fault collection, much of our record fault data set was made up of more subtle faults.

6.2 Lessons learned

We learned a great deal in the process of deploying AFID in our internal development environment.

6.2.1 Build processes

The first lesson is that real world build processes are complex. Our compiler's build process calls javacup to build the java source files for the parser from a cup grammar, then makes multiple invocations to javac, and finally calls javadoc to generate documentation. One problem is that AFID assumes that it can check test cases after the execution of the javac. During our build process, the Java class files may not be completely built until the make file performs the final invocation of javac. This example highlights the need for AFID to provide flexible options that can be used to support a wide range of different build setups.

6.2.2 Development environments

Another lesson is that real world development environments are complex. The students in our group use a wide range of development environments including vi, emacs, eclipse, and netbeans. The initial version of AFID assumed that the compiler would provide a terminal window to allow the developer to tell AFID whether a change was likely to correct a fault. Unfortunately, environments like Eclipse or NetBeans do not provide a terminal window.

6.2.3 Compilation times

During compilation, our initial version of AFID required developers to wait while it executed its testcases. We found that this version AFID's replay takes too long. Our group develops a number of long running analyses. When they fail and generate testcases, running these testcases can take quite some time. Moreover, we have discovered that developers find any extra time waiting for compilation to be distracting.

6.2.4 Forgetting to use AFID

In our experiences, developers often forget to turn AFID on. We have found the best course of action was to setup AFID to always run. The idea is to include an AFID configuration file in the root of the project that AFID is monitoring that tells AFID to monitor the compilation and execution of programs in this part of the directory hierarchy.

The concern then becomes that the developer may forget that AFID is on and accidental disclose private information. This could happen if the developer inputs their personal information into the program under development and the same execution reveals a program fault. In this case, AFID would report a test case that contains the developer's personal information. To make the execution monitoring obvious, AFID now prompts that it is running whenever it monitors the compilation or execution of a program.

6.2.5 Applicability

We have not been able to apply AFID to all of the projects in our group. Some of these projects are performance sensitive. In these projects we are concerned with developing a precise understanding of their performance, and do not wish to introduce performance changes by monitoring them with AFID. Other projects in our group make extensive use of high-bandwidth, latency sensitive network communications, and therefore are not good candidates for AFID. However, in spite of these limitations we have been able to use AFID to monitor the vast majority of our group's software development effort.

6.2.6 Network file system

Our group's development environment consists of a cluster of networked workstations all of which mount a networked filesystem with the user directories. One issue with using in AFID in this environment is that if a developer uses AFID on two different machines, the list of filenames to exclude can be incorrect. We have updated AFID to create a list of files to exclude on each host that it runs on.

6.2.7 Privacy

In our internal development, the hypothetical privacy concerns we discussed earlier have not occurred. The compiler code is freely available and the inputs are typically publicly available benchmarks. A few students have been unable to use AFID due to work involving confidential code. We expect that in practice, confidential code is likely to present a larger challenge.

6.3 Extensions

We next discuss how we extended AFID to address the issues that we discovered while using AFID internally.

6.3.1 Complex build processes

There are two approaches to support complex build processes with AFID. The straightforward approach is to simply wrap the make command with AFID's compilation monitor. We have found that it is sometimes useful to simply modify the make file to explicitly support AFID.

We have also extended AFID's compilation monitor to support two modes: a compilation monitor mode that simply updates the repository without replaying test cases and the normal mode that both updates the repository and replays the test cases.

To address the wide range of development environments some of which provide a terminal window and some of which provide X-windows access, we have extended AFID to use X-windows when available to create a window to ask the use, and to use the terminal when the X-windows support is not available. One of the primary advantages of X-windows support is that AFID can display user dialogs after returning control of the console, and therefore it enables AFID's compilation monitor to perform time consuming operations in the background.

6.3.2 Improved sandbox

In the earlier version, we sandboxed only the files that the original execution accessed. We have found that in practice either fault corrections or other source code changes often cause replayed test cases to generate new output files in the developer's directories. The creation of these output files by the replay component is distracting at best and at worse has the potential to overwrite important files. To address this issue, we have improved AFID's sandbox to sandbox all files that replayed executions write using the same `ptrace`-based technique.

6.3.3 Manual control

In the process of using AFID, we have found times when limited manual interaction was useful. We have observed cases in which over time erroneous inputs cause AFID to store a large number of unresolved test cases. To address this issue, we have provided a mechanism that allows developer to periodically flush the test case archive. We have also found that bad fault information was occasionally uploaded to the server. We have found it straightforward to manually review the faults and remove any faults that are problematic.

7 Related work

Researchers have recently developed tools to mine CVS repositories to collect some of this information (Nagappan et al. 2006; Williams and Hollingsworth 2004;

Neuhaus et al. 2007). The CVS mining research identifies CVS commits that correct software faults through a heuristic analysis of the CVS checkin comments. Researchers have discovered many interesting properties including that code changes on Fridays are more likely to cause problems (Śliwerski et al. 2005). Other research discovers implicit interface rules by searching for code changes that occur together (Livshits and Zimmermann 2005). The primary way that our work differs from previous work on CVS mining is that our work provides fault revealing test cases in a format suitable for automated tools. The extra information provided by these test cases will enable empirical software research to explore software faults in new ways—for example, the test cases will enable researchers to use dynamic analyses to explore the faulty executions.

Developers sometimes commit CVS updates that both correct a software fault and make other changes. Traditional CVS mining techniques do not distinguish between the fault correcting changes and other bundled changes and therefore can extract software fault corrections that are too large. Developers typically compile their code more often than they commit changes to a code repository. As our work is likely to log changes to the code base more frequently, it has the potential to more precisely characterize the changes that correct a software fault. We note that existing techniques such as delta debugging used in conjunction with CVS mining could help to minimize the failure producing changes (Zeller 1999).

The Marmoset project course submission system records snapshots of student's code development (Spacco et al. 2005). While both systems can collect information about software faults, they target different development environments. The two systems differ in how they detect which source files comprise an application. Marmoset functions as a plugin to Eclipse and can therefore use Eclipse's internal project management functionality to detect source files, while AFID attempts to be compatible with all build environments. A more critical difference is that Marmoset uses a set of test cases provided by an instructor while AFID must monitor the executions of an application to collect fault revealing test cases.

Researchers have also developed data sets of applications with seeded faults (Do et al. 2005). These data sets are limited in size because they are labor intensive to create—researchers must manually seed faults and create test cases that reveal these faults. While these data sets have proven to be a useful tool, potential differences between seeded faults and real software faults can threaten the validity of experiments. Moreover, because the software faults are seeded, the data set does not contain information that can be mined to learn about real-world software faults.

The iBUGS project is based on the observation that after developers correct a bug, they often add regression tests designed to ensure that future changes do not reintroduce similar bugs (Dallmeier and Zimmermann 2007). Their approach searches CVS commit messages for text that indicates that the change corrects a bug. They then build pre-fix and post-fix versions of the application and run the versions on the test suite to identify any test cases that reveal the given fault. They have successfully used this technique to build a repository of software bugs.

BugBench is a collection of large scale programs and test cases to trigger bugs (Lu et al. 2005). The collection contains 19 bugs from 17 different applications. These bugs include 13 memory-related bugs, 4 concurrent bugs, and 2 semantic bugs. AFID could automate the construction of such test suites—we note that many of the bugs in BugBench cause crashes and therefore AFID's crash detection technique would work for these bugs.

Researchers have developed many replay systems for debugging applications (Choi and Srinivasan 1998; Steven et al. 2000; LeBlanc and Mellor-Crummey 1987). These other systems replay the exact execution, often with the goal to help developers deterministically replay software bugs in multi-threaded programs. AFID's goal is to execute new versions of the application on the same test case. As a result of these goals, the two system designs are very different. Replay systems incur significant overheads to ensure that they replay the execution of threads in the exact same order. Because AFID must support replaying a test case on a modified version of the program, there cannot be a similar notion of preserving the exact order that threads execute in. Replay systems can simply record the exact outputs of the sequence of system calls an application makes while AFID must replay a test case even if an application has been modified to perform system calls in different orders.

AFID relies on the `ptrace` interface to monitor both application compilation and execution. Researchers have used the `ptrace` interface to inject faults into applications (Some et al. 2001) and to safely execute untrusted code (Sekar et al. 2003). Researchers have also used similar program monitoring techniques to implement user space file systems (Spillane et al. 2007).

Cooperative Bug Isolation monitors the execution of applications by the end user to provide the developer with information to help isolate and correct bugs (Liblit et al. 2003). CBI is constrained in that it must make strong guarantees about maintaining end user's privacy. We expect that adapting techniques like AFID to monitor end users could be very useful for studying and replicating software bugs if the considerable privacy concerns could be addressed. Automatically collecting enough information to replicate software bugs without divulging personal information remains an open problem.

Techniques based on symbolic execution can be used to generate test cases that drive an application into a failing state (Cadar and Engler 2005). AFID differs from these techniques in that it simply monitors the inputs to actual program executions and if the program crashes uses these inputs to generate a test case that reproduces the failure while these other techniques attempt to find unknown bugs.

This paper extends our previous work on AFID (Edwards et al. 2008) with our experiences using AFID in real world development. This experience has led to an evolution of the basic technique to improve its usability in the real world. It has also validated that the approach is a viable approach to collect data on software faults.

8 Conclusion

Data sets of real software faults have the potential to enable the creation of new tools for software engineering and programming language researchers. Our previous experience shows that manual efforts to collect such data are tedious. The AFID tool is a new approach for recording software fault data. A key benefit of AFID is that the data it collects includes fault revealing test cases in addition to a faulty version of the application and the fault correcting source code change. This key results include (1) a technique to automatically record software faults without requiring developer intervention, (2) the implementation of this technique in the AFID tool, (3) an evaluation of the overhead of these techniques, (4) our experiences using the tool to record real software faults, and (5) our experiences using AFID in the daily development environment of our research group, and (6) how we have improved AFID in response to these experiences. Our study results indicate that AFID can automatically record software faults and we continue to build a repository of software faults.

Acknowledgements This research was supported by the National Science Foundation under grants CCF-0846195 and CCF-0725350. We would like to thank the anonymous reviewers for their helpful comments.

References

Cadar, C., Engler, D.: Execution generated test cases: how to make systems code crash itself. In: Proceedings of the SPIN Workshop, pp. 2–23 (2005)

Chacon, S.: Git—The fast version control system. http://www.git-scm.com/ (2010)

Choi, J.D., Srinivasan, H.: Deterministic replay of Java multithreaded applications. In: Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools, pp. 48–59. ACM, New York (1998)

Collins-Sussman, B.: The subversion project: building a better CVS. Linux J. **2002**(94), 3 (2002)

Dallmeier, V., Zimmermann, T.: Extraction of bug localization benchmarks from history. In: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering, pp. 433–436 (2007)

Do, H., Elbaum, S., Rothermel, G.: Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact. Empir. Softw. Eng. Int. J. **10**(4), 405–435 (2005)

Edwards, A., Tucker, S., Worms, S., Vaidya, R., Demsky, B.: AFID: an automated fault identification tool. In: Proceedings of the 2008 International Symposium on Software Testing and Analysis, pp. 179–188 (2008)

Haardt, M., Coleman, M.: Ptrace(2). Linux programmer's manual, Section 2 (1999)

LeBlanc, T.J., Mellor-Crummey, J.M.: Debugging parallel programs with instant replay. IEEE Trans. Comput. **36**(4), 471–482 (1987). doi:10.1109/TC.1987.1676929

Liblit, B., Aiken, A., Zheng, A.X., Jordan, M.I.: Bug isolation via remote program sampling. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, pp. 141–154 (2003)

Livshits, B., Zimmermann, T.: Dynamine: finding common error patterns by mining software revision histories. In: ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 296–305 (2005)

Lu, S., Li, Z., Qin, F., Tan, L., Zhou, P., Zhou, Y.: BugBench: Benchmarks for evaluating bug detection tools. In: Proceedings of the Workshop on the Evaluation of Software Defect Detection Tools (2005)

Nagappan, N., Ball, T., Zeller, A.: Mining metrics to predict component failures. In: Proceeding of the 28th International Conference on Software Engineering, pp. 452–461 (2006)

Neuhaus, S., Zimmermann, T., Holler, C., Zeller, A.: Predicting vulnerable software components. In: Proceedings of the 14th ACM Conference on Computer and Communications Security, pp. 529–540 (2007)

Sekar, R., Venkatakrishnan, V.N., Basu, S., Bhatkar, S., DuVarney, D.C.: Model-carrying code: a practical approach for safe execution of untrusted applications (2003)

Śliwerski, J., Zimmermann, T., Zeller, A.: When do changes induce fixes? On Fridays. In: Proceedings of the International Workshop on Mining Software Repositories, pp. 1–5 (2005)

Some, R.R., Kim, W.S., Khanoyan, G., Callum, L., Agrawal, A., Beahan, J.J.: A software-implemented fault injection methodology for design and validation of system fault tolerance. In: Proceedings of the 2001 International Conference on Dependable Systems and Networks, pp. 501–506 (2001)

Spacco, J., Strecker, J., Hovemeyer, D., Pugh, W.: Software repository mining with Marmoset: an automated programming project snapshot and testing system. In: Proceedings of the Mining Software Repositories Workshop (2005)

Spillane, R.P., Wright, C.P., Sivathanu, G., Zadok, E.: Rapid file system development using ptrace. In: Proceedings of the 2007 Workshop on Experimental Computer Science, pp. 1–13 (2007)

Steven, J., Chandra, P., Fleck, B., Podgurski, A.: jRapture: a capture/replay tool for observation-based testing. In: Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 158–167. Portland, OR, USA (2000)

Williams, C., Hollingsworth, J.K.: Bug driven bug finders. In: Proceedings of the International Workshop on Mining Software Repositories, pp. 70–74 (2004)

Zeller, A.: Yesterday, my program worked. Today, it does not. Why? In: Proceedings of the 7th European Software Engineering Conference/7th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 253–267 (1999)