AFID: AN AUTOMATED FAULT IDENTIFICATION TOOL

Edwards, A; Tucker, S; Worms, S; Vaidya, R; Demsky, B

Introduction: Current Fault Detection

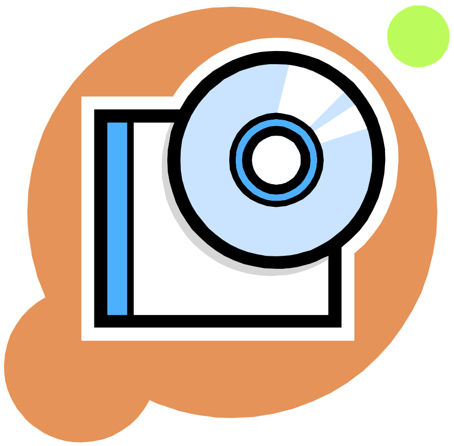
- Traditional approach to evaluate tools
  - Hand-selected & seeded faults
  - Synthetically-injected faults
- Must *still* provide proof tool
  - Does not miss important faults
  - Discovers both real and important faults
- Community avoids large fault data sets
  - Few datasets available
  - Lack of test cases to reproduce results and reveal faults

Related Work

- CVS Repository mining [spacco05, nagappan06, williams04, ying04 nehaus07]
  - Code Correcting commits v. General Application Additions
- Sets of Applications with Seeded Faults [do05]
  - Real v. Seeded software faults
- iBUGS [dallmeier07]
  - Regression testing and software bug repository
- Replay systems [choi98, steven00, leblanc87]
  - Exact execution and deterministic replays

Importance of Fault Data Sets

- Extract several real instances of practical faults
- Lead to creation of sophisticated analyses
- Use by researchers to evaluate their tools



Solution Ideology

- Remember: most existing data sets lack test cases to reveal faults
  - Manually create data set of real software faults
- Record:
  - Test cases that reveal the fault
  - Copy of source code that contained fault
  - Source code that change/removed fault

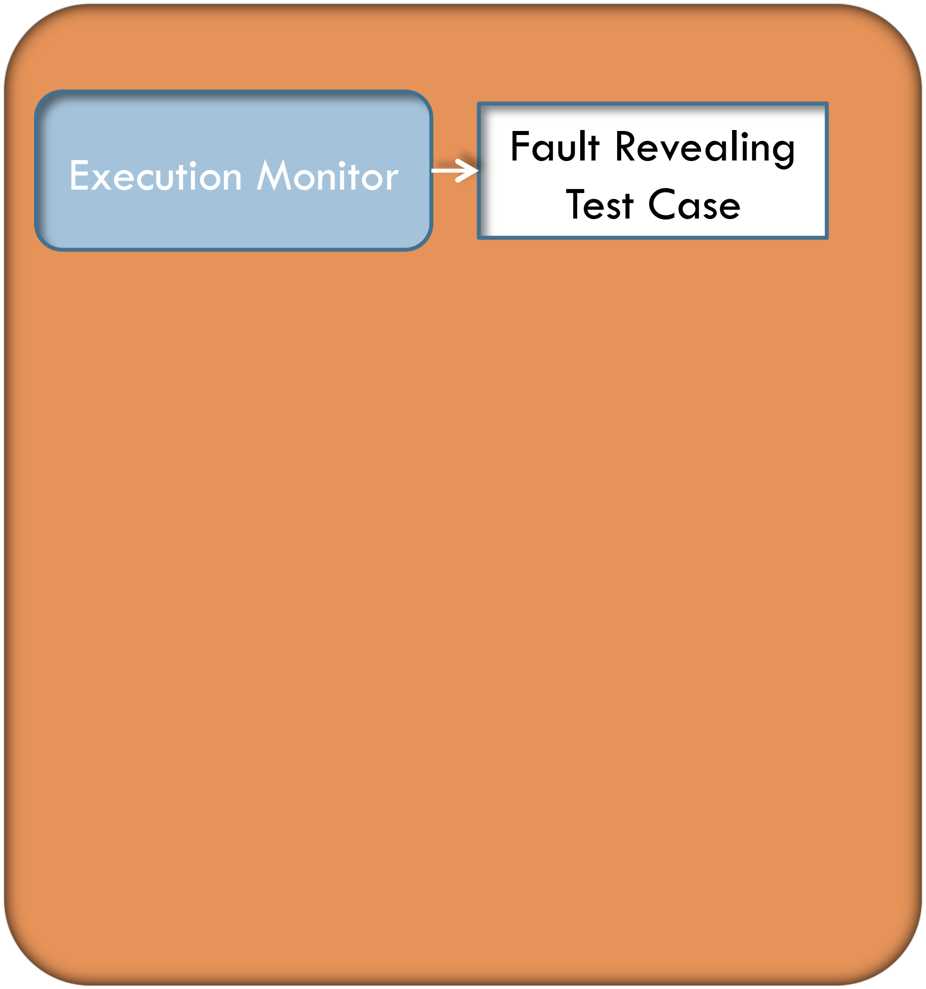
Introduction to the AFID System

- Collect complete information for software faults
  - Wide range of developers
  - Real projects
- Automatically records software faults
  - Monitoring the compilation and execution steps of the software development process
  - Record as much as possible
  - Minimal runtime overheads

Automating Ideology with AFID

Execution Monitor

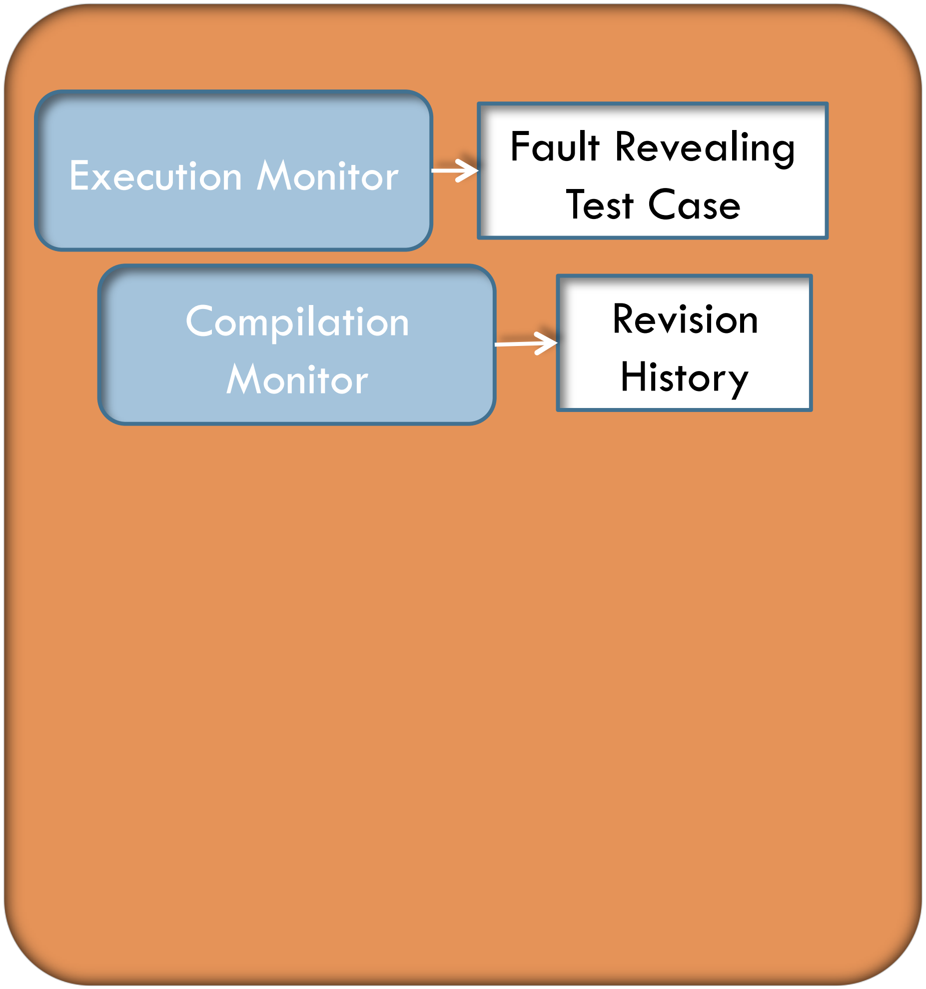
- Traces application execution
- Records input
  - ↳ create test case emulating failure
- Records
  - Test case containing input-revealing fault
  - Source code *version ID* where fault discovered


Execution Monitor → Fault Revealing Test Case

Automating Ideology with AFID

Compilation Monitor

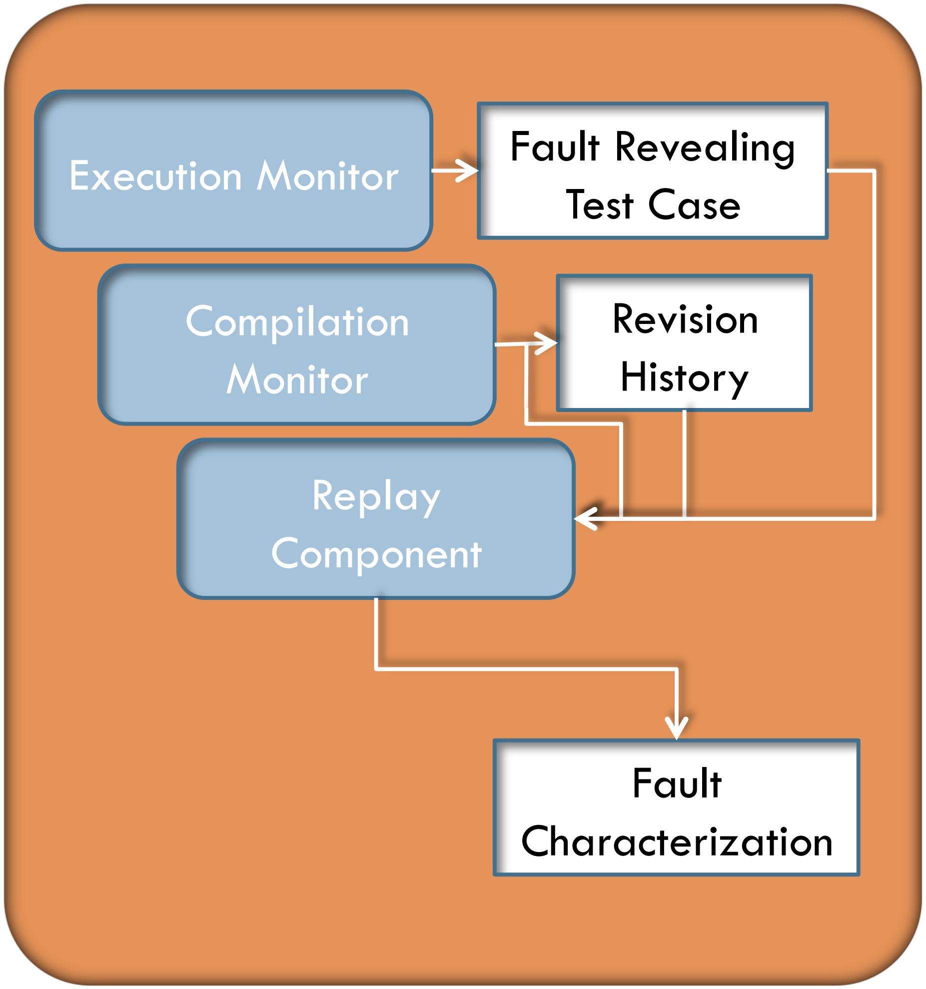
- Traces compiler execution
- Records
  - Any new source files discovered
  - All source files edited since last compilation
- Updates subversion repository



Automating Ideology with AFID

Replay Component

- Executes newly compiled application
- If no test cases crash
  - Records version ID as fault correcting code
  - Marks test case as resolved



Replaying Test Cases: Sandboxing Replay

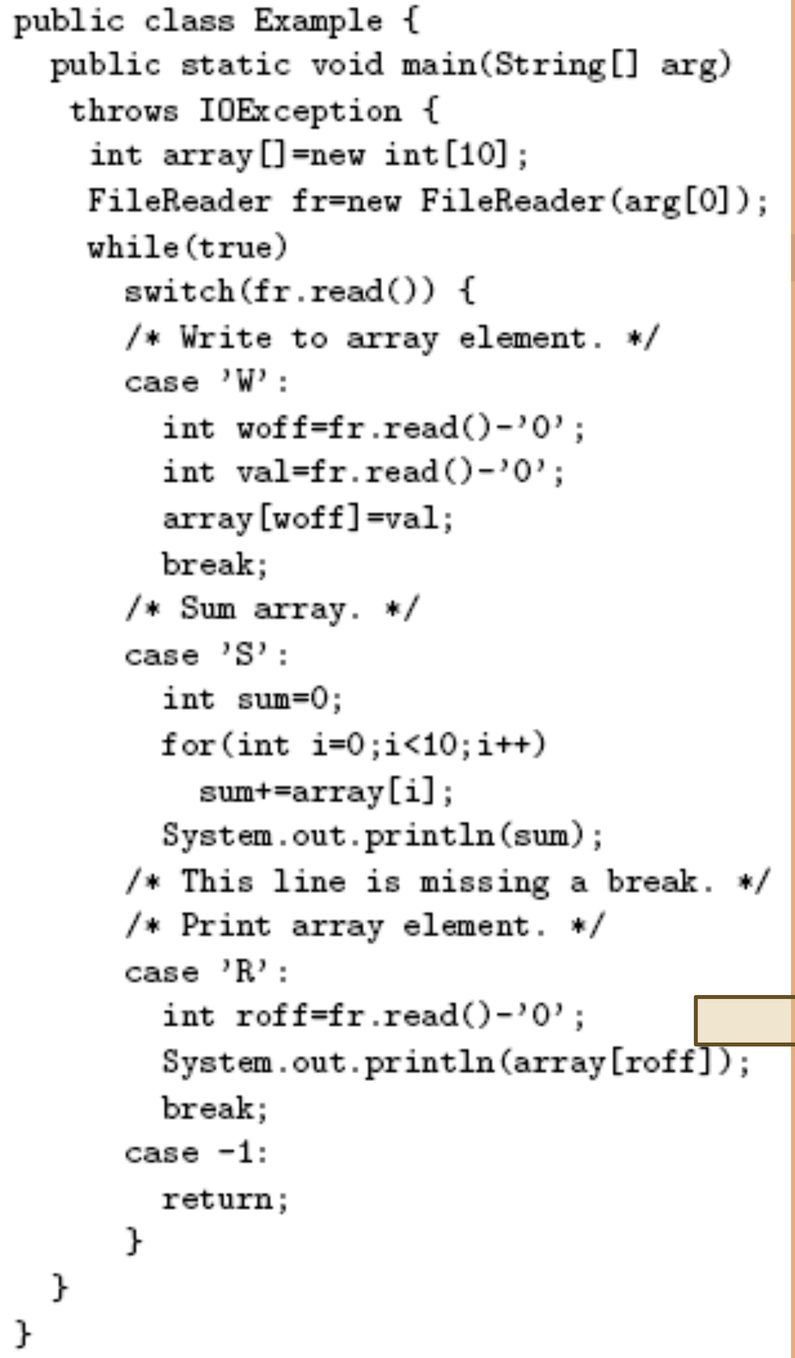
- Intercepts open(`file`) requests
  - Test case file request – redirect to file in test case
  - Excluded file – pass unmodified request to OS
- Modified application/Corrected fault
  - Modify R.C. to copy test case/external file
- *Gives illusion that test case files in same location as original execution*
  - Reproduce faults that depend on exact location of input files

Replaying Test Cases: Termination

- Developer makes source code change that causes loop on unresolved case
- AFID records running times for each execution
  - Computes upper bound
  - Assumes program is looping when execution extends past upper bound
- Worst case:
  - Time-out incorrectly identifies looping → only fault correction unrecognized by AFID

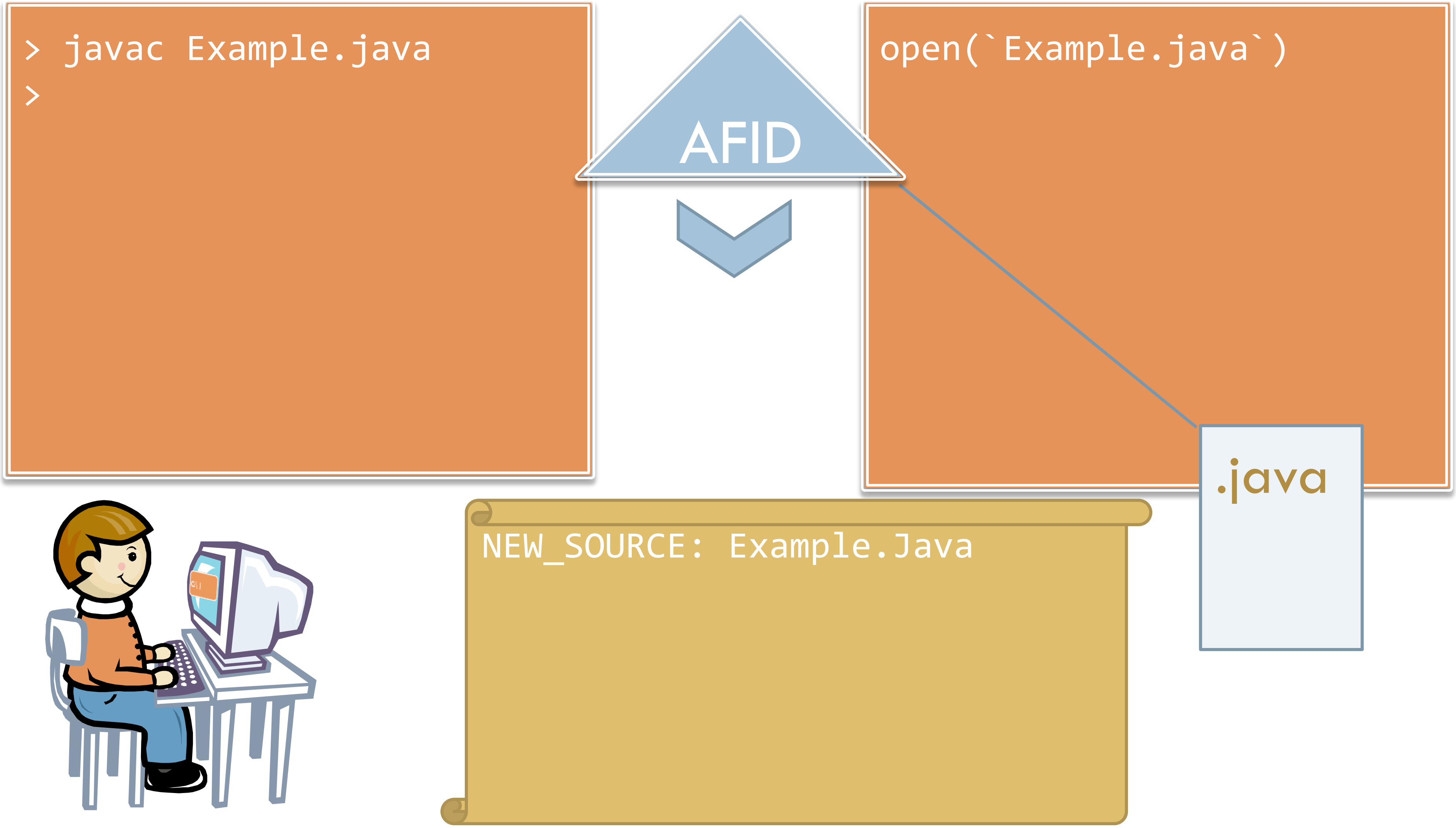
AFID: Qui

- Sample Jav
  - Input: Com parameter of file
  - Execution
    - Open File
    - Reads seri commands
      - Write element
      - Sum ar
      - Print ar



```
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;
      /* Sum array. */
      case 'S':
        int sum=0;
        for(int i=0;i<10;i++)
          sum+=array[i];
        System.out.println(sum);
      /* 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;
      }
  }
}
```

AFID: Monitoring Compilation



```
> javac Example.java
>
```

```
open(`Example.java`)
```

```
NEW_SOURCE: Example.Java
```

AFID: Monitoring Program Execution

```
> javac Example.java
> java Example input.txt
>
```

AFID

W23SR2

```
open(`Example.java`)
open(`input.txt`)
read('W')
read('2')
    '3')
    (&a , 2, 3)
    'S')
    a)
    ('R')
    -1)
```

ERCODE = -1

CRASH!!

```
NEW_SOURCE: Example.Java
CMD: java Example input.txt
ERCMD: java Example input.txt
CPY: input.txt > avid_input.txt
STR: MAP(PATH(input.txt),
   PATH(afid_input.txt))
```

AFID: Detecting Fault Corrections

```
> javac Example.jav                                   mple.java`)
> java Example input                                  d_input.txt`)
> javac Example.jav
> java Example input
                                                      2, 3)

                                                      2)
```

At this point, AFID has collected:

(1) The buggy version of the example program

(2) The test case that reveals a fault in the buggy version of the program

(3) A diff that gives the source code change that corrects the fault

- (a) Replacing line 20th line in the break

(4) Addition to a fine grained revision history

After recording this fault information AFID uploads the information (optionally) to a centralized fault repository.

```
xt
PATH(afid_input.txt))
REPLAY
```

NEW CODE CHNG: 20

REPLAY

GOOD RUN

The AFID Server

- Web based server application
- Aggregates discovered faults by AFID client
  - Automatic/Manual upload after recovery
- Fault Upload Contents
  - Test Case
  - Version ID for source code version whose execution generated the fault-revealing test case
  - Version ID for fault-correcting code
  - Latest version of AFID's internal subversion repository

Recording Test Cases

- Execution Monitor
  - Forking off new child process
    - Child calls ptrace() with PTRACE_TRACEME
    - Child calls exec() to execute application
      - Causes previous ptrace() with PTRACE_TRACEME to stop before executing new application
  - Monitoring process calls ptrace() with PTRACE_SYSCALL and calls wait()
    - OS wakes monitoring process when child makes system call and suspends the child process

Recording Test Cases (cont.)

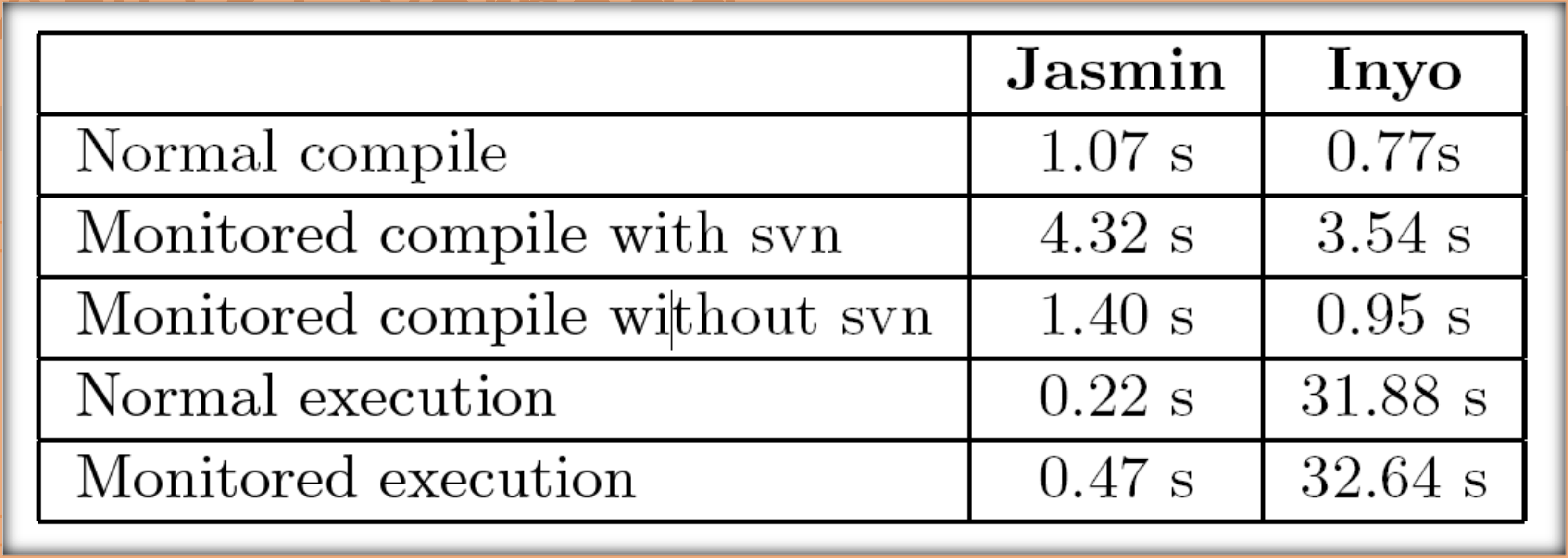
- Monitor awaken → calls ptrace() with PTRACE_GETREGS
  - If child calls open(file), monitor inspect file/access mode by calling ptrace() with PTRACE_PEEKDATA
    - WRITE – make copy of file (immediately)
    - READ – lazy copy

- Monitored application exits
    - Monitor inspects return value for crash
    - On crash – monitor copies all files read by application
    - Stores mapping between application file pathnames and files' copies in text file in test case
- ptrace(), ptrace(), ptrace()

Cleaning Up Records

- User Interaction – fuzzy matching approach
  - Generalization as application output changes
- Duplicate Test Cases
  - Storing multiple copies of same test case
- Filtering Inputs
  - Reading extraneous files not really classified as "inputs"

AFID's Overhead



	Jasmin	**Inyo**
Normal compile	1.07 s	0.77s
Monitored compile with svn	4.32 s	3.54 s
Monitored compile without svn	1.40 s	0.95 s
Normal execution	0.22 s	31.88 s
Monitored execution	0.47 s	32.64 s

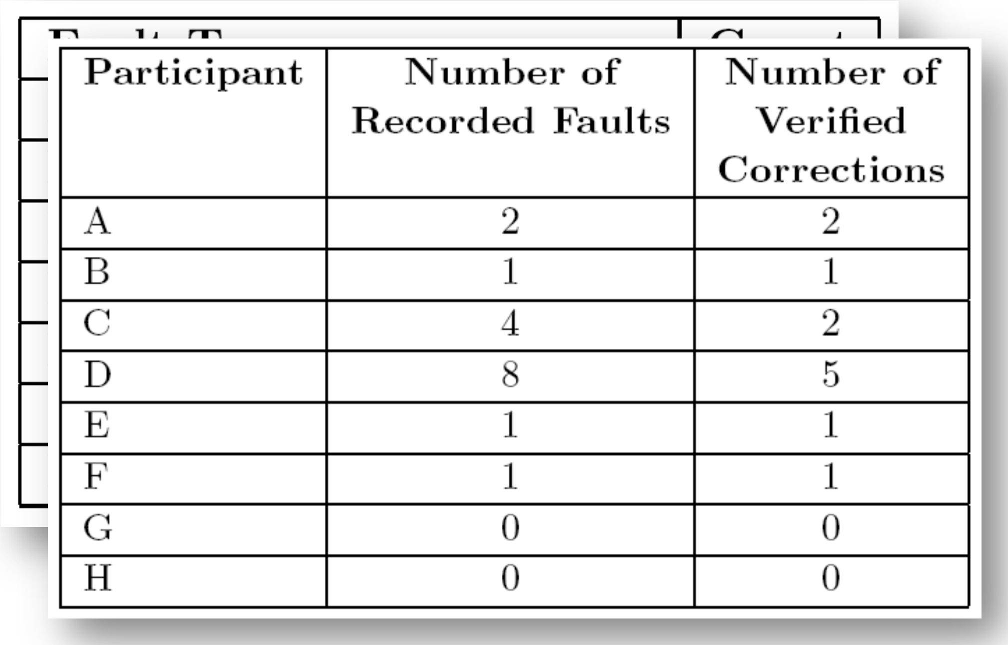
- Jasmin bytecode assemebler
- Inyo

Jasmin Monitoring Overhead – 113%

Inyo Monitoring Overhead – 2 %

Results

- Developer Population
- Methodology
- Fault Breakdown
- Fault Detection Errors
- Multiple Corrections
- Developer Feedback



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

This Work's Contributions

- Automated fault collection strategy
- Process monitoring technique
- Automated recording of test cases
- Monitoring overhead measurement
- Experience

Limitations and Future Work

Limitations

Future Work

- Allow a developer to note when the developer believes that a source code change corrects multiple fault instances
- Address compilation delay by performing both the repository updating and test case replaying in the background.