Tracking Down Software Program Bugs Using Automatic Anomaly Detection

Sudheendra Hangal, Processor Products Group, Sun

Monica S. Lam, Stanford University

ICSE-2002

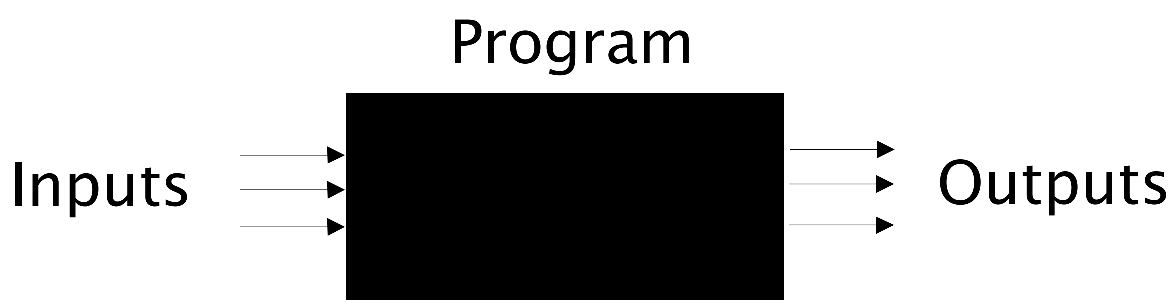
~~Buenos A~~ Orlando, Florida

Outline

- Background & Motivation
- Past Work
- Dynamic Invariants
- DIDUCE
  - Dynamic Invariant Detection $\cup$ Checking Engine
- DIDUCE experiences
- Uses
- Conclusion, Q&A

Background

- **Software reliability is an increasing problem**
- **Traditional input-output interaction with a program is insufficient**



- **What happens inside the black box ?**
  - **Can cheap machine cycles help ?**

Motivation for DIDUCE

- **How do you debug...**
  - a program working correctly on some inputs, failing on others ?
    - "Hmm... What's different about runs which fail ?"
  - a program failing after a long time ?
  - a program you don't even know has a bug ?
  - Large programs written by others and evolved over time (with misleading comments!)
- **DIDUCE successfully tackled these problems on 4 different applications we tried**

Past Work

- Many bug detection tools
- Static approaches
  - e.g. Prefix, Metal, Vault, ESC, etc
  - Exhaustive, conservative
  - No need for test inputs
- Dynamic approaches
  - e.g. Purify, Eraser, assert's, ...
  - Need good test input set, observe only possible behavior

Invariant Specifications

- Many annotation based approaches
- Its manual work - users never do it!
- Usually incomplete
- Users may not even know the invariants...
- ... Or may know them wrongly!
- Programmers rely on passing a test-suite as proof that code works

Dynamic Invariant Detection

- Hypothesize a space of invariants associated with the program
- Test each hypothesis on program runs
  - rule out invariants which do not hold
- Prior work (Daikon) applied to small pro- grams
- Approach is automatic and pervasive...
  - ...but may be unsound!

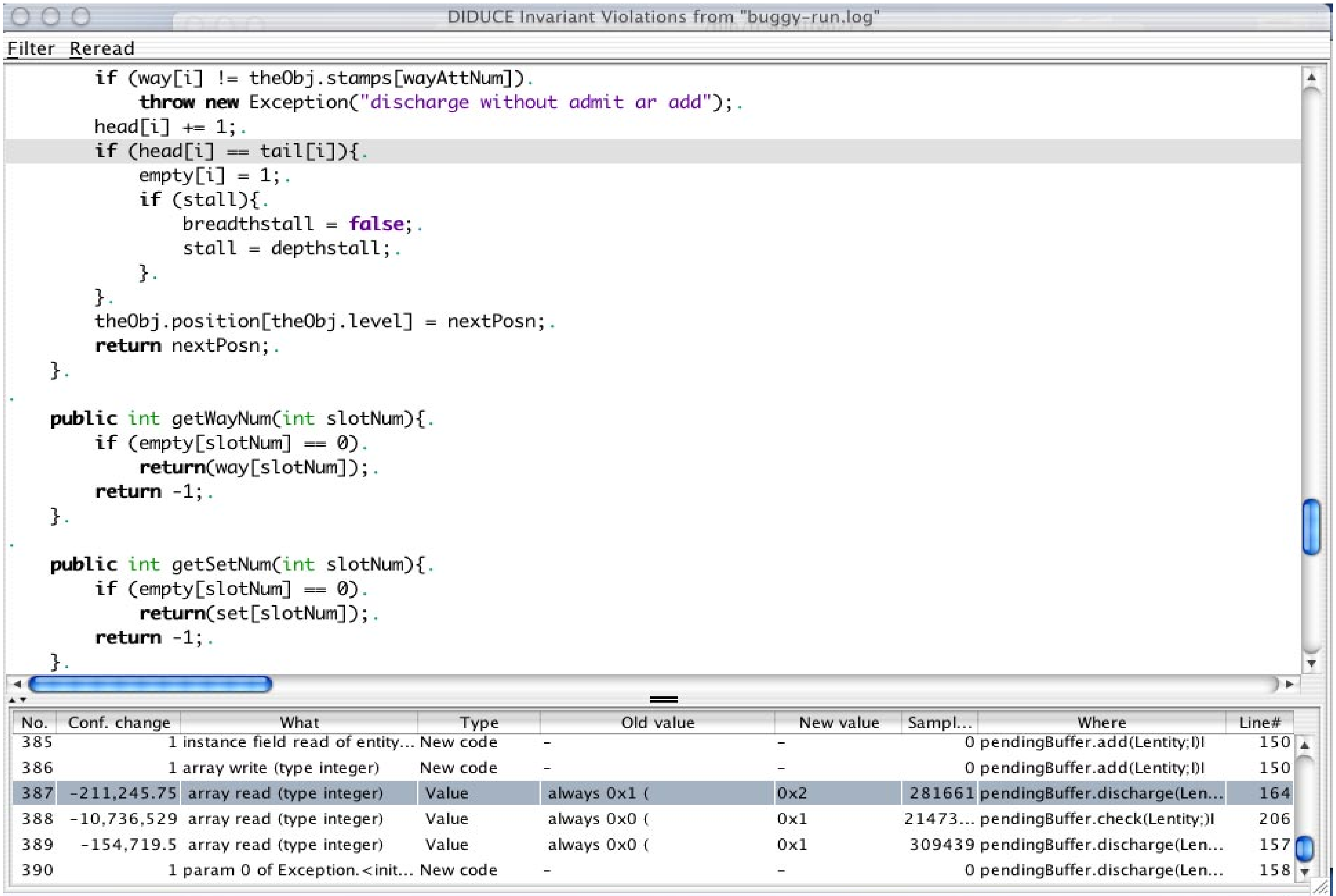
Dynamic Invariant Checking

- Idea: Close the loop
- Detect invariants on "presumed-good" runs
- Automatically check invariants on other runs
  - Report invariant violations
  - Refine invariants, as you check
- Invariant violations signal anomalies, e.g.:
  - New code executed
  - New values seen for variables

Dynamic Invariant Detection U Checking Engine (DIDUCE)

- Simple, practical, and effective tool
- Adds instrumentation to Java bytecode
  - To deduce and check invariants on the fly
  - Works with any compliant JVM
- 2 modes - training and checking
  - Invariant violations suppressed in training mode
  - Training continues in checking mode
- Confidence level associated with invariants and violations

GUI Screenshot



DIDUCE Invariants

- Values at some types of program points
  - Object reads/writes
  - Static var read/writes
  - Method call sites and return values
- Invariants associated with Tracked Expressions (TEs)
  - Value accessed
  - Change in value (for writes)
  - Runtime type of object being accessed
- Requires only "local" computation

Tracked Expressions

Source code

Tracked Expressions

```
Class SomeClass {
 static int x;
 int y;
 ...
}
```

```
Object o = new SomeClass();
Object arr[] = new SomeClass[3];
...
```

Tracked Expressions

Source code

```
Class SomeClass {
 static int x;
 int y;
 ...
}
```

```
Object o = new SomeClass();
Object arr[] = new SomeClass[3];
...
// static field write
 SomeClass.x = ...
```

Tracked Expressions

```
SomeClass.x
SomeClass.x' -  SomeClass.x
```

For writes, the variable name (e.g. o.x) refers to its value before the write, while the name with a ’ suffix (e.g. o.x’) refers to its value after the write.

Tracked Expressions

Source code

```
Class SomeClass {
 static int x;
 int y;
 ...
}
```

```
Object o = new SomeClass();
Object arr[] = new SomeClass[3];
...
// static field write
 SomeClass.x = ...

// object field read
 ... = o.y;
```

Tracked Expressions

SomeClass.x
SomeClass.x' - SomeClass.x

o.y
o

For writes, the variable name (e.g. o.x) refers to its value before the write, while the name with a ’ suffix (e.g. o.x’) refers to its value after the write.

Tracked Expressions

Source code

```
Class SomeClass {
 static int x;
 int y;
 ...
}
```

```
Object o = new SomeClass();
Object arr[] = new SomeClass[3];
...
// static field write
  SomeClass.x = ...

// object field read
  ... = o.y;

// object array write
  arr[i] = ...
```

Tracked Expressions

```
SomeClass.x
SomeClass.x' -  SomeClass.x
```

```
o.y
o
```

```
arr[i]
T(arr[i]') == T(arr[i])
```

For writes, the variable name (e.g. o.x) refers to its value before the write, while the name with a ’ suffix (e.g. o.x’) refers to its value after the write.

Tracked Expressions

Source code

```
Class SomeClass {
 static int x;
 int y;
 ...
}
```

```
Object o = new SomeClass();
Object arr[] = new SomeClass[3];
...
// static field write
  SomeClass.x = ...

// object field read
  ... = o.y;

// object array write
  arr[i] = ...

// procedure call
  foo (x+1, o);
```

Tracked Expressions

Source code	Tracked Expressions
// static field write SomeClass.x = ...	SomeClass.x SomeClass.x' - SomeClass.x
// object field read ... = o.y;	o.y o
// object array write arr[i] = ...	arr[i] T(arr[i]') == T(arr[i])
// procedure call foo (x+1, o);	x+1 o return value of foo

For writes, the variable name (e.g. o.x) refers to its value before the write, while the name with a ’ suffix (e.g. o.x’) refers to its value after the write.

Default Invariant Representation

- Compact, for speed
  - Space overhead $\propto$ static size of program
- For each tracked expression
  - Convert to integer
    - References map to hashcode of class name
  - Each expression stores sample value and mask
    - Mask tracks which bits have remained invariant
  - Meeting value incompatible with current hypothesis relaxes the mask/invariant
    - Mask can be relaxed up to #bits times

Invariant Confidence

- Defined as:
  - # Samples/# of bits marked invariant by mask
- High if same value observed many times, or with small number of bits change
- Users look at invariant violations with large confidence changes first

User Extensions

- Default modes work well in practice
  - our experiments run in this mode
- User can change defaults
  - based on class, method, target field/method name, read v/s write, type of value accessed, line number...
  - Change set of tracked expressions
  - Change invariant representation
  - Change confidence computation
- Writing a new invariant is 20-30 LoC
  - Examples: Min/Max values, Self loops

DIDUCE Experiences

Program	Lines of code	# Classes instr.	# Program points instrumented	Slowdown
Sun MAJC MP Simulator	3300	10/28	3204	8–12 (10 proc)
MailManage + Javamail lib (SourceForge)	21700	214 /214	13014	6
JSSE lib (Java Secure Sockets Layer)	30000 +RSA libraries	384 /384	34844	8
Joeq JVM+JIT (SourceForge)	31500	18/137	3371	20

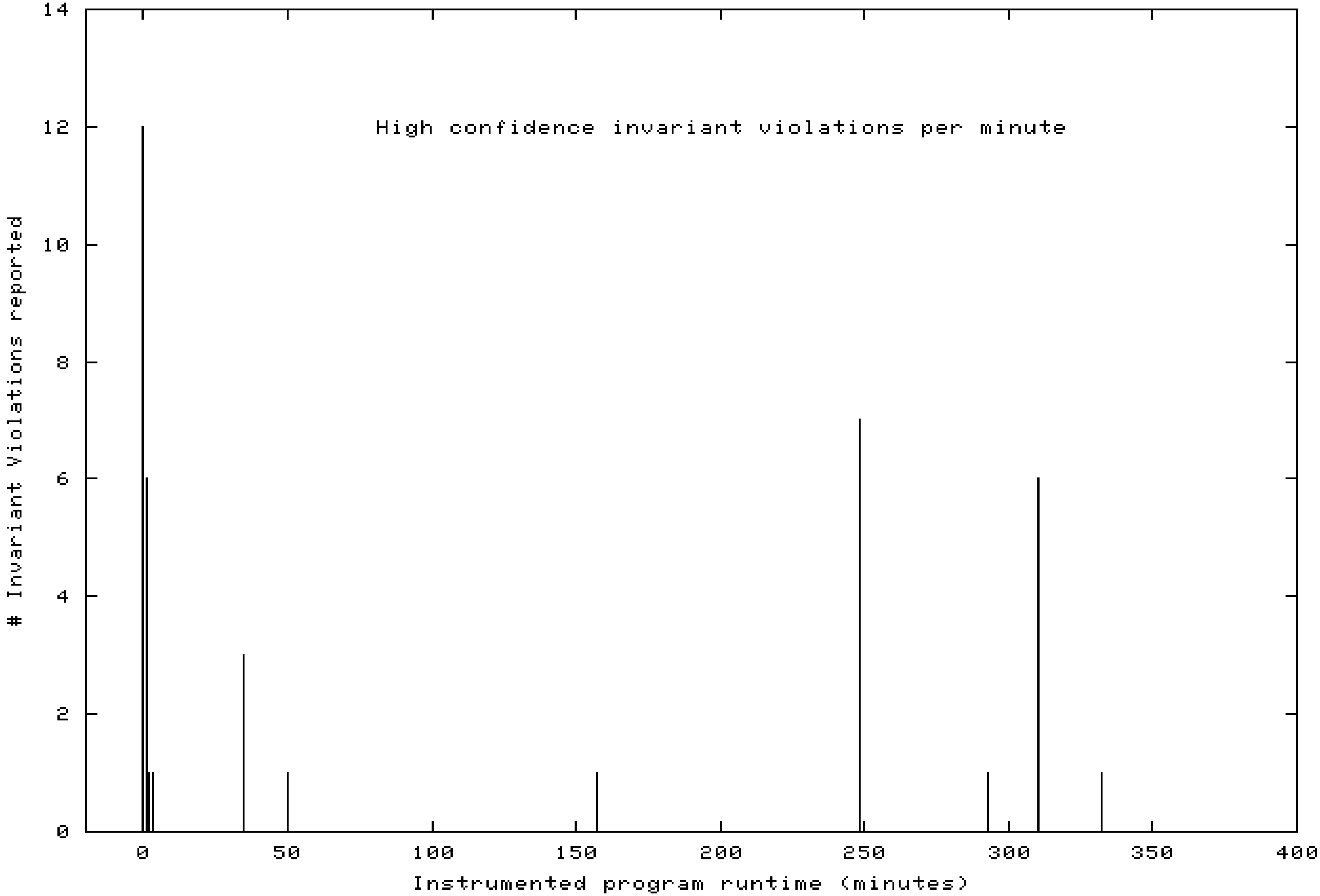
MAJC Simulator

- Simulator stable, in active use
- DIDUCE used initial part of run for training; ignored violations in this phase
- DIDUCE found 2 otherwise undetected errors
- DIDUCE accurately root-caused 3 errors
- Reported violation on 1 error was missed
  - Happened early, incorrect model was built!
- Reported 10 corner cases, all interesting to programmer!

Finding Simulator Errors

```
for (replaced = 0; replaced < assoc; replaced++) {
    // Bug – shd have checked for 0 or 2
    if (status[replaced][curset] == 0)
        break;
}
```

- **Occurred after 1 hour of uninstrumented execution time**
- **Another error detected even later: store to cache line in invalid state**


High confidence invariant violations per minute
Invariant Violations reported
Instrumented program runtime (minutes)
0
2
4
6
8
10
12
14
0
50
100
150
200
250
300
350
400

Debugging Known Errors

- Correctly root-caused 3 errors which resulted in assertion failures
- One of these was extremely hard to under-stand for programmer
  - Occurred after an hour of execution time
  - Tried several iterations of debugging, gave up
- Ran DIDUCE overnight:
  - 6 invariant violations reported just before error
  - Precisely pinpointed failing scenario

JSSE Example

- Java Secure Sockets Extension (JSSE) v. 1.0.2 code, ships with JDK 1.2/1.3
- Programmer tried adding a proxy server, changed timings
- Saw intermittent failures
- Spent 2 days working backwards from failure to root cause, through unfamiliar code

JSSE Bug

```
InputStream s = x;
// x is instance of SocketInputStream
if (...) {
   int len = ...; // const expr. = 74;
   byte[] hdr = new byte[len];
   s.read(hdr);
}
```

- **On passing runs, invariant on return value of read() as 74**
- **Failing run had a different value!**
- **Original programmer misunderstood contract with InputStream.read(byte[]) - not guaranteed to completely fill array**

Observations

- Invariant violations tend to occur in clumps
- Many clues due to an anomaly
  - Can trade-off accuracy for overhead
- Debug forwards
  - Source of error may be far away from failure
- Debug backwards:
  - trail of interesting events along the way
- Random data tends to have low confidence

Noise and Overhead

- Users can select which program points to watch
  - Reduces instrumentation overhead
  - Also reduces noise due to unimportant invariants
- Overhead parallelized by using different machines with different parts instrumented
- No overhead on network or I/O operations

Summary

- Need automatic bug detection and debugging tools
- User assertions are insufficient
- DIDUCE approach finds hard bugs in large programs, real-life situations
- No up-front investment required
- Works for different application domains
- Knowing corner cases gives programmer better feel for what the program is doing

Uses (1)

- Automatic Root-Cause Analysis of bugs
  - Check invariants learnt on runs which pass
- Debug long-running programs
  - Train on early part of program
- Debug component-based software
  - Train components on other configurations
- Watch invariant violations when correct answer may be unknown

Uses (2)

- Understand legacy code
  - "Can this ever happen ?"
- Analyze test suite completeness
  - Incorrect invariants indicate holes in test suite
- Assist in program evolution
  - "Did I break anything ?"

Questions ?

Contact Information:

sudheendra.hangal@sun.com

lam@cs.stanford.edu

DIDUCE now open sourced and available at:

http://diduce.sf.net

Backup: Mailmanage Example

- Open source project to manipulate email
  - http://sf.net/mailmanage
- Uses JavaMail library
- Library threw cryptic IOException on 1 mail-box (of 300)
- Did not have source to JavaMail
  - Blindly instrumented all classes
- Trained on mailboxes which worked
  - Hit one invariant violation just before failure

Backup: Mailmanage Error

```
do {
  switch (buffer[index]) {
      case 'E': ...; index +=...; break;
      // other cases
} while (buffer[index++] != ')');
```

- **Obtained source code of class in question**
- **DIDUCE invariant violation was:**
  - **buffer[index] at this point = Ctrl-A (0x10), instead of ')' or ' '**
- **Bug: Length mismatch - Solaris IMAP server returning wrong data for a DOS attachment**

Backup: Mailmanage Lessons

- Error not in user code, not even in JavaMail library
- Error in IMAP server on a different machine
- Error detected as soon as it propagated into instrumentation domain
- DIDUCE helped zoom in to error in completely unfamiliar code!
- Helped to quickly identify faulty component and replace it
- Trade off accuracy for overhead

Backup: Joeq

- Research Java VM, written in Java
- DIDUCE hit a bug, which caused an assert later:
  - JAR file had duplicate entries for same filename
  - Caused inconsistency between memory image and Jar file
- Detected because return value of Hashtable.put () not null!

Backup: Future Work

- Focus on strong invariants - enable invariant checking in deployed code ?
  - Overhead needs to be lowered
  - Identify critical "state" variables
- Get DIDUCE deployed in testing/bug analysis environments