Systematic Testing

Mooly Sagiv

Slides taken from : John Heasman (NCC)
The Apple “goto fail bug”

... 
if ((err = SSLHashSHA1.update(&hashCtx,
&signedParams)) != 0)
    goto fail;
    goto fail;
    goto fail;
    ... other checks ...
fail:
    ... buffer frees (cleanups) ...
    return err;
<table>
<thead>
<tr>
<th>Problem</th>
<th>Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>Propositional SAT solving</td>
<td>MiniSat, Z3</td>
</tr>
<tr>
<td>First order solving with theories (SMT)</td>
<td>Z3, CVC3</td>
</tr>
<tr>
<td>Bounded Model Checking</td>
<td>CBMC, JBMC</td>
</tr>
<tr>
<td>Concolic Execution</td>
<td>DART, KLEE, SAGE, Cloud9, Mayhem</td>
</tr>
<tr>
<td>Static analysis</td>
<td>SLAM(SDV), Astrée, TVLA, CSSV</td>
</tr>
<tr>
<td>Testing</td>
<td>PITTEST, AFL</td>
</tr>
<tr>
<td>Program Synthesis</td>
<td>SKETCH(MIT), Rosettee(UWASH)</td>
</tr>
</tbody>
</table>
Verification vs. Testing

Program P

Desired Properties $\varphi$

Solver

Is there a behavior of P that violates $\varphi$?

Counterexample

Proof
Testing

Program P

Input Tests I

Desired Properties $\varphi$

Checker

Does the execution of $P$ on $I$ violate $\varphi$?

Bug

No Bug
The Testing Goal

• Input: A program
• Output: An input to the program which demonstrates fault
  • Assertion violation
  • Runtime error
    • Buffer overrun
  • Exception

Sometimes faults can be demonstrated by changing the original program
Testing Terminology

• White vs. Blackbox testing
• Testing levels:
  • Unit
  • Integration
  • System
Adequacy

• How do you know that the set of input tests suffice?
• Coverage
• Mutation testing
• ...
Simple Example (Node Coverage, Edge Coverage, Path Coverage)

\[
p(n_1, n_2)
\]

\[
x := 0
\]

\[
n_1 > 0
\]

\[
x := x - 9
\]

\[
x := x + 7
\]

\[
n_2 > 0
\]

\[
x := x \times 8
\]

\[
x := x \times 16
\]

\[
exit
\]
Mutation Testing

• Measures the adequacy of the test suit
• Faults are introduced into the program by creating many versions of the program called *mutants*
• Each mutant contains a single fault
• The test inputs are applied to the original program and to the mutant program
• If mutant programs fail on the input test ➔ the test suit is adequate
  • Otherwise need more tests
Test Case Adequacy

• A test case is *adequate* if it is useful in detecting faults in a program.
• A test case can be shown to be adequate by finding at least one mutant program that generates a different output than does the original program for that test case.
• If the original program and all mutant programs generate the same output, the test case is *inadequate*. 
Mutant Programs

• Mutation testing involves the creation of a set of mutant programs of the program being tested
• Each mutant differs from the original program by one mutation
• A mutation is a single syntactic change that is made to a program statement/condition
Simple Example

\[
\begin{align*}
\text{max}(x, y) & \quad x < y \\
\text{res} & := y \\
\text{res} & := x \\
\text{return } z
\end{align*}
\]

\(<2, 3>, <3, 2>\)

- \(x \leq y\)
- \(x > y\)
- \(x \geq y\)
Categories of Mutation Operators

• Operand Replacement Operators:
  • Replace a single operand with another operand or constant. *E.g.*,
    • if \(5 > y\) Replacing \(x\) by constant 5.
    • if \(x > 5\) Replacing \(y\) by constant 5.
    • if \(y > x\) Replacing \(x\) and \(y\) with each other.
  • *E.g.*, if all operators are \{+,-,\*,\**,/\} then the following expression \(a = b * (c - d)\) will generate 8 mutants:
    • 4 by replacing \(*\)
    • 4 by replacing \(-\).
Categories of Mutation Operators

• Expression Modification Operators:
  • Replace an operator or insert new operators. *E.g.,*
    • if (x == y)
    • if (x >= y) Replacing == by >=.
    • if (x == ++y) Inserting ++.
Categories of Mutation Operators

• Statement Modification Operators
  • Delete the else part of the if-else statement
  • Delete the entire if-else statement
  • Replace line 3 by a return statement
Why Does Mutation Testing Work?

- The operators are limited to simple single syntactic changes

The basis of the *competent programmer hypothesis*
The Competent Programmer Hypothesis

• Programmers are generally very competent and do not create “random” programs

• For a given problem, a programmer, if mistaken, will create a program that is very close to a correct program

• An incorrect program can be created from a correct program by making some minor change to the correct program
Mutation Testing Procedure

• Generate program test cases
• Run each test case against the original program.
  • If the output is incorrect, the program must be modified and re-tested
  • If the output is correct go to the next step ...
• Construct mutants using a tool like Pitest http://pitest.org/
Mutation Testing Procedure (Cont)

• Execute each test case against each alive mutant
  • If the output of the mutant differs from the output of the original program, the mutant is considered incorrect and is killed

• Two kinds of mutants survive:
  • *Functionally equivalent to the original program*
    • Cannot be killed
  • *Killable*: Test cases are insufficient to kill the mutant
    • New test cases must be created
Another Example

```c
main(argc, argv)
2. int argc, r, i;
3. char *argv[];
4. { r = 1;
5. for i = 2 to 3 do
6. if (atoi(argv[i]) > atoi(argv[r])) r = i;
7. printf("Value of the rank is %d \n", r);
8. exit(0); }
```

Tests:

- **Test 1:** 1, 2, 3
- **Test 2:** 1, 2, 1
- **Test 3:** 3, 1, 2

```c
Mut1: 5’. for i = 1 to 3 do
Mut2: 6’. if (i > atoi(argv[r])) r = i;
Mut3: 6’. if (atoi(argv[i]) >= atoi(argv[r])) r = i;
Mut4: 6’. if (atoi(argv[r]) > atoi(argv[r])) r = i;;
```

- **r = 3**
- **r = 2**
- **r = 1**
- **r = 1 vs r = 2**
Mutation Score

• The *mutation score* for a set of test cases is the percentage of non-equivalent mutants killed by the test data

• *Mutation Score* = \( 100 \times \frac{D}{N - E} \)
  - \( D \) = Dead mutants
  - \( N \) = Number of mutants
  - \( E \) = Number of equivalent mutant

• A set of test cases is *mutation adequate* if its mutation score is 100%
Evaluation

• Theoretical and experimental results have shown that mutation testing is an effective approach to measuring the adequacy of test cases

• The major drawback of mutation testing is the cost of generating the mutants and executing each test case against them
Selected References

• Pitest http://pitest.org/
• MuJava: An Automated Class Mutation System by Yu-Seung Ma, Jeff Offutt and Yong Rae Kwo.
• Mutation Operators for Concurrent Java (J2SE 5.0) by Jeremy S. Bradbury, James R. Cordy, Juergen Dingel.
• Mutation of Java Objects by Roger T. Alexander, James M. Bieman, Sudipto Ghosh, Bixia Ji.
• Mutation-based Testing of Buffer Overflows, SQL Injections, and Format String Bugs by H. Shahriar and M. Zulkernine.
Fuzz Testing
Fuzz testing

• Providing invalid, unexpected, or random data to the inputs

• Observe faults
  • Memory crashes
  • Violations of assertions
  • Security violations

• Sometimes applied by modifying the program based on some assumptions
Fuzzing Unix Utilities

• Begins in 1998 class project: Wisconsin Bart Miller
• Bombard unix utilities with random data until they crashed
• Repeated in many domains:
  • Windows/NT
  • MacOS
  • Networks

Summary of malloc Test Results

• Tested programs in /bin and /usr/ucb on our SunOS 4.1.3 system
• 53 of these programs used malloc()
• We could crash 25 of the 53 (47%)

<table>
<thead>
<tr>
<th>Utilities that Crashed</th>
</tr>
</thead>
<tbody>
<tr>
<td>bar</td>
</tr>
<tr>
<td>df</td>
</tr>
<tr>
<td>login</td>
</tr>
<tr>
<td>rup</td>
</tr>
<tr>
<td>tsort</td>
</tr>
<tr>
<td>cc</td>
</tr>
<tr>
<td>finger</td>
</tr>
<tr>
<td>ls</td>
</tr>
<tr>
<td>ruptime</td>
</tr>
<tr>
<td>users</td>
</tr>
<tr>
<td>checknr</td>
</tr>
<tr>
<td>graph</td>
</tr>
<tr>
<td>man</td>
</tr>
<tr>
<td>rusers</td>
</tr>
<tr>
<td>vplot</td>
</tr>
<tr>
<td>ctags</td>
</tr>
<tr>
<td>iostat</td>
</tr>
<tr>
<td>mkstr</td>
</tr>
<tr>
<td>sdiff</td>
</tr>
<tr>
<td>w</td>
</tr>
<tr>
<td>derooff</td>
</tr>
<tr>
<td>last</td>
</tr>
<tr>
<td>rsh</td>
</tr>
<tr>
<td>symorder</td>
</tr>
<tr>
<td>xsend</td>
</tr>
</tbody>
</table>
Malloc and Friends

• Intercept the calls to malloc()

  program
  
  malloc(...);

  libjig
  void *
  malloc(...) {
  
  rv=FZ_malloc(...);
  
  randomly return zero or rv;
  
  }

  C library
  (sym table modified)
  void *
  FZ_malloc(...) {
  
  ...;
  
  }

• Randomly change the return value to zero: simulating the lack of virtual memory
# Summary of X Window Test Results

## List of Utilities Tested

<table>
<thead>
<tr>
<th>Utility</th>
<th>bitmap</th>
<th>emacs</th>
<th>ghostview</th>
<th>idraw</th>
<th>mosaic</th>
<th>mxn</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>netscape</td>
<td>puzzle</td>
<td>xclock</td>
<td>xcutsel</td>
<td>xclock</td>
<td>xev</td>
</tr>
<tr>
<td></td>
<td>xclock</td>
<td>xconsole</td>
<td>xev</td>
<td>xman</td>
<td>xseparator</td>
<td>xpostit</td>
</tr>
<tr>
<td></td>
<td>rxvt</td>
<td>xcutview</td>
<td>xfontsel</td>
<td>xman</td>
<td>xseparator</td>
<td>xpostit</td>
</tr>
<tr>
<td></td>
<td>xboard</td>
<td>xddview</td>
<td>xgas</td>
<td>xmn</td>
<td>xseparator</td>
<td>xpostit</td>
</tr>
<tr>
<td></td>
<td>xcalc</td>
<td>xddview</td>
<td>xge</td>
<td>xneko</td>
<td>xseparator</td>
<td>xspread</td>
</tr>
<tr>
<td></td>
<td>xclipboard</td>
<td>xddview</td>
<td>xmg</td>
<td>xneko</td>
<td>xseparator</td>
<td>xspread</td>
</tr>
<tr>
<td></td>
<td>xclipboard</td>
<td>xddview</td>
<td>xpbiff</td>
<td>xneko</td>
<td>xseparator</td>
<td>xspread</td>
</tr>
<tr>
<td></td>
<td>xclipboard</td>
<td>xddview</td>
<td>xv</td>
<td>xneko</td>
<td>xseparator</td>
<td>xspread</td>
</tr>
</tbody>
</table>

## Input Data Stream Type

<table>
<thead>
<tr>
<th>X Utility</th>
<th>Random Messages (Type 1)</th>
<th>Garbled Messages (Type 2)</th>
<th>Random Events (Type 3)</th>
<th>Legal Events (Type 4)</th>
</tr>
</thead>
<tbody>
<tr>
<td># tested</td>
<td>38</td>
<td>38</td>
<td>38</td>
<td>38</td>
</tr>
<tr>
<td># crash/hang</td>
<td>1</td>
<td>10</td>
<td>18</td>
<td>16</td>
</tr>
<tr>
<td>%</td>
<td>3%</td>
<td>26%</td>
<td>47%</td>
<td>42%</td>
</tr>
</tbody>
</table>
Four Types of X Testing

• **Completely Random Messages:**
  • A random series of bytes in a message.

• **Garbled Messages:**
  • Randomly insert, delete, or modify parts of the message stream.

• **Random Events:**
  • Keeps track of message protocol message. Randomly insert or modify events with
  • valid size and opcodes. Sequence number, time stamp, and payload may be random.

• **Legal Events:**
  • Protocol conformant messages, logically correct individually and in sequence
  • Valid values X-Y coordinates, window geometry, parent/child relationships, event
    time stamps, and sequence numbers
Intercepting the X Windows Message Stream

• We control the messages going to the X application and server by interposing our “xjig” tester
void null_terminate(char *s)
{
    while (*s != ' ' ) s++;
}

Pointer vulnerability (1)
Pointer vulnerability (2)

```
char string[200];
...
while (cc = getch()) != c) {
    string[j++] = cc;
    ...
}
```

The termination condition ignores the size of the buffer (string)
Pointer vulnerability  ctags

```
char line[4*BUFSIZ];
...
sp = line;
...
do {
    *++sp = c = getc(inf);
} while ((c != '\n') && (c != EOF));
```
Instrumentation

• Automatically modify the input program to create certain behaviors

• Examples
  • Checking undefined behaviors in C
    • Purify, Valgrid
  • Fuzzing
Type of bugs exposed by Fuzzing

• Crashes
• Memory leaks
• Uncaught exceptions
• Incorrect resource management
• Assertion violation
What is fuzzing?

- Feed target automatically generated malformed data designed to trigger implementation flaws
  - A fuzzer is the programmatic construct
- A fuzzing framework typically includes library code to:
  - Generate fuzzed data
  - Deliver test cases
  - Monitor the target
- Publicly available fuzzing frameworks:
  - Spike, Peach Fuzz, Sulley, Schemer, American Fuzzy Lop
- Requirement of Microsoft’s Secure Development Lifecycle program
- Still a long way to go - many vendors do no fuzzing!
What data can be fuzzed?

- Virtually anything!
- Basic types: bit, byte, word, dword, qword
- Common language specific types: strings, structs, arrays
- High level data representations: text, xml
What does fuzzed data consist of?

• Fuzzing at the type level:
  • Long strings, strings containing special characters, format strings
  • Boundary case byte, word, dword, qword values
  • Random fuzzing of data buffers

• Fuzzing at the sequence level
  • Fuzzing types within sequences
  • Nesting sequences a large number of times
  • Adding and removing sequences
  • Random combinations

• **Always record the random seed!!**
When to fuzz?

• Fuzzing typically finds implementation flaws, e.g.:
  • Memory corruption in native code
    • Stack and heap buffer overflows
    • Un-validated pointer arithmetic (attacker controlled offset)
    • Integer overflows
    • Resource exhaustion (disk, CPU, memory)
  • Unhandled exceptions in managed code
    • Format exceptions (e.g. parsing unexpected types)
    • Memory exceptions
    • Null reference exceptions
  • Injection in web applications
    • SQL injection against backend database
    • LDAP injection
    • HTML injection (Cross-site scripting)
    • Code injection
When not to fuzz

• Fuzzing typically does not find logic flaws
  • Malformed data likely to lead to crashes, not logic flaws
  • e.g. Missing authentication / authorization checks
• Fuzzing does not find design/repurposing flaws
  • e.g. A sitelocked ActiveX control with a method named “RunCmd”.
Fuzzing in practice: the basic steps

1. Start
2. Monitor Target
3. Generate next test case
4. Deliver test case
5. Target crashed?
   - Yes: Save crash dump
   - No: Any more test cases?
     - Yes: Deliver test case
     - No: Finish
Monitoring the target

1. Attach a debugger
   - Leverage existing functionality
   - Scripting, logging, crash dumps etc.
Monitoring the target

2. Write your own debugger
   • Actually easy to do
   • Lightweight, fast, full control

C++

```c++
BOOL WINAPI WaitForDebugEvent(_out LPDEBUG_EVENT lpDebugEvent,
__in DWORD dwMilliseconds);
```

typedef struct _DEBUG_EVENT { /* de */
  DWORD dwDebugEventCode;
  DWORD dwProcessId;
  DWORD dwThreadId;
  union { EXCEPTION_DEBUG_INFO Exception;
         CREATE_THREAD_DEBUG_INFO CreateThread;
         CREATE_PROCESS_DEBUG_INFO CreateProcess;
         EXIT_THREAD_DEBUG_INFO ExitThread;
         EXIT_PROCESS_DEBUG_INFO ExitProcess;
         LOAD_DLL_DEBUG_INFO LoadDll;
         UNLOAD_DLL_DEBUG_INFO UnloadDll;
         OUTPUT_DEBUG_STRING_INFO DebugString; } u; } DEBUG_EVENT, *LPDEBUG_EVENT;
```
Monitoring the target

3. Monitor resources:
   • File, registry, memory, CPU, logs
Deliver the test case

1. Standalone test harness
   - E.g. to launch to client application and have it load fuzzed file format

2. Instrumented client
   - Inject function hooking code into target client
   - Intercept data and substitute with fuzzed data
   - Useful if:
     - State machine is complex
     - Data is encoded in a non-standard format
     - Data is signed or encrypted
Evaluation

• Fuzzing is an effective technique for finding bugs in huge software
• But has many limitations
  • Cannot find interesting bugs with correlations
  • Scaling is an issue
Projects with Z3

• Explore the ability of propositional/first order to concisely describe problems
  • Reductions between NP-complete problems
  • Correct SQL queries
    • Bugs in SQL queries
      • Empty join
  • Correct configurations
  • ...

Projects with CBMC/KEE/JBMC/Pittest/AFL/Astree

- Take a small application from Github
  - Instructors can help
Projects with Dafny

• Prove the correctness of parts of Minisat
• Prove the correctness of a data structure from the Data structure course
  • union-find
Projects with IVY/Alloy/TVLA

• Garbage collection algorithms
• Shared memory concurrency
  • Concurrent queue
  • ...
• Distributed applications
• Software defined networks

Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, Asaf Valadarsky: 
VeriCon: towards verifying controller programs in software-defined networks. PLDI 2014: 282-293
Projects with Sketch/Rosette

• Develop a small language for cloud utilization
• .... Next week