Systematic Testing

Mooly Sagiv

Slides taken from :John Heasman(NCC)

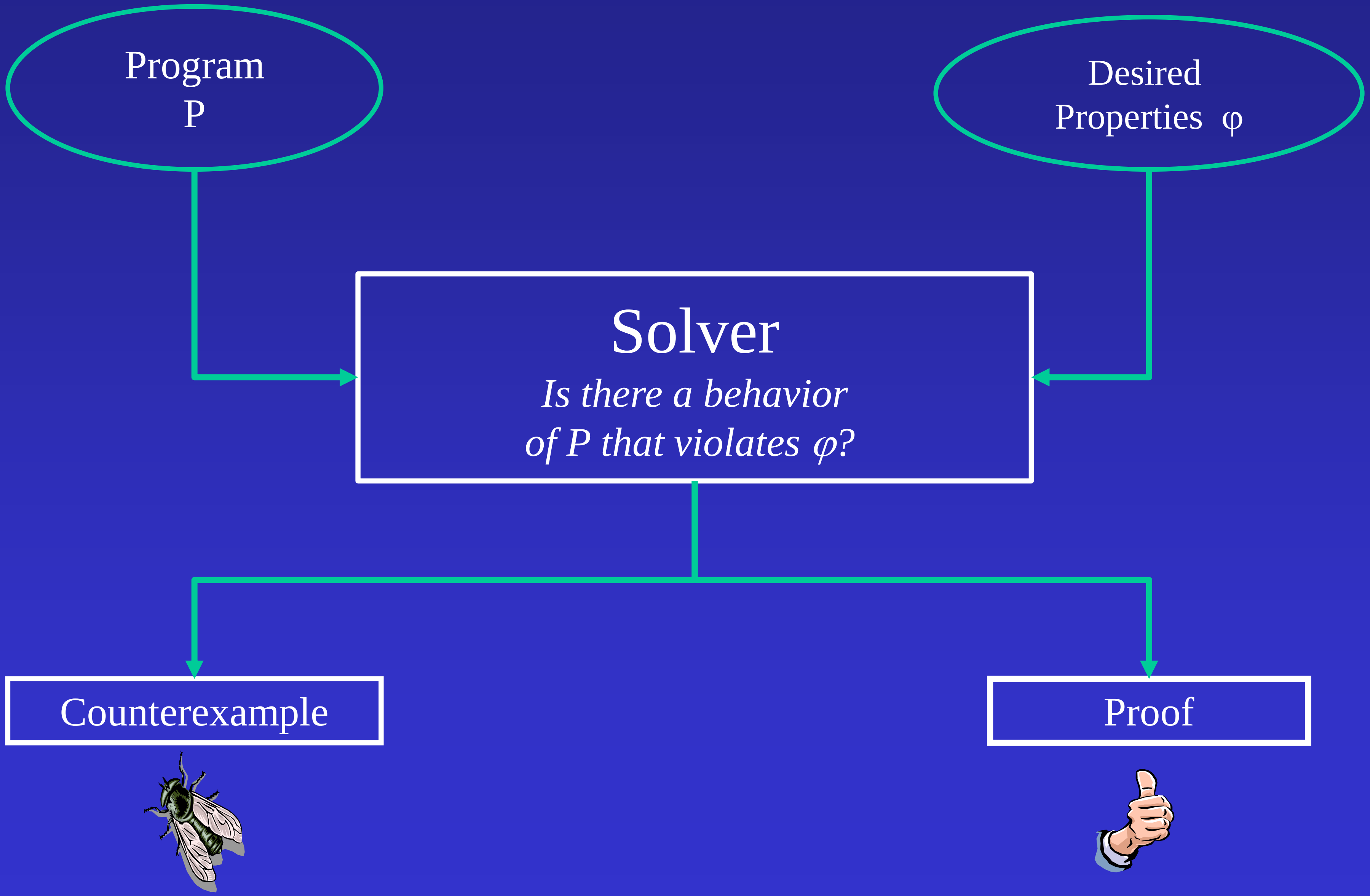
The Apple "goto fail bug"

```
…
if ((err = SSLHashSHA1.update(&hashCtx,
&signedParams)) != 0)
    goto fail;
    goto fail;
  ... other checks ...
  fail:
    ... buffer frees (cleanups) ...
    return err;
```

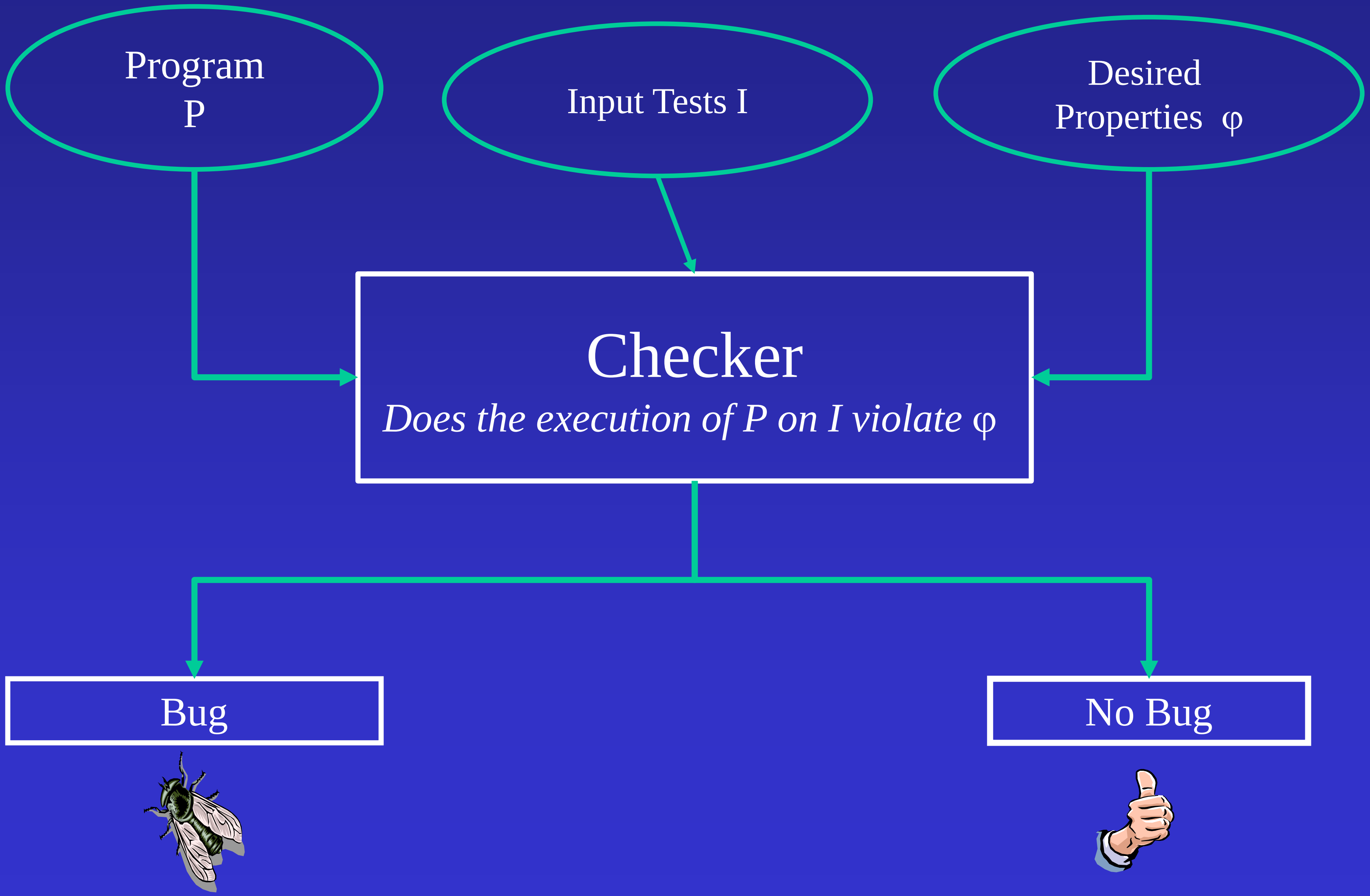
Recap

Problem	Tools
Propositional SAT solving	MiniSat, Z3
First order solving with theories (SMT)	Z3, CVC3
Bounded Model Checking	CBMC, JBMC
Concolic Execution	DART, KLEE, SAGE, Cloud9, Mayhem
Static analysis	SLAM(SDV), Astrée, TVLA, CSSV
Testing	PITTEST, AFL
Program Synthesis	SKETCH(MIT), Rosettee(UWASH)

Verification vs. Testing



Testing



Program
P
Input Tests I
Desired
Properties φ
Checker
Does the execution of P on I violate φ
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

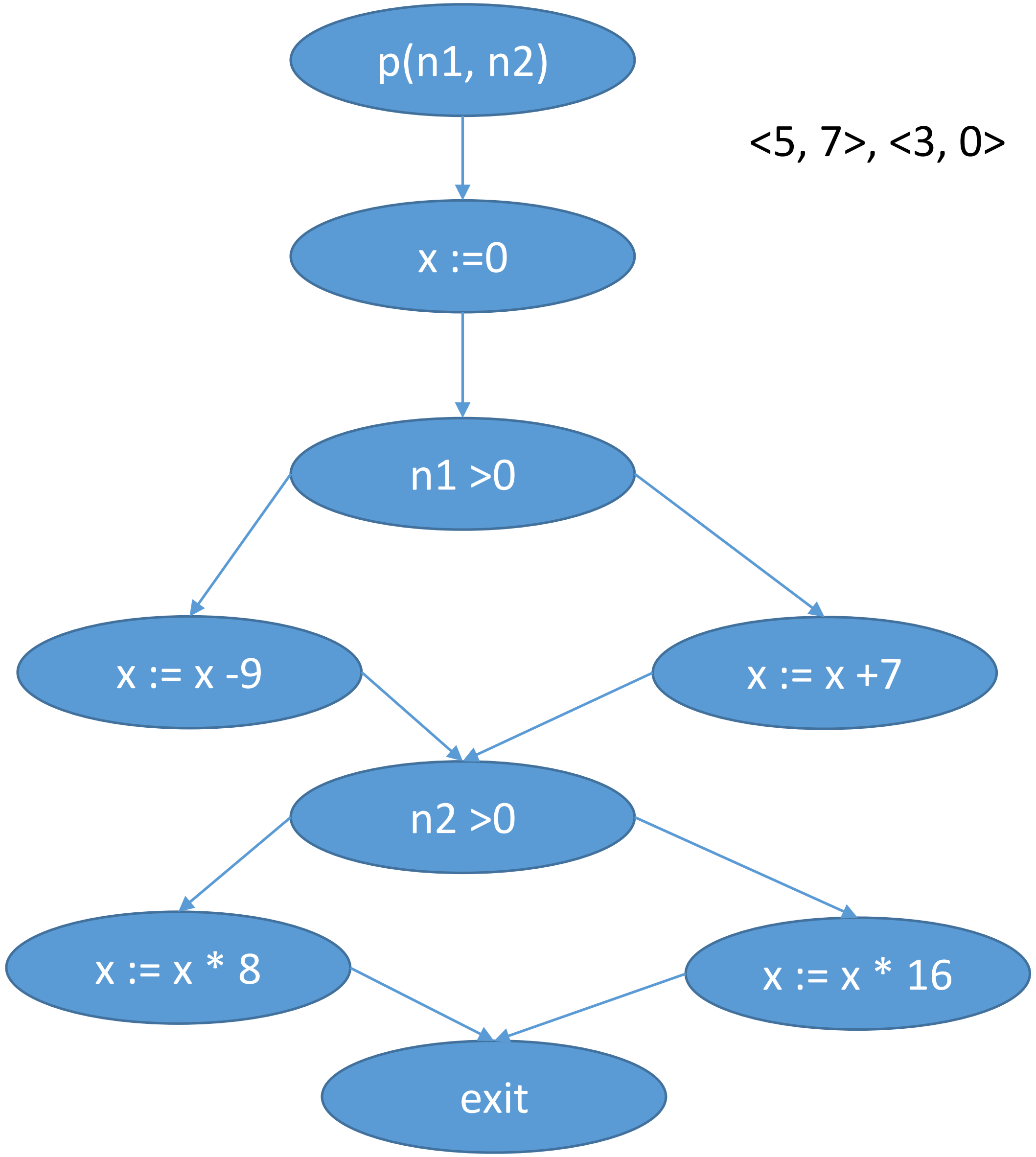
Testing Terminology

- 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)



<5, 7>, <3, 0>

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

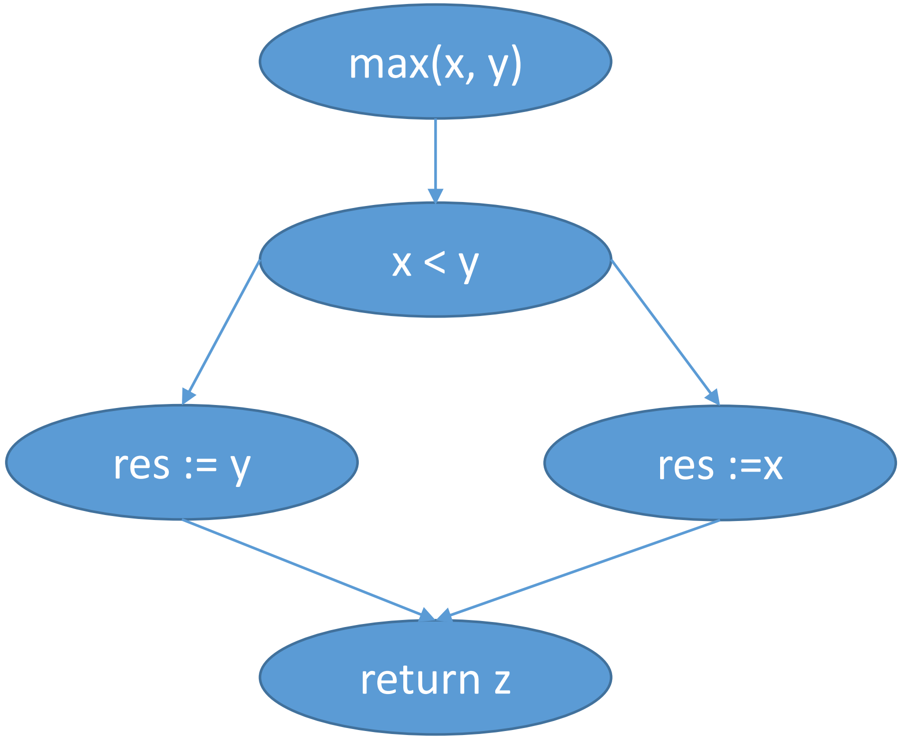
- A test case is *adequate* if it is useful in detecting faults in a program
- A test case can be shown to be adequate by finding at least one mutant program that generates a different output than does the original program for that test case
- If the original program and all mutant programs generate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

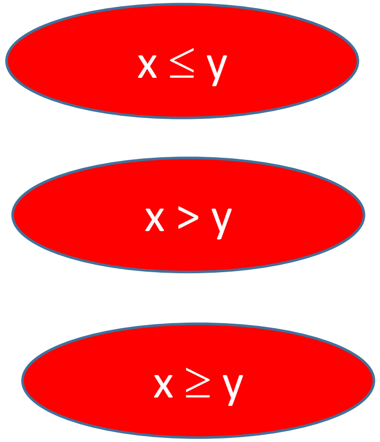
Mutant Programs

- 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

<2, 3>, <3, 2>





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` statemen

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

```
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); }
```

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;;

Test1: 1, 2, 3

r=3

Test2: 1, 2, 1

r=2

Test3: 3, 1, 2

r=1

Test1: 2, 2, 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 * 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

- Richard A. DeMillo, Richard J. Lipton, and Fred G. Sayward. Hints on test data selection: Help for the practicing programmer. IEEE Computer, 11(4):34-41. April 1978.
- Mathur, A., P., *Mutation Testing*, In the Encyclopedia of Software Engineering, John Wiley, 1994
- Paul Ammann and Jeff Offutt. Introduction to Software Testing. Cambridge University Press, 2008.
- 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

Barton Miller (2008). "Preface". In Ari Takanen, Jared DeMott and Charlie Miller, *Fuzzing for Software Security Testing and Quality Assurance*
Michael Sutton; Adam Greene; Pedram Amini (2007). *Fuzzing: Brute Force Vulnerability Discovery. Addison-Wesley.*

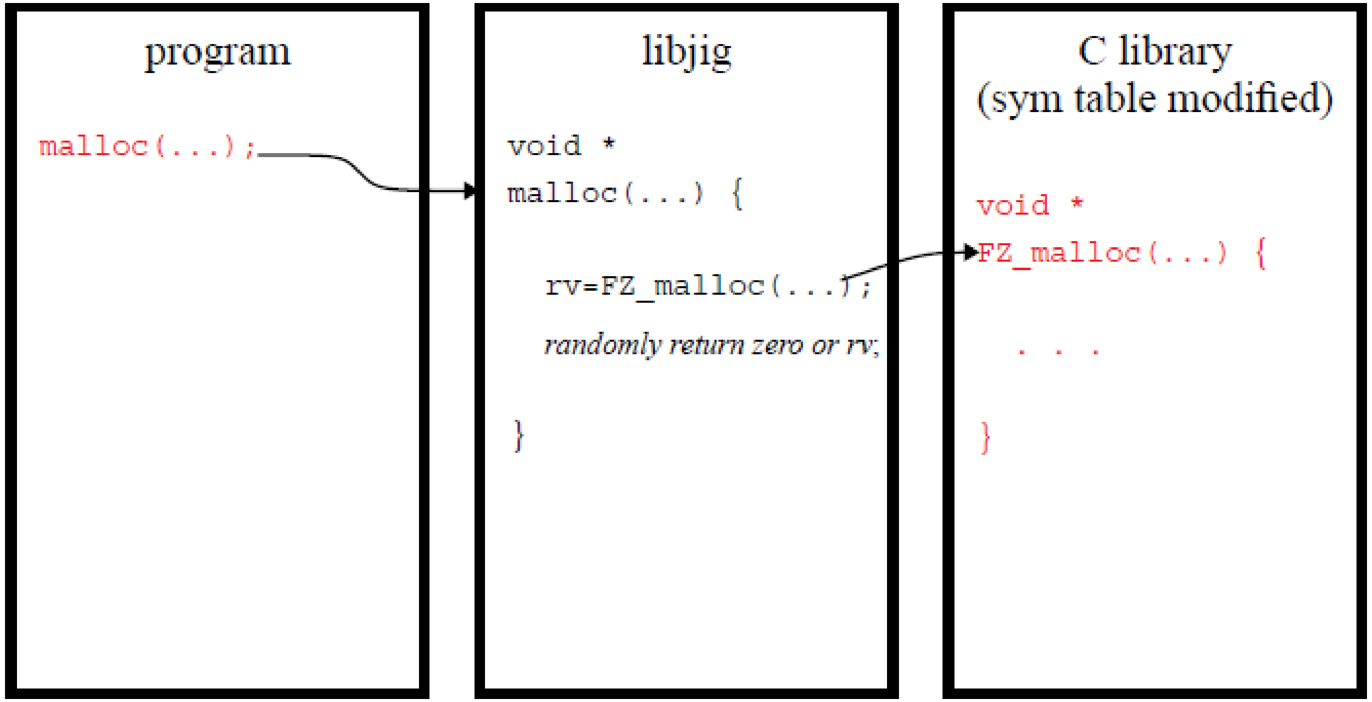
Summary of malloc Test Results

- 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%)

Utilities that Crashed				
bar	df	login	rup	tsort
cc	finger	ls	ruptime	users
checknr	graph	man	rusers	vplot
ctags	iostat	mkstr	sdiff	w
deroff	last	rsh	symorder	xsend

Malloc and Friends

- Intercept the calls to malloc()



- Randomly change the return value to zero: simulating the lack of virtual memory

Summary of X Window Test Results

List of Utilities Tested						
bitmap	netscape	xclock	xev	xman	xpostit	xweather
emacs	puzzle	xconsole	xfig	xmh	xsnow	xxgdb
ghostview	rxvt	xcutsel	xfontsel	xminesweep	xspread	
idraw	xboard	xditview	xgas	xneko	xterm	
mosaic	xcalc	xdvi	xgc	xpaint	xtv	
mxrn	xclipboard	xedit	xmag	xpbiff	xv	

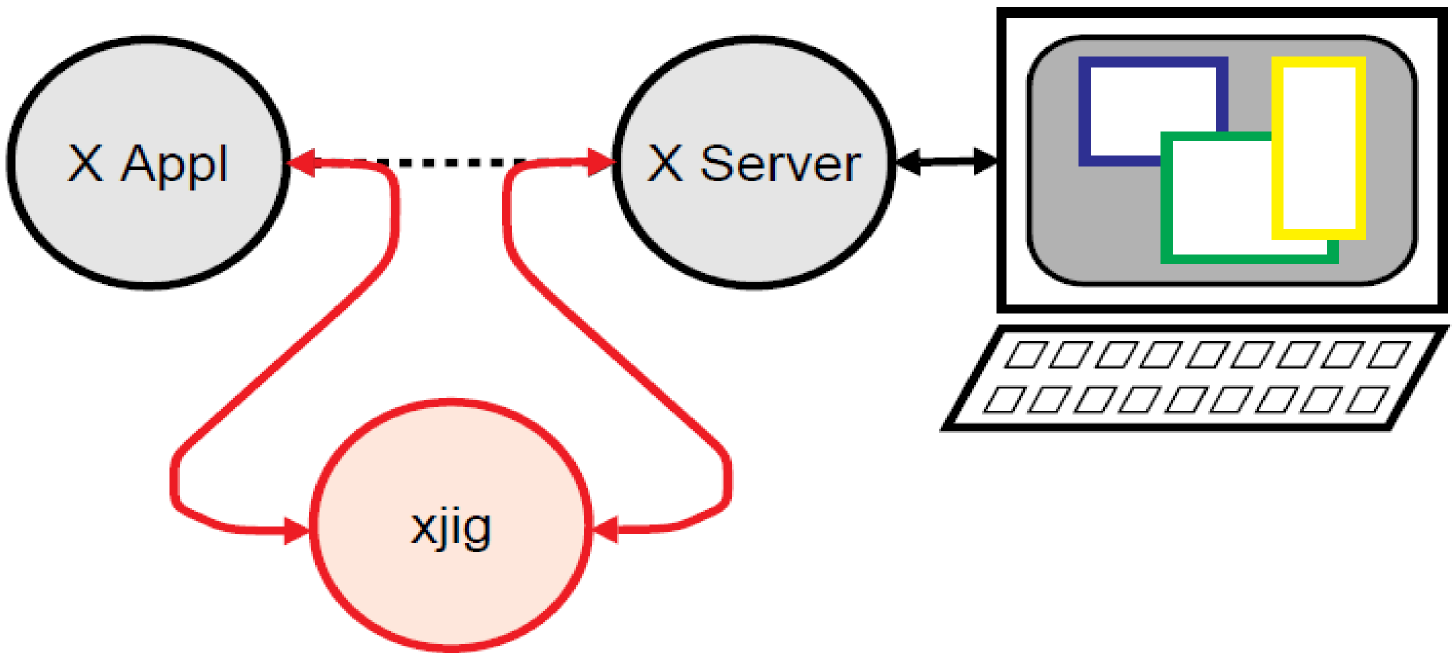
X Utility	Input Data Stream Type			
	Random Messages (Type 1)	Garbled Messges (Type 2)	Random Events (Type 3)	Legal Events (Type 4)
# tested	38	38	38	38
# crash/hang	1	10	18	16
%	3%	26%	47%	42%

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 messageX 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



Pointer vulnerability (1)

```
void null_terminate(char *s)
{
   while (*s  != '  ') s++ ;
}
```

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!!

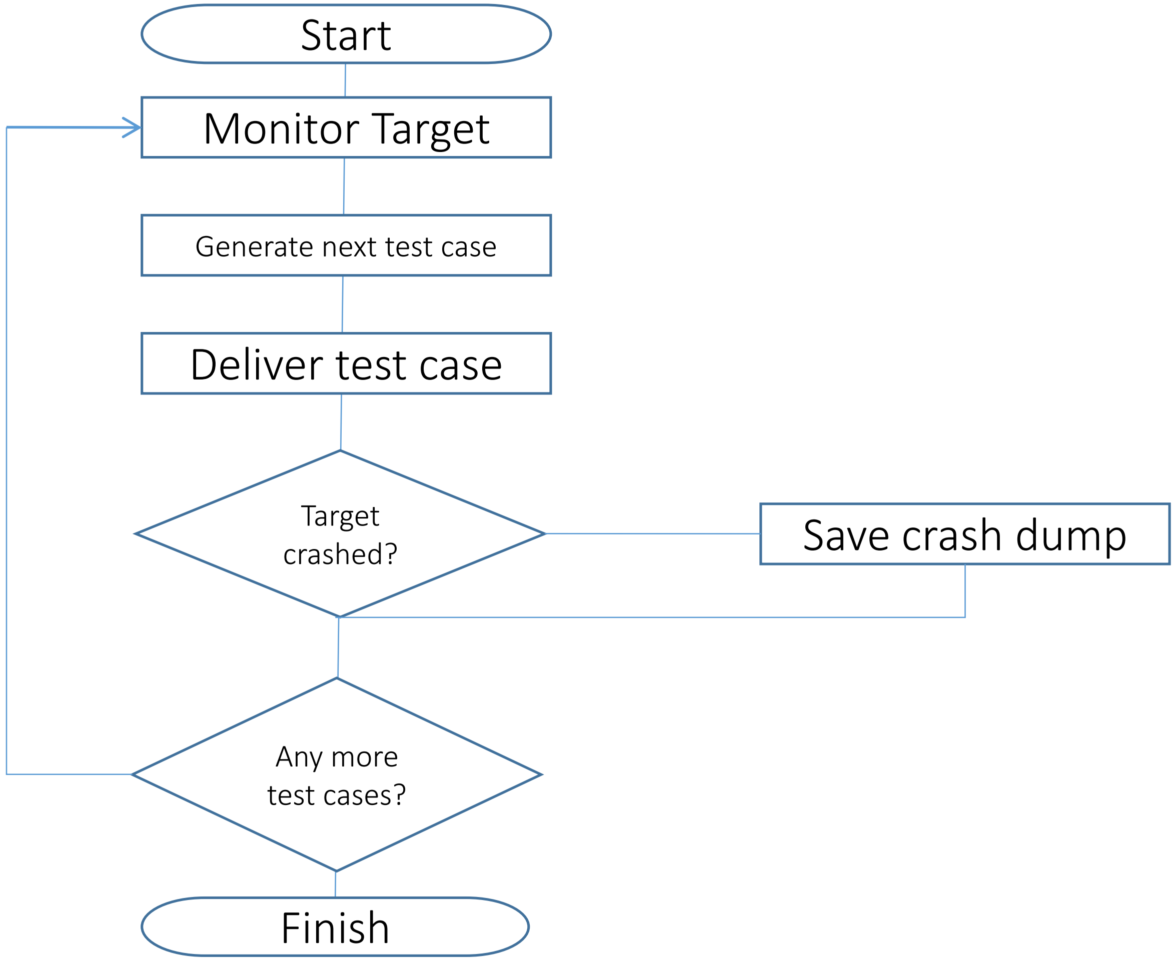
When to fuzz?

- 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

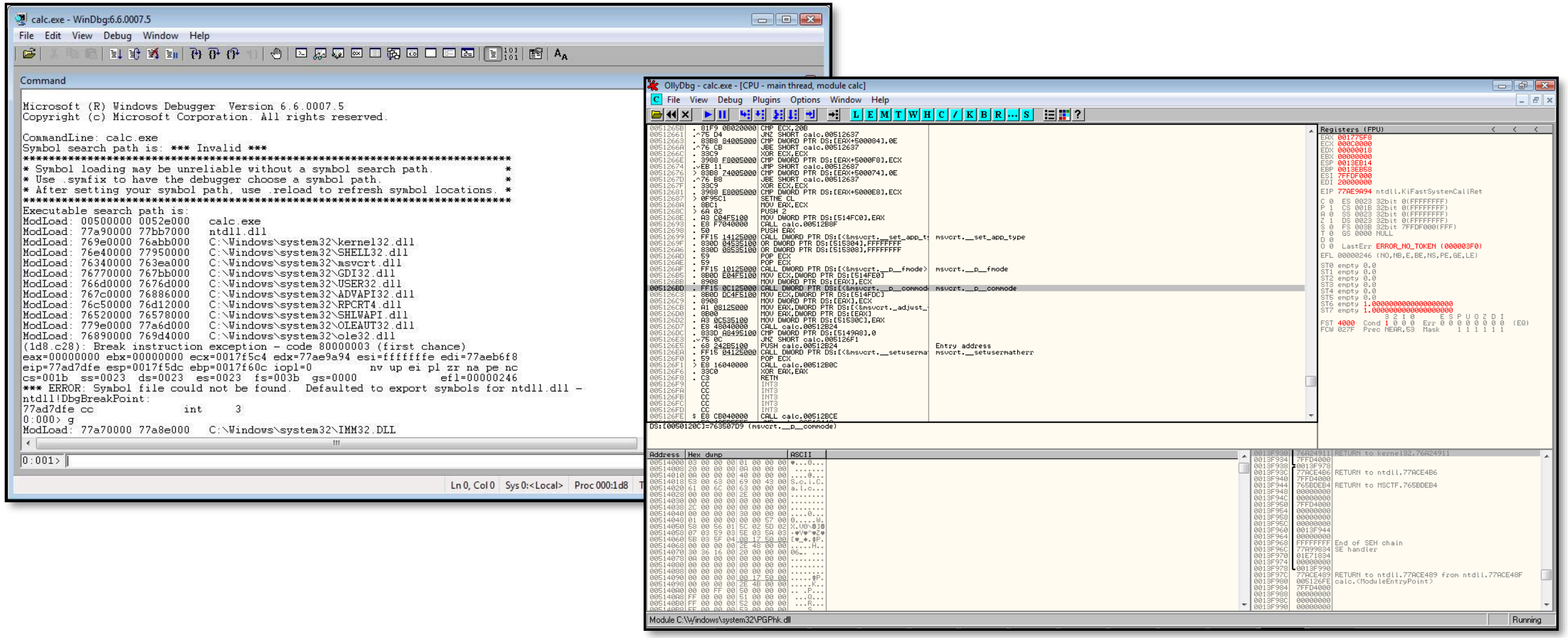
- Fuzzing typically does not find logic flaws
  - Malformed data likely to lead to crashes, not logic flaws
  - e.g. Missing authentication / authorization checks
- Fuzzing does not find design/repurposing flaws
  - e.g. A sitelocked ActiveX control with a method named “RunCmd”.

Fuzzing in practice: the basic steps



Monitoring the target

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



Monitoring the target

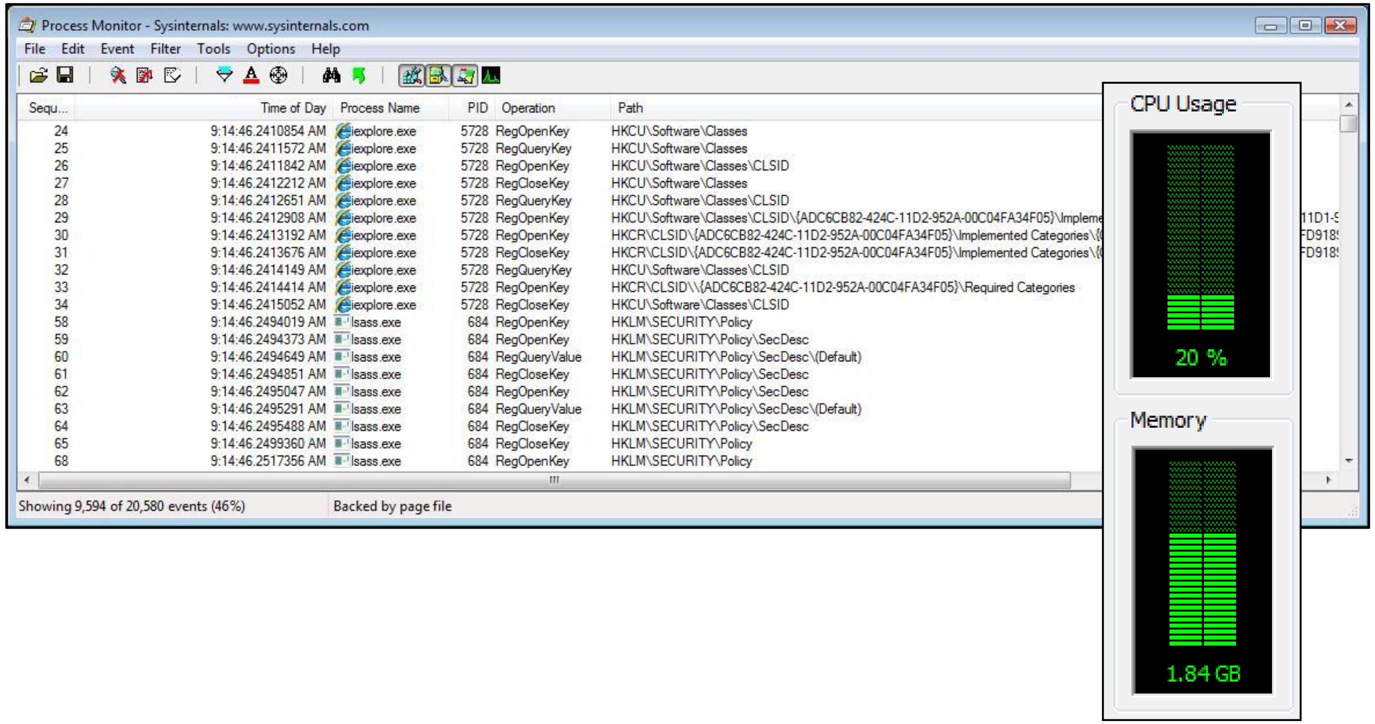
2. Write your own debugger
   - Actually easy to do
   - Lightweight, fast, full control

```
C++
BOOL WINAPI WaitForDebugEver
  __out  LPDEBUG_EVENT lpDe
  __in   DWORD dwMillisecor
);
```

```
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