

# CS 6431: Finding vulnerabilities

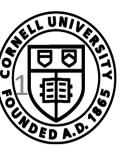
Instructor: Tom Ristenpart

<https://github.com/tomrist/cs6431-fall2021>



**CORNELL  
TECH**

HOME OF THE  
**JACOBS  
INSTITUTE**



# Running demo example (modified from Gray hat hacking book)

```
1 #include <stdio.h>
2 #include <string.h>
3
4
5 void greeting( char* temp1 )
6 {
7     char name[400];
8     memset(name, 0, 400);
9     strcpy(name, temp1);
10    printf( "Hi %s\n", name );
11 }
12
13
14 int main(int argc, char* argv[] )
15 {
16     greeting( argv[1] );
17     printf( "Bye %s\n", argv[1] );
18 }
```



# Language-based approaches

- Low-level languages such as C / C++ fast, but make it easy to make memory safety errors
- (More) type-safe languages prevent some vulnerabilities by design
  - Type-safety: “A language is type-safe if the only operations that can be performed on data in the language are those sanctioned by the type of the data.”
  - Traditionally less performance
- New generation of safer high-performance languages:
  - Rust (Mozilla), Swift (Apple), Go (Google)
- Efforts to improve security of unsafe languages
  - Safe pointer libraries in C / C++
  - Coding standards, defensive programming, unit testing approaches

# Most software very, very complex

In a Nutshell, Apache HTTP Server...

... has had 39,732 commits made by 125 contributors

representing 1,494,342 lines of code

... is mostly written in C with an average number of source code comments

Linux kernel v.4.1:  
~19.5 million lines of code  
14,000 developers contributing

OpenSSL:  
~608,000 lines of code  
572 developers contributing

# Heartbleed

OpenSSL implements TLS. Used in Apache and Nginx

March 2014: researchers discover vulnerability in heartbleed protocol implementation in OpenSSL

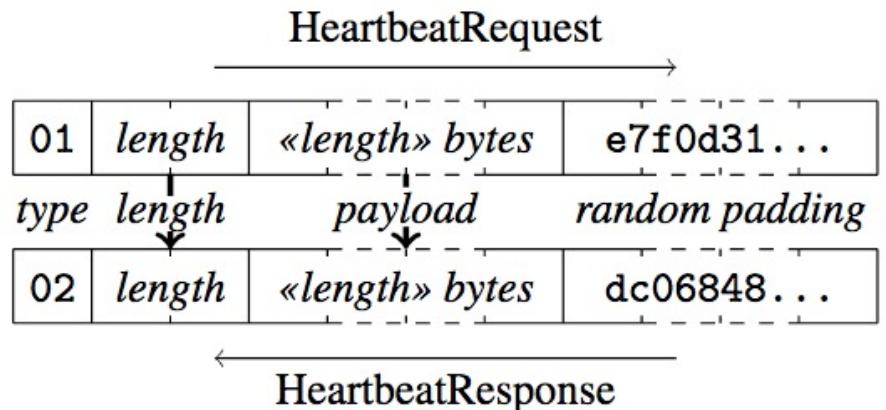


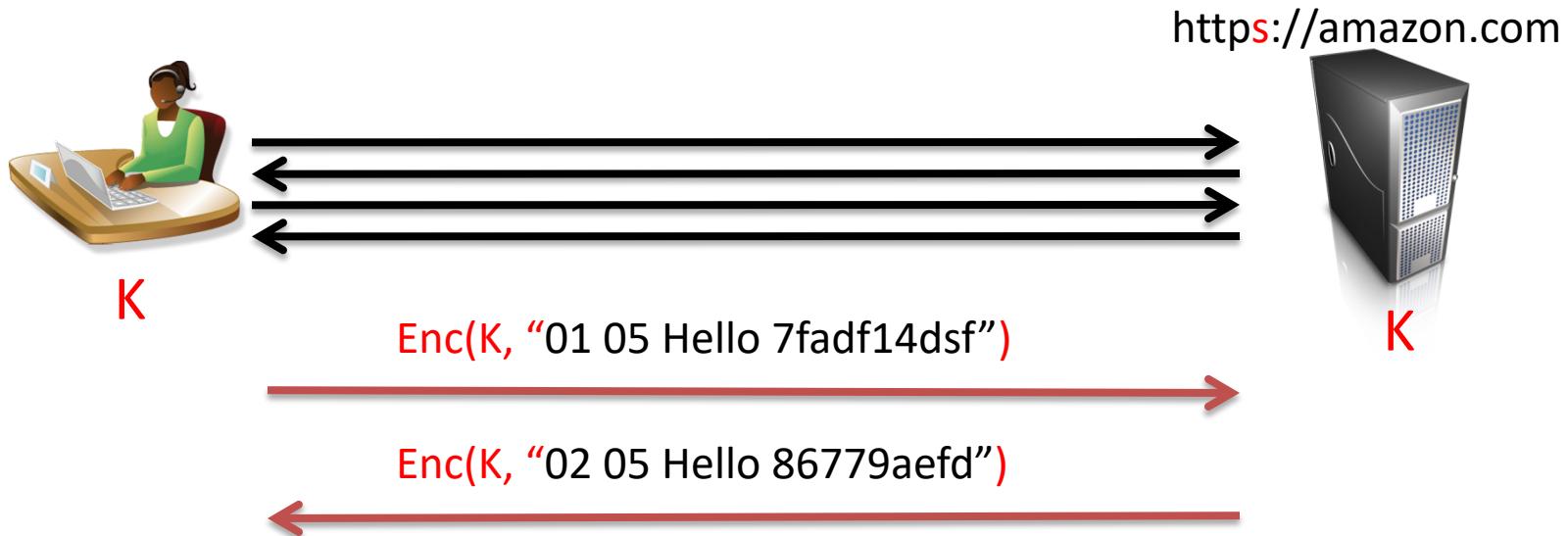
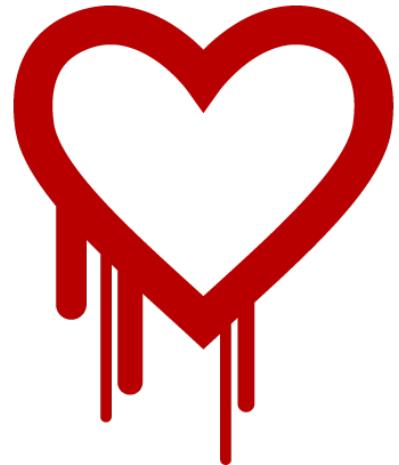
Figure 1: **Heartbeat Protocol.** Heartbeat requests include user data and random padding. The receiving peer responds by echoing back the data in the initial request along with its own padding.

[Durumeric et al. 2014]

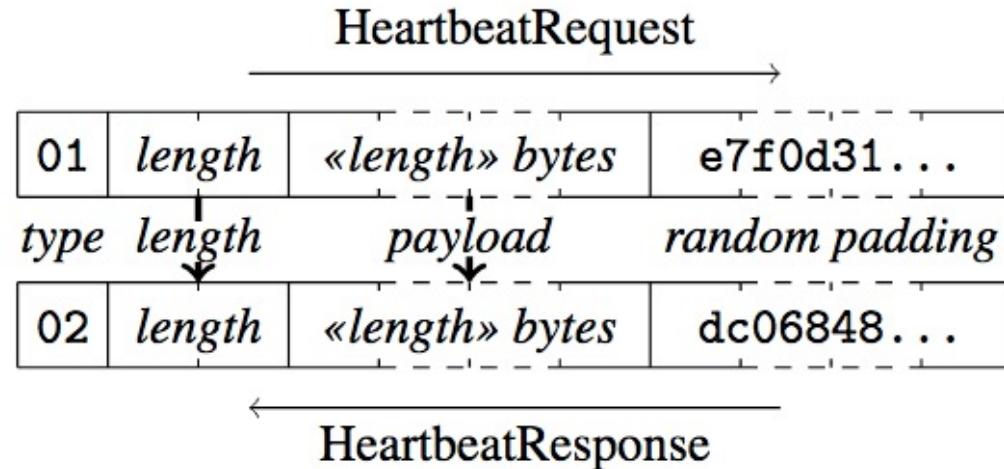
# Heartbleed

OpenSSL implements TLS. Used in Apache and Nginx

March 2014: researchers discover vulnerability in heartbleed protocol implementation in OpenSSL



# Heartbleed



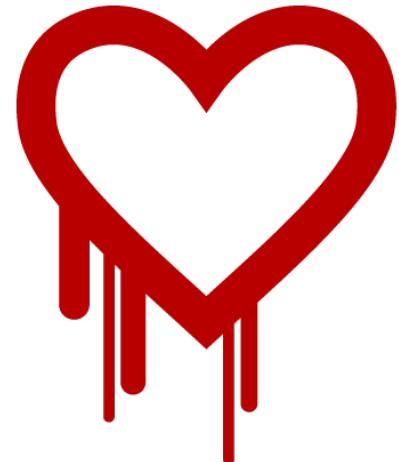
Buffer overread vulnerability  
Copy up to almost  $2^{16}$  bytes of data  
from memory

```
1448 dtls1_process_heartbeat(SSL *s)
1449 {
1450     unsigned char *p = &s->s3->rrec.data[0], *pl;
1451     unsigned short hbtype;
1452     unsigned int payload;
1453     unsigned int padding = 16; /* Use minimum padding */
1454
1455     /* Read type and payload length first */
1456     hbtype = *p++;
1457     n2s(p, payload);
1458     pl = p;
1459
1460     if (s->msg_callback)
1461         s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
1462                         &s->s3->rrec.data[0], s->s3->rrec.length,
1463                         s, s->msg_callback_arg);
1464
1465     if (hbtype == TLS1_HB_REQUEST)
1466     {
1467         unsigned char *buffer, *bp;
1468         int r;
1469
1470         /* Allocate memory for the response, size is 1 byte
1471          * message type, plus 2 bytes payload length, plus
1472          * payload, plus padding
1473          */
1474         buffer = OPENSSL_malloc(1 + 2 + payload + padding);
1475         bp = buffer;
1476
1477         /* Enter response type, length and copy payload */
1478         *bp++ = TLS1_HB_RESPONSE;
1479         s2n(payload, bp);
1480         memcpy(bp, pl, payload);
1481         bp += payload;
```

# Heartbleed

OpenSSL implements TLS. Used in Apache and Nginx

March 2014: researchers discover vulnerability in heartbleed protocol implementation in OpenSSL

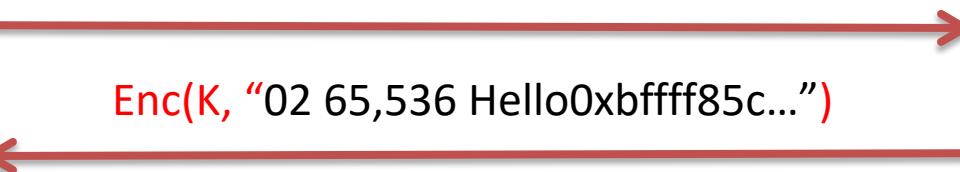


“I was doing laborious auditing of OpenSSL, going through the [Secure Sockets Layer] stack line by line”



K

Enc(K, “01 65,536 Hello 7fadf14dsf”)



https://amazon.com



K

Can dump large swathes of Apache process memory.  
What is at risk?

# Heartbleed

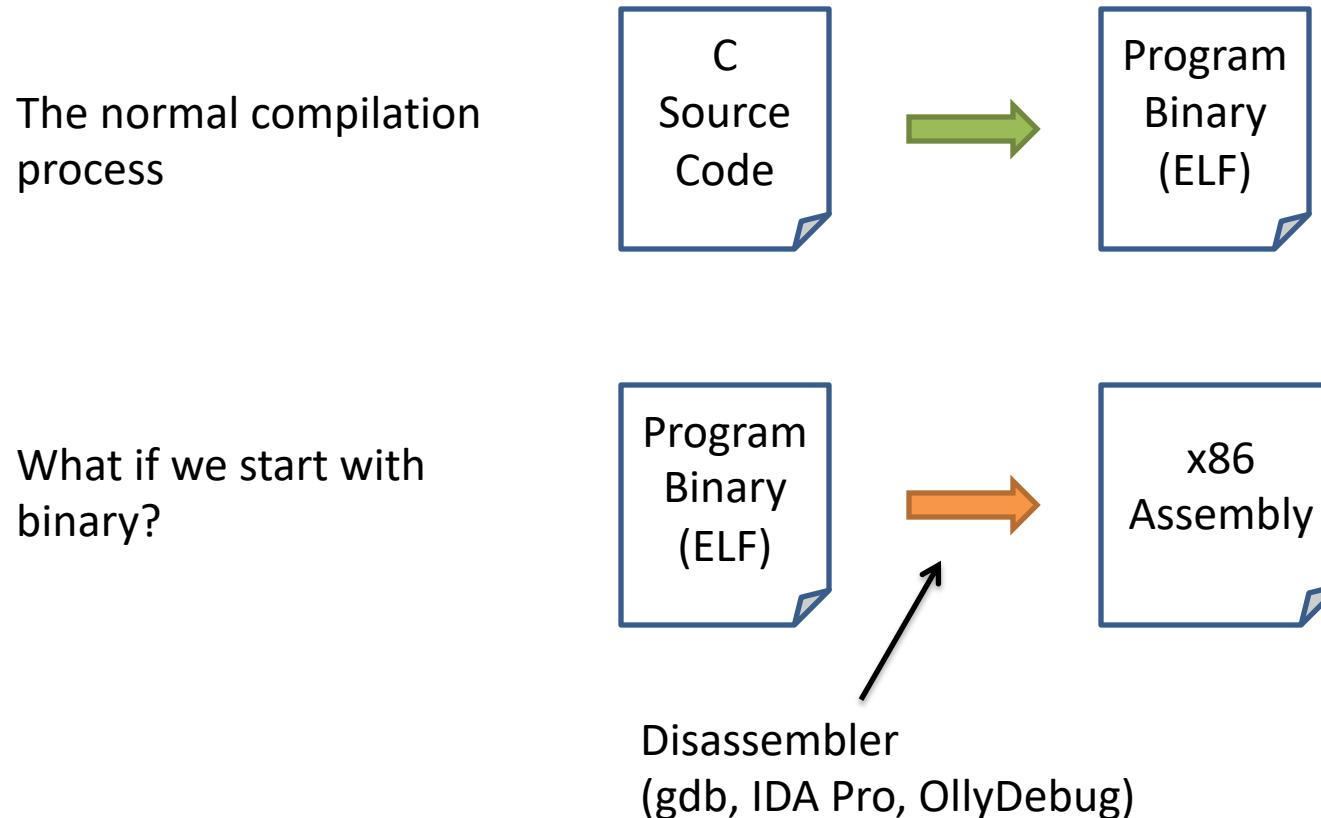
Internet scanning to determine vulnerability:  
Send heartbeat request with zero length (indicates vulnerable system)

Site	Vuln.	Site	Vuln.	Site	Vuln.
Google	Yes	Bing	No	Wordpress	Yes
Facebook	No	Pinterest	Yes	Huff. Post	?
Youtube	Yes	Blogspot	Yes	ESPN	?
Yahoo	Yes	Go.com	?	Reddit	Yes
Amazon	No	Live	No	Netflix	Yes
Wikipedia	Yes	CNN	?	MSN.com	No
LinkedIn	No	Instagram	Yes	Weather.com	?
eBay	No	Paypal	No	IMDB	No
Twitter	No	Tumblr	Yes	Apple	No
Craigslist	?	Imgur	Yes	Yelp	?

[Durumeric et al. 2014]

# Disassembly and decompiling

- Heartbleed discovered by direct C code inspection
- What if you only have binary?



```
0 <main+9>:    movl $0xf8, (%esp)
1 <main+16>:    call 0x8048364 <malloc@plt>
2 <main+21>:    mov %eax, 0x14(%esp)
3 <main+25>:    movl $0xf8, (%esp)
4 <main+32>:    call 0x8048364 <malloc@plt>
5 <main+37>:    mov %eax, 0x18(%esp)
6 <main+41>:    mov 0x14(%esp), %eax
7 <main+45>:    mov %eax, (%esp)
8 <main+48>:    call 0x8048354 <free@plt>
9 <main+53>:    mov 0x18(%esp), %eax
d <main+57>:    mov %eax, (%esp)
0 <main+60>:    call 0x8048354 <free@plt>
5 <main+65>:    movl $0x200, (%esp)
c <main+72>:    call 0x8048364 <malloc@plt>
1 <main+77>:    mov %eax, 0x1c(%esp)
5 <main+81>
3 <main+84> Double-free vulnerability
b <main+87>
d <main+89> Exploit can trick heap management
5 <main+97> software into writing adversary-controlled
3 <main+101> value to adversary-controlled address
d <main+105>
0 <main+108>
5 <main+113>:    mov 0x18(%esp), %eax
3 <main+117>:    mov %eax, (%esp)
c <main+120>:    call 0x8048354 <free@plt>
1 <main+125>:    mov 0x1c(%esp), %eax
5 <main+129>:    mov %eax, (%esp)
3 <main+132>:    call 0x8048354 <free@plt>
d <main+137>:    leave
e <main+138>:    ret
sembler dump.
```

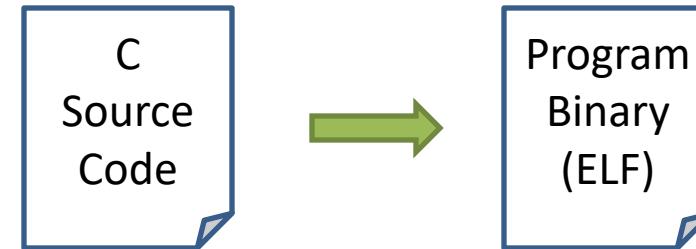
What type of vulnerability might this be?

```
main( int argc, char* argv[] ) {
    char* b1;
    char* b2;
    char* b3;

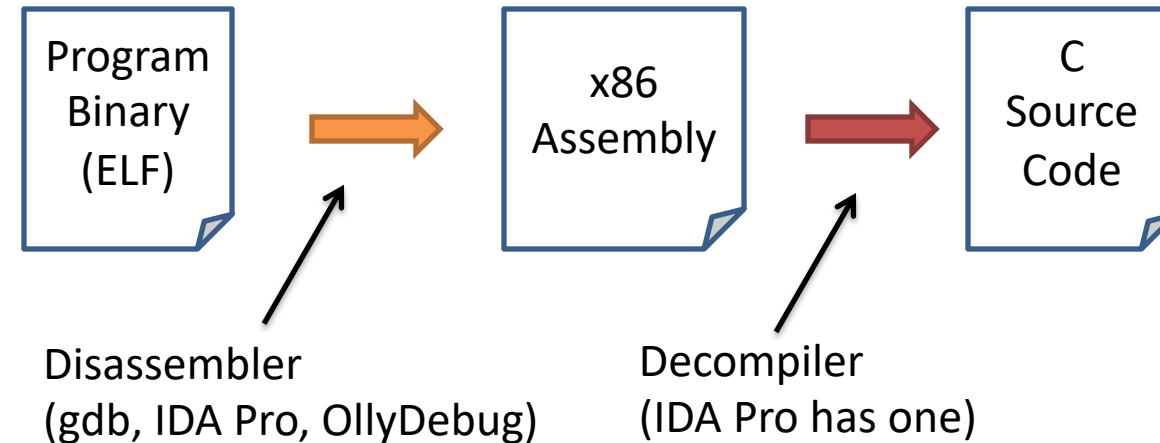
    if( argc != 3 ) return 0;
    if( atoi(argv[2]) != 31337 )
        complicatedFunction();
    else {
        b1 = (char*)malloc(248);
        b2 = (char*)malloc(248);
        free(b1);
        free(b2);
        b3 = (char*)malloc(512);
        strncpy( b3, argv[1], 511 );
        free(b2);
        free(b3);
    }
}
```

# Disassembly and decompiling

The normal compilation process



What if we start with binary?



Very complex, usually poor results

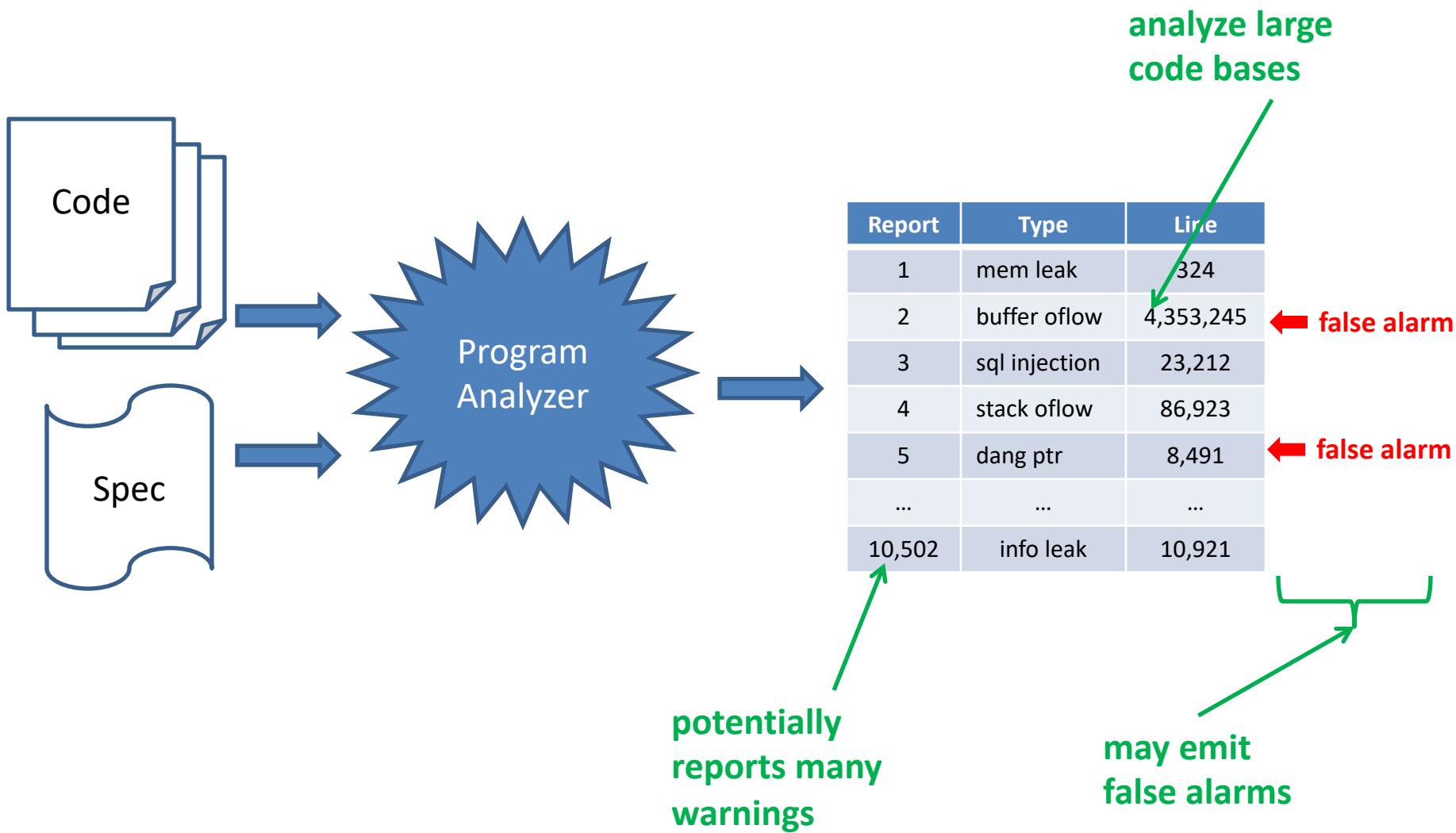
# Reverse engineering and vulnerability discovery

- Experienced analysts according to Aitel:
  - 1 hour of binary analysis:
    - Simple backdoors, coding style, bad API calls (`strcpy`)
  - 1 week of binary analysis:
    - Likely to find 1 good vulnerability
  - 1 month of binary analysis:
    - Likely to find 1 vulnerability *no one else will ever find*

# How do we find vulnerabilities?

- Manual analysis
  - Source code review
  - Reverse engineering
- Program analysis tools:
  - Static analysis
  - Fuzzing
  - Symbolic analysis
- After vulnerabilities get used:
  - Release of patches, network captures, etc.

# Program analyzers



# Program analysis: false positives and false negatives

Term	Definition
False positive	A spurious warning that does not indicate an actual vulnerability
False negative	Does not emit a warning for an actual vulnerability

Complete analysis: no false negatives

Sound analysis: no false positives

# Example tools / approaches

Approach	Type	Comment
Lexical analyzers	Static analysis	Perform syntactic checks Ex: LINT, RATS, ITS4
Fuzz testing	Dynamic analysis	Run on specially crafted inputs to test
Symbolic execution	Emulated execution	Run program on many inputs at once, by Ex: KLEE, S2E, FiE
Model checking	Static analysis	Abstract program to a model, check that model satisfies security properties Ex: MOPS, SLAM, etc.

# Source code scanners

Program that looks at source code, flags suspicious constructs

```
...  
strcpy( ptr1, ptr2 );  
...
```

Warning: Don't use strcpy

Simplest example: grep

Lint is early example

RATS (Rough auditing tool for security)

ITS4 (It's the Software Stupid Security Scanner)

Circa 1990's technology: **shouldn't** work for reasonable modern codebases

(... but probably will)

# Dynamic analysis: fuzzing

Choose a bunch of inputs

See if they cause program to crash

Key challenge: finding good inputs

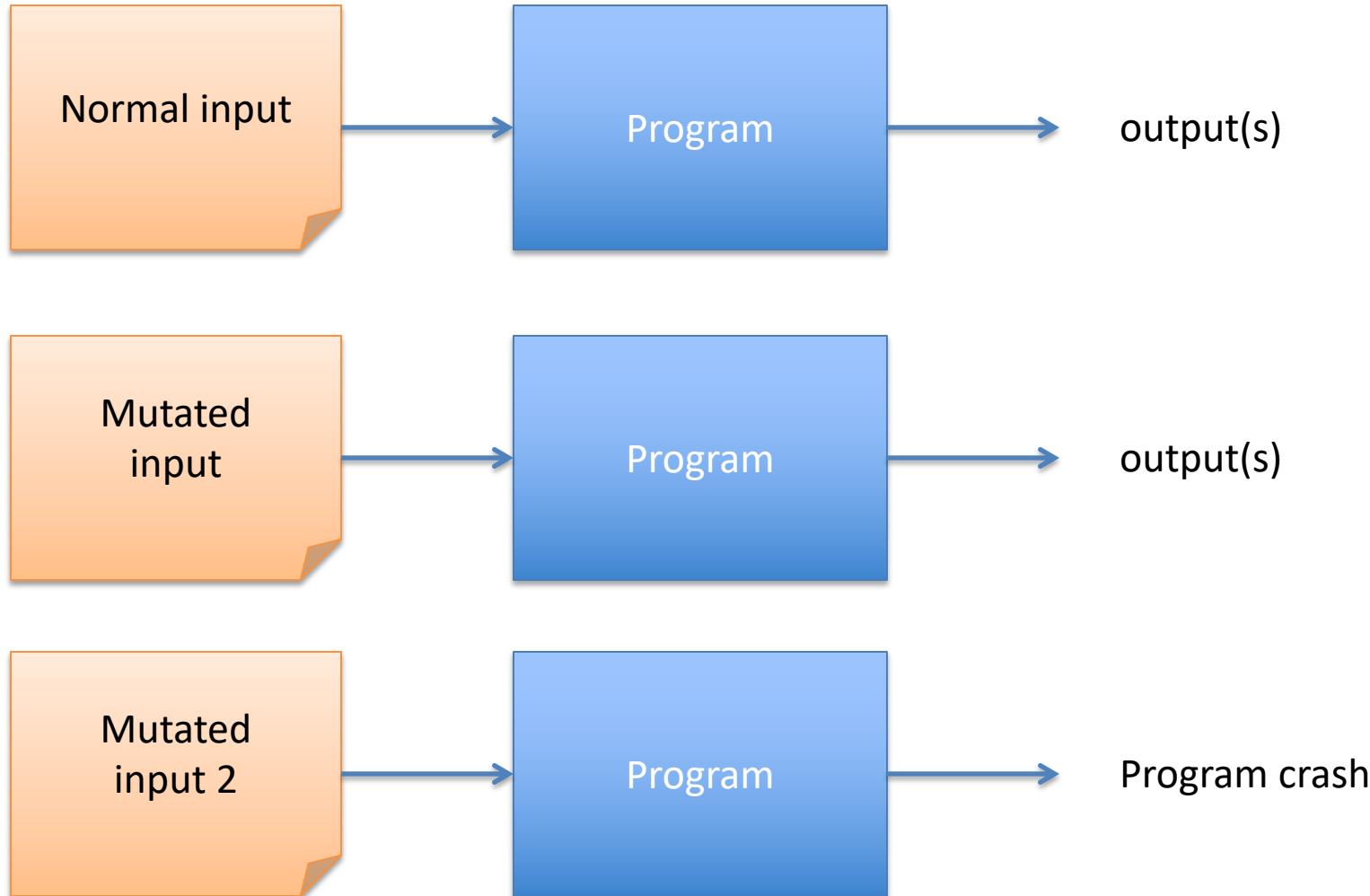


“The term first originates from a class project at the University of Wisconsin 1988 although similar techniques have been used in the field of quality assurance, where they are referred to as robustness testing, syntax testing or negative testing.”

Wikipedia

[http://en.wikipedia.org/wiki/Fuzz\\_testing](http://en.wikipedia.org/wiki/Fuzz_testing)

# Dynamic analysis: fuzzing



# Dynamic analysis: fuzzing

```
argv[1] = "AAAAA"  
argv[2] = 1
```



```
argv[1] = random str  
argv[2] = random 32-bit int
```



If integers are 32 bits, then probability of crashing is **at most what?**  $1/2^{32}$

Achieving code coverage can be very difficult

```
main( int argc, char* argv[] ) {  
    char* b1;  
    char* b2;  
    char* b3;  
  
    if( argc != 3 ) then return 0;  
    if( atoi(argv[2]) != 31337 )  
        complicatedFunction();  
    else {  
        b1 = (char*)malloc(248);  
        b2 = (char*)malloc(248);  
        free(b1);  
        free(b2);  
        b3 = (char*)malloc(512);  
        strncpy( b3, argv[1], 511 );  
        free(b2);  
        free(b3);  
    }  
}
```

# Code coverage and fuzzing

- Code coverage defined in many ways
  - # of basic blocks reached
  - # of paths followed
  - # of conditionals followed
  - gcov is useful standard tool
- Mutation based
  - Start with known-good examples
  - Mutate them to new test cases
    - heuristics: increase string lengths (AAAAAAAAA...)
    - randomly change items
- Generative
  - Start with specification of protocol, file format
  - Build test case files from it
    - Rarely used parts of spec

# American Fuzzy Lop (AFL) fuzzer

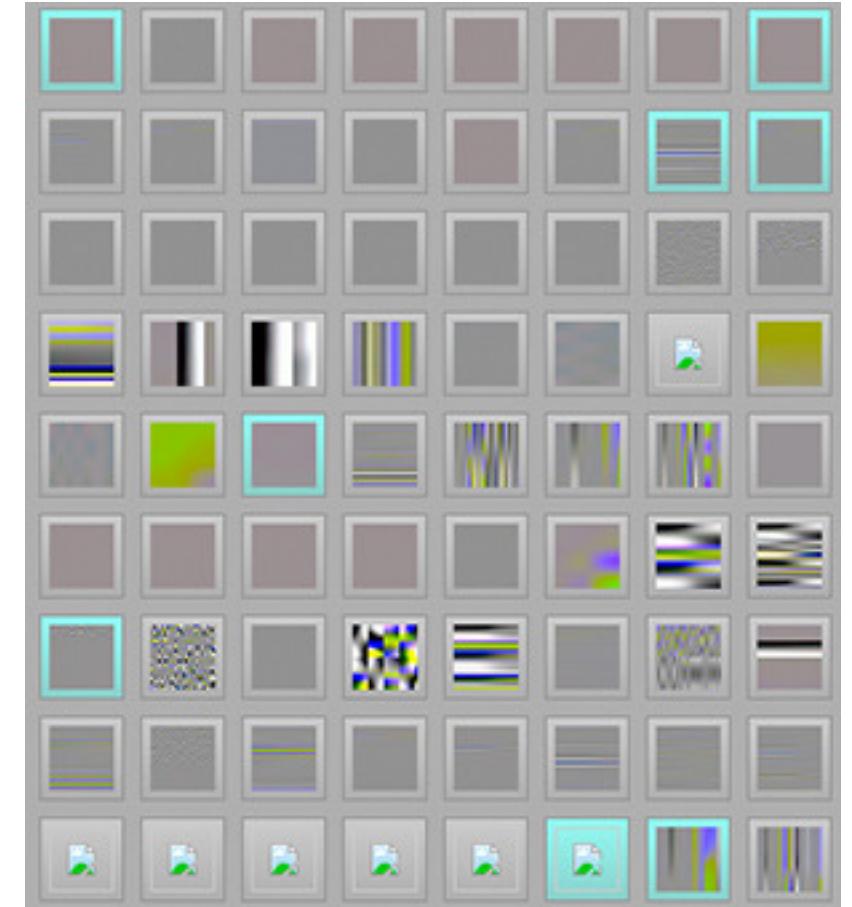
Widely used, highly effective fuzzing tool

- Specify example inputs
- Compile program with special afl compiler
- Run it

Performs mutation-based fuzzing:

- Deterministic transforms to input (flip each bit, “walking byte flips”,)
- Randomized stacked transforms
- Measure (approximation of) path coverage, keep and mutate set of files that increase coverage

Really fast & simple. Used to find bugs in:  
Firefox, OpenSSH, BIND, ImageMagick, iOS, ...



<https://lcamtuf.blogspot.com/2014/11/pulling-jpegs-out-of-thin-air.html>

# Custom fuzzer example: IDTech Firmware

Point-of-sale credit card reader for smart phones

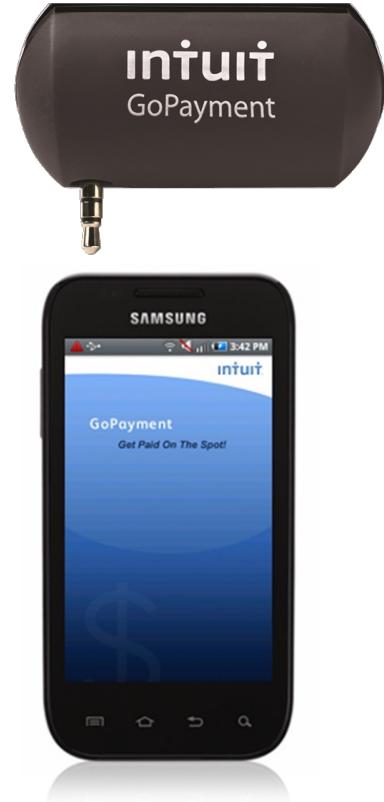
- No access to firmware binary, let alone source code

## What we did:

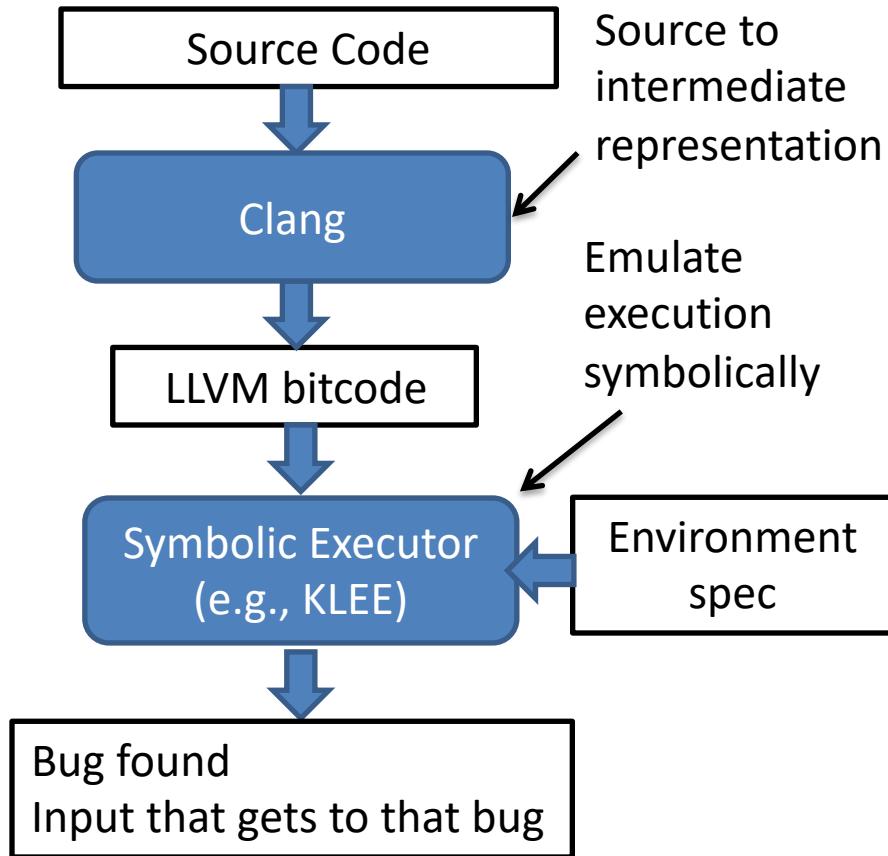
- Scour web for documentation
- Download SDK for app developers (bingo!)
  - Use to build fuzzing functionality for each API call we could find
  - Use to discover undocumented API calls

## Results:

- Found unauthenticated API calls, buffer bound check problems, ultimately brick device, reveals crypto keys, cleartext credit card data



# Symbolic execution



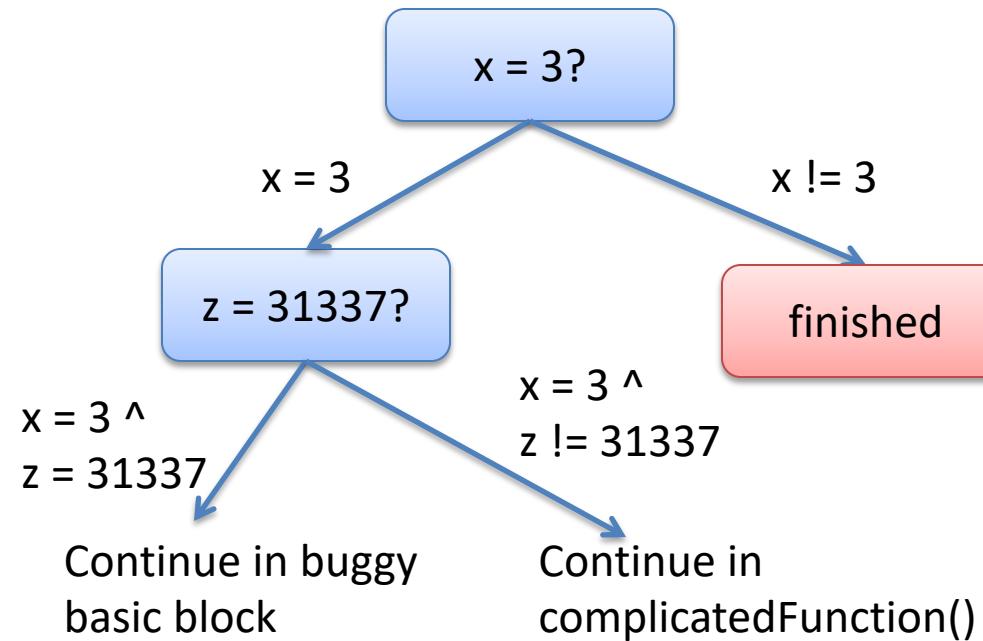
- Technique for analyzing code paths and finding inputs
- Associate to input variables *symbols*
  - called symbolic variable
- Simulate execution symbolically
  - Update symbolic variable's value appropriately
  - Conditionals add constraints on possible values
- Cast constraints as satisfiability, and use SAT solver to find inputs
- Perform security checks at each execution state

# Symbolic execution

```
main( int argc, char* argv[] ) {  
    char* b1;  
    char* b2;  
    char* b3;  
  
    if( argc != 3 ) then return 0;  
    if( argv[2] != 31337 )  
        complicatedFunction();  
    else {  
        b1 = (char*)malloc(248);  
        b2 = (char*)malloc(248);  
        free(b1);  
        free(b2);  
        b3 = (char*)malloc(512);  
        strncpy( b3, argv[1], 511 );  
        free(b2);  
        free(b3);  
    }  
}
```

Initially:

argc =  $x$  (unconstrained int)  
argv[2] =  $z$  (memory array)



- Eventually emulation hits a double free
- Path to error gives constraint equation over inputs, solve to give test inputs

# Symbolic execution challenges

- **Can we complete analyses?**
  - Yes, but only for very simple programs
  - Exponential # of paths to explore
  - Each branch increases state size of symbolic emulator
- **Path selection**
  - Which state to explore next?
  - Might get stuck in complicatedFunction()
- **Encoding checks on symbolic states**
  - Must include logic for double free check
  - Symbolic execution on binary more challenging (lose most memory semantics)
- **Dealing with the environment (“the environment problem”)**
  - What are the inputs? How to handle system calls to network, file system, etc.

# Klee paper

- Prior work by group on EXE symbolic executor
  - Builds off innovations in fast SMT solving
- New system Klee
  - Paper introduces design and evaluation of new symbolic executor
  - Real tool development (it is a pain to use, but lots have used it, still being used in research)
  - Significant attention paid to evaluation
- >3k citations (a lot in this area) and lots of academic impact

# Klee paper: the writeup

- Motivation & framing
  - How does the intro motivate & excite the reader about Klee?
  - How are they evidencing its efficacy?
- Case study / example (Overview)
- Architecture, then Environment problem
- Evaluation (almost 7 pages of evaluation)
  - Not so obvious how to measure coverage
  - Lots of different programs used, mix of commercial plus research

# Example tools / approaches

Approach	Type	Comment
Lexical analyzers	Static analysis	Perform syntactic checks Ex: LINT, RATS, ITS4
Fuzz testing	Dynamic analysis	Run on specially crafted inputs to test
Symbolic execution	Emulated execution	Run program on many inputs at once, by Ex: KLEE, S2E, FiE
Model checking	Static analysis	Abstract program to a model, check that model satisfies security properties Ex: MOPS, SLAM, etc.

# Bug finding is a big business

- Grammatech (Cornell startup, 1988)
- Coverity (Stanford startup)
- Fortify
- many, many others...

Great article on static analysis in the real world:  
<http://web.stanford.edu/~engler/BLOC-coverity.pdf>

- Reverse engineers, exploit developers, & zero-day markets
  - Hacking team
- Bug bounty programs