greybox fuzzing / static analysis / taint tracking

on testing

challenges with testing for security:

security bugs use “unrealistic” inputs — e.g. > 8000 character name

memory errors often don’t crash

on testing

challenges with testing for security:

security bugs use “unrealistic” inputs — e.g. > 8000 character name

~~memory errors often don’t crash~~

bounds checking, etc. tools will fix

automatic testing tools

basic idea: generate lots of random inputs — "*fuzzing*"

- easy to generate weird inputs

look for memory errors

- segfaults, or
- use memory error detector, or
- add (slow) 'assertions' or other checks to code

one of the most common ways to find security bugs

‘blackbox’ fuzzing

```
void fuzzTestImageParser(std::vector<byte> &originalImage) {
  for (int i = 0; i < NUM_TRIES; ++i) {
    std::vector<byte> testImage;
    testImage = originalImage;
    int numberOfChanges = rand() % MAX_CHANGES;
    for (int j = 0; j < numberOfChanges; ++j) {
      /* flip some random bits */
      testImage[rand() % testImage.size()] ^= rand() % 256;
    }
    int result = TryToParseImage(testImage);
    if (result == CRASH) ...
  }
}
```

‘blackbox’ fuzzing

```
void fuzzTestImageParser(std::vector<byte> &originalImage) {
  for (int i = 0; i < NUM_TRIES; ++i) {
    std::vector<byte> testImage;
    testImage = originalImage;
    int numberOfChanges = rand() % MAX_CHANGES;
    for (int j = 0; j < numberOfChanges; ++j) {
      /* flip some random bits */
      testImage[rand() % testImage.size()] ^= rand() % 256;
    }
    int result = TryToParseImage(testImage);
    if (result == CRASH) ...
  }
}
```

‘blackbox’ fuzzing

```
void fuzzTestImageParser(std::vector<byte> &originalImage) {
  for (int i = 0; i < NUM_TRIES; ++i) {
    std::vector<byte> testImage;
    testImage = originalImage;
    int numberOfChanges = rand() % MAX_CHANGES;
    for (int j = 0; j < numberOfChanges; ++j) {
      /* flip some random bits */
      testImage[rand() % testImage.size()] ^= rand() % 256;
    }
    int result = TryToParseImage(testImage);
    if (result == CRASH) ...
  }
}
```

blackbox fuzzing pros

works with *unmodified software*

even with embedded assembly, etc.

works with many kinds of input

don't need to understand input format

easy to *parallelize*

has actually found lots of bugs

‘blackbox’?

the program is a “black box” — can’t look inside

we only run it, see if it works

for memory errors — works $\approx$ doesn’t crash

what can fuzzing find

easiest to find crashes

- intuition: segfault could be security problem

otherwise: how do we know if test cases are useful?

need some way to know if test result is correct

example: fuzz-testing of C compilers versus other C compilers

- Yang et al, "Finding and Understanding Bugs in C compilers", 2011
- 79 GCC, 209 Clang bugs
- about one third "wrong generated code"
- but using smarter fuzzing strategy (we'll talk about it later)

testing for non-memory flaws?

fuzzing for cross-site scripting bugs?

- run on web application
- assert that HTML is well-formed?

fuzzing for SQL injection?

- assert that no malformed SQL gets executed?

operating system?

- input = requests (system calls) to make to the OS

(less likely) fuzzing for permissions issues?

- assert that admin. data doesn't change?

fuzzing challenges

isolation:

need to *detect crashes*/etc. reliably

want *reproducible test cases*

need to distinguish *hangs* from "machine is randomly slow"

speed:

need to run *many millions of tests*

application startup times are a problem

completeness:

might have to get *really* lucky to make interesting input

fuzzing challenges

isolation:

- need to *detect crashes*/etc. reliably
- want *reproducible test cases*
- need to distinguish *hangs* from "machine is randomly slow"

speed:

- need to run *many millions of tests*
- application startup times are a problem

completeness:

- might have to get *really* lucky to make interesting input

completeness problem

let's say we're testing an HTML parser

what code is **usually** going to when we flip random bits?

(or remove/add random bytes)

completeness problem

let's say we're testing an HTML parser

what code is **usually** going to when we flip random bits?

(or remove/add random bytes)

how often are we going to generate tags not in starting document?

how often are we going to generate new almost-valid documents?

HTML with changes

```
<html><head><title>A</title></head><body>B</body></html>
<html*<head><title>A</title></head><body>B</body></html>
<html><iead><title>C</title></head><body>B</body></html>
```

CSmith

Yang et al wrote a random C program generator

“Finding and Understanding Ubgs in C compilers” (PLDI 2011)

carefully avoided code with unspecified effects

most of the work was about doing this

don’t need to know what program does: comparing two compilers

or one compiler with different settings

random selection of types, operators, etc.

...instead of just random bytes

CReduce

Regher et al (including Yang)'s follow-up work

"Test-Case Reduction for C Compiler Bugs" (PLDI 2012)

take a C program that triggers bug...

try removing things to make it smaller

needed: automated way of checking "is bug still there"

same idea applies to security bugs

remove as much as possible and get it to still segfault

thinking about testing

```
void expand(char *arg) {
    if (arg[0] == '[') {
        if (arg[2] != '-' || arg[4] != ']') {
            putchar('[');
            expand(&arg[1]);
        } else {
            for (int i = arg[1]; i <= arg[3]; ++i) {
                putchar(i);
            }
            expand(&arg[5]);
        }
    } else if (arg[0] != '\0') {
        putchar(arg[0]);
        expand(&arg[1]);
    }
}
```

coverage

"coverage": metric for how good tests are

% of code reached

easy to measure

correlates with bugs found

but not the same thing as finding all bugs

automated test generation

conceptual idea: look at code, go down *all paths*

seems automatable?

- just need to identify conditions for each path

a compromise: coverage-guided fuzzing

symbolic execution: try to maximize paths run...

by finding potential paths, solving to run them

observation: easy to measure which paths a test case uses

way, way, way easier than solving eqn to find a case for that path

can make random tests *biased towards finding new paths*

coverage-guided example

```
void foo(int a, int b) {
    if (a != 0) {
        // W
        b -= 2;
        a += b;
    } else {
        // X
    }
    if (b < 5) {
        // Y
        b += 4;
        if (a + b > 50) {
            // Q
            ...
        }
    } else {
        // Z
    }
}
```

initial test case A:
a = 0x17, b = 0x08; covers: WZ

coverage-guided example

```
void foo(int a, int b) {
    if (a != 0) {
        // W
        b -= 2;
        a += b;
    } else {
        // X
    }
    if (b < 5) {
        // Y
        b += 4;
        if (a + b > 50) {
            // Q
            ...
        }
    } else {
        // Z
    }
}
```

initial test case A:
a = 0x17, b = 0x08; covers: WZ

generate random tests based on A

a = 0x37, b = 0x08; covers: WZ
a = 0x15, b = 0x08; covers: WZ
a = 0x17, b = 0x0c; covers: WZ
a = 0x13, b = 0x08; covers: WZ
a = 0x17, b = 0x08; covers: WZ
…
a = 0x17, b = 0x00; covers: *WY*

coverage-guided example

```
void foo(int a, int b) {
    if (a != 0) {
        // W
        b -= 2;
        a += b;
    } else {
        // X
    }
    if (b < 5) {
        // Y
        b += 4;
        if (a + b > 50) {
            // Q
            ...
        }
    } else {
        // Z
    }
}
```

initial test case A:
a = 0x17, b = 0x08; covers: WZ

found test case B:
a = 0x17, b = 0x00; covers: WY

coverage-guided example

```
void foo(int a, int b) {
    if (a != 0) {
        // W
        b -= 2;
        a += b;
    } else {
        // X
    }
    if (b < 5) {
        // Y
        b += 4;
        if (a + b > 50) {
            // Q
            ...
        }
    } else {
        // Z
    }
}
```

initial test case A:
a = 0x17, b = 0x08; covers: WZ

found test case B:
a = 0x17, b = 0x00; covers: WY

generate random tests based on A, B

a = 0x37, b = 0x08; covers: WZ
a = 0x04, b = 0x00; covers: WY
a = 0x17, b = 0x01; covers: WZ
a = 0x16, b = 0x00; covers: WY
...
a = 0x97, b = 0x00; covers: *WYQ*
...
a = 0x00, b = 0x08; covers: *XY*

coverage-guided example

```
void foo(unsigned a,
         unsigned b,
         unsigned c) {
    if (a != 0) {
        b -= c; // W
    }
    if (b < 5) {
        if (a > c) {
            a += b; // X
        }
        b += 4; // Y
    } else {
        a += 1; // Z
    }
    assert(a + b != 7);
}
```

initial test case A:
a = 0x17, b = 0x08, c = 0x00; covers: WZ

coverage-guided example

```
void foo(unsigned a,
         unsigned b,
         unsigned c) {
    if (a != 0) {
        b -= c; // W
    }
    if (b < 5) {
        if (a > c) {
            a += b; // X
        }
        b += 4; // Y
    } else {
        a += 1; // Z
    }
    assert(a + b != 7);
}
```

initial test case A:
a = 0x17, b = 0x08, c = 0x00; covers: WZ

generate random tests based on A

a = 0x37, b = 0x08, c = 0x00; covers: WZ
a = 0x15, b = 0x08, c = 0x02; covers: WZ
a = 0x17, b = 0x0c, c = 0x00; covers: WZ
a = 0x13, b = 0x08, c = 0x40; covers: WZ
a = 0x17, b = 0x08, c = 0x10; covers: WZ
…
a = 0x17, b = 0x00, c = 0x01; covers: *WXY*

coverage-guided example

```
void foo(unsigned a,
         unsigned b,
         unsigned c) {
    if (a != 0) {
        b -= c; // W
    }
    if (b < 5) {
        if (a > c) {
            a += b; // X
        }
        b += 4; // Y
    } else {
        a += 1; // Z
    }
    assert(a + b != 7);
}
```

initial test case A:
a = 0x17, b = 0x08, c = 0x00; covers: WZ

found test case B:
a = 0x17, b = 0x00, c = 0x01; covers: WXY

coverage-guided example

```
void foo(unsigned a,
         unsigned b,
         unsigned c) {
    if (a != 0) {
        b -= c; // W
    }
    if (b < 5) {
        if (a > c) {
            a += b; // X
        }
        b += 4; // Y
    } else {
        a += 1; // Z
    }
    assert(a + b != 7);
}
```

initial test case A:
a = 0x17, b = 0x08, c = 0x00; covers: WZ

found test case B:
a = 0x17, b = 0x00, c = 0x01; covers: WXY

generate random tests based on A, B

a = 0x37, b = 0x08, c = 0x00; covers: WZ
a = 0x17, b = 0x00, c = 0x03; covers: WXY
a = 0x17, b = 0x0c, c = 0x00; covers: WZ
a = 0x37, b = 0x00, c = 0x03; covers: WXY
a = 0x17, b = 0x08, c = 0x10; covers: WZ
...
a = 0x17, b = 0x00, c = 0x81; covers: *WY*

exercise: coverage guidance good for?

```
void example1(int a, int b) {
    if (a < 4 && b < 4 && a == b) {
        assert(a + b != 6);
    }
}
void example2(int a, int b) {
    assert(a != 10325);
}
void example3(int a, int b) {
    assert(a != 10325 && b != 10543);
}
```

exercise: for which of these functions would coverage guided fuzzing be most/least better than random testing for making the assertion fail?

american fuzzy lop

one example of a fuzzer that uses this strategy

“whitebox fuzzing”

assembler wrapper to record computed/conditional jumps:

```
CoverageArray[Hash(JumpSource, JumpDest)]++;
```

use values from coverage array to distinguish cases

outputs only *unique* test cases

goal: test case for every possible jump source/dest

american fuzzy lop heuristics

american fuzzy lop does some deterministic testing

- try flipping every bit, every 2 bits, etc. of base input
- overwrite bytes with 0xFF, 0x00, etc.
- etc.

has many strategies for producing new inputs

- bit-flipping
- duplicating important-looking keywords
- combining existing inputs

automatically simplifying test cases

same idea as fuzzing

but look for *same result/coverage*

systematic simplifications:

- try removing every character (one-by-one)
- try decrementing every byte
- ...

keep simplifications that don't change result

AFL uses some of this strategy to help get better 'base' tests

- also has tool to do this on a found test
- prefers simpler 'base' tests

AFL: manual keywords

AFL supports a dictionary

- list of things to add to create test cases
- example: all possible HTML tags

other strategy: test-case template

other strategy: test postprocessing (fix checksums, etc.)