bug-finding

logistics: ROP assignment

2013 memory safety landscape

SoK: Eternal War in Memory

László Szekeres[†], Mathias Payer[‡], Tao Wei[*‡], Dawn Song[‡]
[†]*Stony Brook University*
[‡]*University of California, Berkeley*
[*]*Peking University*

2013 memory safety landscape

	Policy type (main approach)	Technique	Perf. % (avg/max)	Dep.	Compatibility	Primary attack vectors
Generic prot.	Memory Safety	SofBound + CETS	116 / 300	×	Binary	—
		SoftBound	67 / 150	×	Binary	UAF
		Baggy Bounds Checking	60 / 127	×	—	UAF, sub-obj
	Data Integrity	WIT	10 / 25	×	Binary/Modularity	UAF, sub-obj, read corruption
	Data Space Randomization	DSR	15 / 30	×	Binary/Modularity	Information leak
	Data-flow Integrity	DFI	104 / 155	×	Binary/Modularity	Approximation
CF-Hijack prot.	Code Integrity	Page permissions (R)	0 / 0	✓	JIT compilation	Code reuse or code injection
	Non-executable Data	Page permissions (X)	0 / 0	✓	JIT compilation	Code reuse
	Address Space Randomization	ASLR	0 / 0	✓	Relocatable code	Information leak
		ASLR (PIE on 32 bit)	10 / 26	×	Relocatable code	Information leak
	Control-flow Integrity	Stack cookies	0 / 5	✓	—	Direct overwrite
		Shadow stack	5 / 12	×	Exceptions	Corrupt function pointer
		WIT	10 / 25	×	Binary/Modularity	Approximation
		Abadi CFI	16 / 45	×	Binary/Modularity	Weak return policy
		Abadi CFI (w/ shadow stack)	21 / 56	×	Binary/Modularity	Approximation

Table II

THIS TABLE GROUPS THE DIFFERENT PROTECTION TECHNIQUES ACCORDING TO THEIR POLICY AND COMPARES THE PERFORMANCE IMPACT, DEPLOYMENT STATUS (DEP.), COMPATIBILITY ISSUES, AND MAIN ATTACK VECTORS THAT CIRCUMVENT THE PROTECTION.

different design points

memory safety most extreme — disallow out of bounds

usually even making out-of-bounds pointers

relaxations:

separate 'safe' data like buffers and 'unsafe' data like return addresses

instead of all objects from each other

check only writes or only reads

...

the mitigations

things the OS/compiler can do

assume software won't or can't be fixed

goal: make programs better despite lack of effort by developers

in practice: hard to get >10% overhead mitigations deployed

what else can we do?

alternative techniques

memory error detectors — to help with software **testing**

- reliably detect single-byte overwrites, use-after-free
- bitmap for every bit of memory — should this be accessed
- **not** suitable for stopping exploits
- examples: AddressSanitizer, Valgrind MemCheck

automatic testing tools — run programs to trigger memory bugs

static analysis — analyze programs and either

- find likely memory bugs, or
- prove absence of memory bugs

better programming languages

alternative techniques

memory error detectors — to help with software **testing**

- reliably detect single-byte overwrites, use-after-free
- bitmap for every bit of memory — should this be accessed
- **not** suitable for stopping exploits
- examples: AddressSanitizer, Valgrind MemCheck

automatic testing tools — run programs to trigger memory bugs

static analysis — analyze programs and either

- find likely memory bugs, or
- prove absence of memory bugs

better programming languages

recall: baggy bounds

check on pointer manipulation

make sure pointer maps to same object

more typical solution: check on read/write

goal was to run programs safely — can also find bounds errors

testing workflow

use a tool like baggy bounds to make errors **crash**

run thorough tests of software; fix any crashes

idea: overhead is okay when debugging

can you use Baggy Bounds?

not released in useable form as far as I know

but there are alternative tools that are available

...which are better fits for testing

AddressSanitizer

like baggy bounds:

- big lookup table
- lookup table set by memory allocations
- compiler modification: change stack allocations

unlike baggy bounds:

- check reads/writes (instead of pointer computations)
- only detect errors that read/write between objects
- object sizes not padded to power of two
- table has info for every single byte (more precise)

adding bounds-checking example

```
void vulnerable(long value, int offset) {
    long array[10] = {1,2,3,4,5,6,7,8,9,10};
    // generated code: (added by AddressSanitizer)
    if (!lookup_table[&array[offset]] == VALID) FAIL();
    array[offset] = value;
    do_something_with(array);
}
```

AddressSanitizer: crashes only if `array[offset]` isn't part of any object

but no extra space — single-byte precision

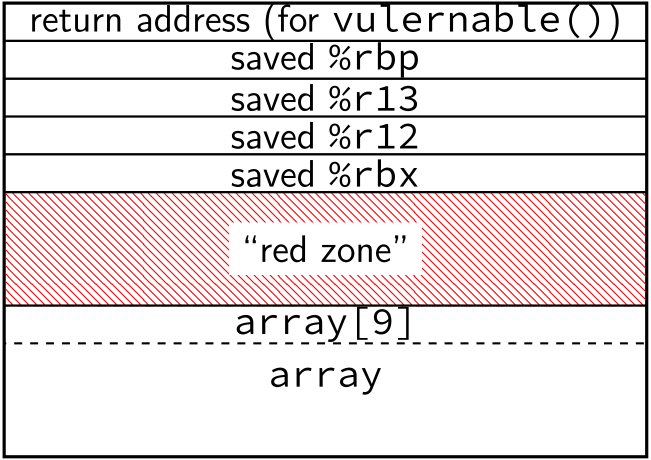
adding bounds-checking example

```
void vulnerable(long value, int offset) {
    long array[10] = {1,2,3,4,5,6,7,8,9,10};
    // generated code: (added by AddressSanitizer)
    if (!lookup_table[&array[offset]] == VALID) FAIL();
    array[offset] = value;
    do_something_with(array);
}
```

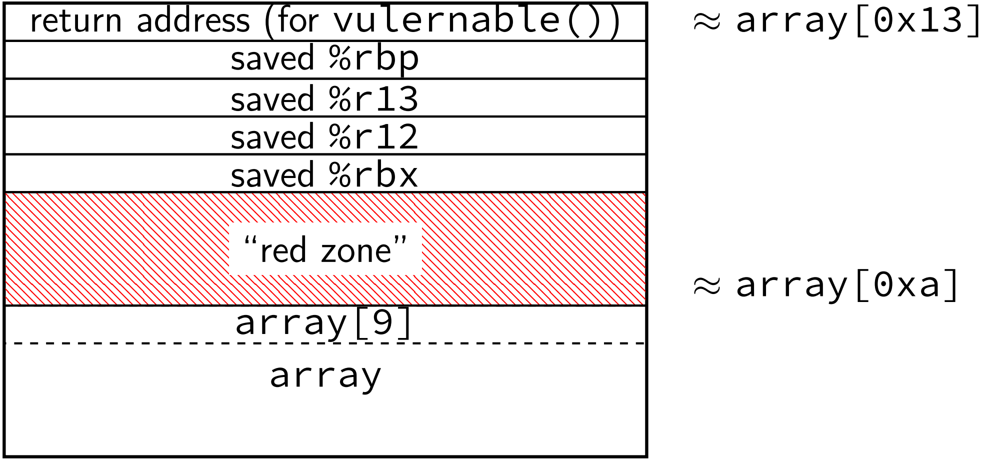
AddressSanitizer: crashes only if `array[offset]` isn't part of any object

but no extra space — single-byte precision

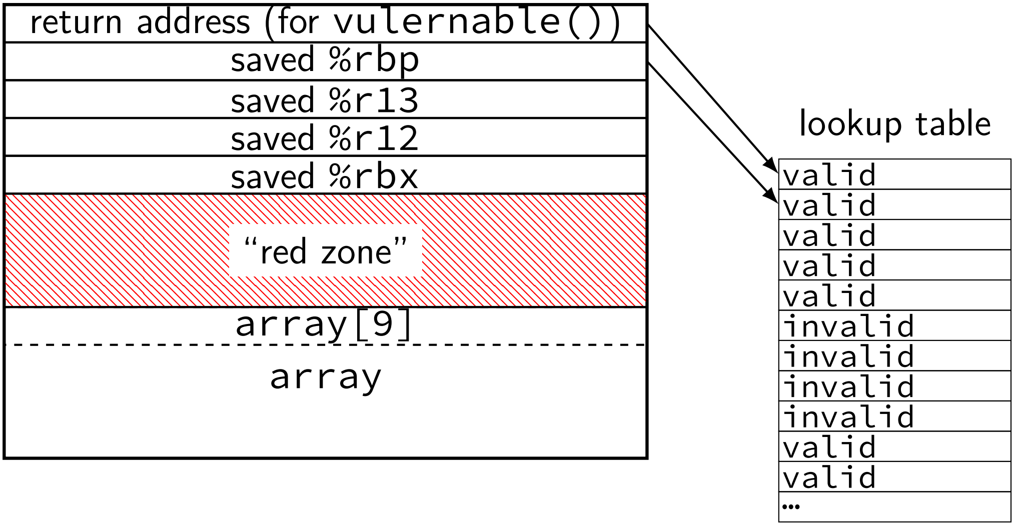
AddressSanitizer stack layout



AddressSanitizer stack layout



AddressSanitizer stack layout



AddressSanitizer versus Baggy Bounds

pros vs baggy bounds:

- you can actually use it (comes with GCC/Clang)
- byte-level precision — no "padding" on objects
- detects use-after-free a lot of the time

cons vs baggy bounds:

- doesn't prevent out-of-bounds "targetted" accesses
- requires extra space between objects
- usually slower

Valgrind Memcheck

similar to AddressSanitizer — but no compiler modificaitons

instead: is a virtual machine (plus alternate malloc/new implementation)

only (reliably) detects errors on heap

but works on unmodified binaries

alternative techniques

memory error detectors — to help with software **testing**

- reliably detect single-byte overwrites, use-after-free
- bitmap for every bit of memory — should this be accessed
- **not** suitable for stopping exploits
- examples: AddressSanitizer, Valgrind MemCheck

automatic testing tools — run programs to trigger memory bugs

static analysis — analyze programs and either

- find likely memory bugs, or
- prove absence of memory bugs

better programming languages

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

fuzz testing to find security holes

common way to find security holes

start with crash, then use debugger

how much control does attacker have?

is out-of-bounds/etc. overwriting important things?

return address? object with VTable? ...

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>
```

fuzzing from format knowledge (1)

make a random document generator

before: small number of manually chosen examples (often 1)

```
String RandomHTML() {
    if (random() > 0.2) {
        String tag = GetRandomTag();
        if (random() > 0.2) {
            return "<" + tag + ">" + RandomHTML() +
                 "</" + tag + ">";
        } else {
            return "<" + tag + ">";
        }
    } else
        return RandomText();
}
```

fuzzing from format knowledge (2)

other fuzzing strategies

identify interesting fields to fuzz

- description of grammar/protocol/etc.
- test different values seperately

(default to) filling in sizes, checksums, type information

- avoid most inputs getting rejected from being malformed
- test specific parts of a larger program

thinking about testing

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

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

symbolic execution

have an emulator/virtual machine

but represent input values as symbolic variables

  like in algebra

choose a path through the program, track constraints

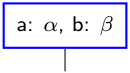
  what values did input need to have to get here?

then solve constraints based on variables to create real test case

  no solution? impossible path

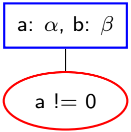
  find solution? test case

example




```
void foo(int a, int b) {
  if (a != 0) {
    b -= 2;
    a += b;
  }
  if (b < 5) {
    b += 4;
  }
  assert(a + b != 5);
}
```

example




```
void foo(int a, int b) {
  if (a != 0) {
    b -= 2;
    a += b;
  }
  if (b < 5) {
    b += 4;
  }
  assert(a + b != 5);
}
```

every variable represented as an equation

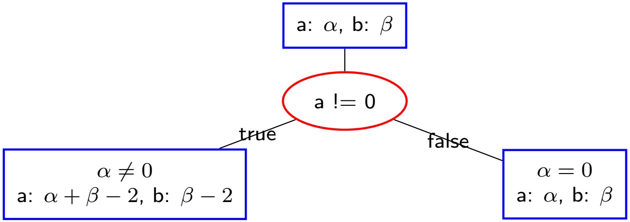
final step: generate solution for each path

100% test coverage

Adapted from Hicks, "Symbolic Execution for Finding Bugs"

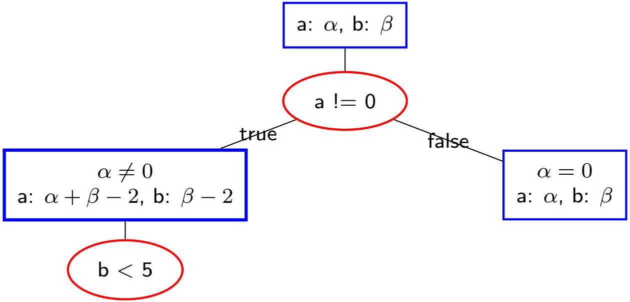
example

```
void foo(int a, int b) {
  if (a != 0) {
    b -= 2;
    a += b;
  }
  if (b < 5) {
    b += 4;
  }
  assert(a + b != 5);
}
```



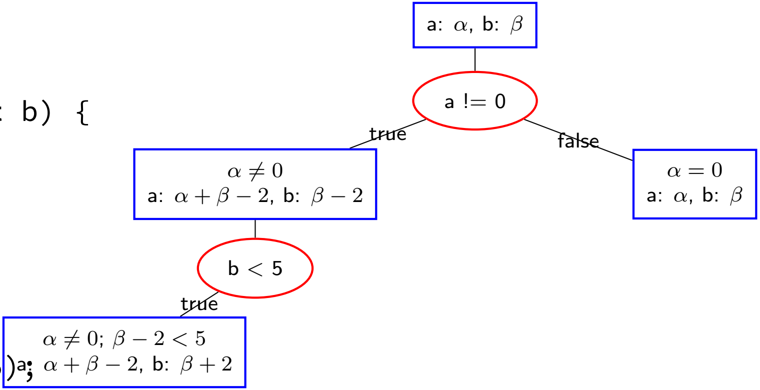
example

```
void foo(int a, int b) {
  if (a != 0) {
    b -= 2;
    a += b;
  }
  if (b < 5) {
    b += 4;
  }
  assert(a + b != 5);
}
```



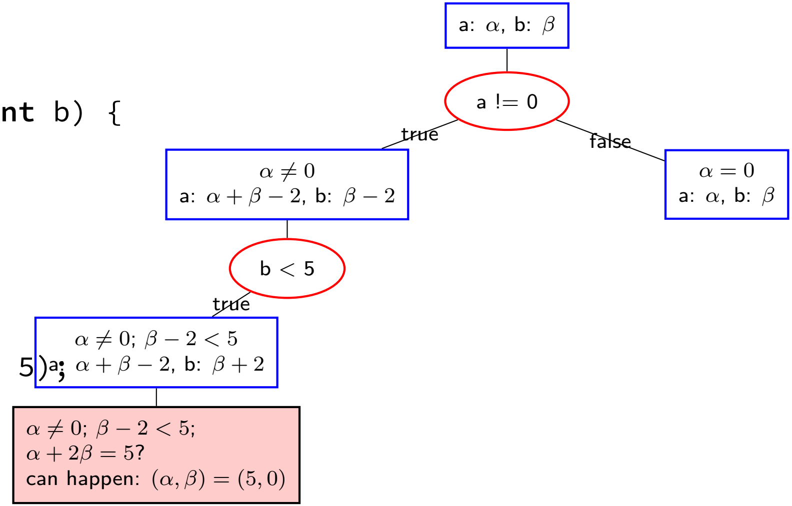
example

```
void foo(int a, int b) {
  if (a != 0) {
    b -= 2;
    a += b;
  }
  if (b < 5) {
    b += 4;
  }
  assert(a + b != 5);
}
```



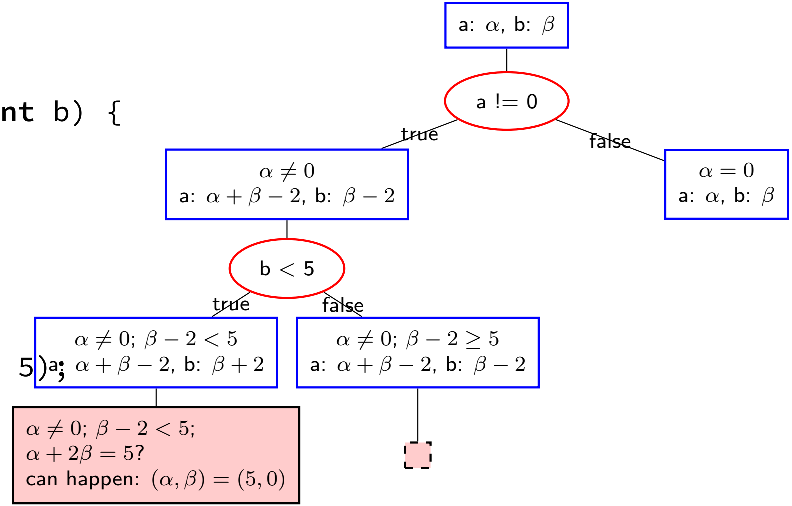
example

```
void foo(int a, int b) {
  if (a != 0) {
    b -= 2;
    a += b;
  }
  if (b < 5) {
    b += 4;
  }
  assert(a + b != 5);
}
```



example

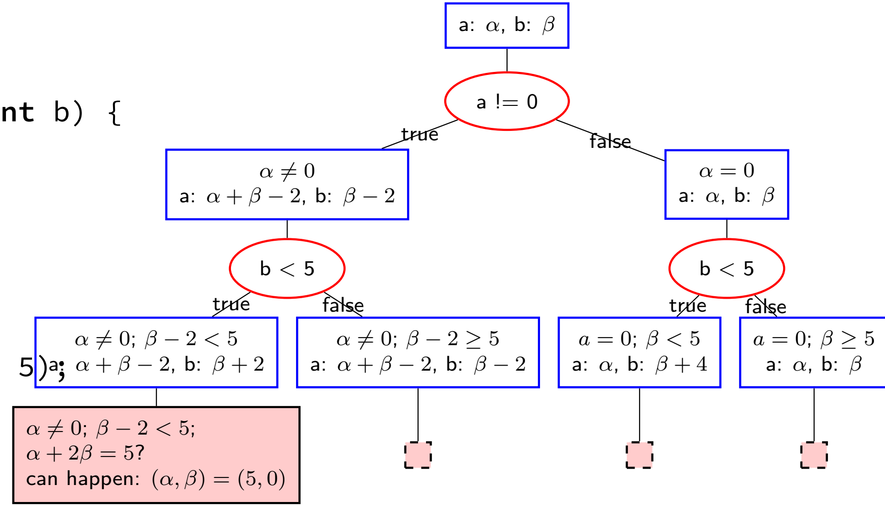
```
void foo(int a, int b) {
  if (a != 0) {
    b -= 2;
    a += b;
  }
  if (b < 5) {
    b += 4;
  }
  assert(a + b != 5);
}
```



Adapted from Hicks, "Symbolic Execution for Finding Bugs"

example

```
void foo(int a, int b) {
  if (a != 0) {
    b -= 2;
    a += b;
  }
  if (b < 5) {
    b += 4;
  }
  assert(a + b != 5);
}
```



symbolic execution challenges

‘solving’ a path’s conditions

generating way too many paths

equation solving

can generate formula with bounded inputs

can always be solved by trying all possibilities

but actually solving is NP-hard (i.e. not generally possible)

luck: there exists solvers that are *often* good enough

...for small programs

...with lots of additional heuristics to make it work

way too many paths

loops mean often really huge number of paths

dealing with array accesses?

easiest way — new path for each index

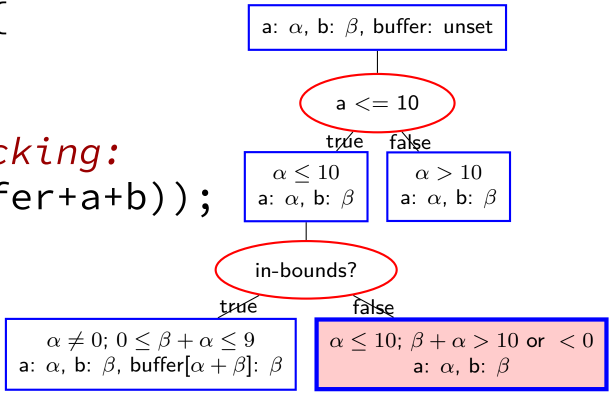
need ways to quickly eliminate impossible paths

won't explore all paths; need to prioritize

can try to similar paths; process together

paths for memory errors

```
void foo(int a, int b) {
  char buffer[10];
  if (a <= 10) {
    // added bounds-checking:
    assert(inBounds(buffer+a+b));
    buffer[a + b] = b;
  }
}
```



add bounds checking assertions — try to solve to satisfy

tricky parts in symbolic execution

dealing with pointers?

one method: one path for each valid value of pointer

solving equations?

NP-hard (boolean satisfiablity) — not practical in general

"good enough" for small enough programs/inputs

...after lots of tricks

how many paths?

$< 100\%$ coverage in practice

small input sizes (limited number of variables)

real symbolic execution

not yet used much outside of research

old technique (1970s), but recent resurgence

- equation solving ('SAT solvers') is now better

useful for more than test-case generation

example usable tool: KLEE (test case generating)

a compromise: coverage-guided fuzzing

idea: generate random test cases based on good test cases

test case goodness based on what code is run

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

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

simplification/keyword finding

see if each character changes coverage

find group of characters which matter — “keyword”?

example: `<html>` versus `<xtml>` etc.

find characters that don’t matter — remove

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.)

other uses of fuzzing tools

easiest to find crashes

but can check correctness if you have a way

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"

fuzzing assignment

target: a program that reindents C source files

tool: american fuzzy lop

along with AddressSanitizer — find crashes

- probably buffer overflows

crashes are easy to find — so won't have to fuzz for long

- but in real scenario would run fuzzer for hours/days

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