last time

"fat" pointers to add bounds-checking to C/C++

- huge performance/space overhead

problematic things C programmers do:

- string past the end of the struct
- casting pointer to X to pointer to struct starting with X
- offset pointers for 1-based indexing
- ...

baggy bounds checking

- round sizes to power of two
- checks on *pointer arithmetic*

AddressSanitizer

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

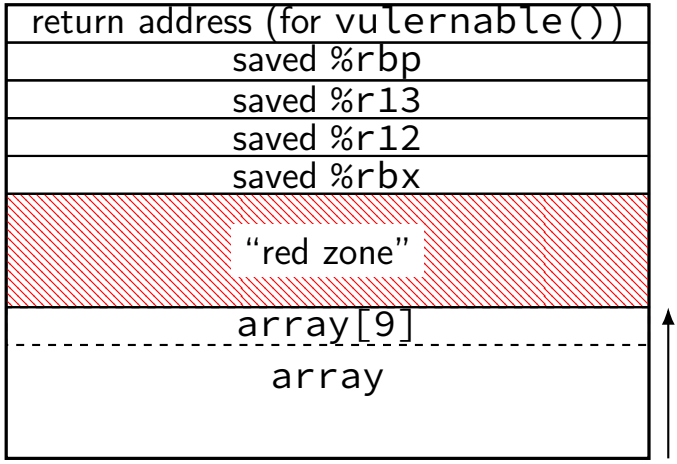
adding bounds-checking example

```
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);
}
```

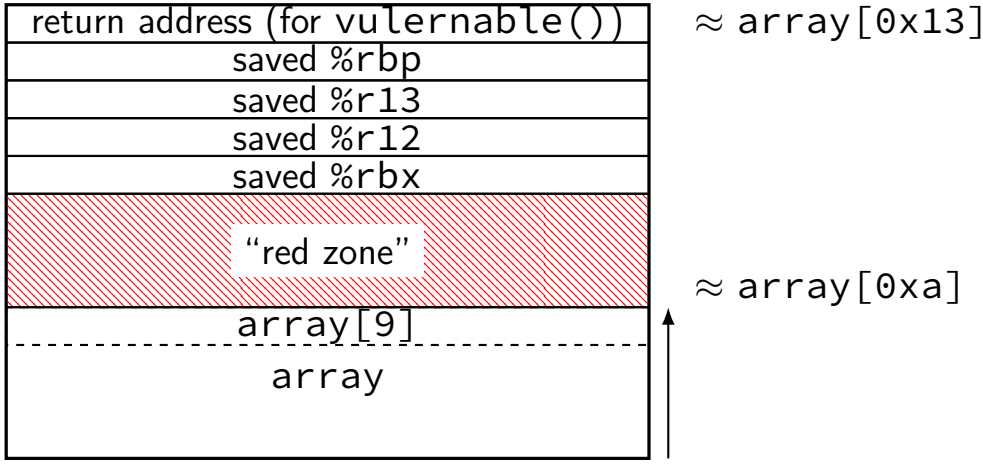
AddressSanitizer: crashes only if `array[offset]` isn't part of any object

- but no extra space — single-byte precision

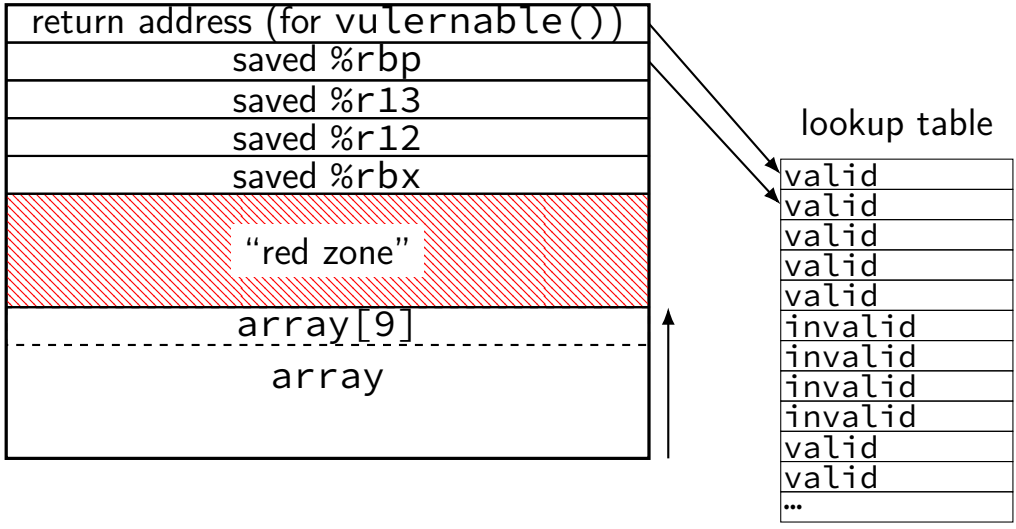
AddressSanitizer stack layout



AddressSanitizer stack layout



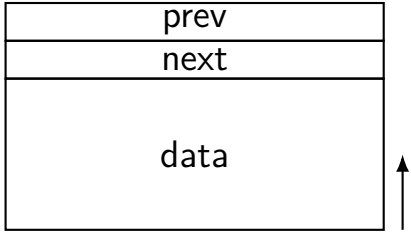
AddressSanitizer stack layout



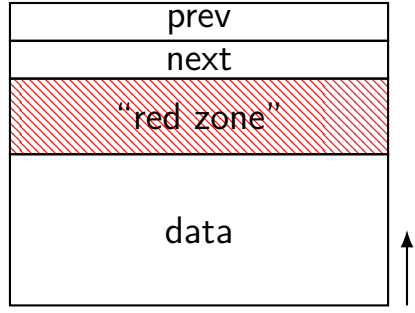
changing object layout?

```
struct string_list {
    char data[100];
    struct string_list *prev;
    struct string_list *next;
};
```

actual layout



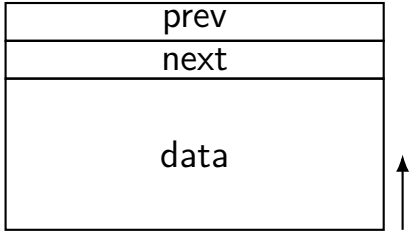
layout wanted for error-finding



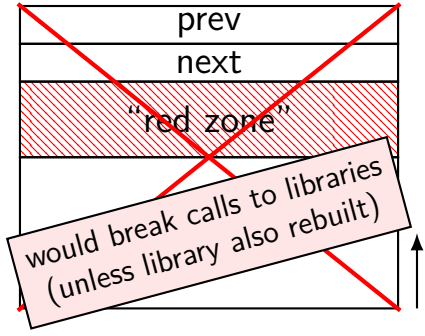
changing object layout?

```
struct string_list {
    char data[100];
    struct string_list *prev;
    struct string_list *next;
};
```

actual layout



layout wanted for error-finding



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

which scheme prevents...?

which schemes detect or prevent from being harmful...?

1. call to assembly code that goes beyond buffer?
2. allowing attacker to insert 150 bytes in 100 byte buffer on heap?
3. allowing attacker to insert 120 bytes in 100 byte buffer on stack?
4. attecker exploiting code that does array[attacker_index] to overwrite something outside heap array?

of:

A. "fat pointers" approach
B. Baggy Bounds checking
C. AddressSanitizer
D. Valgrind Memcheck

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

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

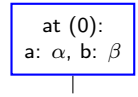
- what values did input need to have to get here?

then solve constraints based on variables to create real test case

- no solution? impossible path
- find solution? test case

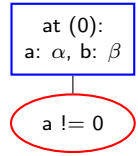
example 1

```
void foo(int a, int b) {
  /* (0) */
  if (a != 0) {
    b -= 2;
    a += b;
  }
  /* (1) */
  if (b < 5) {
    b += 4;
  }
  /* (2) */
  assert(a + b != 5);
}
```



example 1

```
void foo(int a, int b) {
  /* (0) */
  if (a != 0) {
    b -= 2;
    a += b;
  }
  /* (1) */
  if (b < 5) {
    b += 4;
  }
  /* (2) */
  assert(a + b != 5);
}
```



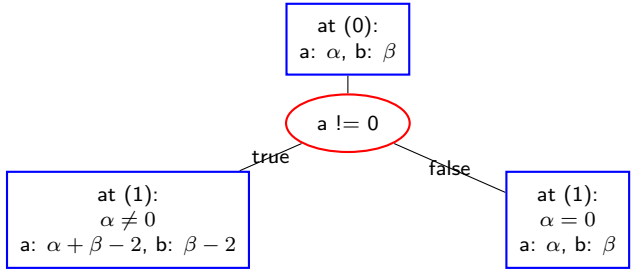
every variable represented as an equation

final step: generate solution for each path

100% test coverage

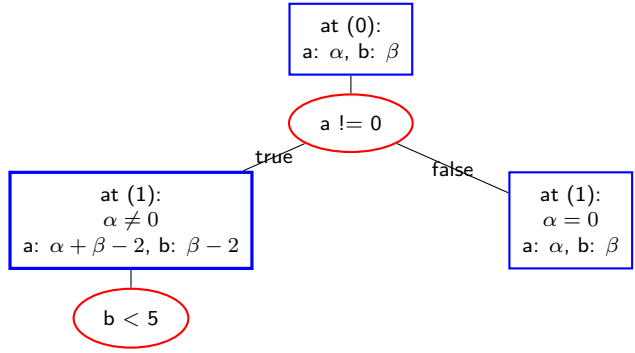
example 1

```
void foo(int a, int b) {
  /* (0) */
  if (a != 0) {
    b -= 2;
    a += b;
  }
  /* (1) */
  if (b < 5) {
    b += 4;
  }
  /* (2) */
  assert(a + b != 5);
}
```



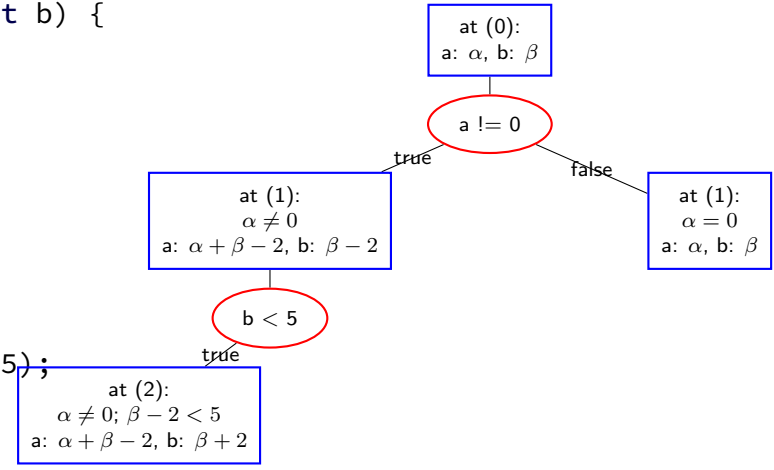
example 1

```
void foo(int a, int b) {
  /* (0) */
  if (a != 0) {
    b -= 2;
    a += b;
  }
  /* (1) */
  if (b < 5) {
    b += 4;
  }
  /* (2) */
  assert(a + b != 5);
}
```



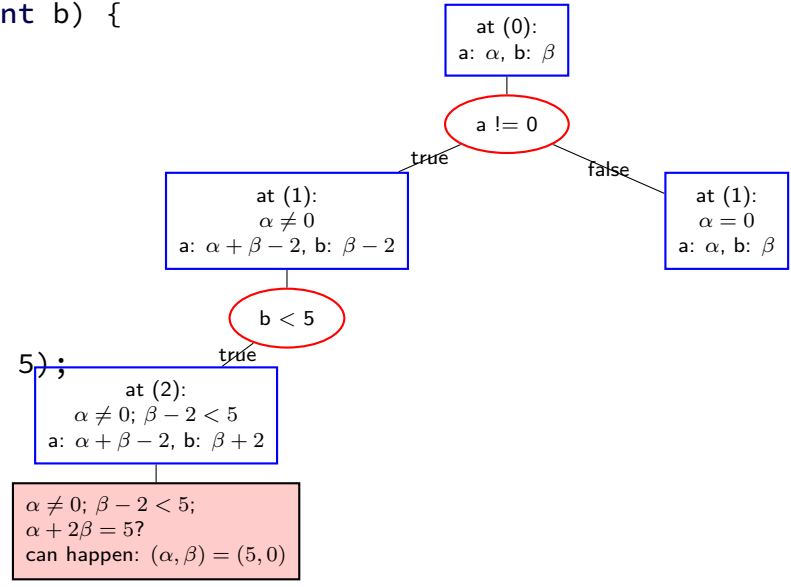
example 1

```
void foo(int a, int b) {
  /* (0) */
  if (a != 0) {
    b -= 2;
    a += b;
  }
  /* (1) */
  if (b < 5) {
    b += 4;
  }
  /* (2) */
  assert(a + b != 5);
}
```



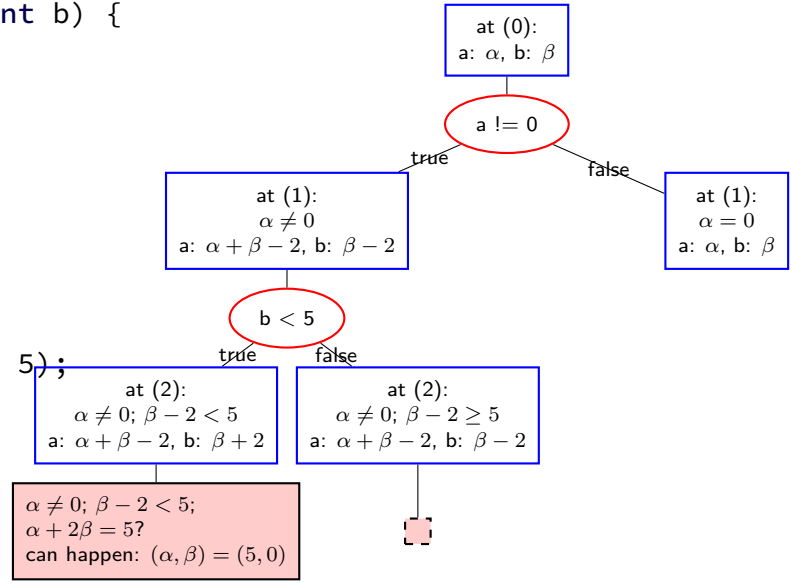
example 1

```
void foo(int a, int b) {
  /* (0) */
  if (a != 0) {
    b -= 2;
    a += b;
  }
  /* (1) */
  if (b < 5) {
    b += 4;
  }
  /* (2) */
  assert(a + b != 5);
}
```



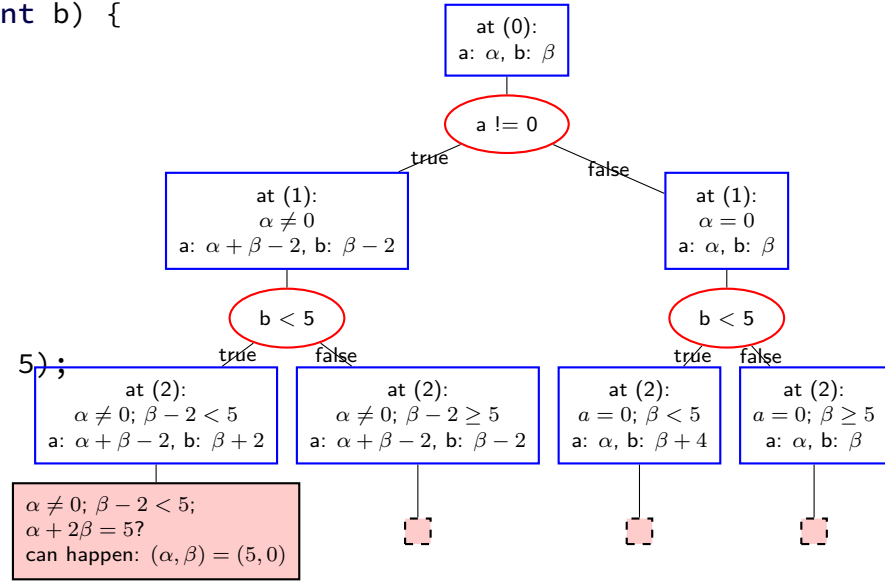
example 1

```
void foo(int a, int b) {
  /* (0) */
  if (a != 0) {
    b -= 2;
    a += b;
  }
  /* (1) */
  if (b < 5) {
    b += 4;
  }
  /* (2) */
  assert(a + b != 5);
}
```



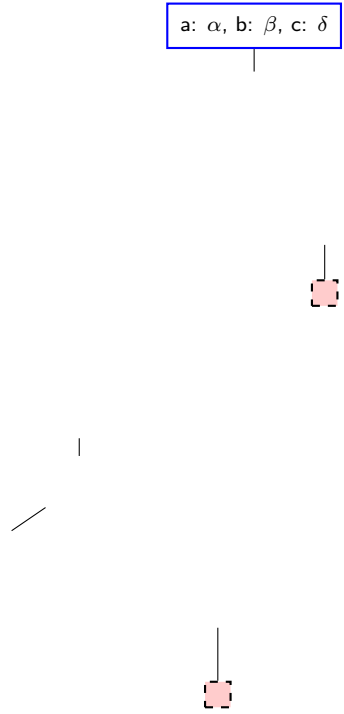
example 1

```
void foo(int a, int b) {
  /* (0) */
  if (a != 0) {
    b -= 2;
    a += b;
  }
  /* (1) */
  if (b < 5) {
    b += 4;
  }
  /* (2) */
  assert(a + b != 5);
}
```



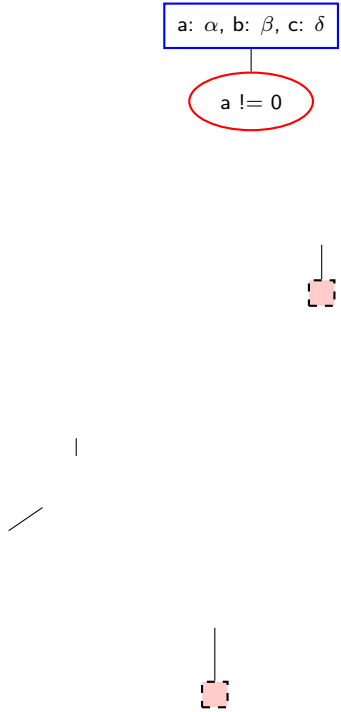
example 2

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



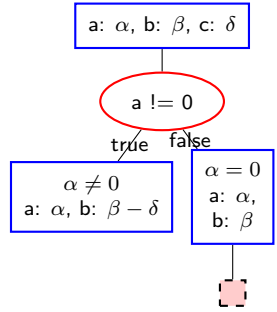
example 2

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



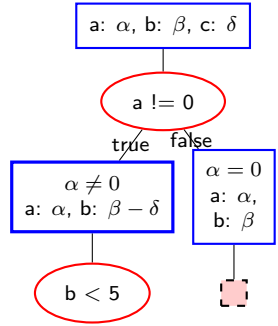
example 2

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



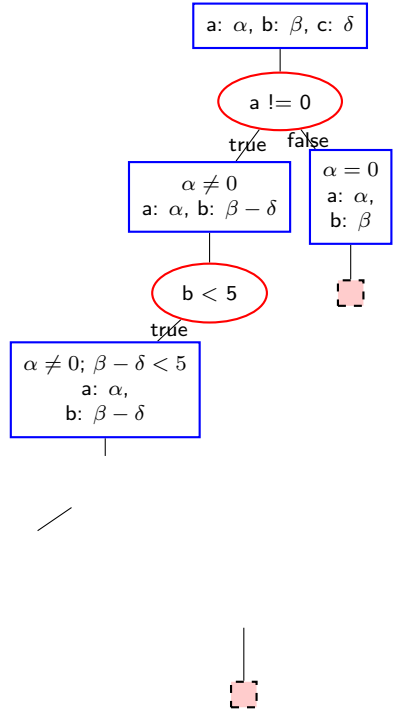
example 2

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



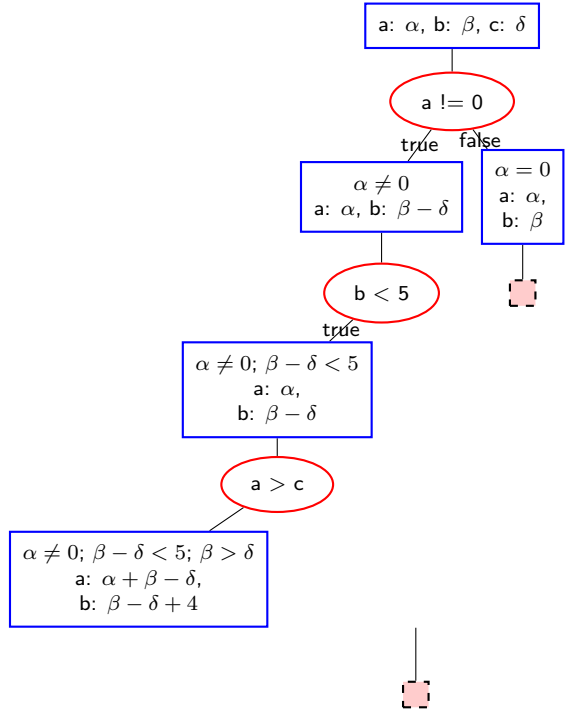
example 2

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



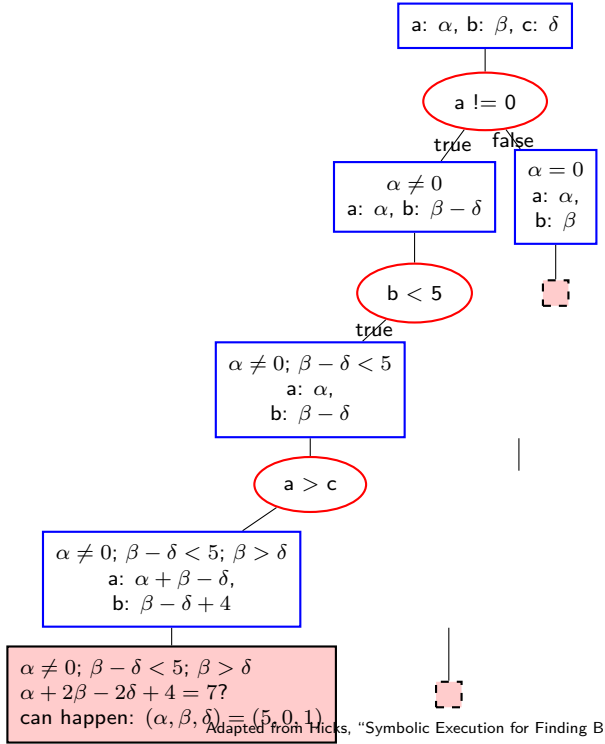
example 2

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



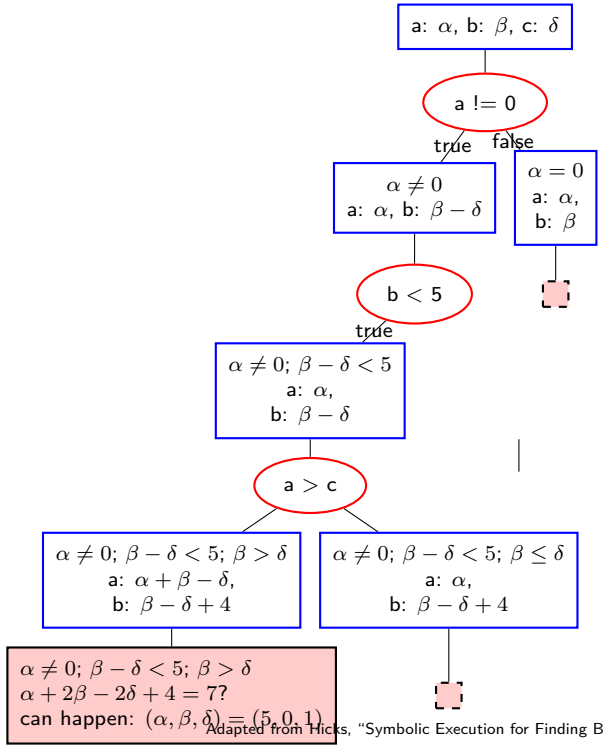
example 2

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



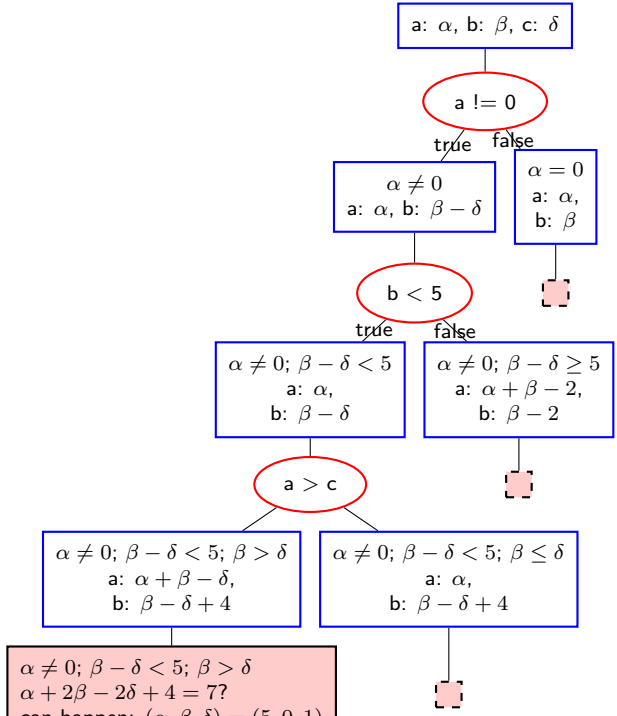
example 2

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



example 2

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



using for bounds checking

```
void foo() {
    char array[100];
    ...
    /* check inserted automatically: */
        assert(i >= 0 && i < 100);
    array[i] = ...;
    ...
}
```

using symbolic execution to find memory bugs?

add assertions for bounds checks

need to track array sizes to do symbolic execution anyways

example 3

```
unsigned a, b;
void foo(unsigned c) {
    int *p;
    if (a > 100) {
        p = &a;
    } else {
        p = &b;
    }
    *p += c;
    assert(a + b == c);
}
```