greybox fuzzing / static analysis / taint tracking

Changelog

12 April 2021: change direction of assertion on symbolic execution equation exercise

12 April 2021: completeness/soundness: correct description

12 April 2021: points-to diagram: correct arrows to C (not via B) and fixup its ID= values

last time (1)

AddressSanitizer, Valgrind Memcheck

- red zones between objects
- lookup table “is location valid”
- instrument memory reads/writes (not pointer arith)

random testing

- way to find memory errors, etc.
- mutating good inputs
- custom generators for formatted input? (e.g. HTML, C code)

last time (2)

symbolic execution

- make program values into algebriac variables
- solve equations to find if paths are possible
- systematic way to generate thorough test cases
- performance problems: slow equation solving, too many paths

exercise

```
void example(unsigned x, unsigned y) {
    if (x > y) return;
    x = x + y;
    assert(x + y + 1 > y);
}
```

1: to see if the assertion is meant, the equation we should solve (if initial values of x, y, are X, Y)?

2: what is an input that fails the assertion? (hint: integer overflow)

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

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’/‘SMT solvers’) is now much better

example usable tools: KLEE, symcc (test case generating)

KLEE optimizations

lots of optimizations to make search time pratical

prioritize paths that produce good tests

- try to execute *new code*
- try to find new paths new root of tree

reuse equation solving results:

- remove irrelevant variables from equation solving queries
  - e.g. if (x == 10) doesn't need variables unrelated to x's value
- cache of prior queries with "no solution"

results from 1 hour of compute time (from 2008 paper):

- avg. 91% coverage on Linux coreutils (basic command line tools)
- versus developer tests: 68% covergae
- (where coverage = % lines of code run ≠ % possible paths run)

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.)

fuzzing/symbolic exec imprecision

symbolic execution had some nice properties:

- could reliably enumerate possible paths
- could figure out inputs
- could prove paths are impossible

but had huge practical problems:

- not enough time/space to explore all those paths
- too complicated to actually solve equations to find inputs

greybox fuzzing: one practical compromise

- replaced equation solving with (educated) guessing
- tried to explore enough paths

complete versus sound

complete:

> if way to reach assertion failure, analysis finds it

sound:

> if analysis finds way to reach assertion failure, it's fails the assertion

symbolic execution, greybox fuzzing: always sound

> because they actually run the program

symbolic execution: complete **if all paths are solved**

> but that isn't practical for a large program

other program analysis designs

other design points than symbolic execution:

tracking all the varaible values

alternative: just track properties of interest

compute precisely what paths through code are possible

alternative: use some approximation

model for use-after-free

model for use-after-free, pointer is:

- allocated
- freed
- (other states?)

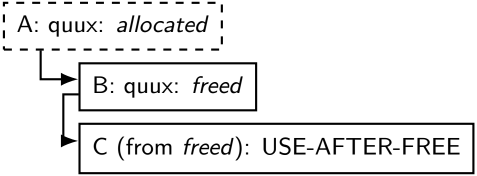
just track this logical state for each pointer

ignore everything else

assume all if statements/loop conditions can be true or false

checking use-after-free (1)

```
void someFunction(int foo, int bar) {
    int *quux = malloc(sizeof(int));
    // A
    ... /* omitted code that doesn't use quux */
    free(quux);
    // B
    ... /* omitted code that doesn't use quux */
    // C
    *quux = bar;
    ...
}
```



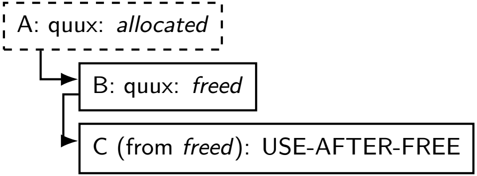
A: quux: *allocated*

B: quux: *freed*

C (from *freed*): USE-AFTER-FREE

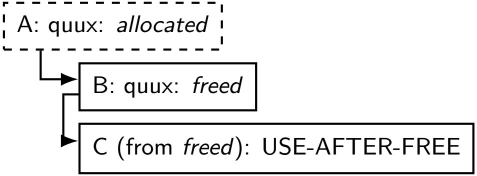
checking use-after-free (1)

```
void someFunction(int foo, int bar) {
    int *quux = malloc(sizeof(int));
    // A
    ... /* omitted code that doesn't use quux */
    free(quux);
    // B
    ... /* omitted code that doesn't use quux */
    // C
    *quux = bar;
    ...
}
```



checking use-after-free (1)

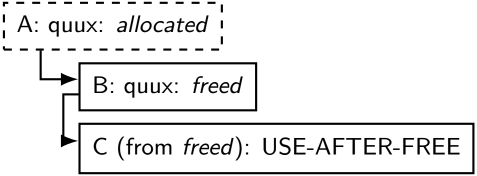
```
void someFunction(int foo, int bar) {
    int *quux = malloc(sizeof(int));
    // A
    ... /* omitted code that doesn't use quux */
    free(quux);
    // B
    ... /* omitted code that doesn't use quux */
    // C
    *quux = bar;
    ...
}
```



analysis can give warning — almost certainly bad

checking use-after-free (1)

```
void someFunction(int foo, int bar) {
    int *quux = malloc(sizeof(int));
    // A
    ... /* omitted code that doesn't use quux */
    free(quux);
    // B
    ... /* omitted code that doesn't use quux */
    // C
    *quux = bar;
    ...
}
```



A: quux: *allocated*

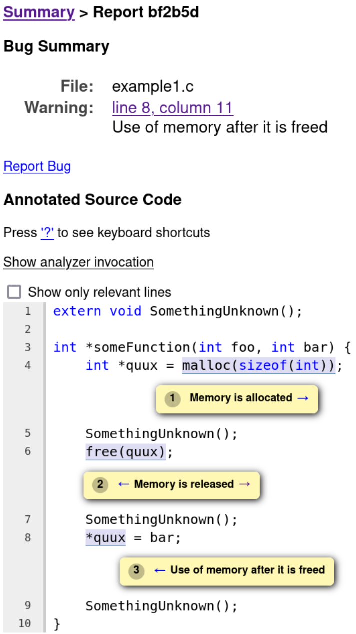
B: quux: *freed*

C (from *freed*): USE-AFTER-FREE

analysis can give warning — almost certainly bad

exercise: how could this be a false positive?

result from clang's scan-build



Summary > Report bf2b5d

Bug Summary

File: example1.c

Warning: line 8, column 11
Use of memory after it is freed

Report Bug

Annotated Source Code

Press '?' to see keyboard shortcuts

Show analyzer invocation

□ Show only relevant lines

```
 1  extern void SomethingUnknown();
 2
 3  int *someFunction(int foo, int bar) {
 4      int *quux = malloc(sizeof(int));
                  1  Memory is allocated →
 5      SomethingUnknown();
 6      free(quux);
        2  ← Memory is released →
 7      SomethingUnknown();
 8      *quux = bar;
             3  ← Use of memory after it is freed
 9      SomethingUnknown();
10  }
```

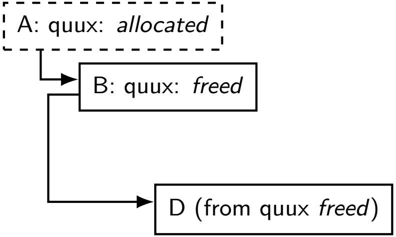
checking use-after-free (2)

```
int *someFunction(int foo, int bar) {
    int *quux = malloc(sizeof(int));
    // A
    if (Complex(foo)) {
        free(quux);
        // B
    }
    ... /* omitted code that doesn't use quux */
    if (Complex(bar)) {
        // C
        *quux = bar;
    }
    ... /* omitted code that doesn't use quux */
    // D
}
```

A: quux: *allocated*

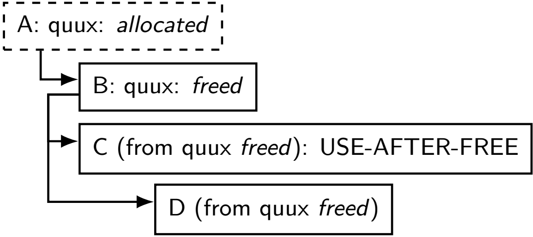
checking use-after-free (2)

```
int *someFunction(int foo, int bar) {
    int *quux = malloc(sizeof(int));
    // A
    if (Complex(foo)) {
        free(quux);
        // B
    }
    ... /* omitted code that doesn't use quux */
    if (Complex(bar)) {
        // C
        *quux = bar;
    }
    ... /* omitted code that doesn't use quux */
    // D
}
```



checking use-after-free (2)

```
int *someFunction(int foo, int bar) {
    int *quux = malloc(sizeof(int));
    // A
    if (Complex(foo)) {
        free(quux);
        // B
    }
    ... /* omitted code that doesn't use quux */
    if (Complex(bar)) {
        // C
        *quux = bar;
    }
    ... /* omitted code that doesn't use quux */
    // D
}
```


A: quux: *allocated*

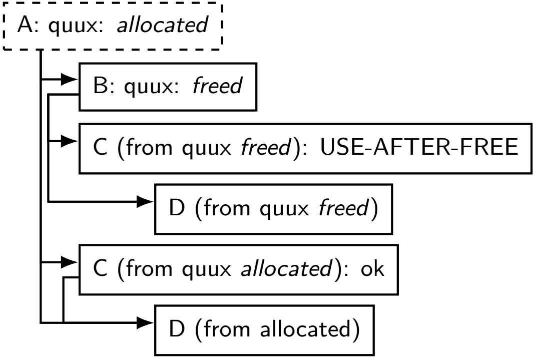
B: quux: *freed*

C (from quux *freed*): USE-AFTER-FREE

D (from quux *freed*)

checking use-after-free (2)

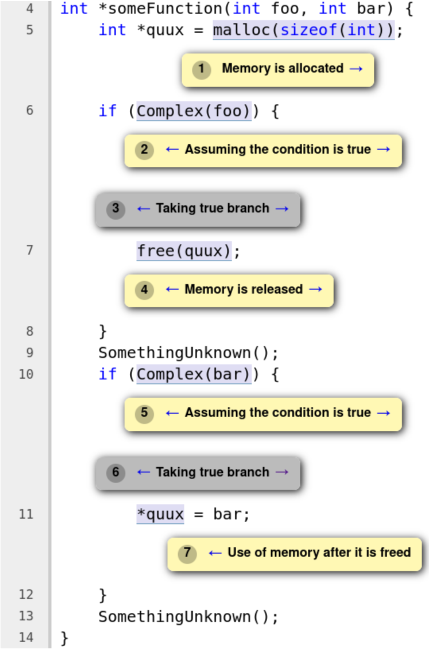
```
int *someFunction(int foo, int bar) {
    int *quux = malloc(sizeof(int));
    // A
    if (Complex(foo)) {
        free(quux);
        // B
    }
    ... /* omitted code that doesn't use quux */
    if (Complex(bar)) {
        // C
        *quux = bar;
    }
    ... /* omitted code that doesn't use quux */
    // D
}
```



one idea: guess that Complex(foo) can be probably be true

option 1: say “something wrong maybe”?
option 2: try to figure out if Complex(foo) is true?)

result from clang's scan-build



```
int *someFunction(int foo, int bar) {
    int *quux = malloc(sizeof(int));
                    1  Memory is allocated →
    if (Complex(foo)) {
              2  ← Assuming the condition is true →
           3  ← Taking true branch →
        free(quux);
              4  ← Memory is released →
    }
    SomethingUnknown();
    if (Complex(bar)) {
              5  ← Assuming the condition is true →
           6  ← Taking true branch →
        *quux = bar;
                   7  ← Use of memory after it is freed
    }
    SomethingUnknown();
}
```

exercise: holes in the model?

```
void example(int a) {
    int *p;
    int *q;
    q = malloc(...);
    p = malloc(...);
    // (A)
    if (a > 0) {
        // (A1)
        p = q;
    }
    // (B)
    free(p);
    // (C)
    ...
}
```

exercise: what should state of pointer q be at C?

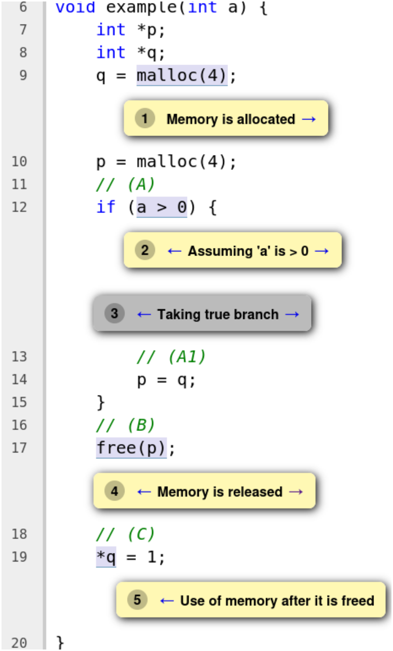
A. allocated B. freed

C. allocated if+only if reached via path with A1

D. freed if+only if reached via path with A1

E. something else?

clang-analyzer output



```
void example(int a) {
    int *p;
    int *q;
    q = malloc(4);

          1  Memory is allocated →

    p = malloc(4);
    // (A)
    if (a > 0) {

          2  ← Assuming 'a' is > 0 →

       3  ← Taking true branch →

        // (A1)
        p = q;
    }
    // (B)
    free(p);

       4  ← Memory is released →

    // (C)
    *q = 1;

         5  ← Use of memory after it is freed

}
```

analysis building blocks

needed to track that p and q could point to same thing

common prerequisite for all sorts of program analysis

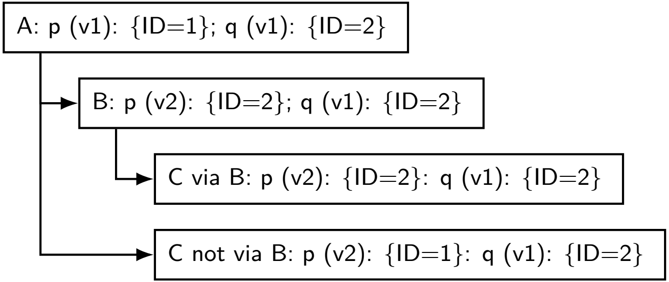
overly simple algorithm for points-to analysis

for each pointer/reference track which objects it can refer to

if multiple paths: take union of all possible

simple points-to analysis

```
void example(int a) {
    int *p;
    int *q;
    q = malloc(...); // ID=1
    p = malloc(...); // ID=2
    // (A)
    if (a > 0) {
        p = q;
        // (B)
    }
    // (C)
    ...
}
```



A: p (v1): {ID=1}; q (v1): {ID=2}

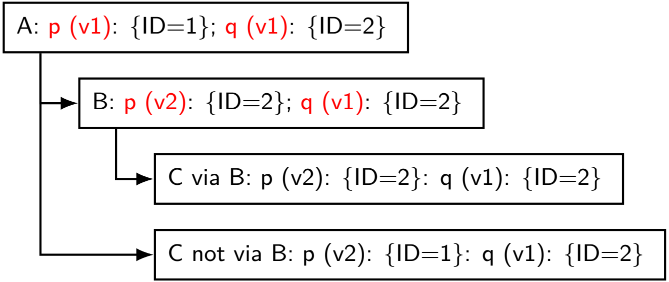
B: p (v2): {ID=2}; q (v1): {ID=2}

C via B: p (v2): {ID=2}: q (v1): {ID=2}

C not via B: p (v2): {ID=1}: q (v1): {ID=2}

simple points-to analysis

```
void example(int a) {
    int *p;
    int *q;
    q = malloc(...); // ID=1
    p = malloc(...); // ID=2
    // (A)
    if (a > 0) {
        p = q;
        // (B)
    }
    // (C)
    ...
}
```



A: p (v1): {ID=1}; q (v1): {ID=2}

B: p (v2): {ID=2}; q (v1): {ID=2}

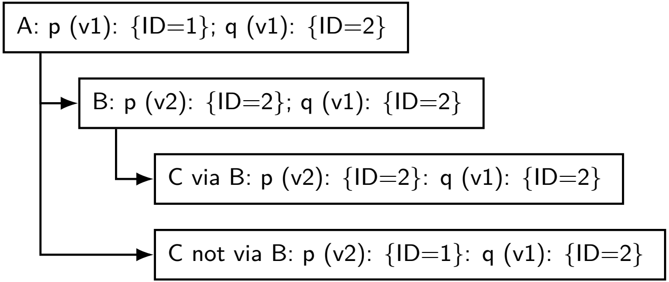
C via B: p (v2): {ID=2}: q (v1): {ID=2}

C not via B: p (v2): {ID=1}: q (v1): {ID=2}

likely first step: mark different versions of p, q
and track them as separate variables
this way: can avoid storing set of values for q for every block of code
(instead just point to q (v1) set)

simple points-to analysis

```
void example(int a) {
    int *p;
    int *q;
    q = malloc(...); // ID=1
    p = malloc(...); // ID=2
    // (A)
    if (a > 0) {
        p = q;
        // (B)
    }
    // (C)
    ...
}
```



A: p (v1): {ID=1}; q (v1): {ID=2}

B: p (v2): {ID=2}; q (v1): {ID=2}

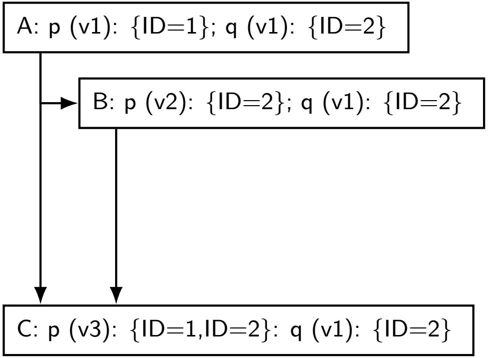
C via B: p (v2): {ID=2}: q (v1): {ID=2}

C not via B: p (v2): {ID=1}: q (v1): {ID=2}

one idea: keep track of each path separately
(but limit to how much one can do this)

simple points-to analysis

```
void example(int a) {
    int *p;
    int *q;
    q = malloc(...); // ID=1
    p = malloc(...); // ID=2
    // (A)
    if (a > 0) {
        p = q;
        // (B)
    }
    // (C)
    ...
}
```



A: p (v1): {ID=1}; q (v1): {ID=2}

B: p (v2): {ID=2}; q (v1): {ID=2}

C: p (v3): {ID=1,ID=2}: q (v1): {ID=2}

alternate idea: avoid path explosion by merging possible sets

complicating points-to analysis

would like to analyze program function-at-a-time, but…

  functions can change values shared by other functions

what about computed array indices?

what about pointers to pointers?

…

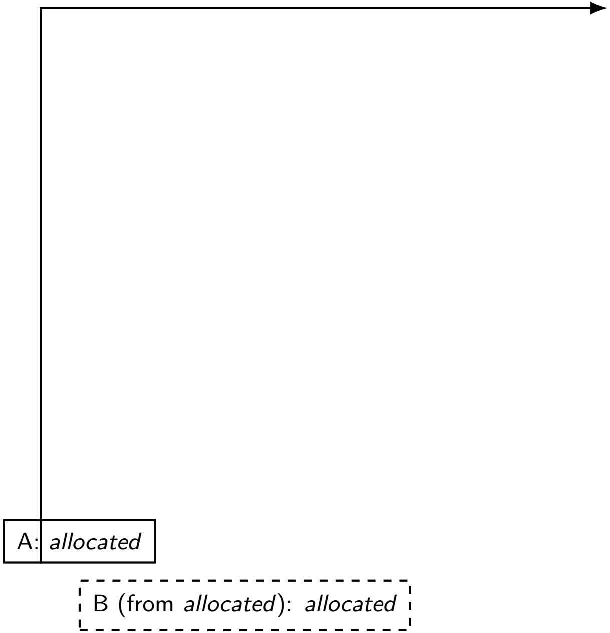
high false-positive solution:

  when incomplete info: assume value points to anything of right type

high false-negative solution:

  when incomplete info: assume value points to nothing

checking use-after-free (3)



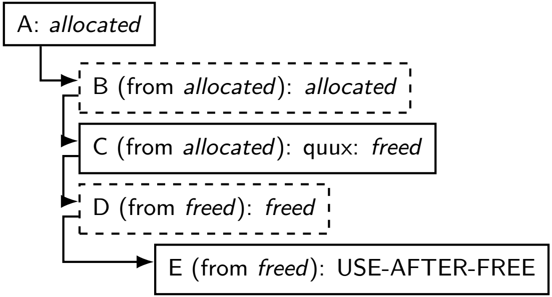
A: *allocated*

B (from *allocated*): *allocated*

```
void someFunction() {
    int *quux = malloc(sizeof(int));
    ...
    // A
    do {
```

checking use-after-free (3)

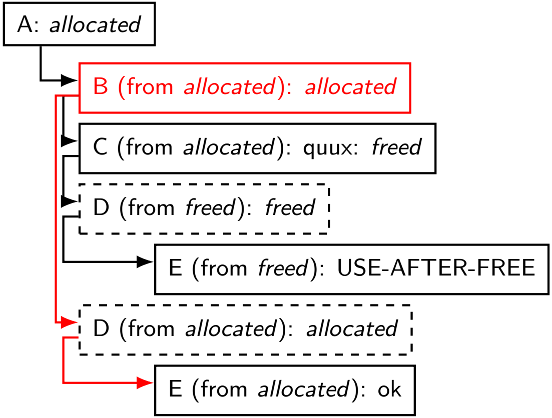
```
void someFunction() {
    int *quux = malloc(sizeof(int));
    ...
    // A
    do {
        // B
        ...
        if (anotherFunction()) {
            free(quux);
            // C
        }
        ...
        // D
    } while (complexFunction());
    ...
    // E
    *quux++;
    ...
}
```



A: *allocated*
B (from *allocated*): *allocated*
C (from *allocated*): quux: *freed*
D (from *freed*): *freed*
E (from *freed*): USE-AFTER-FREE

checking use-after-free (3)

```
void someFunction() {
    int *quux = malloc(sizeof(int));
    ...
    // A
    do {
        // B
        ...
        if (anotherFunction()) {
            free(quux);
            // C
        }
        ...
        // D
    } while (complexFunction());
    ...
    // E
    *quux++;
    ...
}
```



A: *allocated*

B (from *allocated*): *allocated*

C (from *allocated*): quux: *freed*

D (from *freed*): *freed*

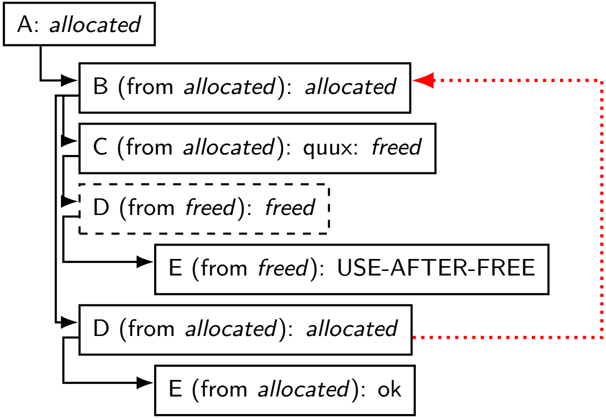
E (from *freed*): USE-AFTER-FREE

D (from *allocated*): *allocated*

E (from *allocated*): ok

checking use-after-free (3)

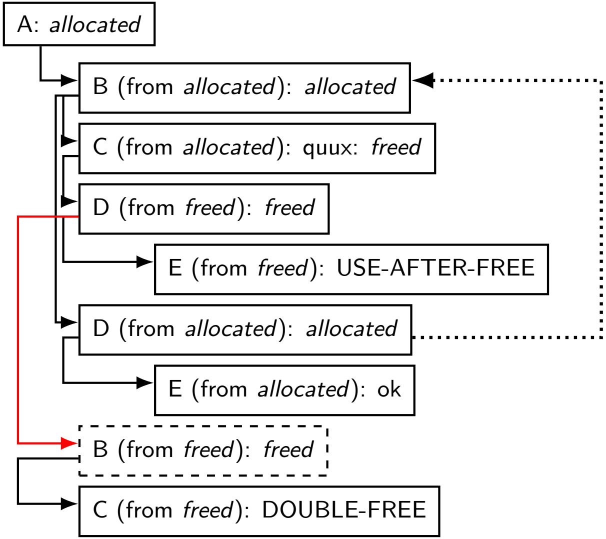
```
void someFunction() {
    int *quux = malloc(sizeof(int));
    ...
    // A
    do {
        // B
        ...
        if (anotherFunction()) {
            free(quux);
            // C
        }
        ...
        // D
    } while (complexFunction());
    ...
    // E
    *quux++;
    ...
}
```



A: *allocated*
B (from *allocated*): *allocated*
C (from *allocated*): quux: *freed*
D (from *freed*): *freed*
E (from *freed*): USE-AFTER-FREE
D (from *allocated*): *allocated*
E (from *allocated*): ok

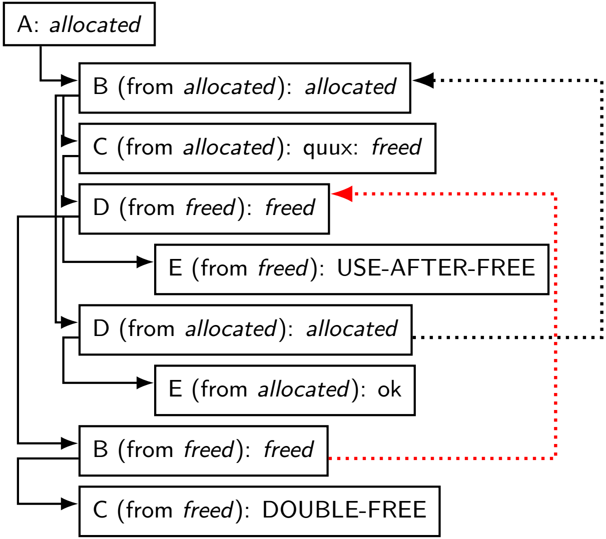
checking use-after-free (3)

```
void someFunction() {
    int *quux = malloc(sizeof(int));
    ...
    // A
    do {
        // B
        ...
        if (anotherFunction()) {
            free(quux);
            // C
        }
        ...
        // D
    } while (complexFunction());
    ...
    // E
    *quux++;
    ...
}
```

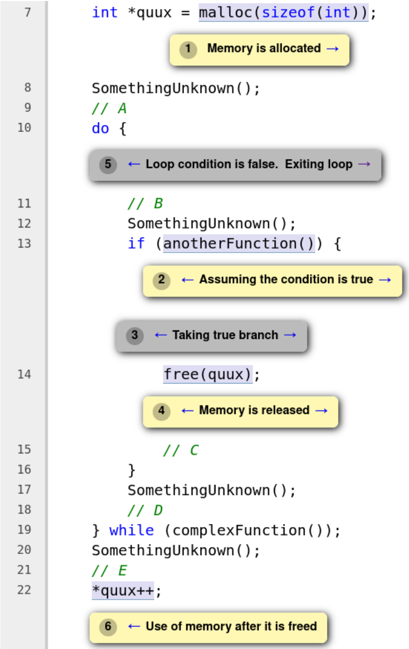

A: *allocated*
B (from *allocated*): *allocated*
C (from *allocated*): quux: *freed*
D (from *freed*): *freed*
E (from *freed*): USE-AFTER-FREE
D (from *allocated*): *allocated*
E (from *allocated*): ok
B (from *freed*): *freed*
C (from *freed*): DOUBLE-FREE

checking use-after-free (3)

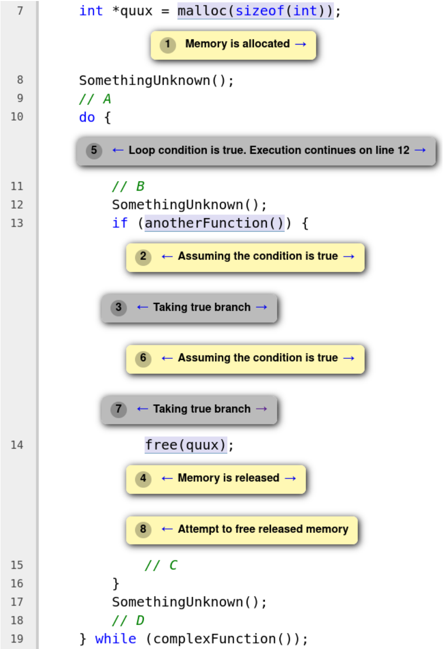
```
void someFunction() {
    int *quux = malloc(sizeof(int));
    ...
    // A
    do {
        // B
        ...
        if (anotherFunction()) {
            free(quux);
            // C
        }
        ...
        // D
    } while (complexFunction());
    ...
    // E
    *quux++;
    ...
}
```


A: *allocated*
B (from *allocated*): *allocated*
C (from *allocated*): quux: *freed*
D (from *freed*): *freed*
E (from *freed*): USE-AFTER-FREE
D (from *allocated*): *allocated*
E (from *allocated*): ok
B (from *freed*): *freed*
C (from *freed*): DOUBLE-FREE

result from clang's scan-build



```
 7    int *quux = malloc(sizeof(int));
          1  Memory is allocated →
 8    SomethingUnknown();
 9    // A
10    do {
      5  ← Loop condition is false.  Exiting loop →
11        // B
12        SomethingUnknown();
13        if (anotherFunction()) {
            2  ← Assuming the condition is true →
          3  ← Taking true branch →
14            free(quux);
            4  ← Memory is released →
15            // C
16        }
17        SomethingUnknown();
18        // D
19    } while (complexFunction());
20    SomethingUnknown();
21    // E
22    *quux++;
      6  ← Use of memory after it is freed
```



```
 7    int *quux = malloc(sizeof(int));
          1  Memory is allocated →
 8    SomethingUnknown();
 9    // A
10    do {
      5  ← Loop condition is true. Execution continues on line 12 →
11        // B
12        SomethingUnknown();
13        if (anotherFunction()) {
            2  ← Assuming the condition is true →
          3  ← Taking true branch →
            6  ← Assuming the condition is true →
          7  ← Taking true branch →
14            free(quux);
            4  ← Memory is released →
            8  ← Attempt to free released memory
15            // C
16        }
17        SomethingUnknown();
18        // D
19    } while (complexFunction());
```

checking for array bounds

can *try* to apply same technique to array bounds

but much more complicated/more likely to have false positives/negatives

for each array or pointer track:

- minimum number of elements before/after what it points to

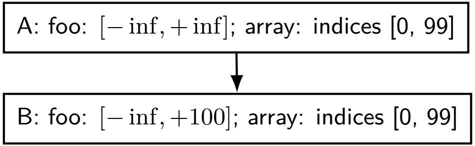
for each integer track:

- minimum bound
- maximum bound

similar analysis looking at paths?

checking array bounds (1)

```
int array[100];
void someFunction(int foo) {
    // A
    if (foo <= 100) {
        return;
    }
    // B
    array[foo] += 1;
}
```

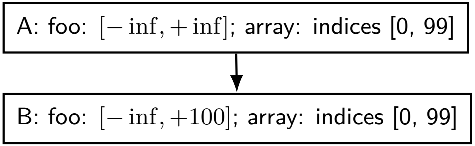


A: foo: $[-\inf, +\inf]$; array: indices [0, 99]

B: foo: $[-\inf, +100]$; array: indices [0, 99]

checking array bounds (1)

```
int array[100];
void someFunction(int foo) {
    // A
    if (foo <= 100) {
        return;
    }
    // B
    array[foo] += 1;
}
```



A: foo: $[-\mathrm{inf}, +\mathrm{inf}]$; array: indices [0, 99]

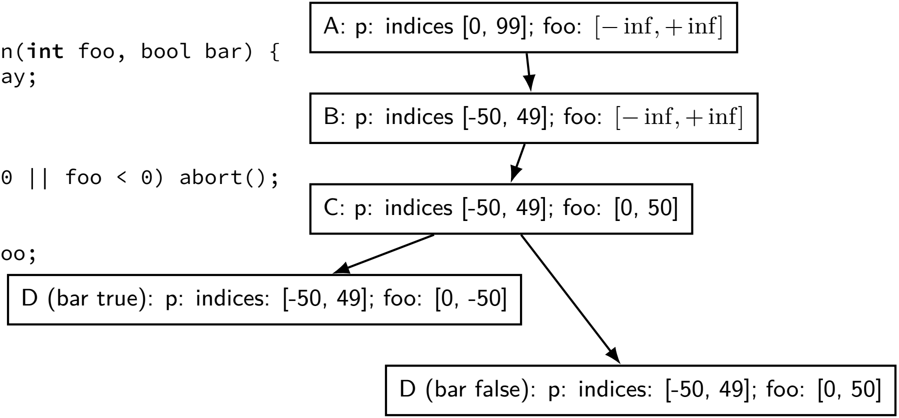
B: foo: $[-\mathrm{inf}, +100]$; array: indices [0, 99]

give warning about `foo == 100`? probably bug!

give warning about `foo < 0`? maybe??

checking array bounds (2)

```
int array[100];
void someFunction(int foo, bool bar) {
    int *p = array;
    // A
    p += 50;
    // B
    if (foo >= 50 || foo < 0) abort();
    // C
    if (bar) {
        foo = –foo;
    }
    // D
    p[foo] = 1;
}
```



A: p: indices [0, 99]; foo: $[-\mathrm{inf}, +\mathrm{inf}]$

B: p: indices [-50, 49]; foo: $[-\mathrm{inf}, +\mathrm{inf}]$

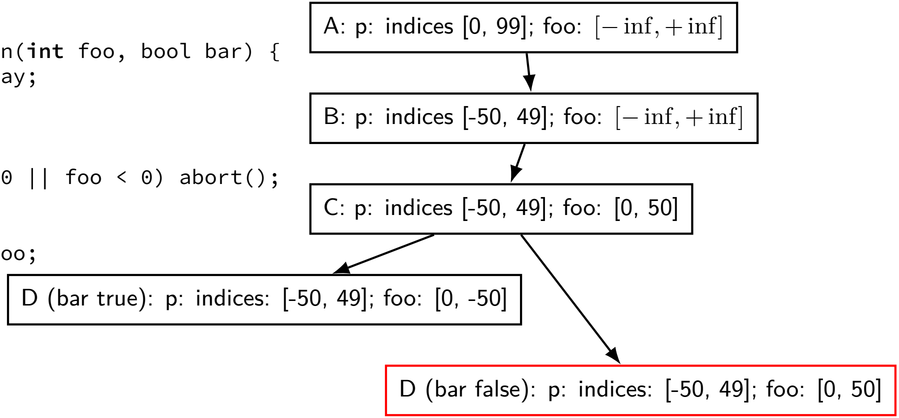
C: p: indices [-50, 49]; foo: [0, 50]

D (bar true): p: indices: [-50, 49]; foo: [0, -50]

D (bar false): p: indices: [-50, 49]; foo: [0, 50]

checking array bounds (2)

```
int array[100];
void someFunction(int foo, bool bar) {
    int *p = array;
    // A
    p += 50;
    // B
    if (foo >= 50 || foo < 0) abort();
    // C
    if (bar) {
        foo = -foo;
    }
    // D
    p[foo] = 1;
}
```



A: p: indices [0, 99]; foo: $[-\mathrm{inf}, +\mathrm{inf}]$

B: p: indices [-50, 49]; foo: $[-\mathrm{inf}, +\mathrm{inf}]$

C: p: indices [-50, 49]; foo: [0, 50]

D (bar true): p: indices: [-50, 49]; foo: [0, -50]

D (bar false): p: indices: [-50, 49]; foo: [0, 50]

warn about possible out-of-bounds?

suspect patterns

some other clang-analyzer checks

observation: mostly simple patterns

```
long *p = malloc(16 * sizeof(short));
    // short != long

int *foo() {
    int x;
    int *p = &x;
    ...
    return p;  // return pointer to stack
}
```

static analysis

need to avoid exploring way too many paths

- clang-analyzer: only a procedure at a time
- other analyzers: some way of pruning paths

need to avoid false positives

- probably can't always assume every if can be true/false
- one idea: apply symbolic-execution like techniques to prune
- clang-analyzer: limited by being procedure-at-a-time

static analysis practicality

good at finding some kinds of bugs

- array out-of-bounds probably not one — complicated tracking needed

excellent for "bug patterns" like:

```
struct Foo* foo;
...
foo = malloc(sizeof(struct Bar));
```

false positive rates are often 20+% or more

some tools assume lots of annotations

not limited to C-like languages

static analysis tools

Coverity, Fortify — commerical static analysis tools

Splint — unmaintained?

- written by David Evans and his research group in the late 90s/early 00s

FindBugs (Java)

clang-analyzer — part of Clang compiler

Microsoft's Static Driver Verifier — required for Windows drivers:

- mostly checks correct usage of Windows APIs

information flow

so far: static analysis concerned with control flow

often, we're really worried about how *data* moves

many applications:

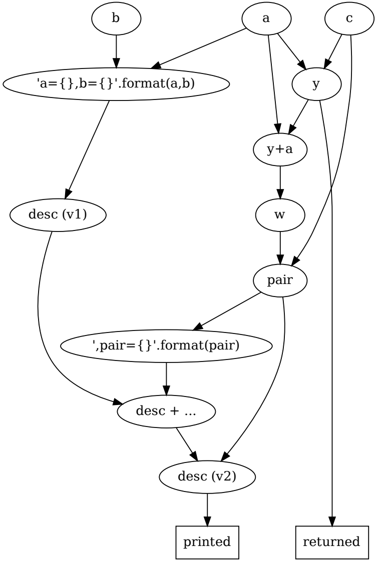
- does an array index depend on user input?
- does an SQL query depend on user input?
- does data sent over network depend on phone number?

…

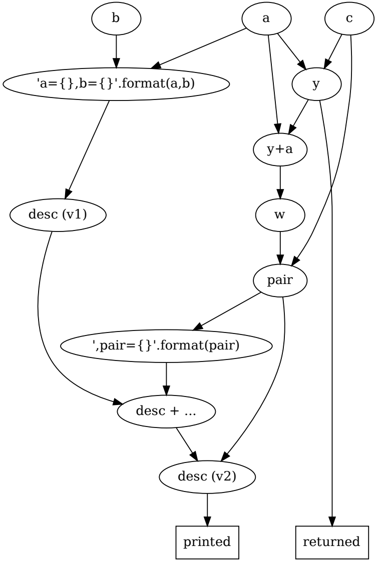
can do this *statically* (potential dependencies)
or *dynamically* (actual dependencies as program runs)

data flow graph (1a)

```
def f(a, b, c):
    desc = 'a={},b={}'.format(a, b)
    if b > 10:
        y = a
    else:
        y = c
    w = y + a
    pair = (w, c)
    desc = desc + \
        ',pair={}'.format(pair)
    print(desc)
    return y
```



data flow graph (1b)

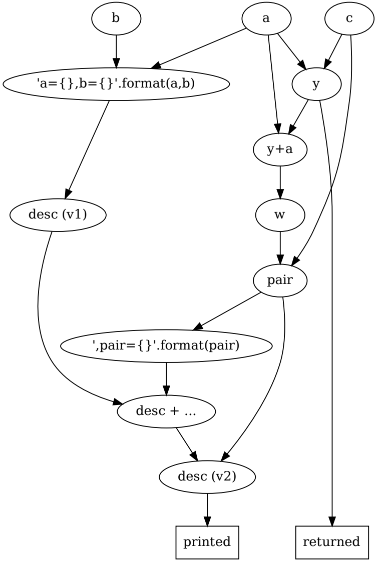


data flow graph (1b)

ex: does returned value depend on a, b, c?

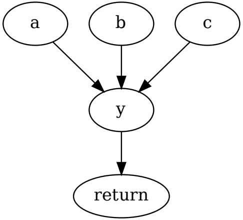
ex: does value of pair depend on a, b, c?

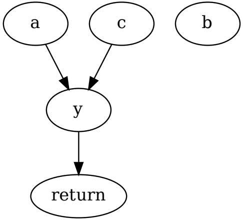
ex: does printed value depend on a, b, c?



information flow and control flow

```
def f(a, b, c):
    if b > 10:
        y = a
    else:
        y = c
    return y
```





Q: which is better ...

if we're trying to see if user input makes it to SQL query?

if we're trying to determine if private info goes out over network?

data flow challenges (1)

```python
# Python example
def stash(a):
    global y
    y = a
x = [0,1,2,3]
stash(x)
x[2] = input()
print(y[2])
```

```c
// C example
int *y;
void stash(int *a) {
    y = a;
}
int main() {
    int x[3];
    stash(x);
    y[2] = GetInput();
    printf("%d\n",x[2]);
}
```

need to realize that x[2] and y[2] are the same!

even if assignment to/usage of y is more cleverly hidden

...or make analysis a lot less prceise

often: make some compromise about how often this case is handled

data flow challenges (2)

```
def retrieve(flag):
    global the_default
    if flag:
        value = input()
    else:
        value = the_default
    value = process(value)
    if not flag:
        print("base on default: ",value)
    return value
retrieve(True)
retrieve(False)
```

input can't make it to print here

...but need *path-sensitive* analysis to tell

often: make some compromise about how often this case is handled

data flow challenges (3)

```
x = int(input())
if x == 0:
    print(0)
elif x == 1:
    print(1)
elif ...
```

does input make it to output?

should we try to detect this?

- probably depends on intended use of analysis

sources and sinks

needed choose *sources* (so far: function arguments)
and *sinks* (so far: print, return)

choice depends on application

SQL injection:

- sources: input from network
- sinks: SQL query functions

private info leak:

- sources: private data: phone number, message history, email, …
- sinks: network output

taint tracking idea

so far: looking at how information makes it from source to sink statically

not actually running the program

can do this as programs are running, trigger error

dynamic taint tracking

taint tracking implementations

for the programmer:

- supported as optional langauge feature — Perl, Ruby
- doesn't seem to have gotten wide adoption?

for the malware analyst/user

- as part of a custom x86 VM (whole system, on machine code)
- as part of a custom Android system
- …

taint tracking in Perl (1)

```perl
#! perl -T
# -T: enable taint tracking
use warnings; use strict;
$ENV{PATH} = '/usr/bin:/bin';

print "Enter name: ";
my $name = readline(STDIN);
my $dir = $name . "-dir";

system("mkdir $dir");
```

“Insecure dependency in system while running with -T switch at perltaint.pl line 10, <STDIN> line 1.”

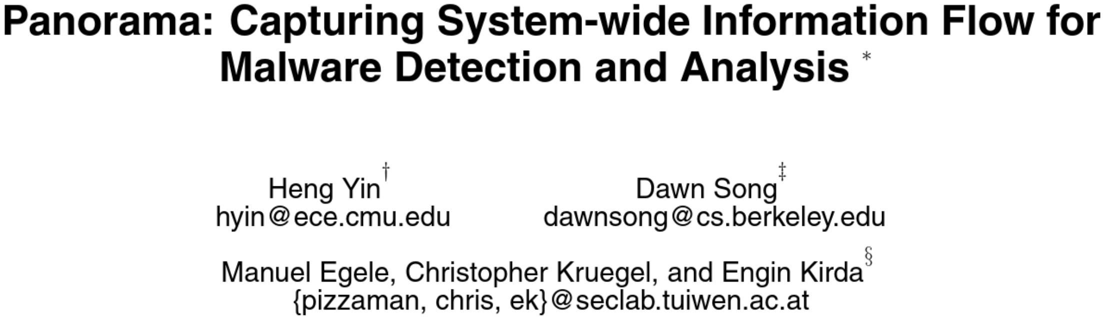
taint tracking in Perl (2)

```
#! perl -T
# -T: enable taint tracking
use warnings; use strict;
$ENV{PATH} = '/usr/bin:/bin';

print "Enter name: ";
my $name = readline(STDIN);
# keep $name only if its all alphanumeric
# this marks $name as untainted
($name) = $name =~ /^([a-zA-Z0-9]+)$/;
my $dir = $name . "-dir";

system("mkdir $name");
```

taint tracking assembly?



Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis [*]

Heng Yin[†]
hyin@ece.cmu.edu

Dawn Song[‡]
dawnsong@cs.berkeley.edu

Manuel Egele, Christopher Kruegel, and Engin Kirda[§]
{pizzaman, chris, ek}@seclab.tuiwen.ac.at

high-level overview

lookup table for each register and byte of memory:

where did this value come from?

`add %r9, (%r8)`:

```
memory-taint-table[register-values[R8]] =
                register-taint-table[R9]
```

custom VM: all applications and the OS run with taint tracking

Panaroma special cases

`xor %eax, %eax`: special case: remove taint from %eax

Windows keyboard input did something like:

```
switch (keycode) {
case KEYCODE_A: return 'a';
case KEYCODE_B: return 'b';
...
}
```

taint tracking for malware analysis

uses proposed by Panaroma authors:

keypresses → network packets

network packets → malware outputs

browser history → network packets

defeating ASM-based checking

if a malware author wanted to defeat this taint checking, what ideas seem promising for confusing the analysis?

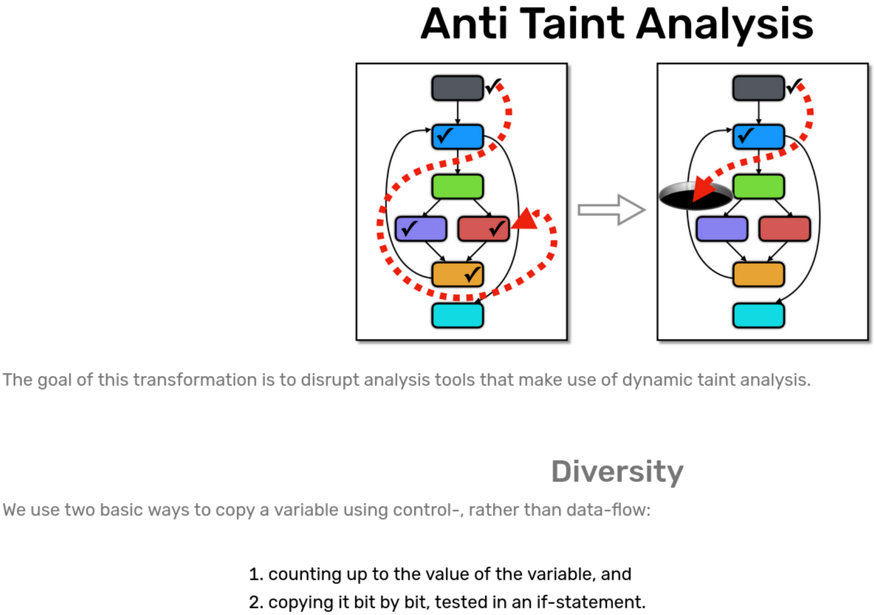
A. timing arithmetic operations to see if the machine is unusually slow

B. computing the hash of the malware's machine code and comparing it to a known value

C. changing `x = y` to
`switch (x) { case 1: y = 1; break; case 2: ...}`

D. changing `x = y` to `x = z + y; x = x – z;`

Tigress's transformation



Anti Taint Analysis

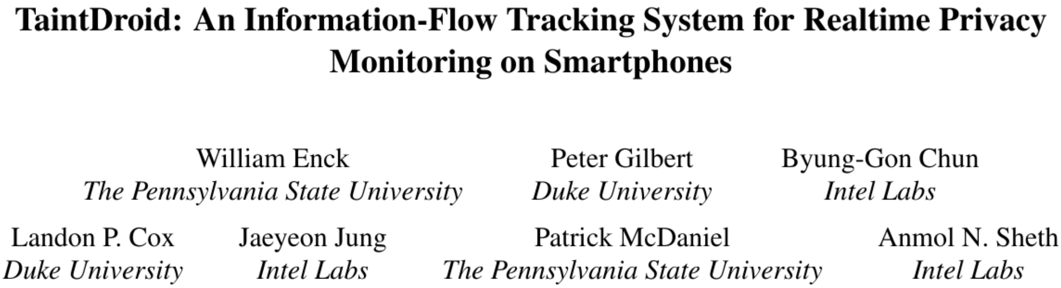
The goal of this transformation is to disrupt analysis tools that make use of dynamic taint analysis.

Diversity

We use two basic ways to copy a variable using control-, rather than data-flow:

1. counting up to the value of the variable, and
2. copying it bit by bit, tested in an if-statement.

example: TaintDroid



TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones

William Enck
The Pennsylvania State University

Peter Gilbert
Duke University

Byung-Gon Chun
Intel Labs

Landon P. Cox
Duke University

Jaeyeon Jung
Intel Labs

Patrick McDaniel
The Pennsylvania State University

Anmol N. Sheth
Intel Labs

TaintDroid instrumentation

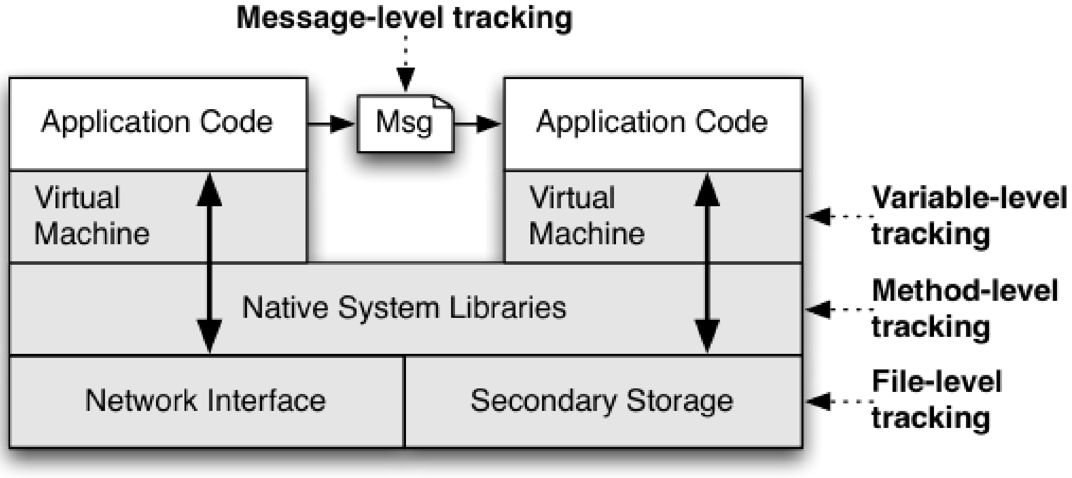


Figure 1: Multi-level approach for performance efficient taint tracking within a common smartphone architecture.

TaintDroid resutls

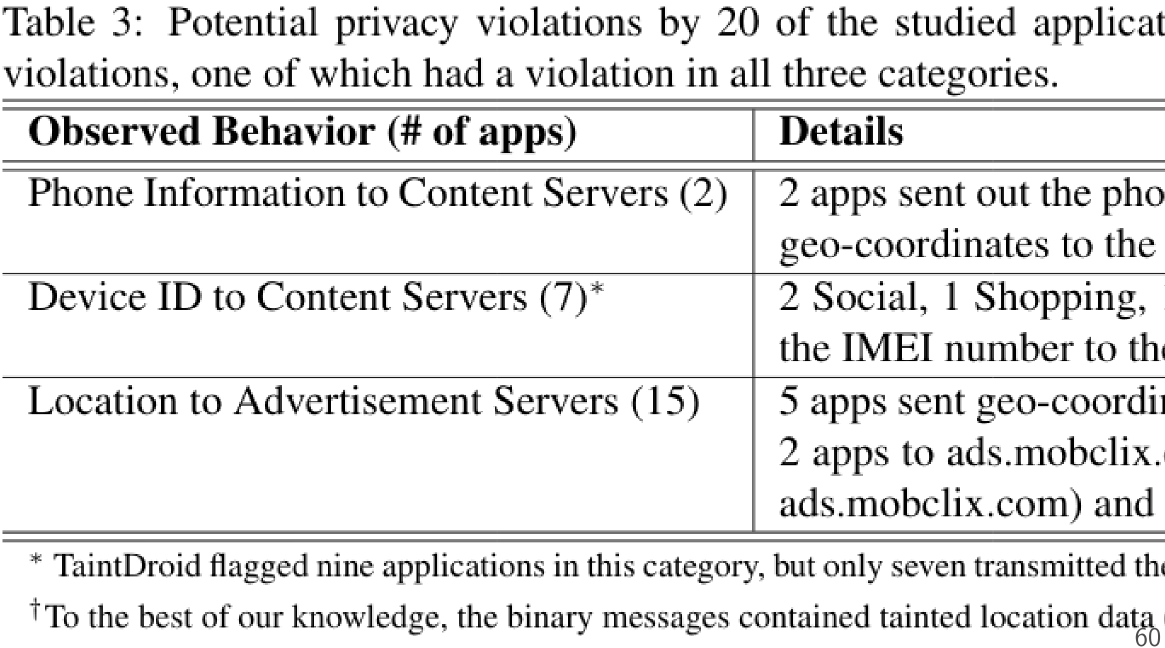


Table 3: Potential privacy violations by 20 of the studied applicat
violations, one of which had a violation in all three categories.

Observed Behavior (# of apps)	Details
Phone Information to Content Servers (2)	2 apps sent out the pho geo-coordinates to the
Device ID to Content Servers (7)*	2 Social, 1 Shopping, the IMEI number to th
Location to Advertisement Servers (15)	5 apps sent geo-coordi 2 apps to ads.mobclix. ads.mobclix.com) and

* TaintDroid flagged nine applications in this category, but only seven transmitted th

† To the best of our knowledge, the binary messages contained tainted location data

Rust philosophy

default rules that only allow 'safe' things

- no dangling pointers
- no out-of-bounds accesses

escape hatch to use "raw" pointers or unchecked libraries

escape hatch can be used to write useful libraries

- e.g. Vector/ArrayList equivalent
- expose interface that is safe

simple Rust syntax (1)

```
fn main() {
    println!("Hello, World!\n");
}
```

simple Rust syntax (2)

```
fn timesTwo(number: i32) -> i32 {
    return number * 2;
}
```

simple Rust syntax (3)

```
struct Student {
    name: String,
    id: i32,
}

fn get_example_student() -> Student {
    return Student {
        name: String::from("Example Fakelastname"),
        id: 42,
    };
}
```

simple Rust syntax (4)

```rust
fn factorial(number: i32) -> i32 {
    let mut result = 1;
    let mut index = 1;
    while index <= number {
        result *= index;
        index = index + 1;
    }
    return result;
}
```

simple Rust syntax (4)

```
fn factorial(number: i32) -> i32 {
    let mut result = 1;
    let mut index = 1;
    while inde
        result
        index
    }
    return result;
}
```

“input” is a mutable variable

type automatically inferred as i32 (32-bit int)

Rust references

```rust
fn main() {
    let mut x: u32 = 42;

    {
        let y: &mut u32 = &mut x;
        *y = 100;
    }

    let z: &u32 = &x;

    println!("x = {}; z = {}", x, x);
}
```

Rust example

```rust
use std::io;

fn main() {
    println!("Enter a number: ");

    let mut input = String::new();
    // could have also written:
    //   let mut input: String = String::new();

    io::stdin().read_line(&mut input);

    // parse number or fail with an error message
    let number: u32 = input.trim().parse()
        .expect("That was not a number!");
    println!("Twice that number is: {}", number * 2);
}
```

Rust example

```
use std::io;

fn main() {
    println!("Ent

    let mut inp
    // could have
    //   let mut input: String = String::new();

    io::stdin().read_line(&mut input);

    // parse number or fail with an error message
    let number: u32 = input.trim().parse()
        .expect("That was not a number!");
    println!("Twice that number is: {}", number * 2);
}
```

“input” is a mutable variable

type is automatically inferred as String

Rust example

```
use std::io;

fn main() {
    println!("Enter a number: ");

    let mut input = String::new();
    // could have also written:
    //   let mut inp

    io::stdin().read_line(&mut input);

    // parse number or fail with an error message
    let number: u32 = input.trim().parse()
        .expect("That was not a number!");
    println!("Twice that number is: {}", number * 2);
}
```

pass mutable reference to input

Rust example

```
use std::io;

fn main() {
    println!("Enter a number: ");

    let mut input = String::new();
    // could have also written:
    //   let mut input: String = String::new();

    io::stdi

    // parse number or fail with an error message
    let number: u32 = input.trim().parse()
        .expect("That was not a number!");
    println!("Twice that number is: {}", number * 2);
}
```

number is an immutable unsigned 32-bit integer

rules to stop dangling pointers (1)

objects have an single owner

owner is the only one allowed to modify an object

owner can give away ownership

simplest version: only owner can access object

never have multiple references to object — always move/copy

Rust objects and ownership (1)

```
fn mysum(vector: Vec<u32>) -> u32 {
    let mut total: u32 = 0
    for value in &vector {
        total += value
    }
    return total
}

fn foo() {
    let vector: Vec<u32> = vec![1, 2, 3];
    let sum = mysum(vector);
    // **moves** vector into mysum()
         // philosophy: no implicit expensive copies

    println!("Sum is {}", sum);
    // ERROR
    println!("vector[0] is {}" , vector[0]);
}
```

Rust objects and ownership (1)

```
fn mysum(vector: Vec<u32>) -> u32 {
    let mut total: u32 = 0
    for value in &vector {
        total += value
    }
    return total
}

fn foo
    let
    let
    //


    pr
    // ERROR
    println!("vector[0] is {}" , vector[0]);
}
```

```
   Compiling lecture-demo v0.1.0 (file:///home/cr4bd/spring2017/cs4630/...
error[E0382]: use of moved value: vector
  --> src/main.rs:16:34
   |
13 |     let sum = mysum(vector);
   |                     ------ value moved here
...
16 |     println!("vector[0] is {}" , vector[0]);
   |                                  ^^^^^^ value used here after move
```

Rust objects and ownership (2)

```
fn mysum(vector: Vec<u32>) -> u32 {
    let mut total: u32 = 0
    for value in &vector {
        total += value
    }
    return total
}

fn foo() {
    let vector: Vec<u32> = vec![1, 2, 3];
    let sum = mysum(vector.clone());
    // give away a copy of vector instead
        // mysum will dispose, since it owns it

    println!("Sum is {}", sum);
    println!("vector[0] is {}" , newVector[0]);
}
```

Rust objects and ownership (2)

```
fn mysum(vector: Vec<u32>) -> u32 {
    let mut total: u32 = 0
    for value in &vector {
        total += value
    }
    return total
}

fn foo() {
    let vector: Vec<u32> = ve
    let sum = mysum(vector.cl
    // give away a copy of vector instead
        // mysum will dispose, since it owns it

    println!("Sum is {}", sum);
    println!("vector[0] is {}" , newVector[0]);
}
```

mysum borrows a copy

moving?

moving a Vec — really copying a pointer to an array and its size

cloning a Vec — making a copy of the array itself, too

Rust defaults to moving non-trivial types

some trivial types (u32, etc.) are copied by default

Rust objects and ownership (3)

```
fn mysum(vector: Vec<u32>) -> (u32, Vec<u32>) {
    let mut total: u32 = 0
    for value in &vector {
        total += value
    }
    return (total, vector)
}

fn foo() {
    let vector: Vec<u32> = vec![1, 2, 3];
    let (sum, newVector) = mysum(vector);
    // give away vector, get it back

    println!("Sum is {}", sum);
    println!("vector[0] is {}" , newVector[0]);
}
```

Rust objects and ownership (3)

```
fn mysum(vector: Vec<u32>) -> (u32, Vec<u32>) {
    let mut total: u32 = 0
    for value in &vector {
        total += value
    }
    return (total, vector)
}

fn foo() {
    let vector: Vec
    let (sum, newVe
    // give away vec

    println!("Sum is {}", sum);
    println!("vector[0] is {}" , newVector[0]);
}
```

mysum “borrows” vector, then gives it back
uses pointers

ownership rules

exactly one owner at a time

giving away ownership means you can't use object

either give object new owner or deallocate

ownership rules

exactly one owner at a time

giving away ownership means you can't use object

- common idiom — temporarily give away object

either give object new owner or deallocate

rules to stop dangling pointers (2)

objects have an single **owner**

owner can give away ownership permanently

- object is “moved”

owner can let someone borrow object **temporarily**

- must know when object is given back

only **modify** object when exactly one user

- owner or exclusive borrower

borrowing

```rust
fn mysum(vector: &Vec<u32>) -> u32 {
    let mut total: u32 = 0
    for value in vector {
        total += value
    }
    return total
}

fn foo() {
    let vector: Vec<u32> = vec![1, 2, 3];
    let sum = mysum(&vector);
    // automates (vector, sum) = mysum(vector) idea

    println!("Sum is {}", sum);
    println!("vector[0] is {}" , vector[0]);
}
```

dangling pointers?

```
int *dangling_pointer() {
    int array[3] = {1,2,3};
    return &array[0]; // not an error
}
```

```
fn dangling_pointer() -> &mut i32 {
    let array = vec![1,2,3];
    return &mut array[0]; // ERROR
}
```

dangling pointers?

```c
int *dangling_pointer() {
    int array[3] = {1,2,3};
    return &array[0]; // not an error
```

```
error[E0106]: missing lifetime specifier
  --> src/main.rs:19:25
   |
19 | fn dangling_pointer() -> &mut i32 {
   |                          ^ expected lifetime parameter
   |
   = help: this function's return type contains a borrowed value,
           but there is no value for it to be borrowed from
```

rules to stop dangling pointers (2)

objects have an single **owner**

owner can give away ownership permanently

  object is “moved”

owner can let someone borrow object **temporarily**

  must know when object is given back

only **modify** object when exactly one user

  owner or exclusive borrower

lifetimes

every reference in Rust has a lifetime

intuitively: how long reference is usable

Rust compiler infers and checks lifetimes

lifetime rules

object is borrowed for duration of reference lifetime

- can't modify object during lifetime
- can't let object go out of scope during lifetime

lifetime of function args must include whole function call

references returned from function must have lifetimes

- based on arguments or static (valid for entire program)

references stored in structs must have lifetime longer than struct

lifetime inference

```
fn get_first(values: &Vec<String>) -> &String {
    return &values[0];
}
```

compiler infers lifetime of return value is same as input

lifetime hard cases

```
// ERROR:
fn get_first_matching(prefix: &str, values: &Vec<String>)
                            -> &String {
    for item in values {
        if item.starts_with(prefix) {
            return item
        }
    }
    panic!()
}
```

this is a compile-error, because of the return value

compiler need to be told lifetime of return value

lifetime annotations

```
fn get_first_matching<'a, 'b>(prefix: &'a str, values: &'b Vec<String>)
                                            -> &'b String {
    for item in values {
        if item.starts_with(prefix) {
            return item
        }
    }
    panic!()
}
```

prefix has lifetime a

values and returned string have lifetime b

lifetime annotations

```
fn get_first_matching<'a, 'b>(prefix: &'a str, values: &'b Vec<String>)
                            -> &'b String {
    for item in values {
        if item.starts_with(prefix) {
            return item
        }
    }
    panic!()
}

fn get_first(values: &Vec<String>) -> &String {
    let prefix: String = compute_prefix();
    return get_first_matching(&prefix, values)
    // prefix deallocated here
}
```

rules to stop dangling pointers (2)

objects have an single **owner**

owner can give away ownership permanently

- object is "moved"

owner can let someone borrow object **temporarily**

- must know when object is given back

only **modify** object when exactly one user

- owner or exclusive borrower

restricting modification

```
fn modifyVector(vector: &mut Vec<u32>) { ... }
fn foo() {
    let vector: Vec<u32> = vec![1, 2, 3];
    for value in &vector {
        if value == 2 {
            modifyVector(&mut vector) // ERROR
        }
    }
}
```

trying to give away mutable reference

…while the for loop has a reference

data races

Rusts rules around modification built assuming concurrency

idea: multiple processes/threads running at same time might use value

safe policy: all reading *or* only one at a time

if multiple at a time: problems are called “data races”

data races for use-after-free

```
Expand Vec             | Read Vec
-------------------------------------------
                       | mov pointer, %rax
                       | ...
mov $100, %rdi         |
call malloc            |
mov pointer, %rdi      |
mov %rax, pointer      |
call free              |
                       | ...
                       | mov (%rax), %rax
```

what about dynamic allocation?

saw Rust's Vec class — equivalent to C++ vector/Java ArrayList

idea: Vec wraps a heap allocation of an array

owner of Vec "owns" heap allocation

- delete when no owner

also Box class — wraps heap allocation of a single value

- basically same as Vec except one element

escape hatch

Rust lets you avoid compiler's mechanisms

implement your own

unsafe keyword

how Vec is implemented

deep inside Vec

```
pub struct Vec<T> {
    buf: RawVec<T>, // interface to malloc
    len: usize,
}

impl<T> Vec<T> {
    ...
    pub fn truncate(&mut self, len: usize) {
        unsafe {
            // drop any extra elements
            while len < self.len {
                // decrement len before the drop_in_place(), so a panic on Drop
                // doesn't re-drop the just-failed value.
                self.len -= 1;
                let len = self.len;
                ptr::drop_in_place(self.get_unchecked_mut(len));
            }
        }
    }
    ...
}
```

Rust escape hatch support

escape hatch: make new reference-like types

callbacks on ownership ending (normally deallocation)

choice of what happens on move/copy

alternative rule: reference counting

keep track of number of references

delete when count goes to zero

- Rust automatically calls destructor — no programmer effort

Rust implement with Rc type (“counted reference”)

Ref Counting Example

```rust
struct Grade {
    score: i32, studentName: String, assignmentName: String,
}
struct Student {
    name: String,
    grades: Vec<Rc<Grade>>,
}
struct Assignment {
    name: String
    grades: Vec<Rc<Grade>>
}

fn add_grade(student: &mut Student, assignment: &mut Assignment, score: i32) {
    let grade = Rc::new(Grade {
        score: i32,
        studentName: student.name,
        assignmentName: assignment.name,
    })
    student.grades.push(grade.clone())
    assignment.grades.push(grade.clone())
}
```

Rust escape hatch support

escape hatch: make new reference-like types

Rc: Rc<T> acts like &T

callbacks on ownership ending (normally deallocation)

Rc: deallocating Rc<T> decrements shared count

choice of what happens on move/copy

Rc: transferring Rc makes new copy, increments shared count

Rc implementationed (annotated) (1)

```
impl<T: ?Sized> Clone for Rc<T> {
    ...
    fn clone(&self) -> Rc<T> {
        self.inc_strong(); // <-- incremenet reference count
        Rc { ptr: self.ptr }
    }
}
```

Rc implementation (annotated) (2)

```
unsafe impl<#[may_dangle] T: ?Sized> Drop for Rc<T> {
    ...
    fn drop(&mut self) { // <-- compilers calls on deallocation
        unsafe {
           let ptr = *self.ptr;

           self.dec_strong(); // <-- decrement reference cont
           if self.strong() == 0 { // if ref count is 0
                // destroy the contained object
                ptr::drop_in_place(&mut (*ptr).value);
                ...
            }
        }
    }
    ...
}
```

other policies Rust supports

RefCell — borrowing, but check at runtime, not compile-time

detect at runtime if used while already used

internally: destructo call when returned object goes out of scope

Weak — reference-counting, but don't contribute to count

detect at runtime if used with count $= 0$

…

other policies Rust supports

RefCell — borrowing, but check at runtime, not compile-time

detect at runtime if used while already used

internally: destructo call when returned object goes out of scope

Weak — reference-counting, but don't contribute to count

detect at runtime if used with count $= 0$

…

zero-overhead

normal case — lifetimes — have no overhead

compiler proves safety, generates code with no bookkeeping

other policies (e.g. reference counting) do

…but can implement new ones if not good enough

backup slides

Rust linked list

not actually a good idea

use `Box<...>` to represent object on the heap

no null, use `Option<Box<...>>` to represent pointer.

Rust linked list (not recommended)

```
struct LinkedListNode {
    value: u32,
    next: Option<Box<LinkedListNode>>,
}

fn allocate_list() -> LinkedListNode {
    return LinkedListNode {
        value: 1,
        next: Some(Box::new(LinkedListNode {
            value: 2,
            next: Some(Box::new(LinkedListNode {
                value: 3,
                next: None
            }))
        }))
    }
}
```

why the box? (1)

```
struct LinkedListNode { // ERROR
    value: u32,
    next: Option<LinkedListNode>,
}

// error[E0072]: recursive type `LinkedListNode` has infinite size
```

why the box? (2)

```
struct LinkedListNode { // ERROR
    value: u32,
    next: Option<&LinkedListNode>,
}
// error[E0106]: missing lifetime specifier
//  --> src/main.rs:48:18
//    |
// 48 |     next: Option<&LinkedListNode>,
//    |                  ^ expected lifetime parameter
```

taint tracking generally

taint tracking for other security issues is a big research area

often by implementing taint tracking for assembly

- much, much higher overhead than implementing for Perl or Ruby

taint tracking generally

taint tracking for other security issues is a big research area

often by implementing taint tracking for assembly

- much, much higher overhead than implementing for Perl or Ruby

---

example: detect private information leaks

- taint personal information
- error if tainted info goes to network

example: check if return addresses are tainted before using