last time

coverage-guided fuzzing

- random tests based on set of base tests
- new path taken: add test to set of base tests

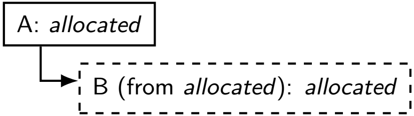
complete (finds any problem) v sound (problems found really problems)

static analysis

- *abstract interpretation*: summary values e.g. allocated/freed
- approximations to avoid analyzing complex if statements, etc.

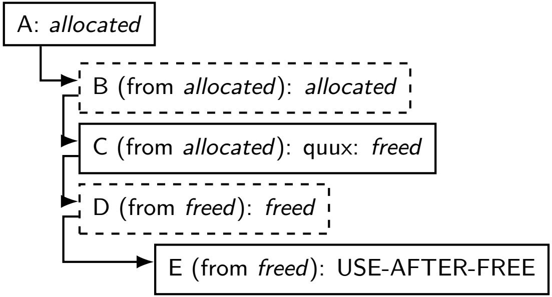
checking use-after-free (3)

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



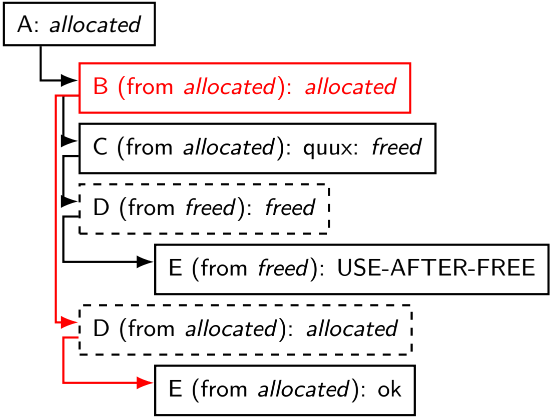
checking use-after-free (3)

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



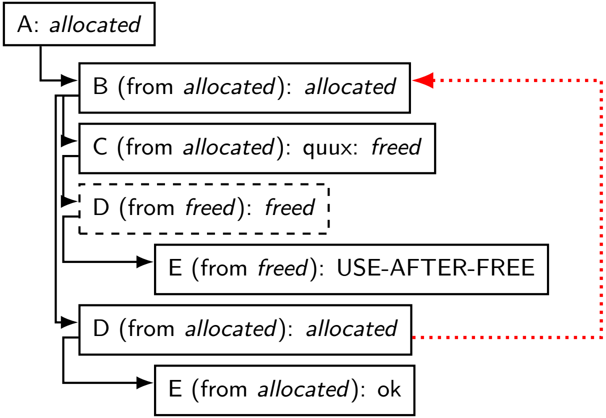
checking use-after-free (3)

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



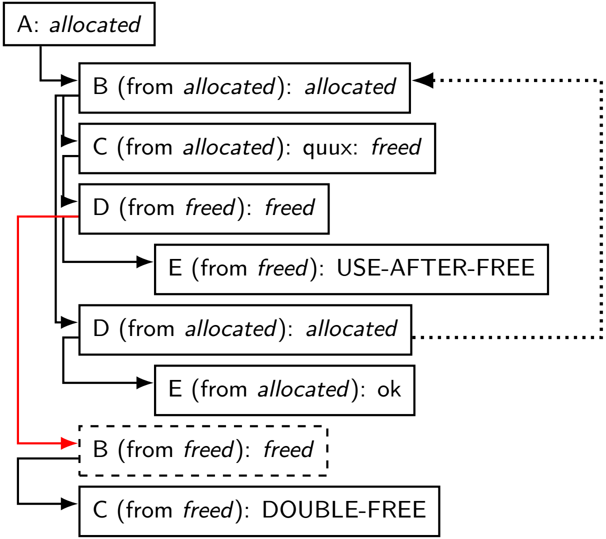
checking use-after-free (3)

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



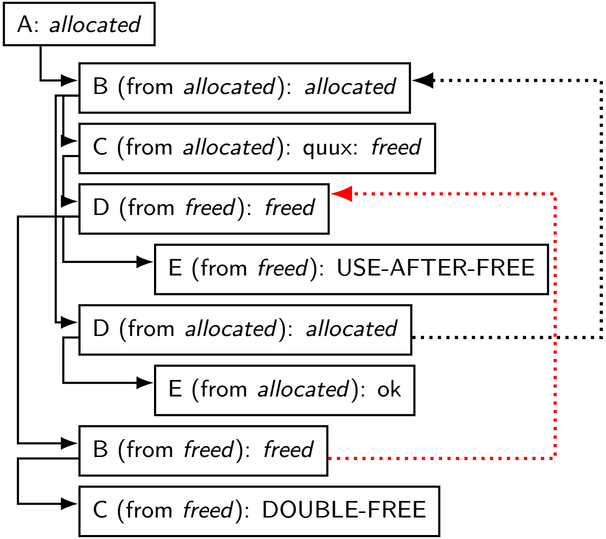
checking use-after-free (3)

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



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++;
    ...
}
```



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)

```c
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: $[-\inf, +\inf]$; array: indices [0, 99]

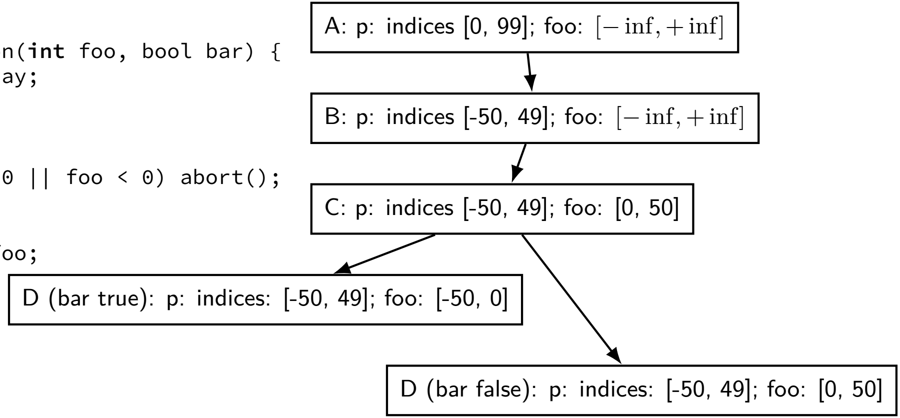
B: foo: $[-\inf, +100]$; array: indices [0, 99]

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;
}
```



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: $[-\inf, +\inf]$

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

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

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

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

warn about possible out-of-bounds?

common bug patterns

effectively detecting things like “arrays are in bounds”
or “values aren't used after being freed”
is not very reliable for large programs

(but analysis tools true and are getting better)

but static analysis tools shine for common bug patterns

patterns clang's analyzer knows

```
struct foo *p = malloc(sizeof(struct foo*)); // meant struct foo?
long *p = malloc(16 * sizeof(int)); // meant sizeof(long)?
```

```
strncat(foo, bar, sizeof(foo));
```

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

more suspect patterns

SpotBugs: Java static analysis tool

```
// pattern: connecting to database with empty password:
connection = DriverManager.getConnection(
    "jdbc:hsqldb:hsql://db.example.com/xdb" /* database ID */,
    "sa" /* username */, "" /* password */);

// pattern: Sql.hasResult()'s second argument isn't a constant
Sql.hasResult(c, "SELECT 1 FROM myTable WHERE code='"+code+"'");

// pattern: new FileReader's argument comes from request
HttpRequest request = ...;
String path = request.getParameter("path");
BufferedReader r = new BufferedReader(
    new FileReader("data/" + path));
```

preview: information flow

really common pattern we want to find:
data from somewhere gets to dangerous place

- pointer to stack escapes function
- input makes it to SQL query, file name

we'll talk about it specially next

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:

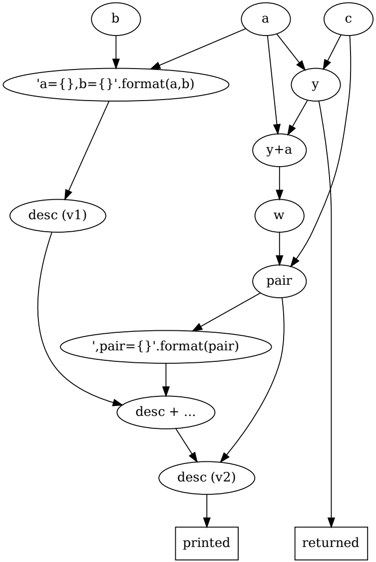
- does an array index depend on user input?
- does an SQL query depend on user input?
- does data sent over network depend on phone number?

…

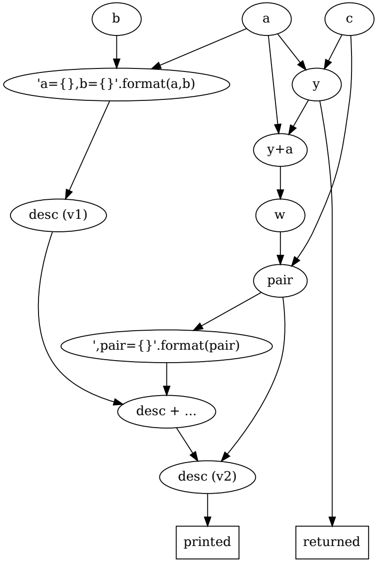
can do this *statically* (potential dependencies)
or *dynamically* (actual dependencies as program runs)

information 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
```



information flow graph (1b)

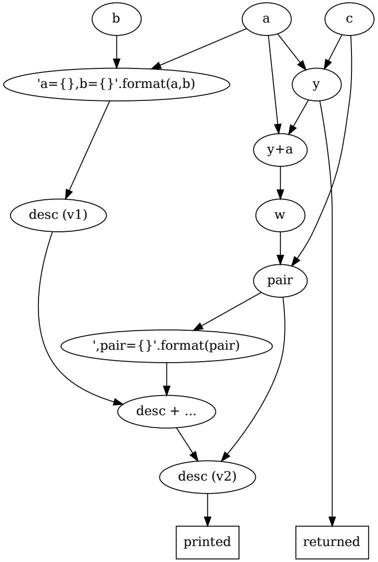


information flow graph (1b)

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

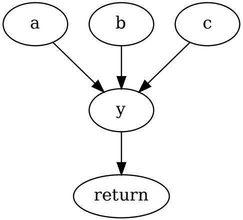
ex: does value of pair depend on a, b, c?

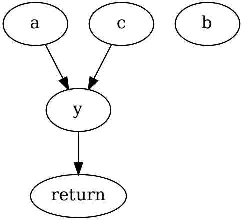
ex: does printed value depend on a, b, c?



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?

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

static info 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]);
}
```

same points-to problem with static analysis

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

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

can fix this with dynamic approach: monitor running program

static info 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

can fix this we dynamic approach: monitor running program

static info 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

harder to fix this issue

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 -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
#! 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]
```

also similar for virtual disk, network, ...

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:

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

taint tracking for malware analysis

uses proposed by Panaroma authors:

keypresses $\rightarrow$ network packets

network packets $\rightarrow$ malware outputs

browser history $\rightarrow$ network packets

defeating ASM-based checking

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

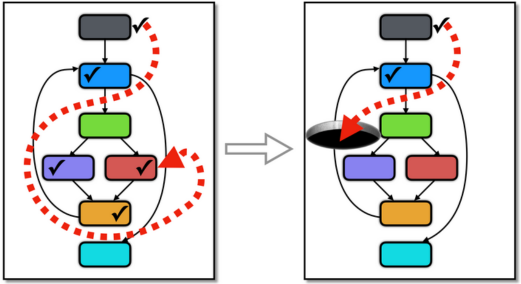
A. timing arithmetic operations to see if the machine is unusually slow

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

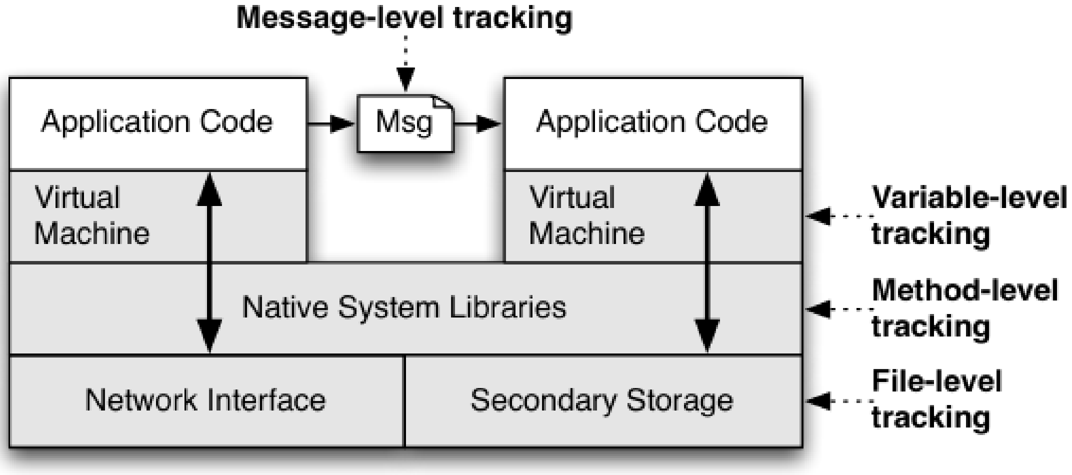


Figure 1: Multi-level approach for performance efficient taint tracking within a common smartphone architecture.

TaintDroid resutls

Table 3: Potential privacy violations by 20 of the studied applications. Note that three applications had multiple 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 phone number, IMSI, and ICC-ID along with the geo-coordinates to the app's content server.
Device ID to Content Servers (7)*	2 Social, 1 Shopping, 1 Reference and three other apps transmitted the IMEI number to the app's content server.
Location to Advertisement Servers (15)	5 apps sent geo-coordinates to ad.qwapi.com, 5 apps to admob.com, 2 apps to ads.mobclix.com (1 sent location both to admob.com and ads.mobclix.com) and 4 apps sent location† to data.flurry.com.

* TaintDroid flagged nine applications in this category, but only seven transmitted the raw IMEI without mentioning such practice in the EULA.

† To the best of our knowledge, the binary messages contained tainted location data (see the discussion below).

TaintDroid and performance

modifying Dalvik ($\sim$ Java) VM allows very good performance

could do this sort of tracking on a “live” system

logistics note

next few planned topics:

(next) systems programming languages with memory safety (Rust as example)

(after) sandboxing / privilege separation

- running code without trusting it as much

hardware support for memory safety + CFI (memory safety mitigation)

concurrency bugs / time of check to time of use

could make adjustments if there are topics people especially want

why are people still using C/C++?

Python, Java, ...are great languages

why are people using C, C++, etc.?
- which seem horrible for security?

history + good support
- lots of libraries in C, C++, ...

"zero overhead"
- safe languages don't make it easy to get "close to the machine"
- e.g. garbage collection overhead
- e.g. array checking overhead

no language VM — easier to distribute

why are people still using C/C++?

Python, Java, ...are great languages

why are people using C, C++, etc.?

- which seem horrible for security?

history + good support

- lots of libraries in C, C++, ...

"zero overhead"

- safe languages don't make it easy to get "close to the machine"
- e.g. garbage collection overhead
- e.g. array checking overhead

no language VM — easier to distribute

safety rules + escape hatch

idea: can avoid out-of-bounds, etc. with safety rules

...but safety rules don't allow us to do some things fast

so: have "escape hatch" to avoid safety checks in those cases

hope: code that uses escape hatch can be tightly checked

- good target for expensive program analysis

Java: unofficial escape hatch

Oracle JDK and OpenJDK come with a class called
`com.sun.Unsafe`

Example methods:

```
public long allocateMemory(long size);
                                              // returns pointer value
public void freeMemory(long address);
public long getLong(long address);
public void putLong(long address, long x);
```

can be used to, e.g., write "fast" IntArray class

so, if Java has escape hatch...

why do people not want to write
their performance-sensitive programs in Java?

hard to integrate code that uses escape hatch with normal Java code

hard to efficiently avoid dangling pointers when using escape hatch
- Is it safe to freeMemory from my FastIntArray class?

slow to pass garbage collected references to/from C/assembly code

hard to avoid using garbage collector
- garbage collector performance can be variable

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

backup slides

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