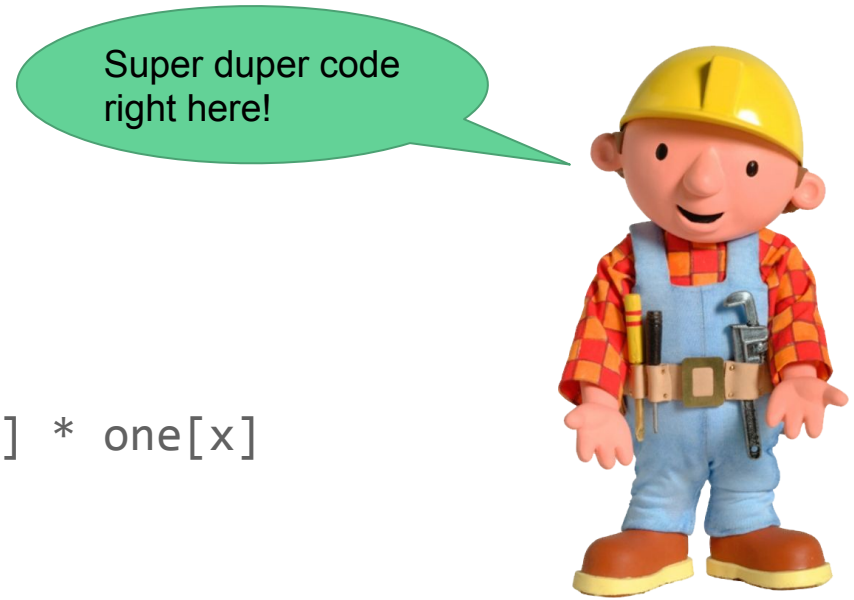
Provable Security

Review: Timing Attacks

```
def secureMix(one, two):
    if len(one) != len(two):
        return None
    x = 0
    while x < len(one):
        if one[x] % 2 == 0:
            two[x] *= one[x]
        else:
            two[x] *= one[x] * one[x] * one[x]
        x = x + 1
    return two
```



Review: Timing Attacks

```
def secureMix(one, two):
    if len(one) != len(two):
        return None
    x = 0
    while x < len(one):
        if one[x] % 2 == 0:
            two[x] *= one[x]
        else:
            two[x] *= one[x] * one[x] * one[x]
        x = x + 1
    return two
```

Code timing depends on *value* of secret data

Reduces bruteforce from exponential time to linear (w.r.t. length)

Preventing Timing Attacks

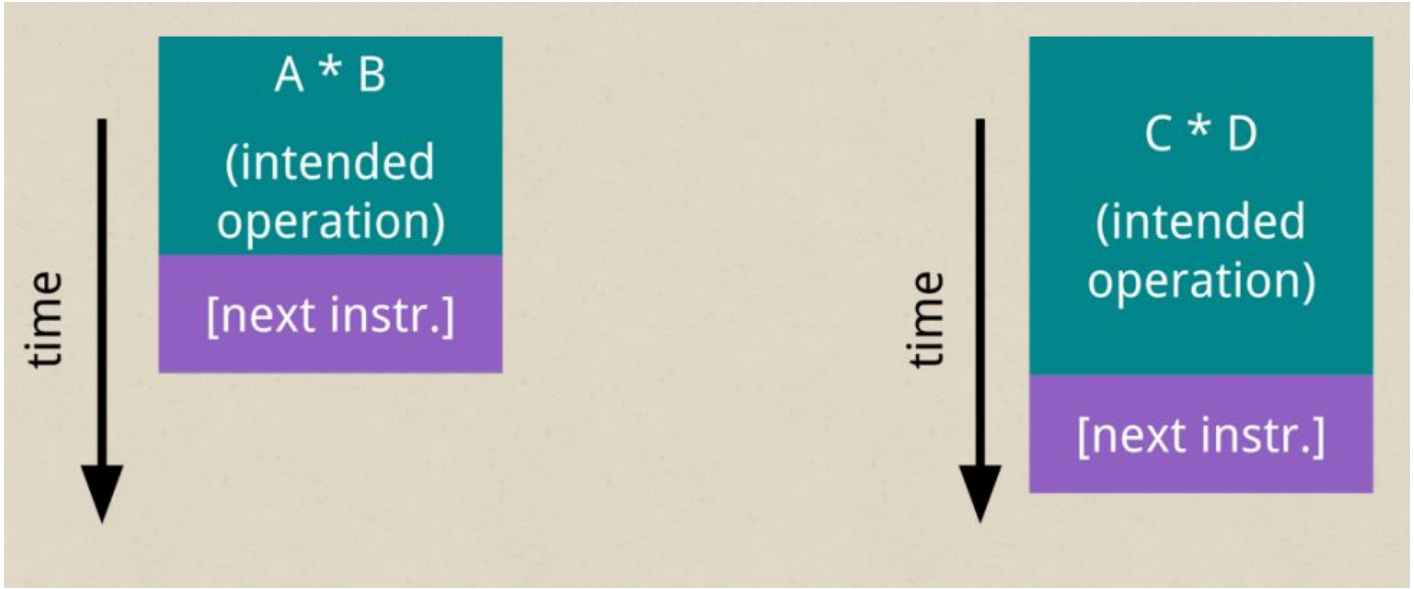
Use fixed processing time for code that involves secret data (every possible code path must take the same time) *

Avoid code branches and memory lookups

Some newer block ciphers use **add-rotate-xor (ARX)** algorithms (like ChaCha20) instead of substitution-boxes (like AES)

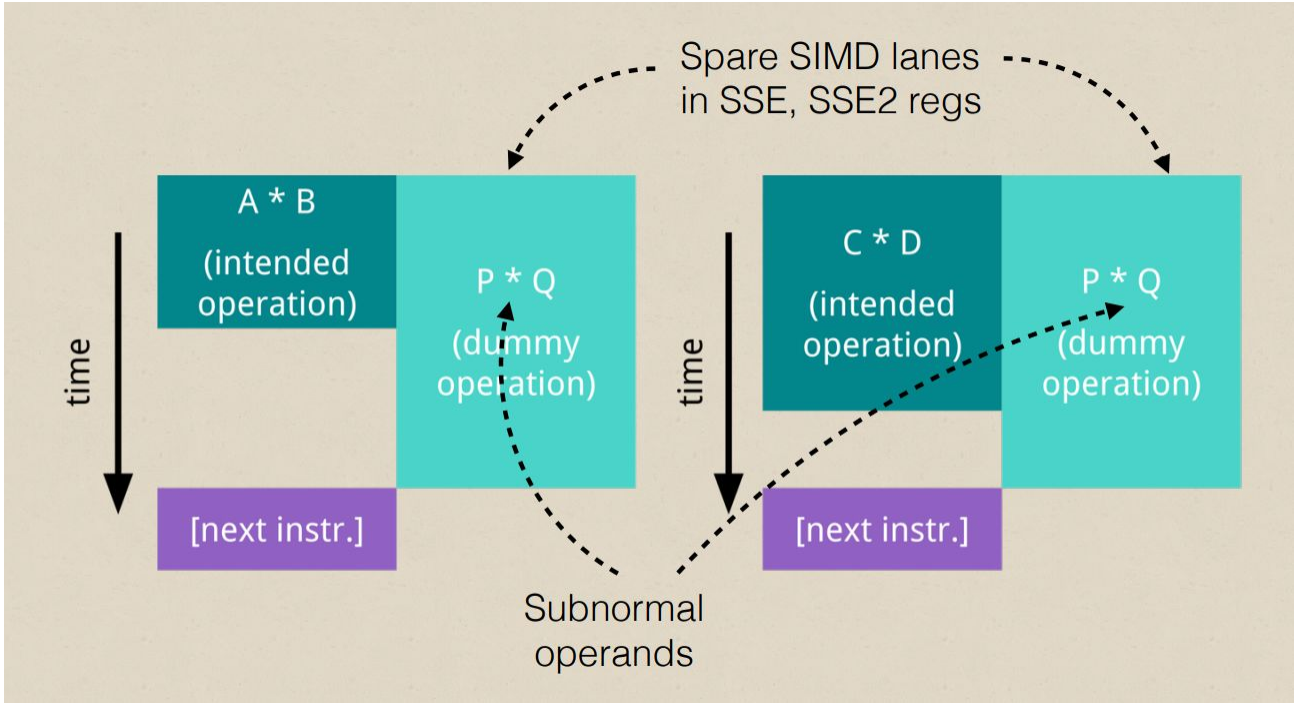
* Easier said than done

Variable-Latency Floating-Point Operations



Rane, A., Lin, C., and Tiwari, M. Secure, Precise, and Fast Floating-Point Operations on x86 Processors

Fixed-Latency Floating-Point Operations



Rane, A., Lin, C., and Tiwari, M. Secure, Precise, and Fast Floating-Point Operations on x86 Processors

Formal Methods

- Mathematical techniques for specification, development, and verification

Abstract Interpretation: Keeping track of RSVPs

Abstract Interpretation: SSNs

Guest List

- 518-67-2432
- 346-28-4287
- 289-54-4596
- 653-03-5012

Abstract Interpretation: First Names

Guest List

- Alice
- Bob
- Carl
- Dora

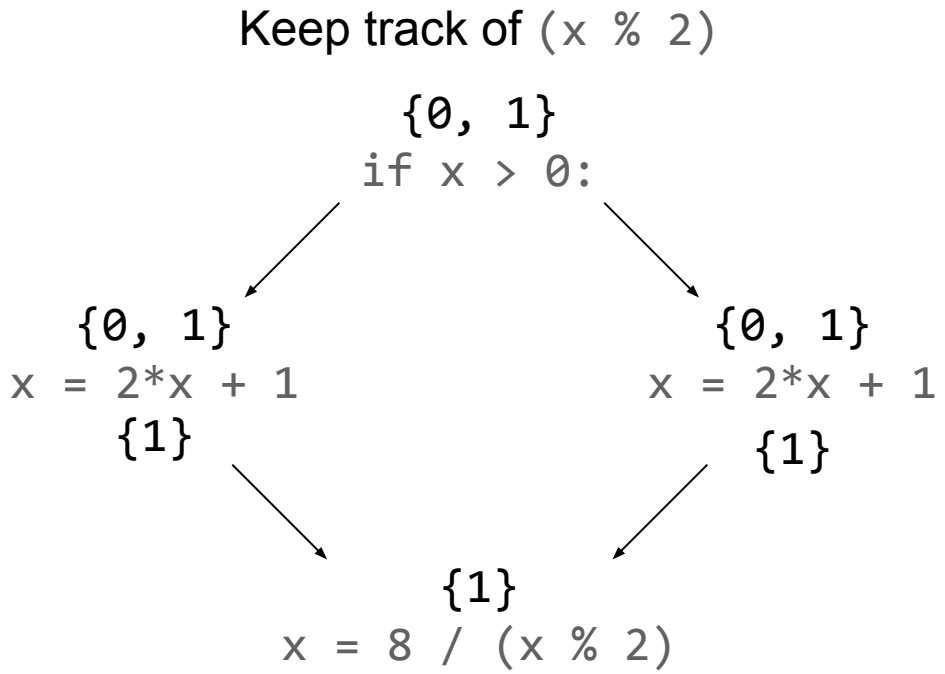
Is Bob Ross coming?

False positives are possible, but not false negatives

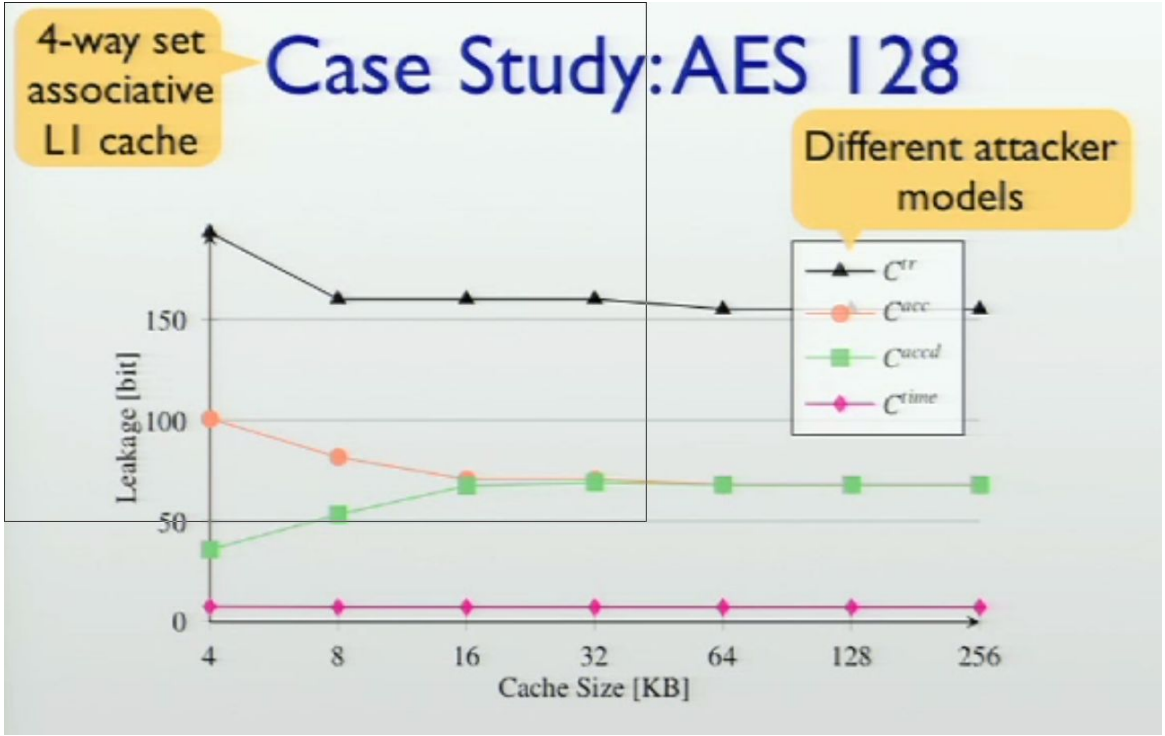
Abstract Interpretation: Static Analysis

```
if x > 0:
    x = 2*x + 1
else:
    x = 1 - 4*x
x = 8 / (x % 2)
```

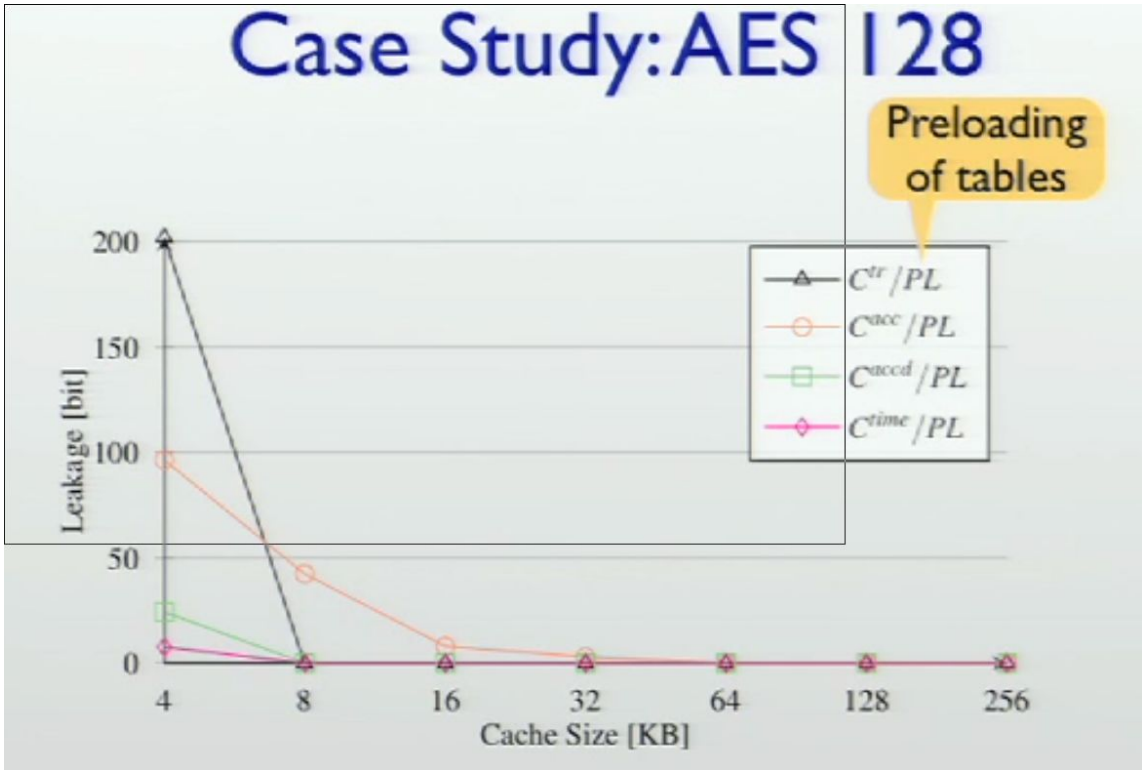
Will this program ever divide by zero?



Abstract Interpretation: Cache Audit



Abstract Interpretation: Cache Audit



Model Checking: HMAC

```
// H((K xor opad) || H((K xor ipad) || message))
hmac hash key message = hash (okey # internal)
 where
  okey = [k ^ 0x5C | k <- key] // K xor opad
  ikey = [k ^ 0x36 | k <- key] // K xor ipad
  // H((K xor ipad) || message)
  internal = split (hash (ikey # message))
```

== https://github.com/awslabs/s2n/blob/master/crypto/s2n_hmac.c

Program Annotations: Swapping Variables

```
X ::= X + Y;;

Y ::= X - Y;;

X ::= X - Y
```

Program Annotations: Swapping Variables

```
{{ X = m /\ Y = n }}

X ::= X + Y;;

Y ::= X - Y;;

X ::= X - Y
{{ X = n /\ Y = m }}
```

Program Annotations: Swapping Variables

```
{{ X = m /\ Y = n }}

X ::= X + Y;;

Y ::= X - Y;;
{{ X - Y = n /\ Y = m }}
X ::= X - Y
{{ X = n /\ Y = m }}
```

Program Annotations: Swapping Variables

```
{{ X = m /\ Y = n }}

X ::= X + Y;;
{{ X - (X - Y) = n /\ X - Y = m }}
Y ::= X - Y;;
{{ X - Y = n /\ Y = m }}
X ::= X - Y
{{ X = n /\ Y = m }}
```

Program Annotations: Swapping Variables

```
{{ X = m /\ Y = n }}
{{ (X + Y) - ((X + Y) - Y) = n /\ (X + Y) - Y = m }}
X ::= X + Y;;
{{ X - (X - Y) = n /\ X - Y = m }}
Y ::= X - Y;;
{{ X - Y = n /\ Y = m }}
X ::= X - Y
{{ X = n /\ Y = m }}
```

Program Annotations: Swapping Variables

```
{{ X = m /\ Y = n }}
{{ (X + Y) - ((X + Y) - Y) = n /\ (X + Y) - Y = m }}
X ::= X + Y;;
{{ X - (X - Y) = n /\ X - Y = m }}
Y ::= X - Y;;
{{ X - Y = n /\ Y = m }}
X ::= X - Y
{{ X = n /\ Y = m }}
```

```
(X + Y) - ((X + Y) - Y) = n /\ (X + Y) - Y = m
(m + n) - ((m + n) - n) = n /\ (m + n) - n = m
n = n /\ m = m
```

Downsides

Formal verification can only be as effective as the underlying **assumptions** made by your system

We must formalize the system semantics correctly (which can be very tricky!) to reason about how they work together in larger programs

We can verify a C program but... what if the compiler has bugs? what if the OS has bugs? what if the hardware has bugs?

💥 State Explosion 💥

As program complexity increases linearly, the number of possible states increases exponentially!
Branches and loops are commonplace in programs but significantly increase state

```
if (a = 1) { /* something */ } else { /* another thing */ }
if (b = 1) { /* something */ } else { /* another thing */ }
if (c = 1) { /* something */ } else { /* another thing */ }
if (d = 1) { /* something */ } else { /* another thing */ }
```

== 2^4 = 16 paths

💥 State Explosion 💥

As program complexity increases linearly, the number of possible states increases exponentially!
Branches and loops are commonplace in programs but significantly increase state

```
unsigned int x = <some value>;
while (x) { /* loop body */ }
```

Up to 2^{32}

Dealing with the 💥 State Explosion 💥 problem

Iterative deepening -- only consider states with depth 1, then 2, then 3, etc

Abstract out or ignore some known features (library functions, functions that are out-of-scope, etc)

Concolic testing: combine concrete testing with symbolic testing to avoid some states

ct-verif

CS4501-007, Proof Engineering

This course will introduce students to type theory and thus to the integration of logic and functional programming. Students will learn to use the Coq proof assistant. Topics to be covered include inductive data type definitions, pure functional programming, propositions as types, logical representation of properties and relations, proof engineering, programs as data, program evaluation, Hoare Logic, lambda calculus, and type checking.

Model-Based Design of Cyber-Physical Systems

Cyber-Physical Systems (CPS) are smart systems that include co-engineered interacting networks of physical and computational components. Examples of CPS include medical devices, cars, and robots. Increasingly, such systems are everywhere. This course will give you the required skills to formally analyze the CPS that are all around us, so that when you contribute to the design of CPS, you are able to understand important safety-critical aspects and feel confident designing and analyzing system models. It will provide an excellent foundation for students who seek industry positions and for students interested in pursuing research.

Homework

- Fix a non-constant-time C function (https://github.com/cnsuva/modernsectopics/blob/master/3-30/hw.c)
- Send an email with your solution (and a 1-paragraph writeup) to cm7bv@virginia.edu with the subject “MST Assignment 9 - <YOUR_UVA_ID>”
- Tips:
  - Carefully consider what is and is not constant time
  - Your result must have the same output
  - Don’t worry about cache or other hardware attacks
- **Don’t hesitate to ask questions!**

Further Reading

- Verifying an HMAC implementation (https://galois.com/blog/2016/09/verifying-s2n-hmac-with-saw/)
- A survey of symbolic methods for establishing equivalence-based properties in cryptographic protocols [Sections 1 and 3] (https://arxiv.org/pdf/1610.08279.pdf)