

Modeling and Verification of Security Properties in HW/SW Systems

Pramod Subramanyan

SAT + SMT Winter School

December 8, 2018

spramod@cse.iitk.ac.in

<https://github.com/pramodsu/satsmt2018>

Commit to Linux circa 2003

```
+ if ((options == (__WCLONE|__WALL)) && (current->uid = 0))  
+     retval = -EINVAL;
```

- Nov 2003: An unknown patch to the wait4 system call added these two lines to the kernel source code
- Question: what is the problem with these modifications?
- A: Creates a backdoor for any program to become root

<https://lwn.net/Articles/57135/>

Way back in 2007

A bug nearly killed a company

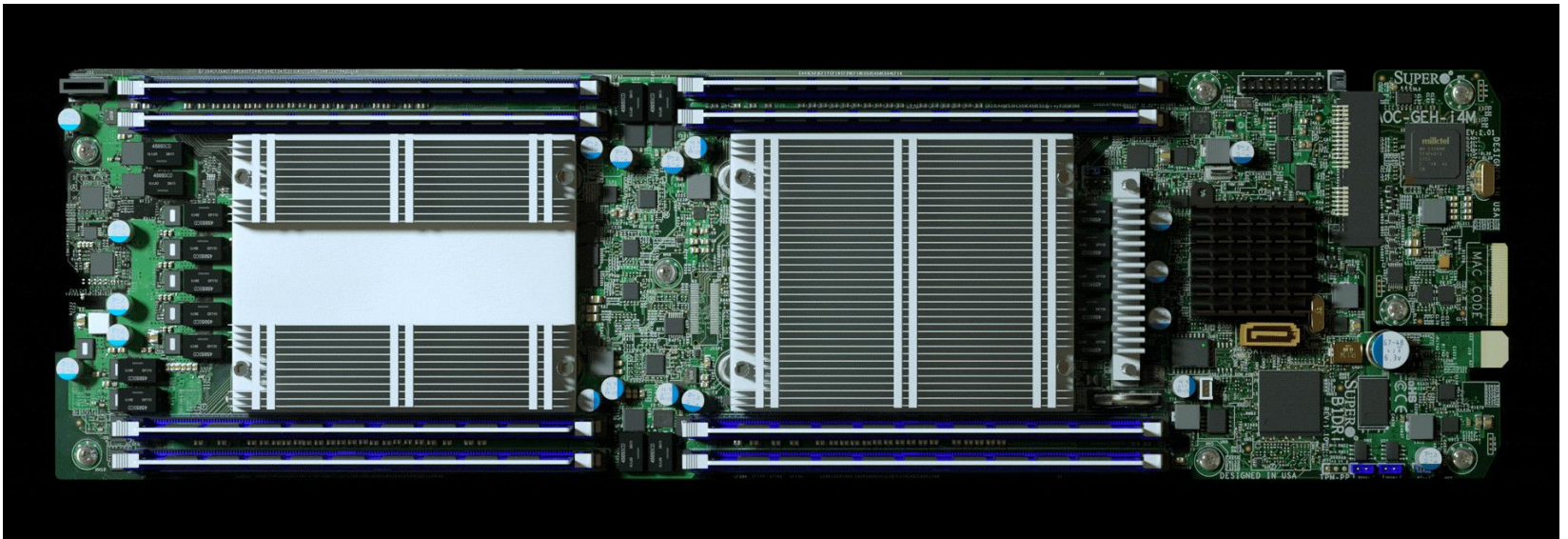


<http://www.techwarelabs.com/reviews/processors/barcelona/>

- AMD slows down shipment of Barcelona chips ([Network World](#))
- AMD delaying Barcelona volume shipments to next year ([CNET](#))
- AMD Phenom, Barcelona Chips Hit By Lock-up Bug ([ExtremeTech](#))
- Bug In AMD's Quad-Core Barcelona And Phenom May Be More Serious Than Previously Suspected ([InformationWeek](#))
- AMD's Barcelona, Phenom suffer early setbacks ([arstechnica](#))
- AMD in trouble: Barcelona bug, ATI write-down cast bad shadows ([betanews](#))

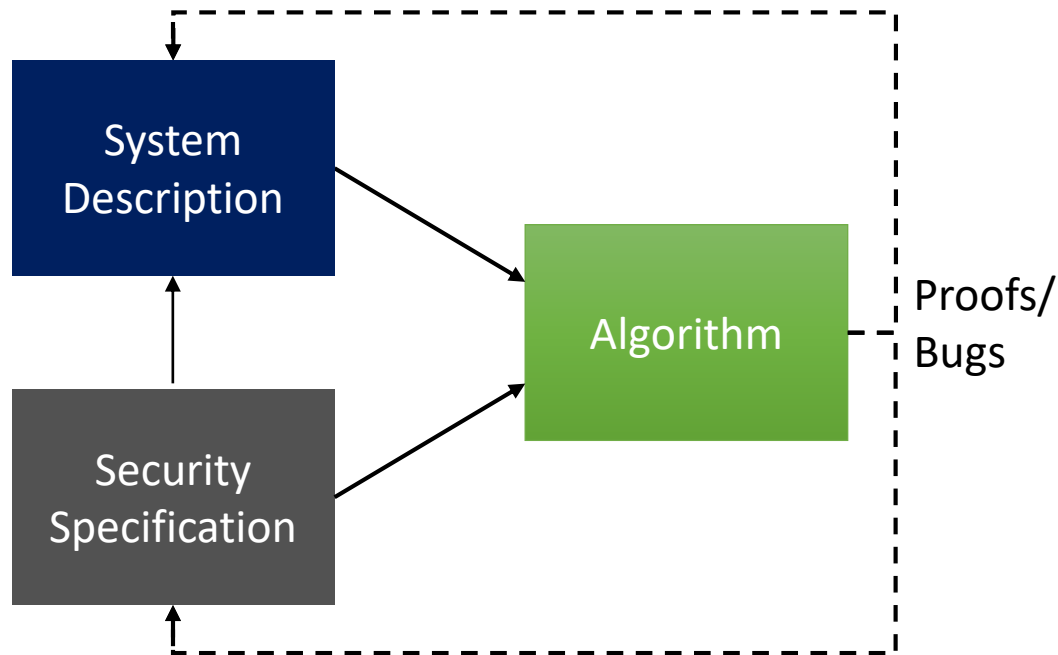
More Recently, “The Big Hack”

How China Used a Tiny Chip to **(maybe)** Infiltrate US Companies



<https://www.bloomberg.com/news/features/2018-10-04/the-big-hack-how-china-used-a-tiny-chip-to-infiltrate-america-s-top-companies>

In an Ideal World



We would algorithmically discover security vulnerabilities

In this tutorial

We will learn:

- How do we automatically find vulnerabilities?
- How do we prove the absence of certain kinds of vulnerabilities?

We'll need to solve three problems

- How to transform a system into something that a computer program can analyze (**modeling**)?
- How to **specify** what constitutes a vulnerability?
- How we **find** these **bugs** or **prove** their **absence**?

Outline of this tutorial

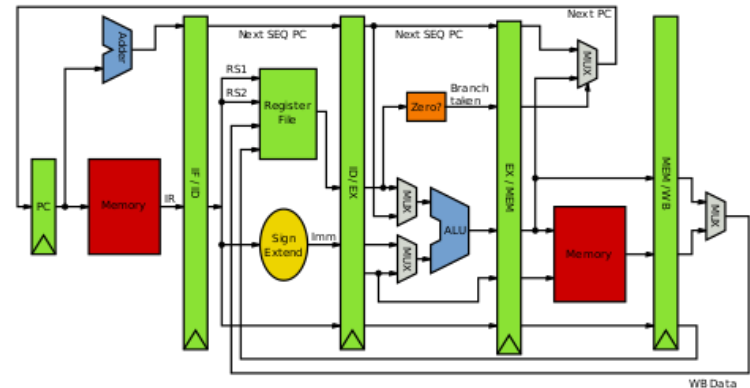
1. Background: Modeling, Specification, Verification
2. Security Property Verification (Hyperproperties)
3. Case Study: Enclave Platforms

1. Background: Modeling, Specification, Verification
 - A Brief Introduction to Modeling
2. Security Property Verification (Hyperproperties)
3. Case Study: Enclave Platforms

What is Modeling?



Reality



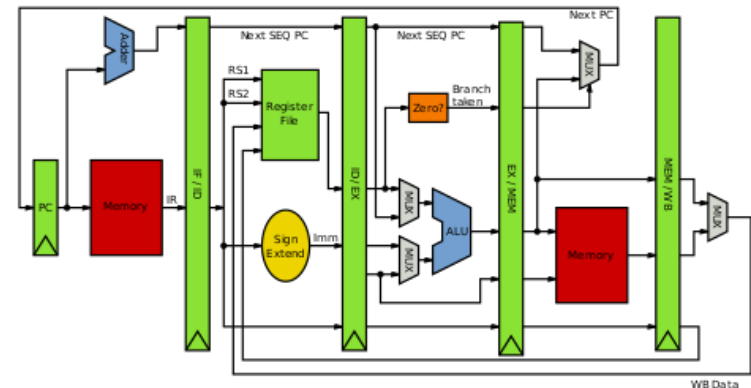
Model aka "Virtuality"

Process of transforming a real-world object into a mathematical (in our case specifically machine-readable) object that we can analyze

What is Modeling?



Reality



Model aka “Virtuality”

Questions

1. Is the model the same as the source code?
2. Is it possible to build a perfectly accurate model of a system?

For us, Models are **Transition Systems**

$$M = (X, Init, R)$$

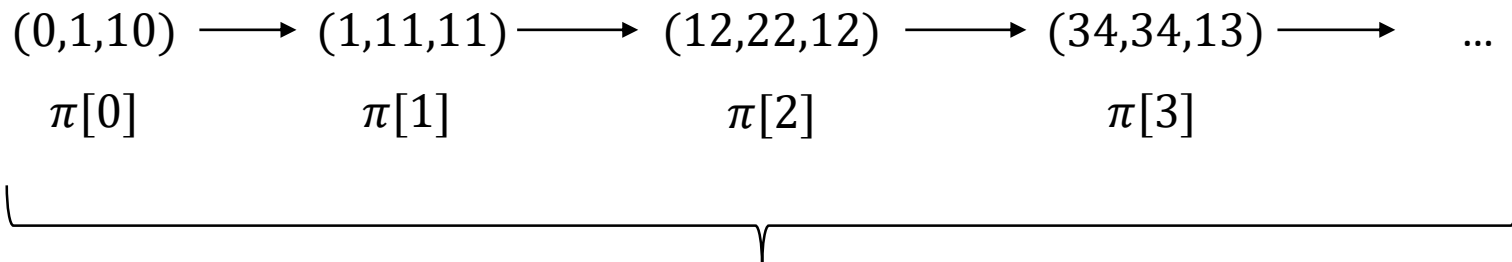
- X is a set of state variables
- $Init(X)$ is a predicate that defines the init state
- $R(X, X')$ defines the transition relation

We look for bugs by defining formulas over runs of the transition system to capture “good”/“bad” states

Example: Transition System

$$M = (X, Init, R)$$

- $X \doteq \{x, y, z\}$
- $Init(X) \doteq x \geq 0 \wedge y > 0 \wedge z > 0$
- $R(X, X') \doteq x' = x + y \wedge y' = y + z \wedge z' = z + 1$



This is called a **trace** or a **run** of the transition system

Example: Transition System

$$M = (X, Init, R)$$

- $X \doteq \{x, y, z\}$
- $Init(X) \doteq x \geq 0 \wedge y > 0 \wedge z > 0$
- $R(X, X') \doteq x' = x + y \wedge y' = y + z \wedge z' = z + 1$

$(0,1,10) \longrightarrow (1,11,11) \longrightarrow (12,22,12) \longrightarrow (34,34,13) \longrightarrow \dots$

$(0,1,1) \longrightarrow (1,2,2) \longrightarrow (3,4,3) \longrightarrow (7,7,4) \longrightarrow \dots$

$(1,2,3) \longrightarrow (3,5,4) \longrightarrow (8,9,5) \longrightarrow (17,14,6) \longrightarrow \dots$

The system has infinitely many traces

Traces of a Transition System (1/3)

$$M = (X, Init, R)$$

Trace π of the transition system is a sequence of assignments to the state variables such that:

- $Init(\pi[0])$ holds
- For all $i \in \mathbb{N}$, $R(\pi[i], \pi[i + 1])$ holds

Traces of a Transition System (2/3)

Question 1: what length are traces?

- They could be of infinite length

Question 2: Why is this useful/necessary?

- Ex: some applications don't terminate

Traces of a Transition System (3/3)

Question 2: Our formulas seem to be in first order logic. Are they in pure first order logic?

$$M = (X, Init, R)$$

- $X \doteq \{x, y, z\}$
- $Init(X) \doteq x \geq 0 \wedge y > 0 \wedge z > 0$
- $R(X, X') \doteq x' = x + y \wedge y' = y + z \wedge z' = z + 1$

1. Background: Modeling, Specification, Verification
 - A Brief Introduction to Modeling
 - Brief Interlude: First-Order Logic
2. Security Property Verification (Hyperproperties)
3. Case Study: Enclave Platforms

First-Order Logic

- A formal notation for mathematics, with expressions involving
 - Propositional symbols
 - Predicates
 - Functions and constant symbols
 - Quantifiers
- In contrast, propositional (Boolean) logic only involves propositional symbols and operators

First-Order Logic: Syntax

- Logical Symbols
 - Parentheses: (,)
 - Propositional connectives: \wedge , \neg , \vee ,
 - Variables: a , b , ...
 - Quantifiers: \exists , \forall
- Non-logical symbols/Parameters
 - Equality: $=$
 - Functions: $+$, $-$, $\%$, $\&$, $f()$, concat , ...
 - Predicates: \cdot , is_substring , ...
 - Constant symbols: 0 , 1.0 , null , ...

First-Order vs. Propositional Logic

- PL only has propositional variables (Booleans)
- And only logical connectives applied to variables
- FOL has terms
 - Which can be 0-ary (constants)
 - Or the application of functions to terms
 - These are not Boolean (can be over arbitrary domains)
- Formulas are constructed using atoms
 - Atoms consist of T, F and predicates over variables
 - And combined using the logical connectives/quantifiers

Some Examples of FOL Formulas

- $\forall x. p(f(x), y) \rightarrow \forall y. p(f(x), y)$
 - x is a variable
 - $p(_, _)$ is a predicate
 - $f(_)$ is a function
 - \rightarrow is a logical connective
- $\forall x, y, z. \text{triangle}(x, y, z) \rightarrow \text{len}(x) < \text{len}(y) + \text{len}(z)$
 - Variables: x, y, z
 - Predicates: $\text{triangle}, <$
 - Functions: $\text{len}, +$
 - Connectives: \rightarrow

Pure FOL and Satisfiability

- FOL itself doesn't define symbols $+$, $<$, $=$, len etc.
- Semantics of FOL are defined in terms of interpretations
- Interpretations provide meanings to symbols like $+$ and $<$ and assignments to variables
 - Defines the set of values from which variables are drawn
 - Defines mapping from variables/functions to this set

Example of FOL Satisfiability

- Consider formula $F : x + y > z \rightarrow y > z - x$
- Interpretation $I : (Z, \alpha_I)$,
 - Z (integers) is the set from which values are drawn
 - $\alpha_I : \{+ \rightarrow +_Z, - \rightarrow -_Z, > \rightarrow >_Z, x \rightarrow 13_Z, y \rightarrow 42_Z, z \rightarrow 1_Z\}$
 - α_I tells what $+$, $-$, etc. mean for this interpretation
- $I \models F$
 1. $I \models x + y > z$ since $\alpha_I [x + y > z] = 13_Z +_Z 42_Z >_Z 1_Z$
 2. $I \models y > z - x$ since $\alpha_I [y > z - x] = 42_Z >_Z 1_Z -_Z 13_Z$
 3. $I \models F$ by 1, 2, and the semantics of \rightarrow

FOL with Background Theories

- Satisfiability in Pure FOL is for any interpretation
- Example: $(x > y) \wedge (y > z) \wedge \neg(x > z)$
 - We want to think of $>$ as the usual greater than operator
 - But FOL doesn't know what $>$ means
 - It's the interpretation that assigns meaning to $>$
 - We could make this formula SAT by picking an some weird interpretations (ex: rock paper scissors!)
- Often we want “standard” meanings for $<$, $+$ etc.
 - These are referred to as **standard interpretations**
 - Referred to as **FOL with background theory T**

Logical Theory

- Defines a set of parameters (non-logical symbols) and their meanings
- This definition is called a **signature**.
- Example of a signature:

Theory of linear arithmetic over integers

Signature is $(0, 1, +, -, \cdot)$ interpreted over \mathbb{Z}

1. Background: Modeling, Specification, Verification
 - A Brief Introduction to Modeling (Continued)
 - Brief Interlude: First-Order Logic
2. Security Property Verification (Hyperproperties)
3. Case Study: Enclave Platforms

Ex1: Programs to Transition Systems

```
// assume (incr > 0) && (n > 0);
int foo(int incr, int n) {
    int i, sum = 0;
    for (i = 0; i < n; i++) sum = sum + incr;
    return sum;
}
```

$M = (X, Init, R)$ where

- $X \doteq \{n, incr, sum, i\}$
- $Init(X) \doteq incr > 0 \wedge n > 0 \wedge i = 0 \wedge sum = 0$
- $R(X, X') \doteq n' = n \wedge incr' = incr \wedge$
 $sum' = ite(i < n, sum + incr, sum) \wedge$
 $i' = ite(i < n, i + 1, i)$

Ex2: Linear Search (Arrays)

```
bool lsearch(int a[], int n, int v) {  
    for (int i=0; i < n; i++) {  
        if (a[i] == v) return true;  
    }  
    return false;  
}
```

See [lsearch.ucl](https://github.com/pramodsu/satsmt2018/blob/master/lsearch.ucl) in the repository:

<https://github.com/pramodsu/satsmt2018/blob/master/lsearch.ucl>

Ex3: Crypto (Uninterpreted Fns)

Model crypto functions as uninterpreted functions

Add axioms that specify crypto functionality

$$\forall k, m. dec(k, enc(k, m)) = m$$

$$\forall k_1, k_2. k_1 \neq k_2 \Rightarrow dec(k_1, enc(k_2, m)) \neq m$$

Axioms of the theory of equality and uninterpreted functions:

- $\forall xy: x = y \Rightarrow f(x) = f(y)$
- Along with properties of the equality operator

<https://github.com/pramodsu/satsmt2018/blob/master/crypto.ucl>

1. Background: Modeling, Specification, Verification

- A Brief Introduction to Modeling
- Brief Interlude: First-Order Logic
- Property Specification

2. Security Property Verification (Hyperproperties)

3. Case Study: Enclave Platforms

Properties of a Transition System

Consider system defined by: $M = (X, Init, R)$ where

- $X \doteq \{x, y, z\}$
- $Init(X) \doteq x \geq 0 \wedge y > 0 \wedge z > 0$
- $R(X, X') \doteq x' = x + y \wedge y' = y + z \wedge z' = z + 1$
- **Claim: for all reachable states of the transition system, $x \geq 0$**
- We will write this as:

$$\underbrace{\forall \pi: (Init(\pi[0]) \wedge \forall i. R(\pi[i], \pi[i + 1]))}_{\text{Valid trace of M}} \Rightarrow \underbrace{\forall i: \pi[i].x \geq 0}_{\text{Property of trace}}$$

Valid trace of M

Property of trace

Properties of a Transition System

Consider system defined by: $M = (X, Init, R)$ where

- $X \doteq \{x, y, z\}$
- $Init(X) \doteq x \geq 0 \wedge y > 0 \wedge z > 0$
- $R(X, X') \doteq x' = x + y \wedge y' = y + z \wedge z' = z + 1$
- **Claim: for all reachable states of the transition system, $x \geq 0$**
- We will write this as:

$$\underbrace{\forall \pi \in Traces(M)}_{\text{Valid trace of M}} : \underbrace{\forall i: \pi[i].x \geq 0}_{\text{Property of trace}}$$

Valid trace of M Property of trace

Safety vs. Liveness

- Safety property
 - “something bad must not happen”
 - E.g.: system should not crash
 - finite-length error trace
- Liveness property
 - “something good must happen”
 - E.g.: every packet sent must be received at its destination
 - infinite-length error trace

Examples: Safety or Liveness?

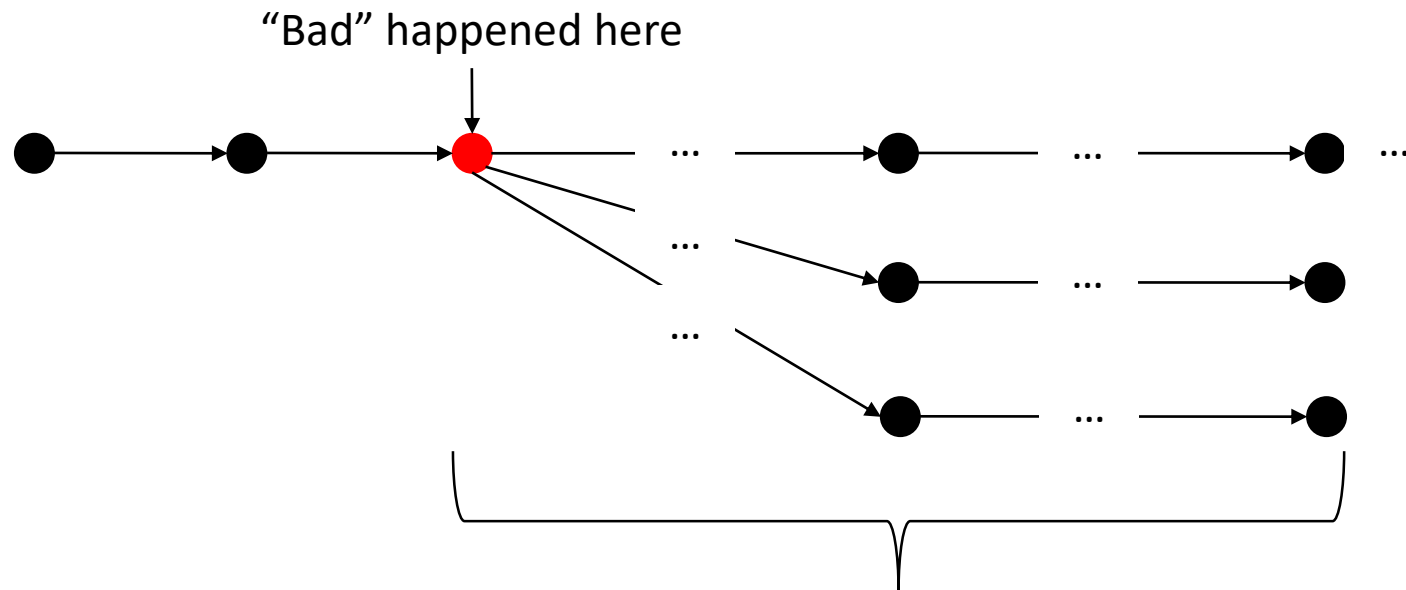
1. “No more than one processor (in a multi-processor system) should have a cache line in write mode”
2. “The grant signal must be asserted at some time after the request signal is asserted”
3. “Every request signal must receive an acknowledge and the request should stay asserted until the acknowledge signal is received”

Defining Safety

- A safety property is a property such that
 - for every trace that violates the property,
 - this trace has a prefix such that,
 - any extension of this prefix
 - also violates the property

“No more than one processor should have a cache line in write/modified state”

Safety in Pictures



Every suffix from the red point on is a violation

Defining Liveness

- A liveness property is a property s.t.
 - for every trace (violating the property),
 - every finite prefix of this trace
 - can be extended into a trace
 - that satisfies the property

“The grant signal must be asserted at some time after the request signal is asserted”

Theorem: Alpern and Schneider

Every trace property is an intersection of some safety property and some liveness property

[Alpern and Schneider, Information Processing Letters, '85]




Trace Properties and Logic

- Properties are typically specified in temporal logic
 - “Temporal” means related to time
 - In this case, time refers to states in a trace
- We won’t learn temporal logics and will focus on safety and variations of safety
- Instead we will use the quantified notation over traces that we saw before

System Invariants

- Some predicate ϕ that is satisfied by every reachable state of the transition system
- In other words, a property of the form:
 - $\forall \pi \in \text{Traces}(M): \forall i: \phi(\pi[i])$
- We will emphasize verifying invariants

Invariant or not?

- If the initial value of x is greater than 0, then x remains greater than 0 in every subsequent state 
 - $\forall \pi \in \text{Traces}(M): \pi[0].x > 0 \Rightarrow \forall i: \pi[i].x > 0$
- The value of x is always greater than or equal to 0 
 - $\forall \pi \in \text{Traces}(M): \forall i: \pi[i].x \geq 0$
- In every state, the value of x is less than its value in the next state 
 - $\forall \pi \in \text{Traces}(M): \forall i: \pi[i].x < \pi[i + 1].x$

Temporal Properties to Invariants

$X = \{ x, y \}$

$\text{Init}(X) = \text{true}$

$R(X, X') = y' > 0 \wedge x' = \text{ite}(x \geq 0, x+y', x-y')$

Claim

- If initial value of x is non-negative, then x remains non-negative
- If initial value of x was negative, then x remains negative

$$\forall \pi \in \text{Traces}(M): (\forall i: \text{sgn}(\pi[0].x) = \text{sgn}(\pi[i].x))$$

- Is this true?
- Can this be converted into an invariant?

Constructing a Monitor

X $= \{ x, y, x_0 \}$
 $\text{Init}(X)$ $= (x_0 = x)$
 $R(X, X')$ $= y' > 0 \wedge$
 $x' = \text{ite}(x \geq 0, x+y', x-y') \wedge$
 $x_0' = x_0$

$$\forall \pi \in \text{Traces}(M): (\forall i: \text{sgn}(\pi[i].x_0) = \text{sgn}(\pi[i].x))$$

[Claessen, Een, Sterin. A Circuit Approach to LTL Model Checking, FMCAD 2013]

1. Background: Modeling, Specification, Verification

- A Brief Introduction to Modeling
- Brief Interlude: First-Order Logic
- Property Specification
- Methods for Verification

2. Security Property Verification (Hyperproperties)

3. Case Study: Enclave Platforms

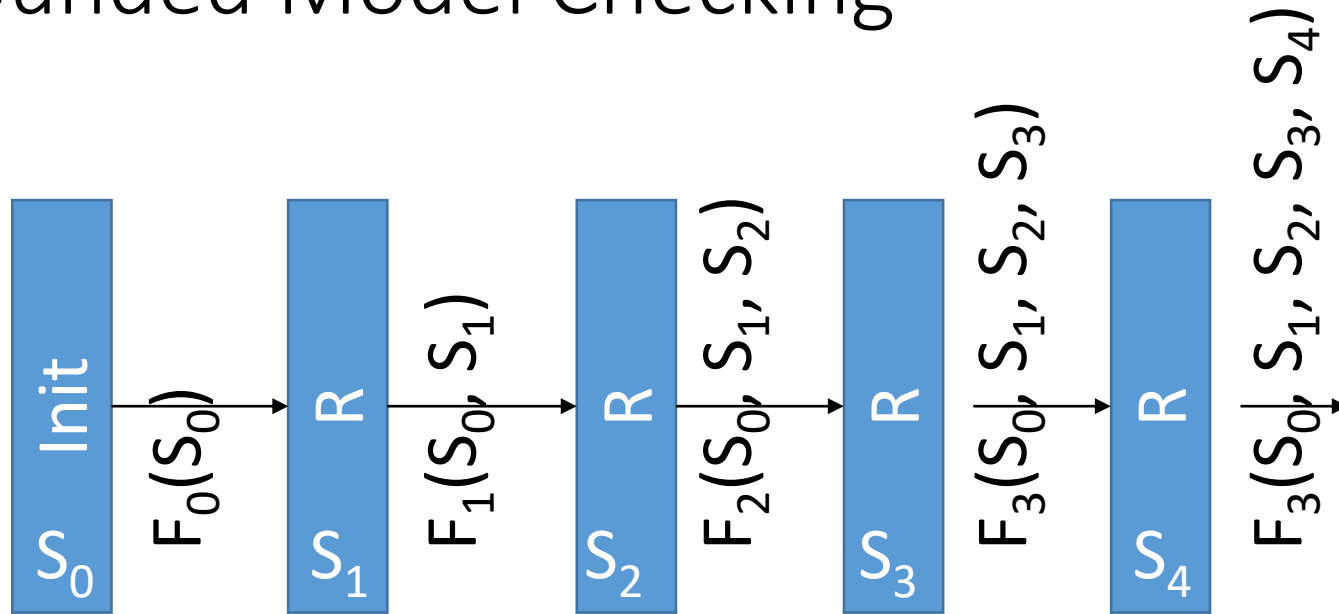
Bounded Model Checking

- We are given a transition system $M = (X, Init, R)$
- And an invariant $\phi(X)$
- Want to prove all reachable states of M satisfy ϕ

Approach

- Check if all initial states satisfy ϕ :
 - Is $Init(S_0) \wedge \neg\phi(S_0)$ UNSAT?
- Check if all states reachable in one step satisfy ϕ :
 - Is $Init(S_0) \wedge R(S_0, S_1) \wedge \neg\phi(S_1)$ UNSAT?
- ...

Bounded Model Checking



- $F_0 \equiv \text{Init}(S_0) \wedge \neg\phi(S_0)$
- $F_1 \equiv \text{Init}(S_0) \wedge R(S_0, S_1) \wedge \neg\phi(S_1)$
- $F_2 \equiv \text{Init}(S_0) \wedge R(S_0, S_1) \wedge R(S_1, S_2) \wedge \neg\phi(S_2)$
- $F_3 \equiv \text{Init}(S_0) \wedge R(S_0, S_1) \wedge R(S_1, S_2) \wedge R(S_2, S_3) \wedge \neg\phi(S_3)$
- $F_4 \equiv \text{Init}(S_0) \wedge R(S_0, S_1) \wedge R(S_1, S_2) \wedge R(S_2, S_3) \wedge R(S_3, S_4) \wedge \neg\phi(S_4)$

Bounded Model Checking

$F_0(s_0) = \text{Init}(s_0)$

$i = 0$

do

if $\neg\phi(s_i) \wedge F_i \neq \perp$:

return UNSAFE

$F_{i+1} = F_i \wedge R(s_{i-1}, s_i)$

$i = i + 1$

while $i \leq B$

return UNKNOWN

What is going on?

- How many variables?
- Can this be complete?
 - How will we know?

BMC Bounds for Completeness

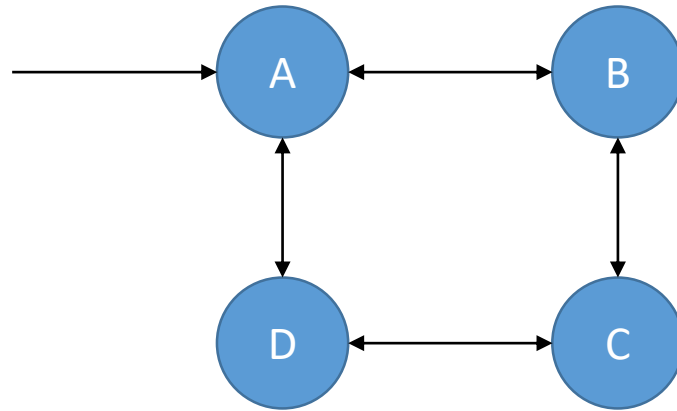
Diameter d of a transition system

- Longest “shortest path” from an initial state to some reachable state

Recurrence diameter rd of a transition system

- Longest loop-free path

Ex: Diameter/Recurrence Diameter



- What is the diameter?
- What is the recurrence diameter?

BMC Pros and Cons

Pros

- Very effective for bug finding
- Can scale to fairly large designs
- Can start from arbitrarily deep states (how?)

Cons

- Not complete
- What about “deep” bugs?

BMC Example

- Gray code conversion:
<https://github.com/pramodsu/satsmt2018/blob/master/bmc-ex1.py>
- Task: Add a check for whether the recurrence diameter has been reached
- Do this by adding a constraints which says that all the state variables S_0, S_1, \dots, S_N must be distinct
 - Use `z3.Distinct(a, b, c, d)` to say that a, b, c, d are pairwise distinct

Induction (First Attempt)

- Given a transition system
 - $M = (X, Init, R)$
 - Invariant $\phi(X)$
- Obvious proof methodology
- We will check this by verifying whether:
 - [Base case] $Init(X) \Rightarrow \phi(X)$ must be valid
 - [Step case] $\phi(X) \wedge R(X, X') \Rightarrow \phi(X')$
- More scalable than even BMC (why?)

Induction: Example 0

- <https://github.com/pramodsu/satsmt2018/blob/master/induction-ex0.py>

Induction: Example 1

- <https://github.com/pramodsu/satsmt2018/blob/master/induction-ex1.py>

Induction Revisited

Need to “strengthen” the inductive hypothesis

Do this by finding a inductive invariant Ψ such that

- $Init(X) \Rightarrow \Psi(X)$
- $\Psi(X) \wedge R(X, X') \Rightarrow \Psi(X')$
- $\Psi(X) \Rightarrow \phi(X)$

Are all valid

We can also do k-induction (what is this?)

Induction: Example 2

- <https://github.com/pramodsu/satsmt2018/blob/master/induction-ex2.py>
- Task: add strengthening invariant. Change the definition of ψ in the code to make the proof work.

Induction using UCLID5


- lsearch.ucl (demo)
- findmin.ucl (exercise)
 - Add strengthening invariants to prove the properties stated in the file

1. Background: Modeling, Specification, Verification
2. Security Property Verification (Hyperproperties)
 - Why we need hyperproperties
3. Case Study: Enclave Platform Verification

Specifying Information Flow (1/3)

```
int secret[N];
int public[N];
int foo(int index) {
    int r = 0;
    if (index >= 0 && index < N)
        if (priv_level == sup_user)
            r = secret[index];
        else
            r = public[index];
    return r;
}
```

Specification: only
super user must
access secret array.



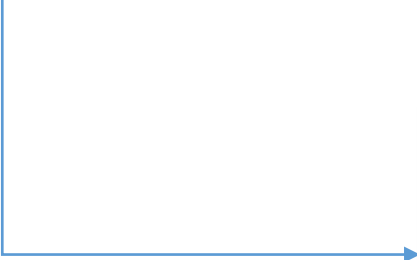
```
mov ebx, secret
mov edi, index
mov eax, [ebx+4*edi]
mov r, eax
```

Idea: instrument each load to ensure that secret array is not accessed

Specifying Information Flow(2/3)

```
define valid (addr) =  
  (addr >= secret &&  
   addr < secret + N)  
  ⇒ priv_level == sup_user)
```

Specification: only
super user must
access secret array.



```
mov ebx, secret  
mov edi, index  
assert valid(ebx+4*edi);  
mov eax, [ebx+4*edi]  
mov r, eax
```

Idea: instrument each load to ensure that secret array is not accessed

Specifying Information Flow(3/3)

```
int secret[N];
int public[N];
int t = 0;
int foo(int index) {
    int r = 0;
    if (index >= 0 && index <= N)
        if (priv_level == sup_user)
            r=t=secret[index];
        else
            r=public[index];
    return r;
}
```

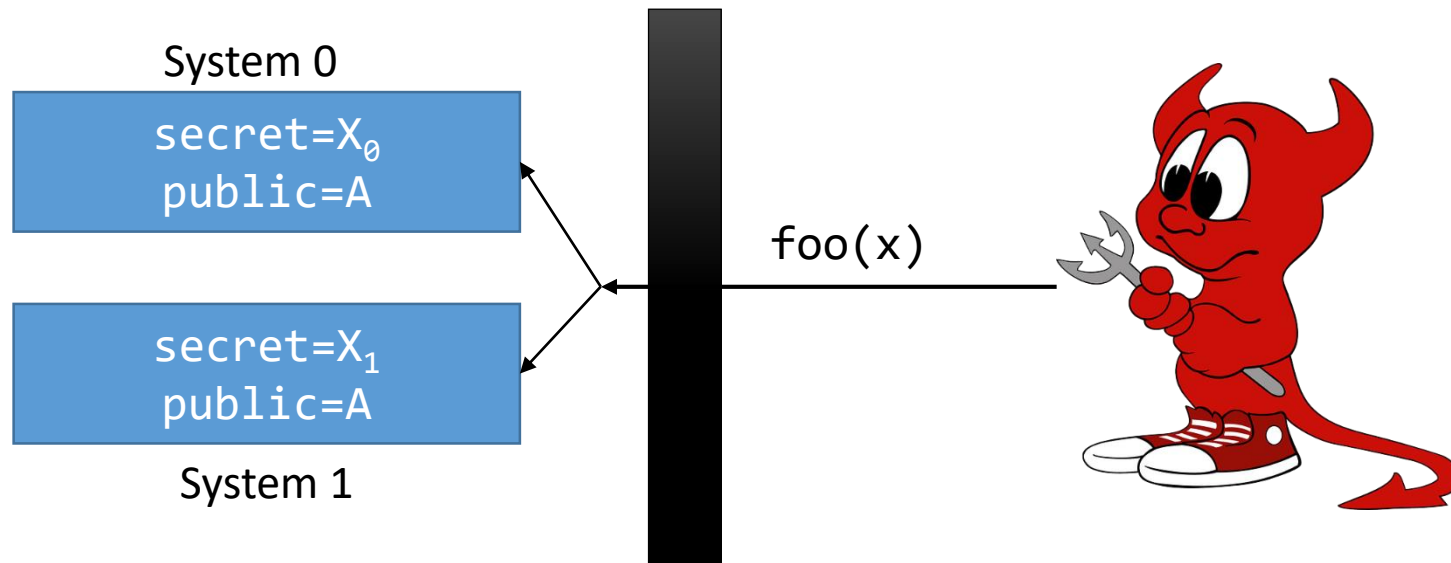
What is the bug?

```
mov ebx, secret
mov edi, index
assert valid(ebx+4*edi);
mov eax, [ebx+4*edi]
mov r, eax
```

Note our property is still satisfied!

1. Background: Modeling, Specification, Verification
2. Security Property Verification (Hyperproperties)
 - Why we need hyperproperties
 - Security as a distinguishability game
3. Case Study: Enclave Platform Verification

Distinguishability Games



- Pose game between an attacker and defender
- Attacker wins game: system is **not secure**
- Defender wins the game: system **is secure**
- Note defender must reason about **all attacker actions**

Example of Distinguishability Game

Game Initialization

- Attacker chooses $secret_0, secret_1, public$
- Defender chooses $b \in \{0,1\}$
- System initial state is defined by $secret_b, public$

Game Execution

- Attacker makes calls to $foo(x)$, and observes result r
 - Arbitrary values of x , in `user_mode`
- Interspersed with calls to $foo(x)$ in `supervisor mode`
 - Arbitrary values of x , but attacker can't observe the result

Finalization

Attacker wins if they guess b correctly, defender wins otherwise

Can the attacker win the game?

```
int secret[N];
int public[N];
int foo(int index) {
    int r = 0;
    if (index >= 0 && index < N)
        if (priv_level == sup_user)
            r = secret[index];
        else
            r = public[index];
    return r;
}
```

Initialization

- $\text{secret}_0 = \{1, 2, 3, 4\}$
- $\text{secret}_1 = \{5, 6, 7, 8\}$
- $\text{public} = \{10, 11, 12, 13\}$
- say $b = 1$

Game

- $\text{priv_level} = \text{sup_user}$, $\text{foo}(1)$, $\text{obs} = \emptyset$
- $\text{priv_level} = \text{user}$, $\text{foo}(1)$, $\text{obs} = 11$
- $\text{priv_level} = \text{sup_user}$, $\text{foo}(2)$, $\text{obs} = \emptyset$
- ...

This can go on forever, nothing will ever reveal secret to the adversary

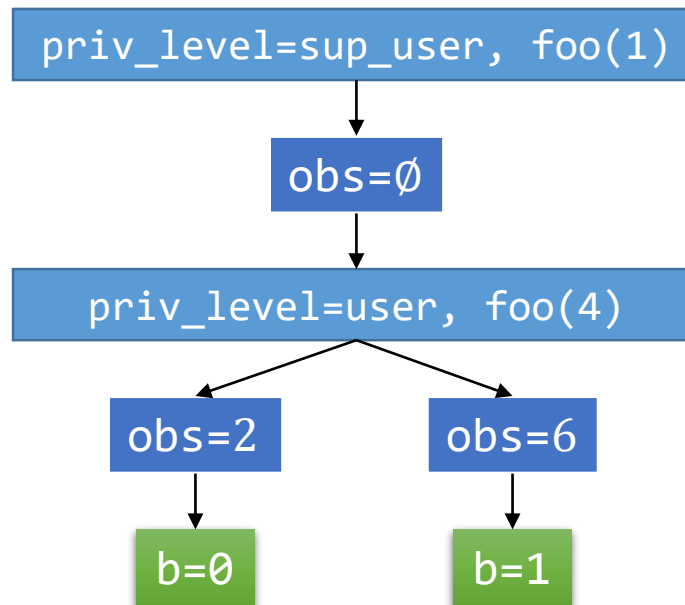
Attacker can win *this* game!

```
int secret[N];
int public[N];
int t = 0;
int foo(int index) {
    int r = 0;
    if (index >= 0 && index <= N)
        if (priv_level == sup_user)
            r=t=secret[index];
        else
            r=public[index];
    return r;
}
```

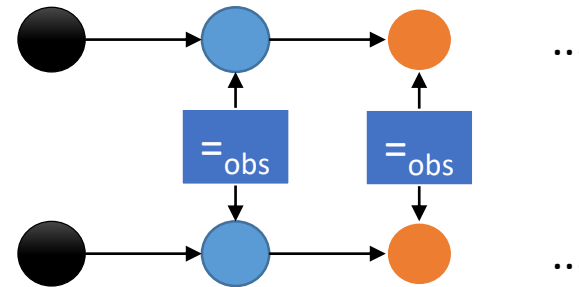
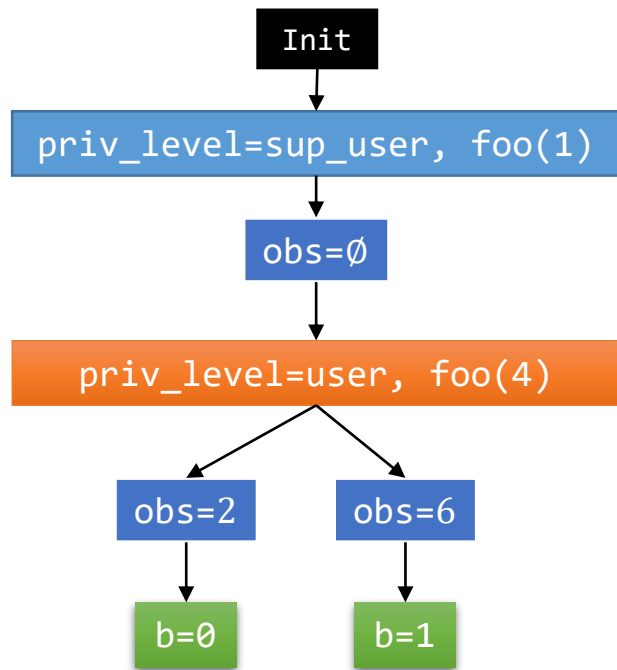
Attacker can easily win this game!

Initialization

- $\text{secret}_0 = \{1,2,3,4\}$
- $\text{secret}_1 = \{5,6,7,8\}$
- $\text{public} = \{10,11,12,13\}$

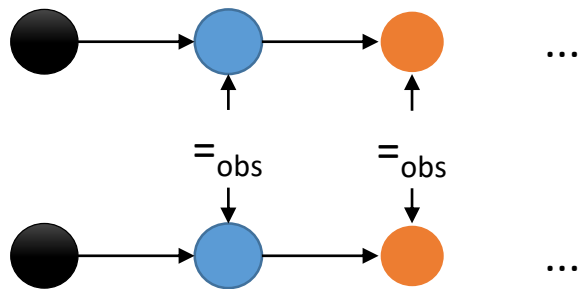


This is a 2 trace property



1. Initial states have different secrets but same public values
2. Attacker takes arbitrary actions at each step
3. Must prove observations are the same at every step of execution

This is a 2 trace property



1. Initial states have different secrets but same public values
2. Attacker takes arbitrary actions at each step
3. Must prove observations are the same at every step of execution

Recall our definition of a trace as a sequence of states: $\pi = s_0.s_1.s_2 \dots$

$\forall \pi_1 \pi_2:$

$$\pi_1[0].public = \pi_2[0].public \Rightarrow$$

$$\forall i. obs(\pi_1[i]) = obs(\pi_2[i])$$

1. Background: Modeling, Specification, Verification
2. Security Property Verification (Hyperproperties)
 - Why we need hyperproperties
 - Security as a distinguishability game
 - Introduction to hyperproperties
3. Case Study: Enclave Platform Verification

Properties vs. Hyperproperties

- Safety and liveness properties are sets of traces
- Given trace π can always tell whether it satisfies the property or not

$$\forall \pi \in \text{Traces}(M): (\forall i: \text{sgn}(\pi[0].x) = \text{sgn}(\pi[i].x))$$

$$X = \{ x, y \}$$

$$\text{Init}(X) = \text{true}$$

$$R(X, X') = y' > 0 \wedge x' = \text{ite}(x \geq 0, x+y', x-y')$$

The above system satisfies the property

Properties vs. Hyperproperties

- Safety and liveness properties are sets of traces
- Given trace π can always tell whether it satisfies the property or not

$$\forall \pi \in \text{Traces}(M): (\forall i: \text{sgn}(\pi[0].x) = \text{sgn}(\pi[i].x))$$

$$X = \{ x, y \}$$

$$\text{Init}(X) = \text{true}$$

$$R(X, X') = y' > 0 \wedge x' = \text{ite}(x \geq 0, x+2y', x-3y')$$

The above system also satisfies the property

Properties vs. Hyperproperties

- Safety and liveness properties are sets of traces
- Given trace π can always tell whether it satisfies the property or not

$$\forall \pi \in \text{Traces}(M): (\forall i: \text{sgn}(\pi[0].x) = \text{sgn}(\pi[i].x))$$

$$\begin{aligned} X &= \{ x \} \\ \text{Init}(X) &= \text{true} \\ R(X, X') &= x' = \text{ite}(x \geq 0, x+x, x-1) \end{aligned}$$

Yet another system that satisfies the property

Properties vs. Hyperproperties

- Safety and liveness properties are sets of traces
- Given trace π can always tell whether it satisfies the property or not

$$\forall \pi \in \text{Traces}(M): (\forall i: \text{sgn}(\pi[0].x) = \text{sgn}(\pi[i].x))$$

- In fact there are infinitely many systems that satisfy the property
- All the traces of all these systems are subsets of the set of traces that satisfy the formula above

Hyperproperties

$\forall \pi_1 \pi_2:$

$$\pi_1[0].public = \pi_2[0].public \Rightarrow \\ \forall i. obs(\pi_1[i]) = obs(\pi_2[i])$$

1. We **cannot** determine whether a single trace satisfies the above or not
2. The above is also **not a set of traces**
3. It is a **set of set of traces!**

Hypersafety Property

- Safety property was set of traces
- Hyperproperties: set of sets of traces
- If T, T' are sets of traces, then
 - T' is a prefix of T if for every trace $\pi \in T$,
 - there exists a trace $\pi' \in T'$ such that π' is a prefix of π
- A hypersafety property is a set of set of traces s.t.
 - For every set of traces T that don't satisfy the property, there exists a set of traces M , a prefix of T , such that any extension of M also doesn't satisfy the property

K-Safety

$\forall \pi_1 \pi_2:$

$$\pi_1[0].public = \pi_2[0].public \Rightarrow \\ \forall i. obs(\pi_1[i]) = obs(\pi_2[i])$$

1. Hyperproperties over k-tuples of traces are called k-safety properties
2. See references for a technical definition
3. The above is a 2-safety property

References

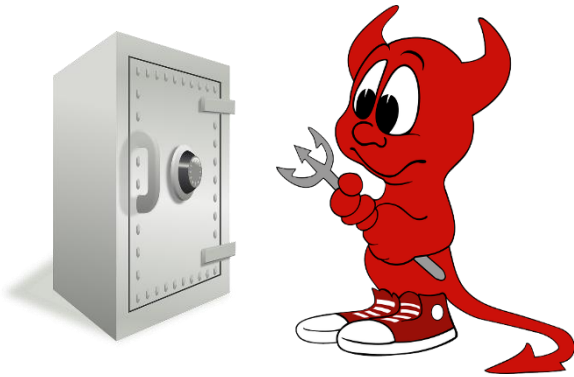
[Hyperproperties by Clarkson and Schneider](#)

First two sections are easy to read

Related: Terauchi and Aiken, Secure Information Flow as a Safety Problem, SAS 2005

1. Background: Modeling, Specification, Verification
2. Security Property Verification (Hyperproperties)
 - Why we need hyperproperties
 - Security as a distinguishability game
 - Introduction to hyperproperties
 - Hyperproperties for security specification
3. Case Study: Enclave Platform Verification

Security Specification: CIA Triad



Confidentiality, Integrity and Availability

[Anderson '72, Saltzer and Schroeder '75]

Secure Information Flow Properties

Adversary

- (Confidentiality)
- (Integrity)
- (Availability)

Can't see what's inside locker

Can't damage items inside locker

Can't destroy the locker wholesale

Confidentiality and Integrity are 2-safety properties

Secure Information Flow

Let us consider a system with two types of users

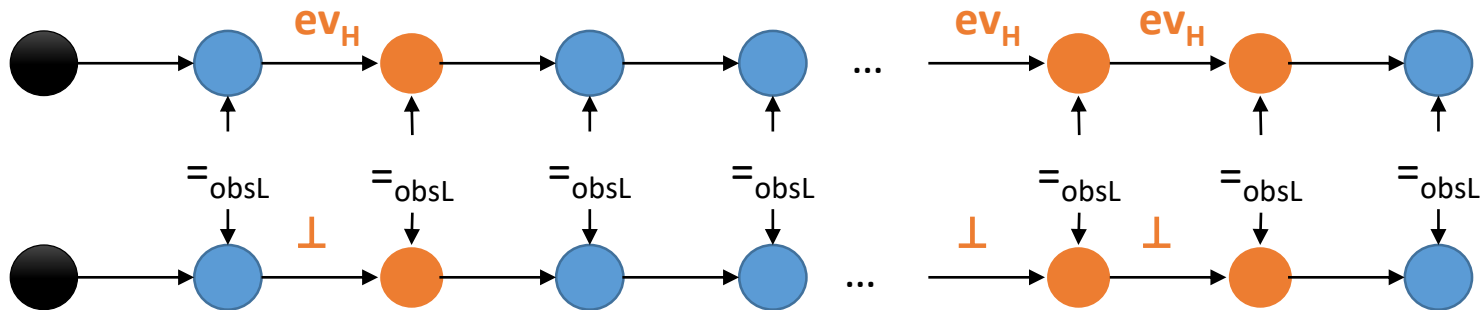
- Low security users (say soldiers aka jawans)
- High security users (say generals)
- Both types of users issue commands to and receive responses from (i.e. make observations) the system

We want to express the idea that high security information shouldn't flow to low security users

Turns out there are many ways of doing this

Noninterference

Commands issued by **high** users must be removeable without affecting the observations of **low** users



$$\begin{aligned}
 &\forall \pi_1: \exists \pi_2: \\
 &\forall i. ev_{lo}(\pi_1[i]) = ev_{lo}(\pi_2[i]) \wedge \\
 &\forall i. ev_{hi}(\pi_2[i]) = \perp \wedge \\
 &\forall i. obs(\pi_1[i]) = obs(\pi_2[i])
 \end{aligned}$$

[Goguen and Meseguer, 1982]

Example of Noninterference

Commands issued by **high** users must be removeable without affecting the observations of **low** users

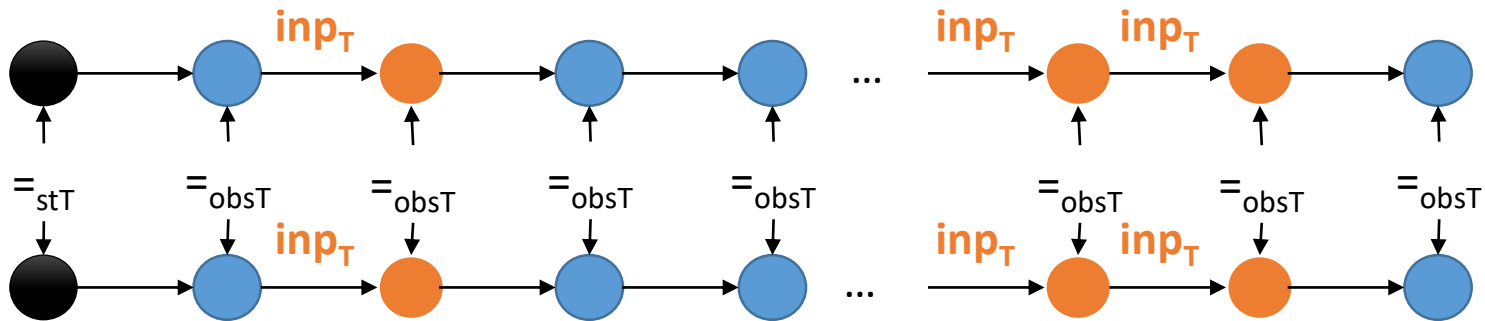
- Two operating systems running within a VM
- One is trusted, the other is not
- Removing actions of the untrusted VM shouldn't affect the observable behavior of the trusted VM

Question: what is included in observations?

- Register values? Cache status? # of cycles?

Observational Determinism

System should appear deterministic to **trusted** users



$$\begin{aligned}
 &\forall \pi_1 \pi_2: \\
 &st_T(\pi_1[0]) = st_T(\pi_2[0]) \Rightarrow \\
 &\forall i. inp_T(\pi_1[i]) = inp_T(\pi_2[i]) \Rightarrow \\
 &\forall i. obs_T(\pi_1[i]) = obs_T(\pi_2[i])
 \end{aligned}$$

[Zdancewic and Myers, 2003]

Example: Observational Determinism

System should appear deterministic to **trusted** users

- OS kernels satisfy observational determinism
- They don't satisfy noninterference because they receive input from untrusted processes

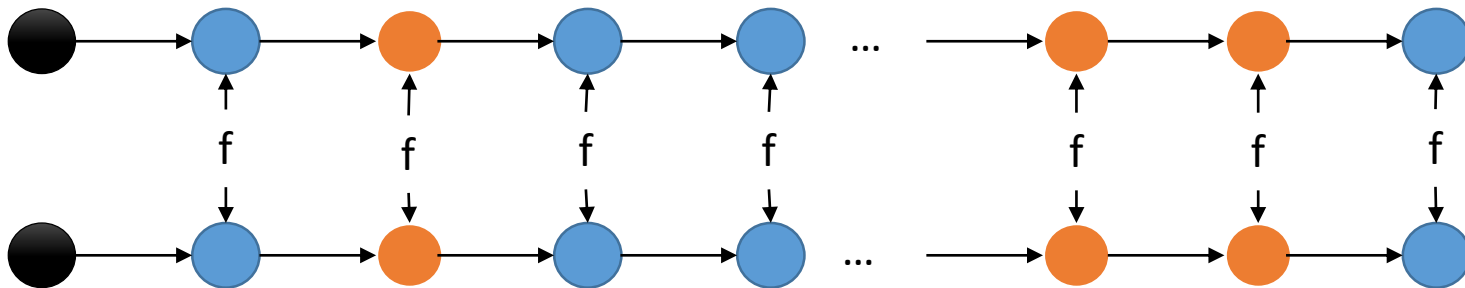
Question: what is included in observations?

- Register values? Cache status? # of cycles?

Deniability

For every observation that **low** users make, there must be at least two **high** user states consistent with it

$$f(s, s') \doteq obs(s) = obs(s') \wedge st_{hi}(s) \neq st_{hi}(s')$$

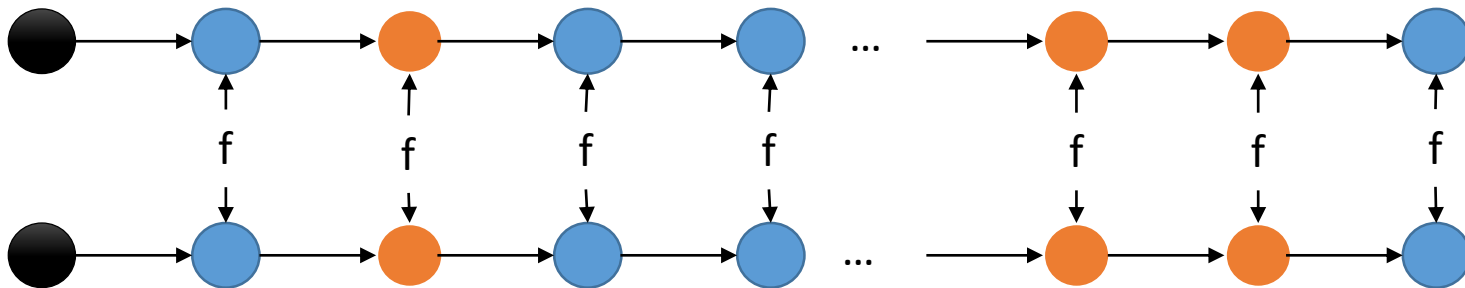


$$\begin{aligned} &\forall \pi_1: \exists \pi_2: \\ &\forall i. obs_{lo}(\pi_1[i]) = obs_{lo}(\pi_2[i]) \wedge \\ &\exists i. st_{hi}(\pi_1[i]) \neq st_{hi}(\pi_2[i]) \end{aligned}$$

Deniability

For every observation that **low** users make, there must be at least two **high** user states consistent with it

$$f(s, s') \doteq obs(s) = obs(s') \wedge st_{hi}(s) \neq st_{hi}(s')$$



$$\forall \pi_1: \exists \pi_2 \pi_3:$$

$$\forall i. obs_{lo}(\pi_1[i]) = obs_{lo}(\pi_2[i]) \wedge$$

$$\forall i. obs_{lo}(\pi_1[i]) = obs_{lo}(\pi_3[i]) \wedge$$

$$\exists i. st_{hi}(\pi_1[i]) \neq st_{hi}(\pi_2[i]) \wedge$$

$$\exists i. st_{hi}(\pi_1[i]) \neq st_{hi}(\pi_3[i])$$

Can generalize this
to m different high
states

Example: Deniability

For every observation that **low** users make, there must be at least ~~two~~ m **high** user states consistent with it

- RSA algorithm that operates on a **secret** key
- Adversary observes cache hit/miss status
- For security (indistinguishability), we want m keys correspond to any single trace of observations
- To be meaningful, m has to be big (say 2^{KeySize})

Another example: ORAM

There are more of these ...

- Opacity
- Non-deducibility
- Generalized non interference
- ...

Cottage industry of papers playing with variations of these ideas

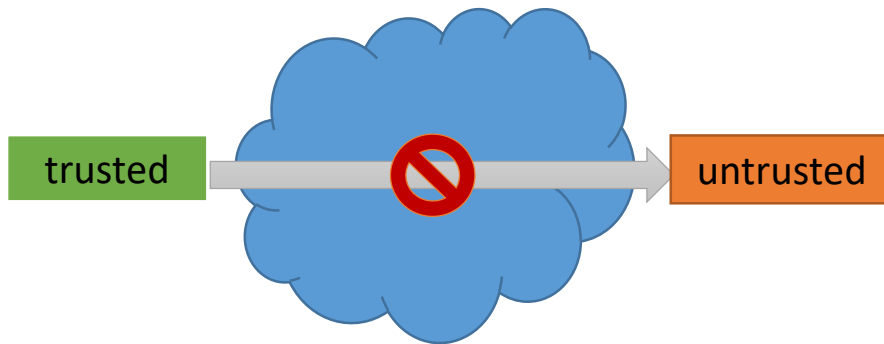
Look them up on Google Scholar if you're interested

Back to Secure Information Flow

Two types of information flow properties

- When we are dealing with confidential information, we don't want it flowing **from trusted** sources **to untrusted** sinks
- Integrity is its dual: information should not flow **from untrusted** sources **to trusted** sinks

Specifying Confidentiality



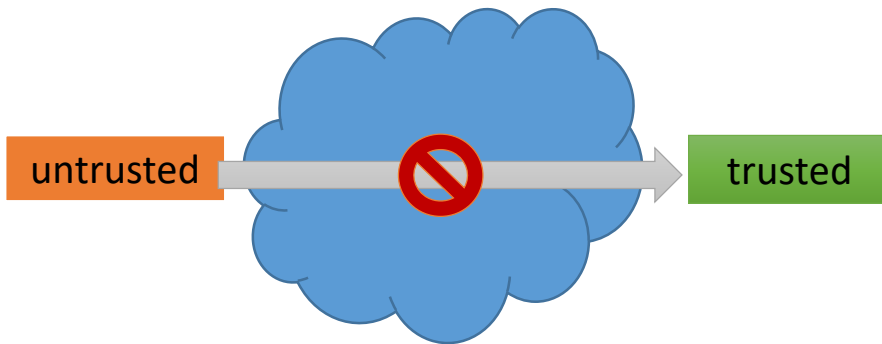
An instance of
observational
determinism

Confidentiality

- Trusted state contains secrets (e.g., cryptographic key)
- Untrusted state is adversary accessible (e.g., cache hit/miss result)
- Ex: crypto key should not flow to adversary cache observations

$$\begin{aligned} \forall \pi_1 \pi_2: st_{untrusted}(\pi_1[0]) &= st_{untrusted}(\pi_2[0]) \\ \forall i: inp_{untrusted}(\pi_1[i]) &= inp_{untrusted}(\pi_2[i]) \Rightarrow \\ \forall i: st_{untrusted}(\pi_1[i]) &= st_{untrusted}(\pi_2[i]) \end{aligned}$$

Specifying Integrity



Also an instance
of observational
determinism

Integrity

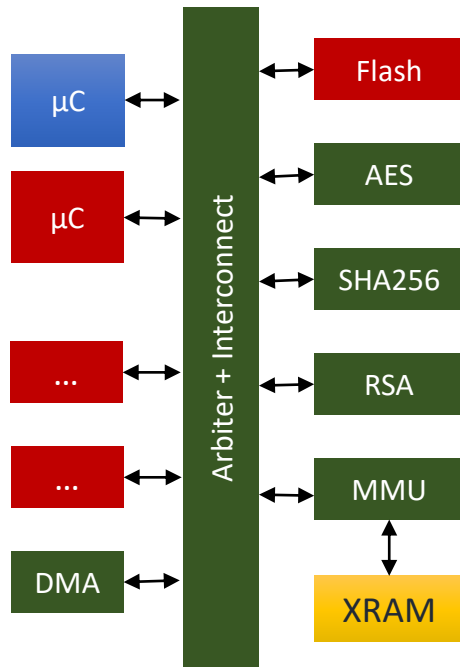
- Source is an untrusted location (e.g., adversary writeable register)
- Destination is a sensitive register (e.g., paging enable register)
- Ex: untrusted entity should not influence the value of sensitive register

$$\begin{aligned} \forall \pi_1 \pi_2: st_{trusted}(\pi_1[0]) &= st_{trusted}(\pi_2[0]) \\ \forall i: inp_{trusted}(\pi_1[i]) &= inp_{trusted}(\pi_2[i]) \Rightarrow \\ \forall i: st_{trusted}(\pi_1[i]) &= st_{trusted}(\pi_2[i]) \end{aligned}$$

Many variants are possible

- We will now see another variation of integrity that is slightly weaker than observational determinism

Integrity: SecureBoot Example (1/3)



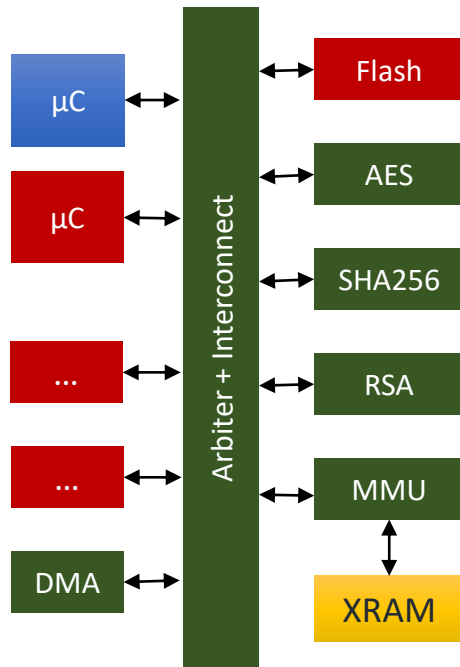
```

Img = LoadImage()
SetMMU(Img, ReadOnly)
Hash = SHA256(Img.Header)
if (!RSAVerify(Img.Sign, PKey, Hash))
    raise Failure;
for (i=0; i < Img.numBlks; i++):
    Copy(Img.Blk[i], Img.BlkLoc[i])
    SetMMU(Img.BlkLoc[i], ReadOnly)
    if (SHA256(Img.BlkLoc[i]) != Img.BlkHash[i]))
        raise Failure;
Boot (Img.BlkLoc[0])
    
```

μC Flash ...
 μC MMU ...
 μC SHA256 ...
 μC RSA ...
 μC ...
 μC ...
 μC DMA ...
 μC MMU ...
 μC SHA256 ...
 μC ...
 μC ...

Note: Untrusted components are shown in **red** while trusted components are shown in **green**.

Integrity: SecureBoot Example (2/3)



Design Driver SoC

What can go wrong?

- Binary in flash modified by malicious user
- Image is modified in memory after loading from flash, and verifying signature but before starting execution (**TOCTOU**)

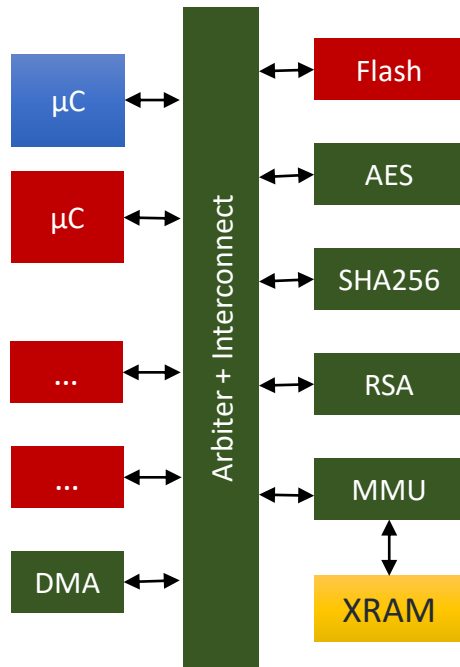
How exactly is image modified?

- Directly by adversary (say μ controller)
- Indirectly – adversary issues commands to AES to decrypt on its behalf and write the result to memory (**Confused Deputy**)

What else could go wrong?

- Weird aliasing and wraparound issues

Integrity: SecureBoot Example (3/3)



$\forall \pi_1 \pi_2:$

$\forall i: (\text{advOp}(\pi_2[i]) = \perp) \Rightarrow$

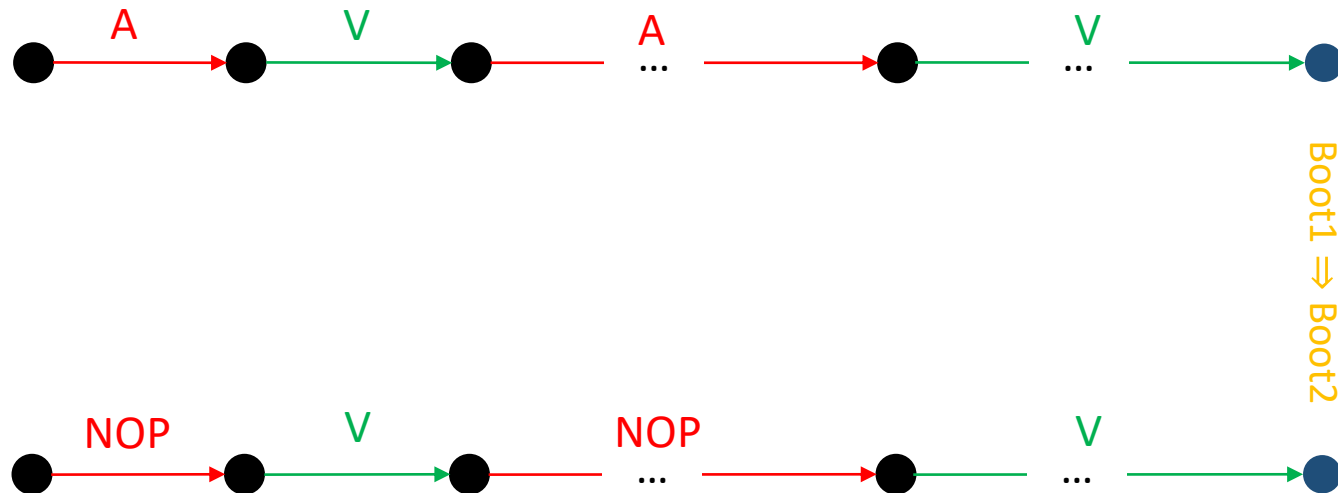
$\forall i: (\text{boot}(\pi_1[i]) \Rightarrow \text{boot}(\pi_2[i]))$

If we consider two executions, and in one the adversary actions are NOPs (i.e., no tampering), and if the execution with tampering boots, so must the execution without tampering

In other words, no way for adversary to take an execution that wouldn't have booted and turn it into something that does boot due to adversary interference

SecureBoot Property in Pictures

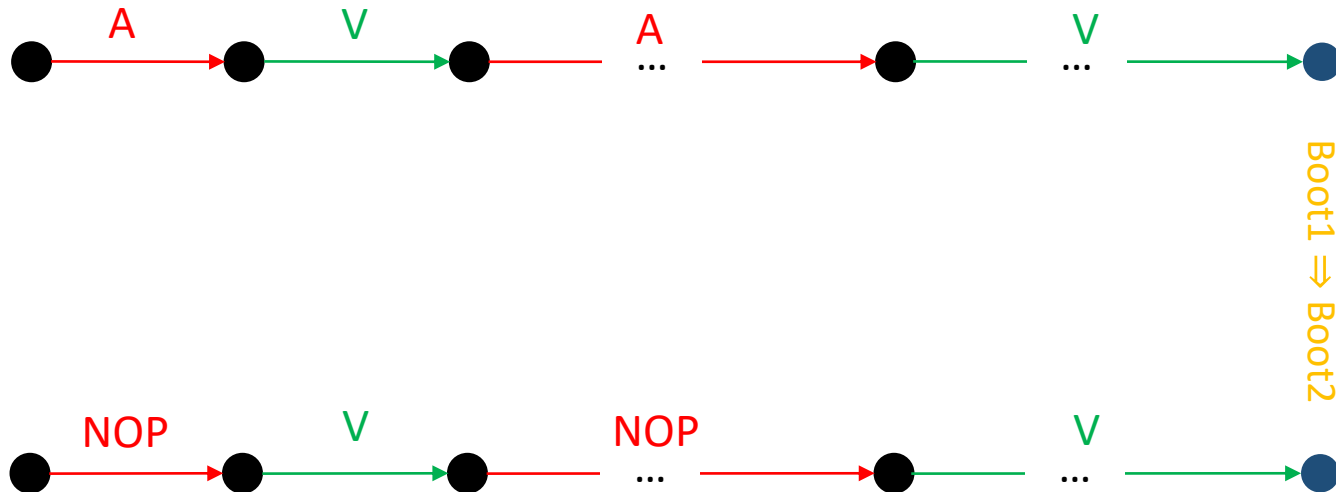
$$\forall \pi_1 \pi_2. \forall i. (\text{advOp}(\pi_2[i]) = \perp) \Rightarrow \forall i. (\text{boot}(\pi_1[i]) \Rightarrow \text{boot}(\pi_2[i]))$$



- V are the actions taken by the trusted bootloader/verifier
- A is the adversary who is attempting to tamper with the verification
- In π_2 all adversary actions are turned in NOPs (i.e. no tampering)
- We want to prove that if π_1 boots then so does π_2

Why \Rightarrow and not \Leftrightarrow

$$\forall \pi_1 \pi_2. \forall i. (\text{advOp}(\pi_2[i]) = \perp) \Rightarrow \forall i. (\text{boot}(\pi_1[i]) \Rightarrow \text{boot}(\pi_2[i]))$$



- Iff (\Leftrightarrow) means system boots regardless of adv actions
- If (\Rightarrow) allows us to **detect** interference and fail to boot

To summarize

- Hyperproperties are a very rich class of properties that can express many different kinds of security requirements
- We have seen a few of the important kinds
 - Observational determinism
 - Noninterference
 - Deniability
- Some minor variants of these may be useful too (e.g. the secureboot property)

Beyond Information Flow

- Determinism: $x_1 = x_2 \Rightarrow f(x_1) = f(x_2)$
- Injectivity: $x_1 \neq x_2 \Rightarrow f(x_1) \neq f(x_2)$
- Monotonicity: $x_1 < x_2 \Rightarrow f(x_1) < f(x_2)$
- Commutativity: $f(x_1, x_2) = f(x_2, x_1)$
- Associativity: $f(x_1, f(x_2, x_3)) = f(f(x_1, x_2), x_3)$

1. Background: Modeling, Specification, Verification
2. Security Property Verification (Hyperproperties)
 - Why we need hyperproperties
 - Security as a distinguishability game
 - Introduction to hyperproperties
 - Hyperproperties for security specification
 - Verification of hyperproperties (k-safety properties)
3. Case Study: Enclave Platform Verification

Verifying k-safety Properties

- The obvious construction is called self-composition
- Make k “copies” of the system
- Impose constraints relating the state in the “copies”
- Run the model checker on this composed system

Example Hyperproperty

$M = (X, \text{Init}, R)$ where

- $X \doteq \{n, \text{incr}, \text{sum}, i\}$
- $\text{Init}(X) \doteq \text{incr} > 0 \wedge n > 0 \wedge i = 0 \wedge \text{sum} = 0$
- $R(X, X') \doteq n' = n \wedge \text{incr}' = \text{incr} \wedge$
 $\text{sum}' = \text{ite}(i < n, \text{sum} + \text{incr}, \text{sum}) \wedge$
 $i' = \text{ite}(i < n, i + 1, i)$

$\forall \pi_1 \pi_2:$

$(\pi_1[0].\text{incr} < \pi_2[0].\text{incr} \wedge \pi_1[0].n = \pi_2[0].n) \Rightarrow$
 $\forall i: (\pi_1[i].\text{sum} < \pi_2[i].\text{sum})$

Example Hyperproperty

- Q: What does this property mean?
- A: For every pair of traces of the system, if the initial values of n are equal and $incr$ in one trace is less than the other, then at every subsequent step, sum in the first trace is less than sum in the other trace

$$\begin{aligned} &\forall \pi_1 \pi_2: \\ &(\pi_1[0].incr < \pi_2[0].incr \wedge \pi_1[0].n = \pi_2[0].n) \Rightarrow \\ &\forall i: (\pi_1[i].sum < \pi_2[i].sum) \end{aligned}$$

Example Hyperproperty Verification

$M = (X, \text{Init}, R)$ where

- $X \doteq \{n, \text{incr}, \text{sum}, i\}$
- $\text{Init}(X) \doteq \text{incr} > 0 \wedge n > 0 \wedge i = 0 \wedge \text{sum} = 0$
- $R(X, X') \doteq n' = n \wedge \text{incr}' = \text{incr} \wedge$
 $\text{sum}' = \text{ite}(i < n, \text{sum} + \text{incr}, \text{sum}) \wedge$
 $i' = \text{ite}(i < n, i + 1, i)$

$\forall \pi_1 \pi_2:$

$(\pi_1[0].\text{incr} < \pi_2[0].\text{incr} \wedge \pi_1[0].n = \pi_2[0].n) \Rightarrow$
 $\forall i: (\pi_1[i].\text{sum} < \pi_2[i].\text{sum})$

Example Hyperproperty Verification

$M = (X, \text{Init}, R)$ where

- $X \doteq \{n, \text{incr}, \text{sum}, i\}$
- $\text{Init}(X) \doteq \text{incr} > 0 \wedge n > 0 \wedge i = 0 \wedge \text{sum} = 0$
- $R(X, X') \doteq n' = n \wedge \text{incr}' = \text{incr} \wedge$
 $\text{sum}' = \text{ite}(i < n, \text{sum} + \text{incr}, \text{sum}) \wedge$
 $i' = \text{ite}(i < n, i + 1, i)$

$\forall \pi_1 \pi_2 i:$

$$(\pi_1[i].\text{incr} < \pi_2[i].\text{incr} \wedge \pi_1[i].n = \pi_2[i].n) \Rightarrow$$
$$(\pi_1[i].\text{sum} < \pi_2[i].\text{sum})$$

Why is this safe to do for the above system?

Self-composition Construction

Given $M = (X, Init, R)$

Define $M_{sc} = (X_1 \cup X_2, Init_{sc}, R_{sc})$ where

- $X_1 \doteq \{n_1, incr_1, sum_1, i_1\}, X_2 = \{n_2, incr_2, sum_2, i_2\}$
- $Init_{sc}(X_1, X_2) \doteq Init(X_1) \wedge Init(X_2)$
- $R_{sc}(X_1 \cup X_2, X'_1 \cup X'_2) \doteq R(X_1, X'_1) \wedge R(X_2, X'_2)$
- $\phi(X_1, X_2) \doteq (incr_1 < incr_2 \wedge n_1 = n_2) \Rightarrow (sum_1 < sum_2)$

We can now check the invariant ϕ on the system M_{sc}

Formal View of Self-Composition

- Given a system $M = (X, Init, R)$
- Suppose we have 2-safety invariant $\phi(X_1, X_2)$
- Define $M_{sc} = (X_1 \cup X_2, Init_{sc}, R_{sc})$ where
- $Init_{sc}(X_1, X_2) \doteq Init(X_1) \wedge Init(X_2)$
- $R_{sc}(X_1 \cup X_2, X'_1 \cup X'_2) \doteq R(X_1, X'_1) \wedge R(X_2, X'_2)$
- Check $\phi(X_1, X_2)$ on M_{sc} using BMC/Induction/PDR

Example Verification

<https://github.com/pramodsu/satsmt2018/blob/master/hyperproperty-ex1.ucl>

Automated verification of 2-safety

- Lots of recent interest in developing tools that fully automatically verify 2-safety properties
- Most are based on self-composition
- Tools work better when the **control flow path taken in the two traces must be identical**
- This particular form of self-composition is called a **product program**

Product Programs: Example

$$\begin{aligned} \forall \pi_1 \pi_2. \pi_1[0].tbl &= \pi_2[0].tbl \Rightarrow \\ \forall i. \pi_1[i].x &= \pi_2[i].x \Rightarrow \\ \forall i. \pi_1[i].IO_REG &= \pi_2[i].IO_REG \end{aligned}$$

```
#define N 3 // typo: 2 is the correct value

uint8_t tbl[] = { 1, 1 }; // &tbl = 0x100
uint8_t data = 3;         // &data=0x102
uint8_t IO_REG = 1;       // &IO_REG=0x200
uint8_t err_idx;

void foo(int x) {
L0:  if (x < 0 || x >= N) {
L1:    err_idx = x;
L2:    return;
L3:  }
L4:  IO_REG = tbl[x];
L5:}
```

stack	L0
PC	??
PathCond	true
	true
err_idx	??
	??
IO_reg	??
	??

Also called product programs

See [Barthe, Crespo, Kunz, FM 2011], [Subramanyan et al., DATE 2016]

Product Programs: Example

$$\forall \pi_1 \pi_2. \pi_1[0].tbl = \pi_2[0].tbl \Rightarrow$$
$$\forall i. \pi_1[i].x = \pi_2[i].x \Rightarrow$$
$$\forall i. \pi_1[i].IO_REG = \pi_2[i].IO_REG$$

```
#define N 3 // typo: 2 is the correct value

uint8_t tbl[] = { 1, 1 }; // &tbl = 0x100
uint8_t data = 3;         // &data=0x102
uint8_t IO_REG = 1;       // &IO_REG=0x200
uint8_t err_idx;

void foo(int x) {
L0:  if (x < 0 || x >= N) {
L1:    err_idx = x;
L2:    return;
L3:  }
L4:  IO_REG = tbl[x];
L5:}
```

stack	empty
PC	L0
PathCond	true
	true
err_idx	??
	??
IO_reg	??
	??

Product Programs: Example

$$\begin{aligned} \forall \pi_1 \pi_2. \pi_1[0].tbl = \pi_2[0].tbl &\Rightarrow \\ \forall i. \pi_1[i].x = \pi_2[i].x &\Rightarrow \\ \forall i. \pi_1[i].IO_REG = \pi_2[i].IO_REG \end{aligned}$$

```
#define N 3 // typo: 2 is the correct value

uint8_t tbl[] = { 1, 1 }; // &tbl = 0x100
uint8_t data = 3;         // &data=0x102
uint8_t IO_REG = 1;       // &IO_REG=0x200
uint8_t err_idx;

void foo(int x) {
L0:  if (x < 0 || x >= N) {
L1:    err_idx = x;
L2:    return;
L3:  }
L4:  IO_REG = tbl[x];
L5:}
```

stack	L4
PC	L1
PathCond	$x_1 < 0 \ \ x_1 \geq 3$
	$x_2 < 0 \ \ x_2 \geq 3$
err_idx	??
	??
IO_reg	??
	??

Product Programs: Example

$$\begin{aligned} \forall \pi_1 \pi_2. \pi_1[0].tbl = \pi_2[0].tbl &\Rightarrow \\ \forall i. \pi_1[i].x = \pi_2[i].x &\Rightarrow \\ \forall i. \pi_1[i].IO_REG = \pi_2[i].IO_REG \end{aligned}$$

```
#define N 3 // typo: 2 is the correct value

uint8_t tbl[] = { 1, 1 }; // &tbl = 0x100
uint8_t data = 3;         // &data=0x102
uint8_t IO_REG = 1;       // &IO_REG=0x200
uint8_t err_idx;

void foo(int x) {
L0:  if (x < 0 || x >= N) {
L1:    err_idx = x;
L2:    return;
L3:  }
L4:  IO_REG = tbl[x];
L5:}
```

stack	L4
PC	L2
PathCond	$x_1 < 0 \ \ x_1 \geq 3$
	$x_2 < 0 \ \ x_2 \geq 3$
err_idx	x_1
	x_2
IO_reg	??
	??

Product Programs: Example

$$\forall \pi_1 \pi_2. \pi_1[0].tbl = \pi_2[0].tbl \Rightarrow$$

$$\forall i. \pi_1[i].x = \pi_2[i].x \Rightarrow$$

$$\forall i. \pi_1[i].IO_REG = \pi_2[i].IO_REG$$

```
#define N 3 // typo: 2 is the correct value

uint8_t tbl[] = { 1, 1 }; // &tbl = 0x100
uint8_t data = 3;         // &data=0x102
uint8_t IO_REG = 1;       // &IO_REG=0x200
uint8_t err_idx;

void foo(int x) {
L0:  if (x < 0 || x >= N) {
L1:    err_idx = x;
L2:    return;
L3:  }
L4:  IO_REG = tbl[x];
L5:}
```

stack	empty
PC	L4
PathCond	$0 \leq x_1 < 3$
	$0 \leq x_2 < 3$
err_idx	??
	??
IO_reg	??
	??

Product Programs: Example

$$\forall \pi_1 \pi_2. \pi_1[0].tbl = \pi_2[0].tbl \Rightarrow$$

$$\forall i. \pi_1[i].x = \pi_2[i].x \Rightarrow$$

$$\forall i. \pi_1[i].IO_REG = \pi_2[i].IO_REG$$

```
#define N 3 // typo: 2 is the correct value

uint8_t tbl[] = { 1, 1 }; // &tbl = 0x100
uint8_t data = 3;         // &data=0x102
uint8_t IO_REG = 1;       // &IO_REG=0x200
uint8_t err_idx;

void foo(int x) {
L0:  if (x < 0 || x >= N) {
L1:    err_idx = x;
L2:    return;
L3:  }
L4:  IO_REG = tbl[x];
L5:}
```

stack	empty
PC	L5
PathCond	$0 \leq x_1 < 3$
	$0 \leq x_2 < 3$
err_idx	??
	??
IO_reg	data ₁
	data ₂

(data₁ != data₂)

Hyperproperties: Queue Example

<https://github.com/pramodsu/satsmt2018/blob/master/queue.ucl>

$\forall \pi_1 \pi_2:$

$\forall i: (\pi_1[i].op = \pi_2[i].op \wedge \pi_1[i].user = \pi_2[i].user)$
 $\Rightarrow (\pi_1[i].user = u1 \Rightarrow \pi_1[i].datain = \pi_2[i].datain)$
 $\Rightarrow (\pi_1[i].user = u1 \Rightarrow \pi_1[i].dataout = \pi_2[i].dataout)$

Q: What does the above property mean?

A: If we consider two traces with the same sequence of pushes/pops by the same users, **and** $u1$ pushes the same data in both traces, then $u1$ must pop the same data

Hyperproperties: Queue Example

<https://github.com/pramodsu/satsmt2018/blob/master/queue.ucl>

Task: add strengthening invariants that relate the values in the two traces

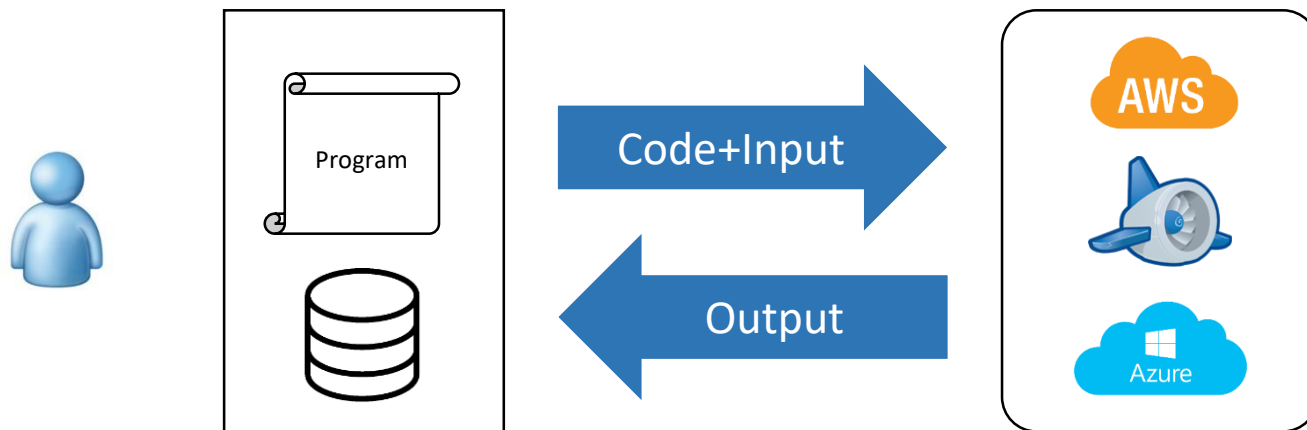
- Both queues have the same user array
- Both data arrays are same for items which belong to u1

Summarizing Hyperproperty Verification

- Hyperproperties are an extremely important class of properties for the specification of security requirements
- Lot of recent research has been focusing on the specification and verification of these properties for various computer systems
- In practice, induction is still the only scalable approach that works for largish systems

1. Background: Modeling, Specification, Verification
2. Security Property Verification (Hyperproperties)
3. Case Study: Enclave Platform Verification
 - Introduction to Enclave Platforms

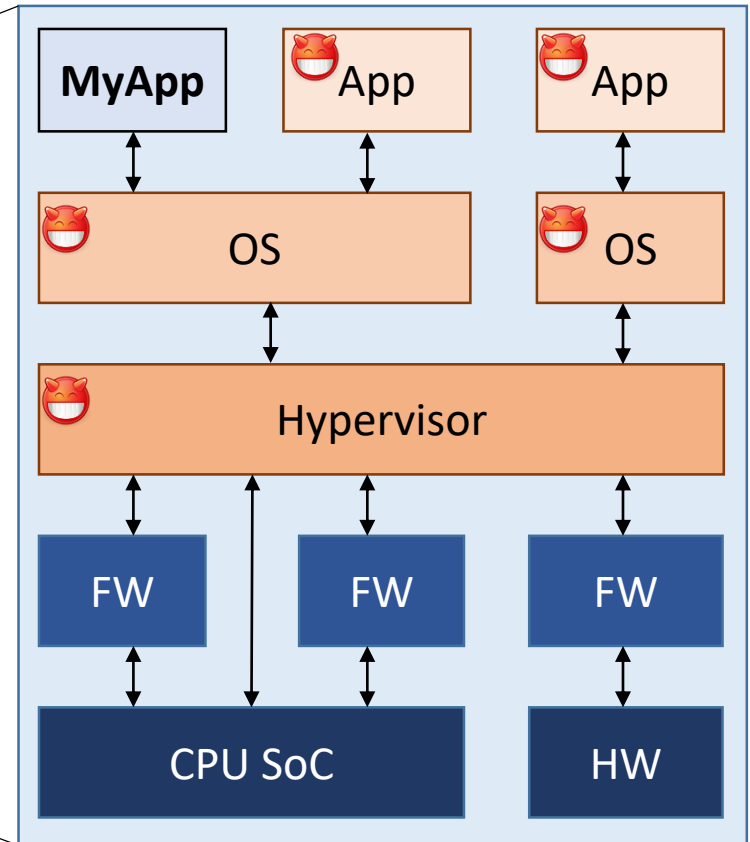
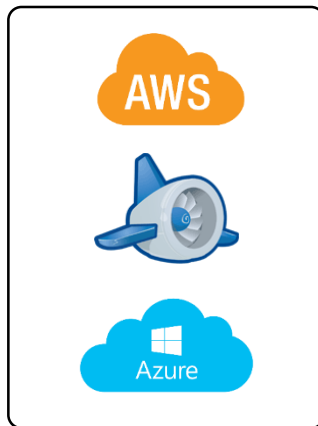
Enclave Platforms 101 (1/3)



What guarantees do we want for **secure remote execution**?

Enclave Platforms 101 (2/3)

Typical public cloud implies untrusted OS, hypervisor and co-located apps



How to ensure security in this scenario?

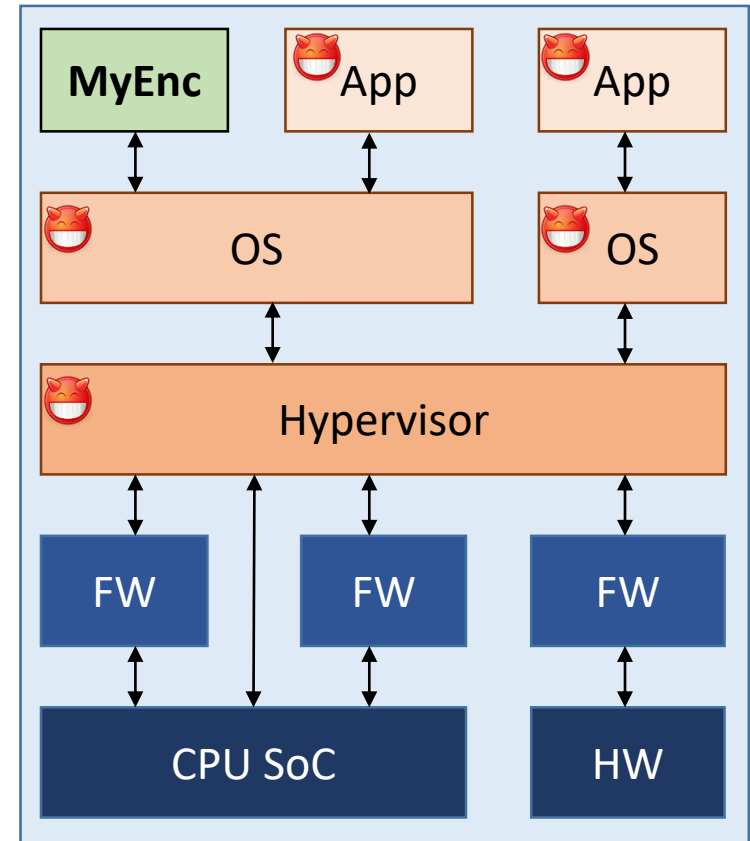
Enclave Platforms 101 (3/3)

Enclave Platforms like Intel SGX and MIT Sanctum provide ISA-level primitives for “secure” remote execution

Enclave consists of:

- Protected mem region for code/data
- Even OS/VM can't access enclave mem
- HW “measurement” operator

Research Question: How do enclave primitives translate to secure remote execution?



How could platforms betray us?

- Access control bugs: untrusted code is able to read/write from enclave memory
- Side channels: caches, page tables, speculation, etc. might leak information about enclave memory
- Measurements: different enclaves with the same measurement
- “Bad” operations like cloning of enclaves

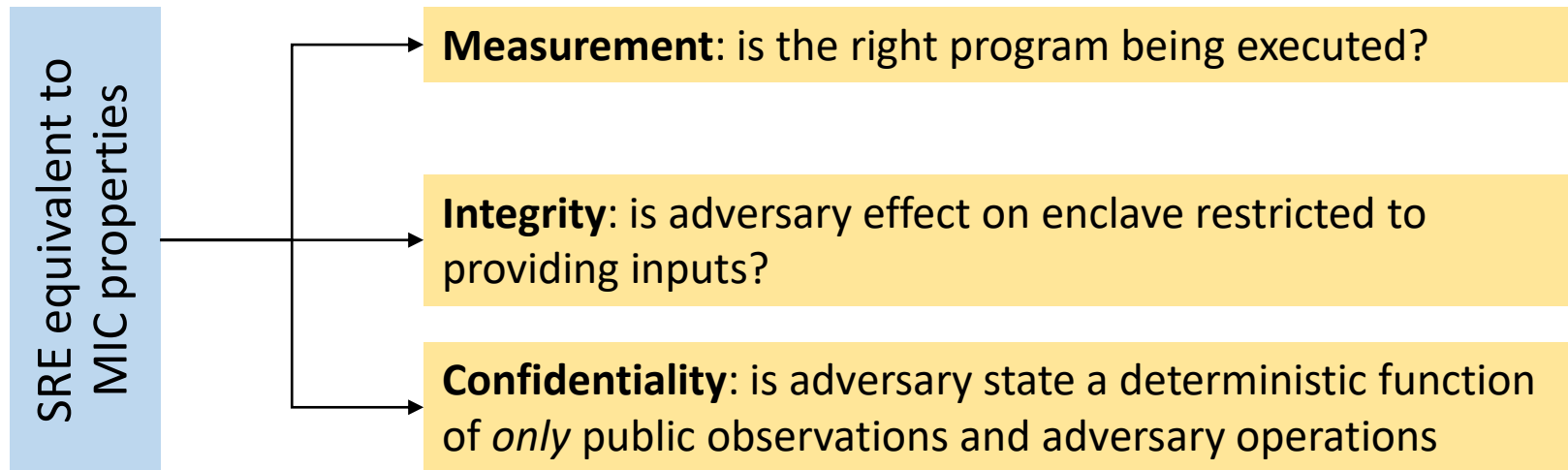
1. Background: Modeling, Specification, Verification
2. Security Property Verification (Hyperproperties)
3. Case Study: Enclave Platform Verification
 - Introduction to Enclave Platforms
 - Defining Security for Enclave Platforms

Secure Remote Execution of Enclaves

(Definition) Secure Remote Execution (SRE)

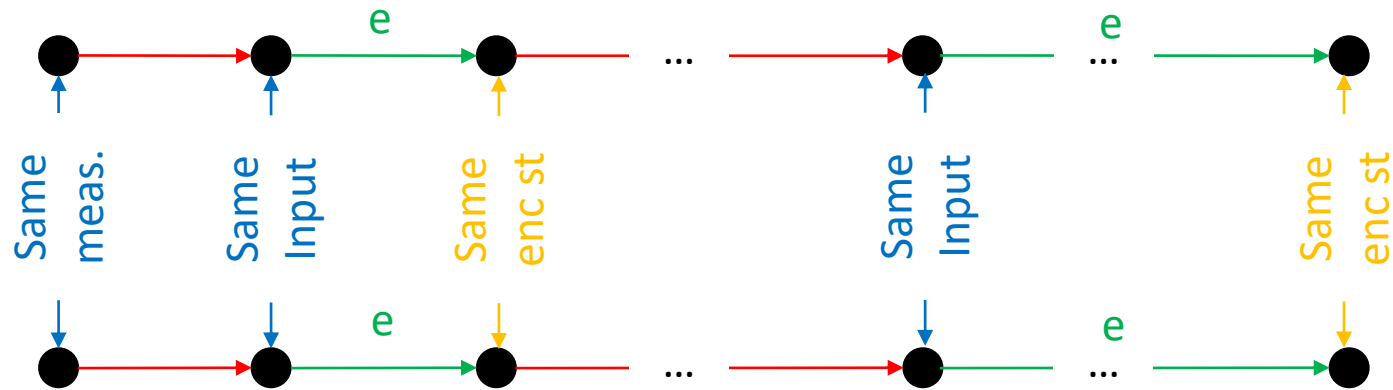
Remote server's execution of enclave program e must be identical to trusted local execution modulo inputs provided to the enclave.

(Theorem) SRE Decomposition



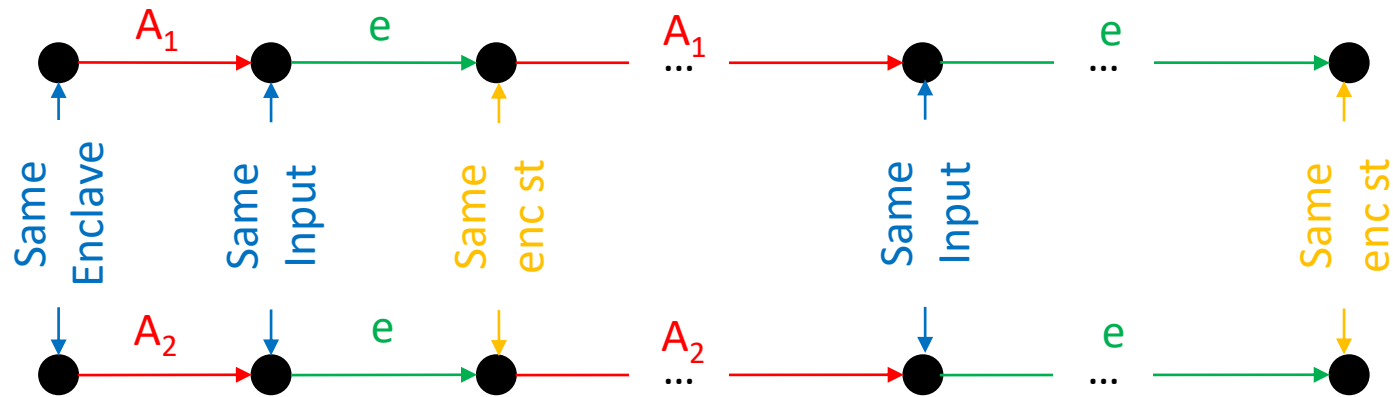
[Subramanyan et al., CCS'17]

Enclave Measurement Property



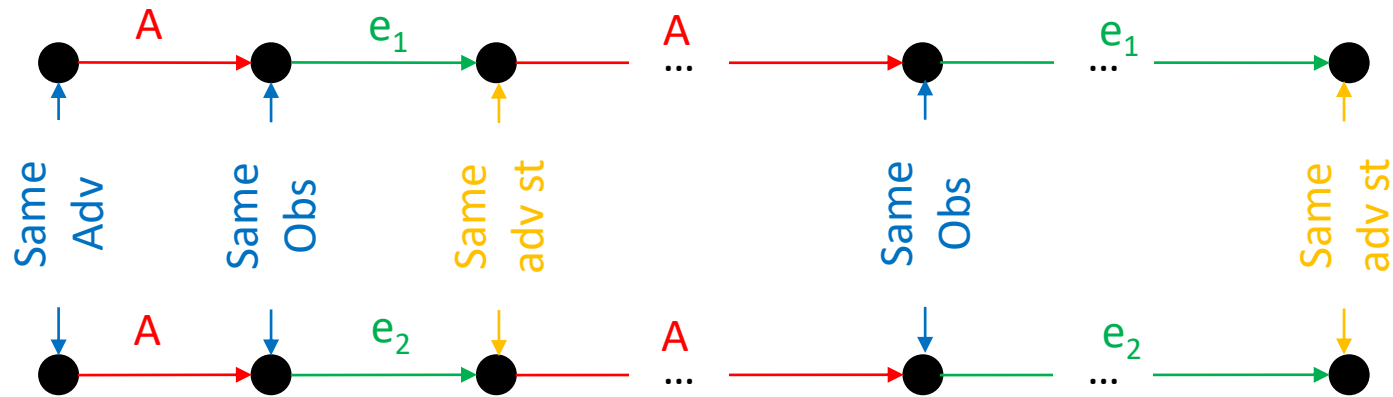
If we start two enclaves with the same measurement and, and if at each step the enclave executes, the inputs to the enclave are identical, then these two enclaves must have the same state and outputs at every step

Enclave Integrity Property



If we start two identical enclaves, and if at each step the enclave executes, the inputs to the enclave are identical, then regardless of adversary actions, the enclave computation must also be identical

Enclave Confidentiality Property



If we start two different enclaves, and if at each step the enclave executes, the outputs of the enclave are identical, and the adversary actions are the same, then regardless of adversary actions, the adversary state must also be identical

The Trusted Abstract Platform

- Registers, program counter
- Virtual to physical mappings
- Enclave metadata
- Memory and associated cache

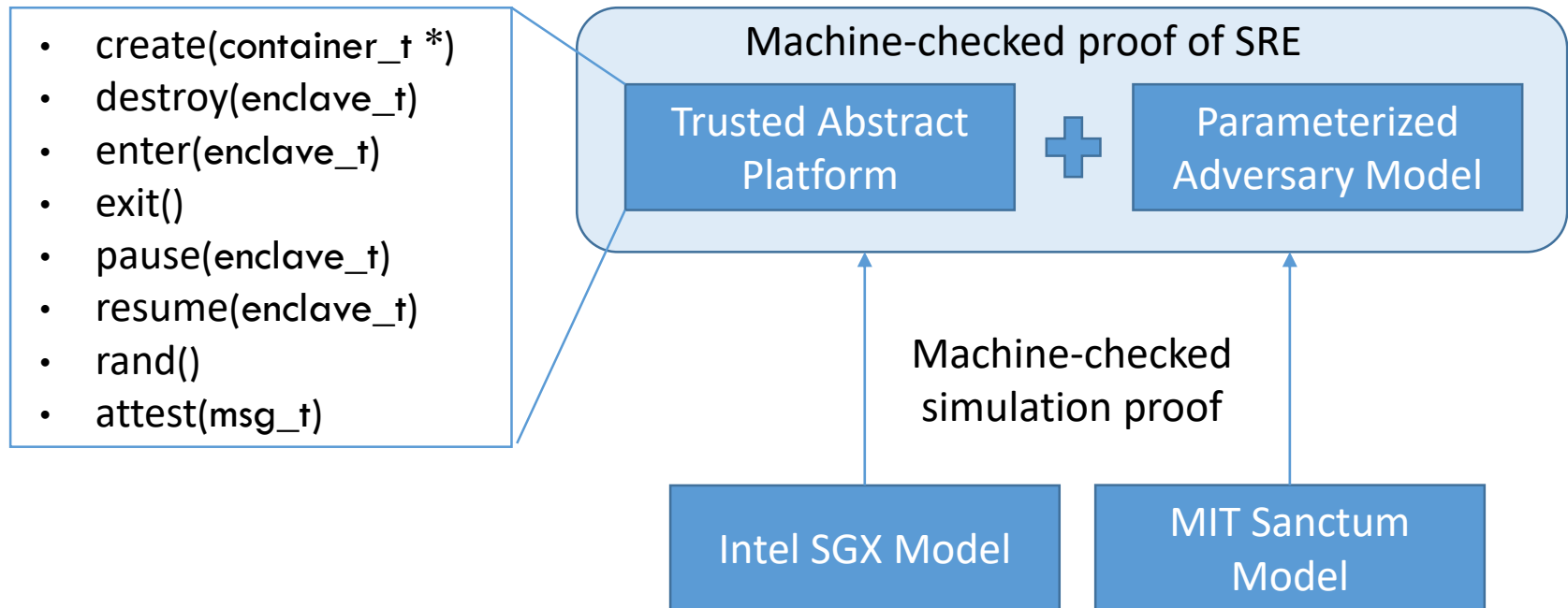
TAP State Variables

- Create, Destroy
- Enter, Exit
- Pause, Resume
- Load/Store/ALU
- Update page tables

TAP Operations

A model of an enclave platform that is not specific to a particular implementation like SGX, or Sanctum

Verification of Enclave Platforms



The TAP provides a common framework for reasoning about security of enclave platform primitives and adversary models.

1. Background: Modeling, Specification, Verification
2. Security Property Verification (Hyperproperties)
3. Case Study: Enclave Platform Verification
 - Introduction to Enclave Platforms
 - Defining Security for Enclave Platforms
 - Simulation and Refinement

Simulation --- Intuition (1/2)

First, a minor modification to our definition of transition systems

- $M = (X, Init, R, Out)$
- $Out(X)$ is the “output” of the system at each step of execution

Simulation --- Intuition (2/2)

$M = (X, Init, R, Out)$

- $X \doteq \{x, y\}$
- $Init(X) \doteq x = 0 \wedge y = 1$
- $R(X, X') \doteq x' = x + y \wedge y' = y$
- $Out(X) \doteq x$

$M' = (X', Init', R', Out')$

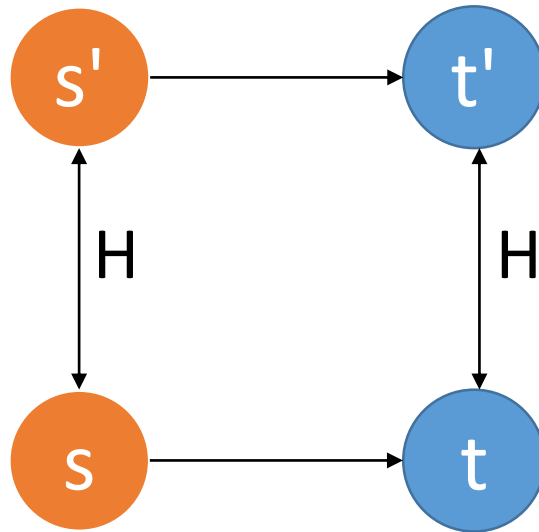
- $X' \doteq \{x\}$
- $Init'(X') \doteq x = 0$
- $R(X, X') \doteq x' = x + 1$
- $Out'(X') = x$

- Trace of values of Out in M and M' are identical
 - In fact M' is an optimization of M
- Simulation captures the notion of one system's traces being a subset of another system's traces

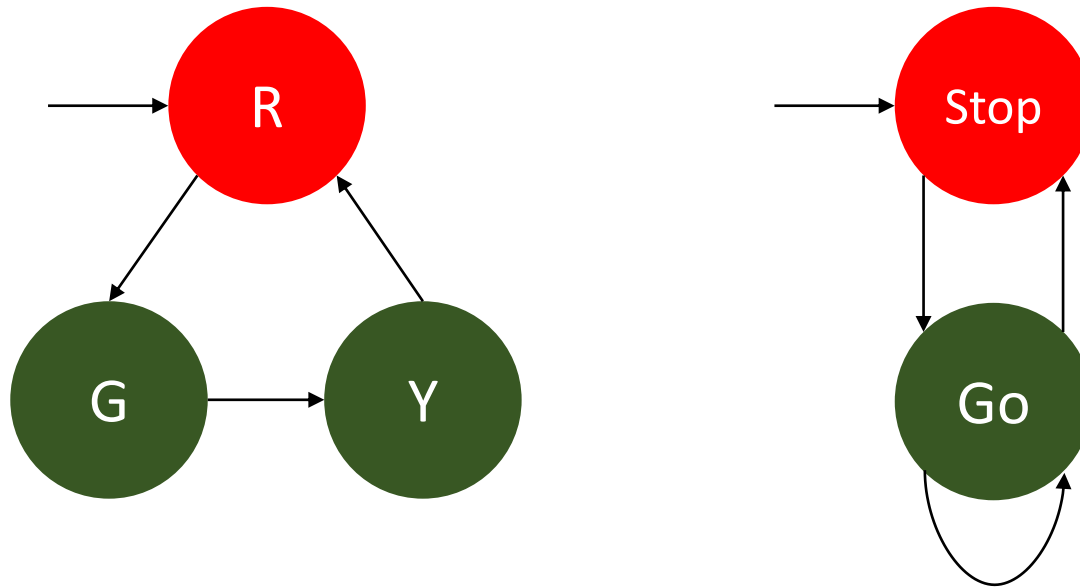
Simulation

- $M = (X, Init, R, Out)$ and $M' = (X', Init', R', Out')$
- $H \subseteq (X \times X')$ is a simulation relation if
 - For all (s, s') , if $H(s, s')$ then:
 - $Out'(s') = Out(s)$
 - For every state t such that $R(s, t)$, there is a state t' s.t. $R'(s', t')$ and $H(t, t')$
- M' simulates M if
 - there exists a simulation relation H and
 - For each s , s.t. $Init(s)$, there exists s' s.t. $H(s, s') \wedge Init'(s')$

Simulation Visualized



Example: Traffic Lights



- Which machine simulates which machine?
- What is the simulation relation?

Why bother with simulation?

- If M' simulates M , then all safety properties over *Out* satisfied by M' also satisfied by M
- M' could be simpler than M as we saw with traffic light
- Proof sketch

Note on Terminology

- We say M' simulates M if the behaviors of M are a subset of the behaviors of M'
- Equivalently to M' simulates M , we can say that M refines M'

Refinement and Hyperproperties

- Q: Are hyperproperties preserved by refinement?
- Specifically, are k-safety properties preserved by refinement?
- A: Yes, k-safety is preserved by refinement
- But in general hyperproperties **are not preserved** by refinement. For example, deniability is not preserved by refinement

Simulation Exercise

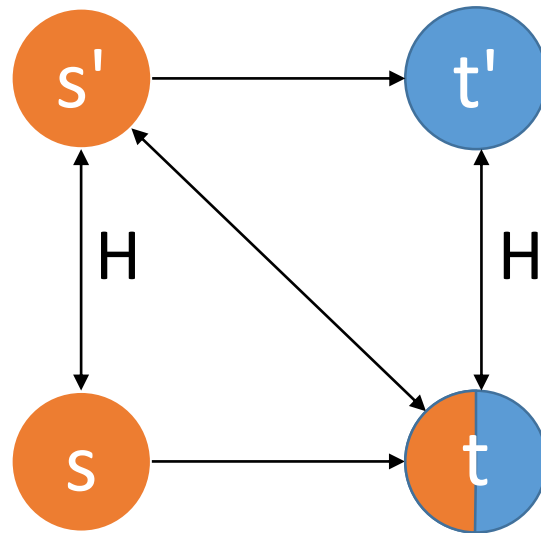
- trafficlight1.ucl
- Encodes the 3 (R, Y, G) vs. 2 (R, G) traffic light model

Task: Add strengthening invariants to prove by induction that module light2 simulates module light3

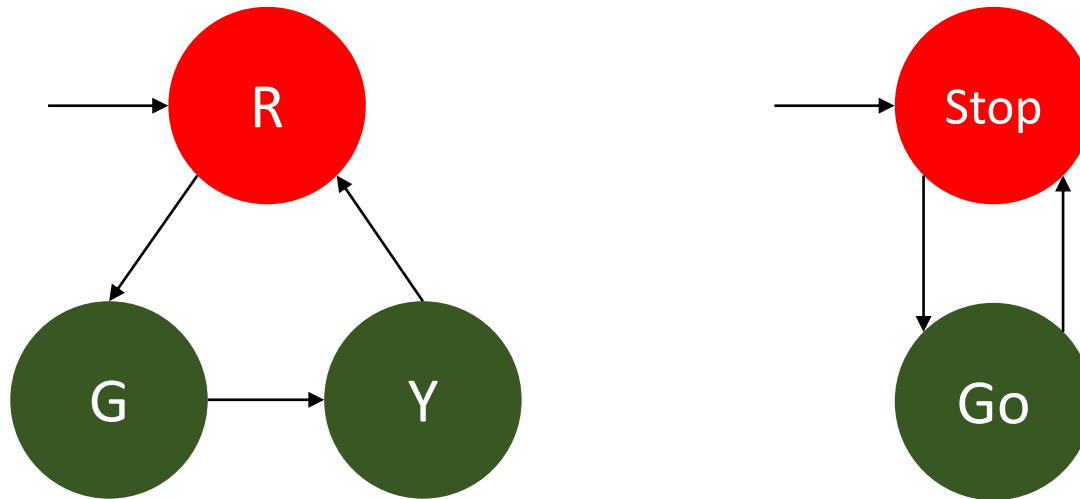
Stuttering Simulation

- $M = (X, Init, R, Out)$ and $M' = (X', Init', R', Out')$
- $H \subseteq (X \times X')$ is a stuttering simulation relation if
 - For all (s, s') , if $H(s, s')$ then:
 - $Out'(s') = Out(s)$
 - For every state t such that $R(s, t)$, there is a state t' s.t. $R'(s', t')$ and $H(t, t')$ **or** $H(t, s')$
- M' simulates M stutteringly if
 - there exists a stuttering simulation relation H and
 - For each s , s.t. $Init(s)$, there exists s' s.t. $H(s, s') \wedge Init'(s')$

Stuttering Simulation Visualized

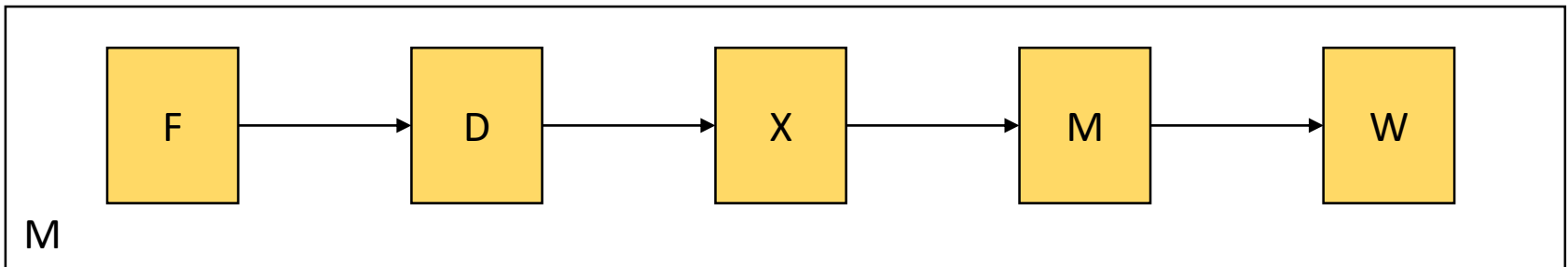
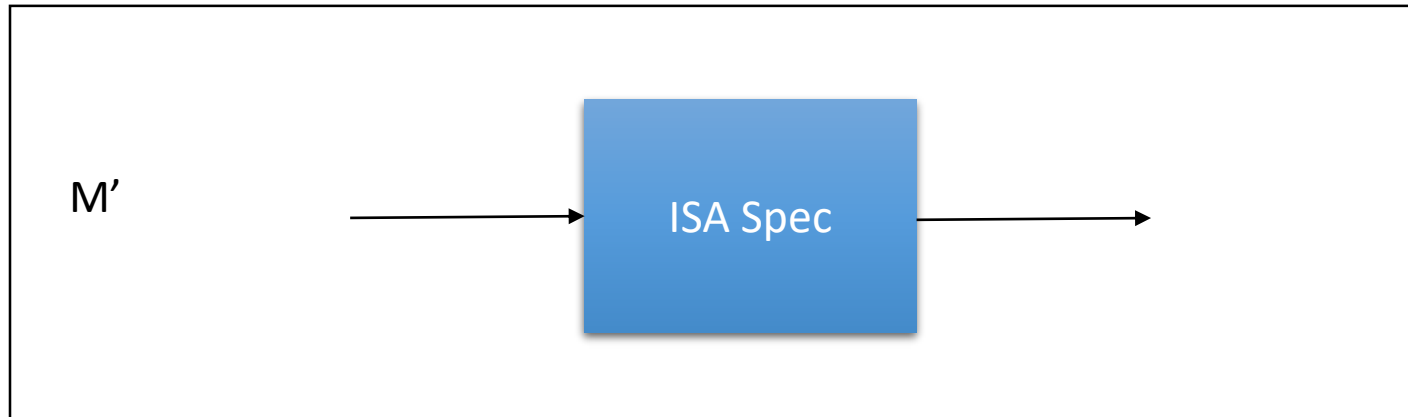


Example: Traffic Lights



- Which machine simulates which machine?
- What is the stuttering simulation relation?

What's the point of stuttering?



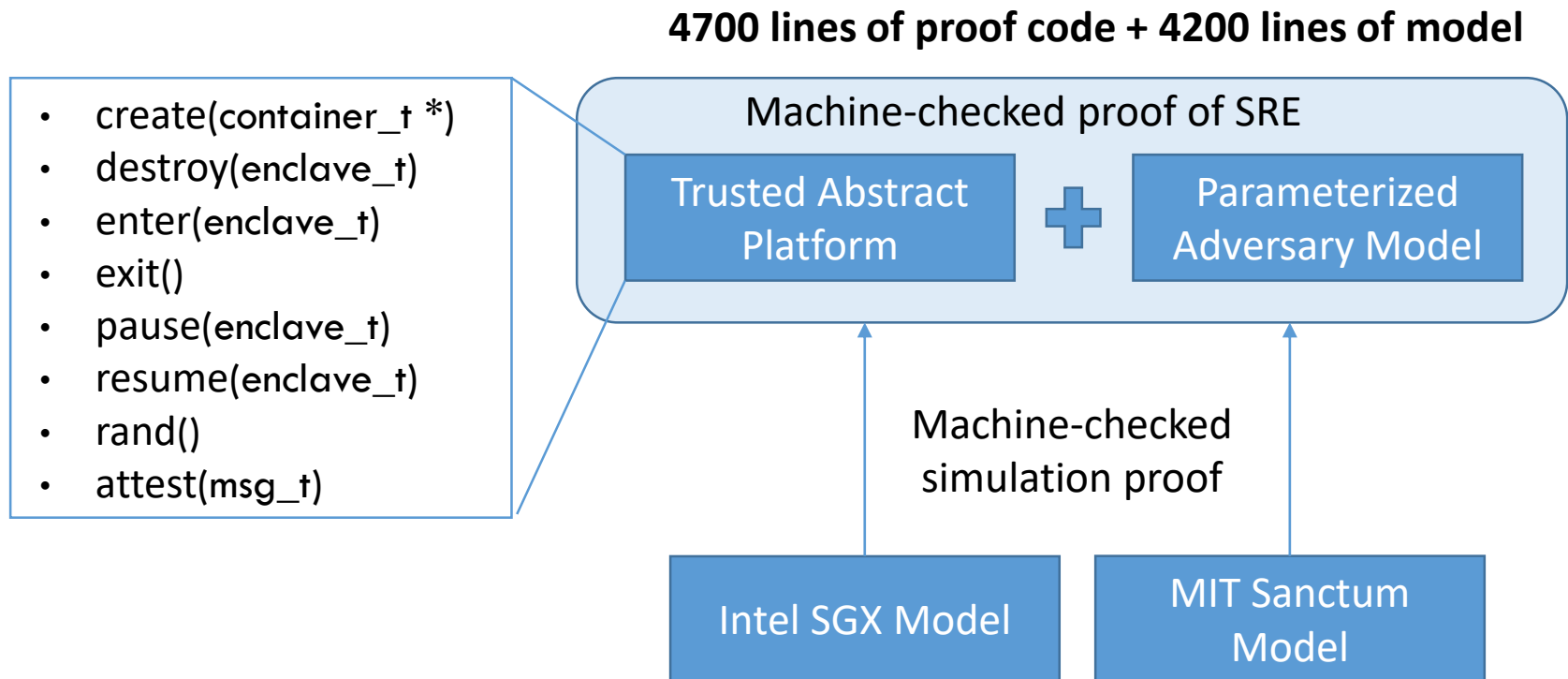
M may take many steps to execute an instruction, that's why we need stuttering

Exercise

- trafficlight2.ucl
- Prove stuttering simulation by induction

1. Background: Modeling, Specification, Verification
2. Security Property Verification (Hyperproperties)
3. Case Study: Enclave Platform Verification
 - Introduction to Enclave Platforms
 - Defining Security for Enclave Platforms
 - Simulation and Refinement
 - Experiments and Wrap-up

Verification of Enclave Platforms



- Proofs were done by induction, just like this tutorial
- Two safety properties proven using self-composition

In this tutorial

What I promised that we will learn:

- How do we automatically find vulnerabilities?
- How do we prove the absence of certain kinds of vulnerabilities?

The 3 Problems (Revisited)

- How to transform a system into something that a computer program can analyze (**modeling**)?
 - **Transition systems**
- How to **specify** what constitutes a vulnerability?
 - **Hyperproperties** – specifically non-interference and observational determinism for specifying security properties
- How we **find** these **bugs** or **prove** their **absence**?
 - **BMC and Induction + Simulation/Refinement**

Thank You for Your Attention

<https://github.com/pramodsu/satsmt2018>

- Public repository contains all problems
- Email me for solutions
- I will put up slides in the repository too
- Feel free to email me with questions
- I am around all day and will be happy to talk

Final Bonus Exercise

`cpu_isolated_mode.ucl`