OREGON PROGRAMMING LANGUAGES SUMMER SCHOOL AT BOSTON UNIVERSITY

Harvard John A. Paulson School of Engineering and Applied Sciences

Language-Based Security

Lecture 4: Enforcing Language Abstractions

Stephen Chong, Harvard University

Road Map

- Intro
  - Formal Methods for Security
  - Language-Based Security
  - Case Study: Noninterference
- Primer on Computer Security
- Information Flow
  - Semantics
  - Enforcement
  - Beyond confidentiality
- Enforcing Language Abstractions

Enforcing Language Abstractions

- Programming Languages are a very useful abstraction!
  - Programmers reason about systems using that abstraction
- But language abstractions can be violated
  - When compiled down to lower-level abstractions and composed with other code
  - Due to "strange" language features, e.g.,
    - reflection
    - unsafe code
    - dynamic code (e.g., `eval`)
    - foreign-function calls
    - ...
  - ...
- If language abstractions violated then language-level reasoning may not hold 😔
- Variety of existing techniques to enforce language abstractions

No Executable Data

- Prevent execution of unauthorized code
- E.g.,
  - Do not have an `eval` operator in your language
    - But limited forms of reflection are often useful!
  - Database interface: use prepared statements instead of arbitrary strings
    - Prevents SQL injection attacks

Enforce Memory Safety

- Fat pointers
  - Pointers to memory include upper and lower bounds
    - Prevents buffer overflow
- Software Fault Isolation (SFI)
  - Low-level rewriting/restriction of code execution to ensure it (approximately) matches intended execution
  - Key idea: confine what code can execute and what memory can be accessed
  - E.g., Control Flow Integrity (CFI): ensure jumps only to suitable code targets
    - Maybe aligned on 32-byte boundaries, maybe a list of permitted addresses
  - E.g., ensure that all memory access is aligned and restricted to appropriate segment
  - Lots of low-level tricks to be efficient
- …

Compilation

- Those previous techniques are mainly ad hoc, and don't actually guarantee enforcement of language semantics
- Let's think about compilation from high-level language to low-level language
  - Discrepancy between language abstractions of low-level and high-level

```java
package Bank;

public class Account{
  private int balance = 0;

  public void deposit( int amount ) {
    this.balance += amount;
  }
}
```

```c
typedef struct account_t {
  int balance = 0;
  void ( *deposit ) ( struct Account*, int ) = deposit_f;
} Account;

void deposit_f( Account* a, int amount ) {
  a→balance += amount;
  return;
}
```

Patrignani et al., ACM Comput. Surv. 51(6), 2019.

Secure Compilation

- The goal of **secure compilation** is to develop compiler techniques that preserve security properties of program components
  - i.e., program components that are composed with other (potentially malicious) components

Full Abstraction

- Various formal statements of what secure compilation means
- One common approach is **full abstraction**
- Compiler is fully abstract when it translates equivalent source-level components into equivalent target-level ones
  - **Preserves** and **reflects** observational equivalence between source and target programs

Contexts

- To define full abstraction, we first define *contexts* and *contextual equivalence*
- A context $\mathbb{C}$ is a program with a hole (denoted $[\cdot]$) that can be filled with a program component P, generating a whole program $\mathbb{C}[P]$
  - You can think of a context $\mathbb{C}$ as a function from component to whole program
- Contexts can model external code that is interacting with a component

Context Examples

- In an ML-like language:

```
let f = [·] in
f 0
```

- Plugging in the component `fun x -> x + 7` gives us the whole program

```
let f = fun x -> x + 7 in
f 0
```

Context Examples

- In Java:

```java
package main;
import Bank.Account;

public class Main{
  public static void main( String [] args ){
    Account acc = new Account();
  }
}
```

- Composing with component from earlier gives us whole program:

```java
package Bank;

public class Account{
  private int balance = 0;

  public void deposit( int amount ) {
    this.balance += amount;
  }
}
```

Contextual Equivalence

- Write $P{\Downarrow}o$ if (whole) program P produces observation o
  - e.g., diverges
  - e.g., terminates with output 42
- Two components P_1 and P_2 are **contextually equivalent** if for all contexts $\mathbb{C}$,and observations o, $\mathbb{C}[P_1]{\Downarrow}o$ if and only if $\mathbb{C}[P_2]{\Downarrow}o$
  - Written $P_1 \simeq_{\mathrm{ctx}} P_2$

Example

- Are these OCaml programs contextually equivalent?

```
let rec factorial n =
  match n with
  | 0 -> 1
  | _ -> n * factorial (n - 1)
```

```
let rec factorial n =
  if n <= 0 then 1
  else n * factorial (n - 1)
```

- No, here is a context that distinguishes them

```
[·]
factorial -1
```

  - One program diverges, one evaluates to 1

Example

- Are these OCaml programs contextually equivalent?

```
let sum_to_n n =
  let result = ref 0 in
  for i = 1 to n do
    result := !result + i
  done;
  !result
```

```
let rec sum_to_n n =
  if n <= 0 then 0
  else n + sum_to_n (n - 1)
```

- Yes, can't distinguish their behavior using (standard) OCaml contexts

Example

- Are these Java programs contextually equivalent?

```
private secret : Int = 0;

public setSecret( ) : Int {
  secret = 0;
  return 0;
}
```

```
private secret : Int = 0;

public setSecret( ) : Int {
  secret = 1;
  return 0;
}
```

Contextual Equivalence

- Often definitions of contextual equivalence limited to whether program terminates or diverges
  - Can convert other behavior into termination/divergence
    - E.g., have a context that diverges if the component returns 42, terminates otherwise
  - Note: cannot capture timing channels
- Key idea is that contexts are capturing a notion of observability
  - Contextual equivalence means the components are indistinguishable
- Reasoning about contexts is typically very hard!
  - Can use other equivalences (e.g., trace-based, bisimilarity, ...) so long as they are exactly as precise as contextual equivalence

Full Abstraction

- A compiler is **fully abstract** if it preserves and reflects contextual equivalence:
  For all P_1, P_2, $\quad P_1 \simeq_{\text{ctx}} P_2 \quad$ if and only if $\quad [\![P_1]\!] \simeq_{\text{ctx}} [\![P_2]\!]$

Source components contextually equivalent

Compiled components contextually equivalent

Full Abstraction

- A compiler is **fully abstract** if it preserves and reflects contextual equivalence:
  For all P_1, P_2, $\quad P_1 \simeq_{ctx} P_2 \quad$ if and only if $\quad [\![P_1]\!] \simeq_{ctx} [\![P_2]\!]$
- Reflection is backward direction: follows from compiler correctness (assuming determinism)
  - i.e., if $[\![P_1]\!] \simeq_{ctx} [\![P_2]\!]$ then $P_1 \simeq_{ctx} P_2$
  - e.g., not satisfied by compiling every program to "return 42;"
- Preservation is forward direction: implies target language can not make any additional distinctions between P_1 and P_2
  - i.e., if $P_1 \simeq_{ctx} P_2$ then $[\![P_1]\!] \simeq_{ctx} [\![P_2]\!]$
  - i.e., source-level abstractions are preserved

Achieving Full Abstraction

- May require **back translation**: proving any target-language context can be expressed as a source-language context
- Statically:
  - Use a typed target language, and show the compilation preserves typing
- Dynamically:
  - Use cryptography in target language
  - Insert runtime checks
    - Must ensure attacker cannot avoid/tamper with these checks
  - Use security architectures
    - Address-space layout randomization
    - Trusted Execution Environments, e.g., Intel's SGX, ARM's TrustZone, ...

Beyond Full Abstraction

- Full abstraction preserves (and reflects) contextual equivalence
- But they may not be the only property we are interested in preserving
- What about safety and liveness properties?
- What about hyperproperties?
  - E.g., noninterference-like security guarantees
- Full abstraction not strong enough to enforce these
- And may be too hard to enforce if all you care about is, e.g., safety and not contextual equivalence

Full Abstraction Not Enough

- Consider a compiler that translates programs of the form $\mathsf{f(x{:}Bool)} \mapsto \mathsf{e}$
  to $\mathbf{f(x{:}Nat){\mapsto}if\ x{<}2\ then}\ \mathsf{e}{\downarrow}\ \mathbf{else\ if\ x{<}3\ then\ f(x)\ else\ 42}$
  - i.e., checks if input is a boolean, and if so behaves correctly, but is insecure on other inputs
- Is fully abstract!
- But doesn't preserve safety property "Never output 42"
  - E.g., when compiling $\mathsf{f(x{:}Bool){\mapsto}0}$

Robust Trace Preservation

- A compiler satisfies RTP iff compilation preserves every trace-based property:

$$\forall \pi \in 2^{\mathit{Trace}}.\ \forall \mathsf{P}.\ (\forall \mathsf{C_S}\ t.\ \mathsf{C_S}[\mathsf{P}] \rightsquigarrow t \Rightarrow t \in \pi) \Rightarrow$$
$$(\forall \mathbf{C_T}\ t.\ \mathbf{C_T}[\mathsf{P}\!\downarrow] \rightsquigarrow t \Rightarrow t \in \pi)$$

A trace-based property

Compilation of program P

Robust Trace Preservation

- A compiler satisfies RTP iff compilation preserves every trace-based property:

$$\forall \pi \in 2^{\mathit{Trace}}.\ \forall \mathsf{P}.\ (\forall \mathsf{C_S}\ t.\ \mathsf{C_S}\left[\mathsf{P}\right] \rightsquigarrow t \Rightarrow t \in \pi) \Rightarrow$$
$$(\forall \mathbf{C_T}\ t.\ \mathbf{C_T}\left[\mathsf{P}\!\downarrow\right] \rightsquigarrow t \Rightarrow t \in \pi)$$

- An equivalent “property-free” characterization:

$$\forall \mathsf{P}.\ \forall \mathbf{C_T}.\ \forall t.\ \ \mathbf{C_T}\left[\mathsf{P}\!\downarrow\right] \rightsquigarrow t \Rightarrow \exists \mathsf{C_S}.\ \mathsf{C_S}\left[\mathsf{P}\right] \rightsquigarrow t$$

Robust Safety Preservation

- A compiler satisfies RSP iff compilation preserves every trace-based safety property:

$$\mathsf{RSP}: \quad \forall \pi \in \mathit{Safety}.\ \forall \mathsf{P}.\ (\forall \mathsf{C_S}\ t.\ \mathsf{C_S}[\mathsf{P}] \rightsquigarrow t \Rightarrow t \in \pi) \Rightarrow$$
$$(\forall \mathbf{C_T}\ t.\ \mathbf{C_T}[\mathsf{P}{\downarrow}] \rightsquigarrow t \Rightarrow t \in \pi)$$

- An equivalent "property-free" characterization:

$$\forall \mathsf{P}.\ \forall \mathbf{C_T}.\ \forall m.\ \mathbf{C_T}[\mathsf{P}{\downarrow}] \rightsquigarrow m \Rightarrow \exists \mathsf{C_S}.\ \mathsf{C_S}[\mathsf{P}] \rightsquigarrow m$$

Finite trace prefix
(Intuitively, the "bad" trace)

Robust Hyperproperty Preservation

- A compiler satisfies RHP iff compilation preserves every trace-based hyperproperty:

$$\text{RHP}: \quad \forall H \in 2^{2^{\mathit{Trace}}}.\ \forall \mathsf{P}.\ (\forall \mathsf{C_S}.\ \mathtt{Behav}\,(\mathsf{C_S}\,[\mathsf{P}]) \in H) \Rightarrow (\forall \mathbf{C_T}.\ \mathtt{Behav}\,(\mathbf{C_T}\,[\mathsf{P}{\downarrow}]) \in H)$$

- Equivalent "property-free" characterizations:

$$\forall \mathsf{P}.\ \forall \mathbf{C_T}.\ \exists \mathsf{C_S}.\ \mathtt{Behav}\,(\mathbf{C_T}\,[\mathsf{P}{\downarrow}]) = \mathtt{Behav}\,(\mathsf{C_S}\,[\mathsf{P}])$$

$$\forall \mathsf{P}.\ \forall \mathbf{C_T}.\ \exists \mathsf{C_S}.\ \forall t.\ \mathbf{C_T}\,[\mathsf{P}{\downarrow}] \rightsquigarrow t \iff \mathsf{C_S}\,[\mathsf{P}] \rightsquigarrow t$$

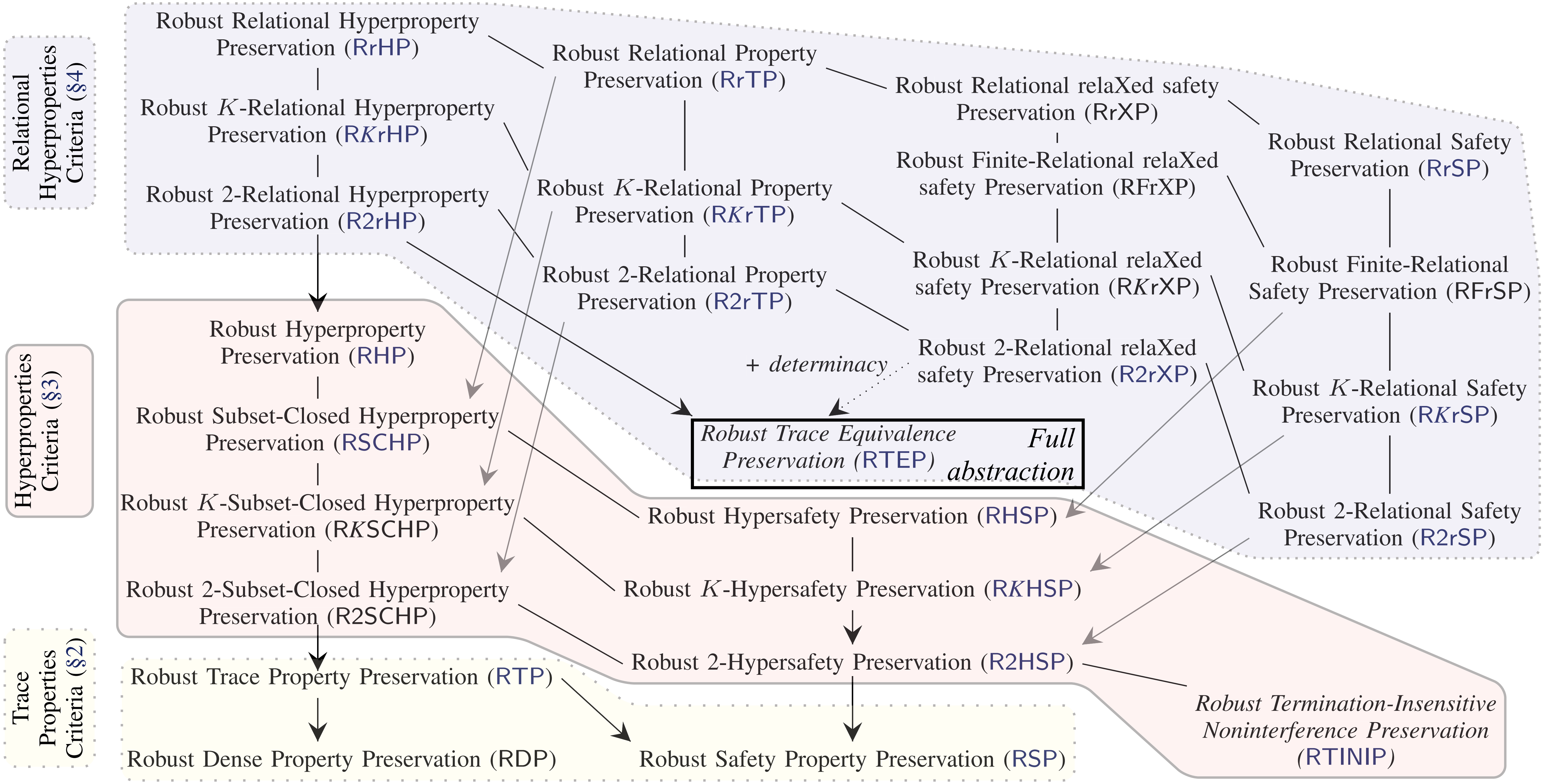


Fig. 1: Partial order with the secure compilation criteria studied in this paper. Criteria higher in the diagram imply the lower ones to which they are connected by edges. Criteria based on trace properties are grouped in a yellow area, those based on hyperproperties are in a red area, and those based on relational hyperproperties are in a blue area. Criteria with an *italics name* preserve a *single* property that belongs to the class they are connected to; the dotted edge requires an additional determinacy assumption. Finally, each edge with a thick arrow denotes a *strict* implication that we have proved as a separation result.

Abate et al., CSF 2019.

References/Further Reading

- Patrignani, M., A. Ahmed, and D. Clarke (2019, feb). Formal approaches to secure compilation: A survey of fully abstract compilation and related work. ACM Comput. Surv. 51(6).
- Abate, C., R. Blanco, D. Garg, C. Hritcu, M. Patrignani, and J. Thibault (2019). Journey beyond full abstraction: Exploring robust property preservation for secure compilation. CSF 2019, pp. 256–271. IEEE.
- Amal Ahmed OPLSS 2019 lectures: https://www.youtube.com/watch?v=yP29TKmK3_o

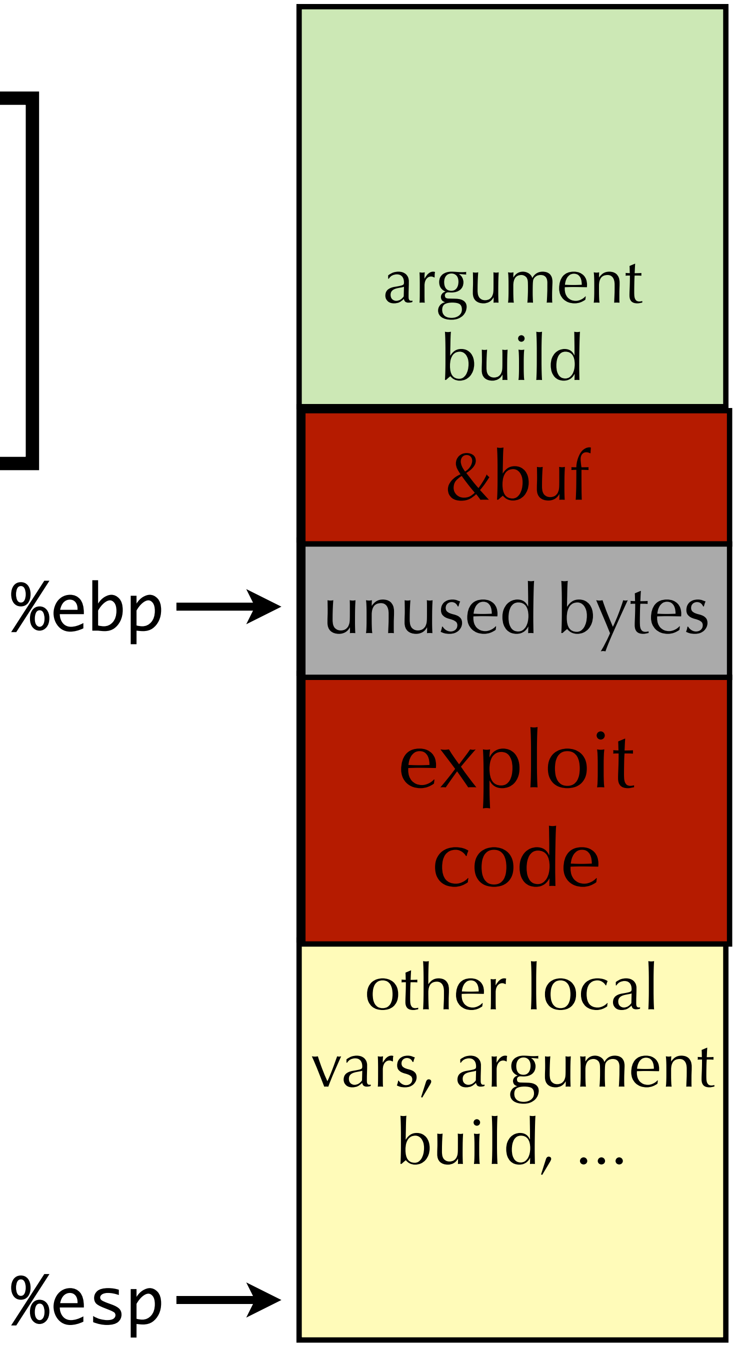
Weird Machines

Buffer Overflow Exploit

- Consider the following vulnerable C code

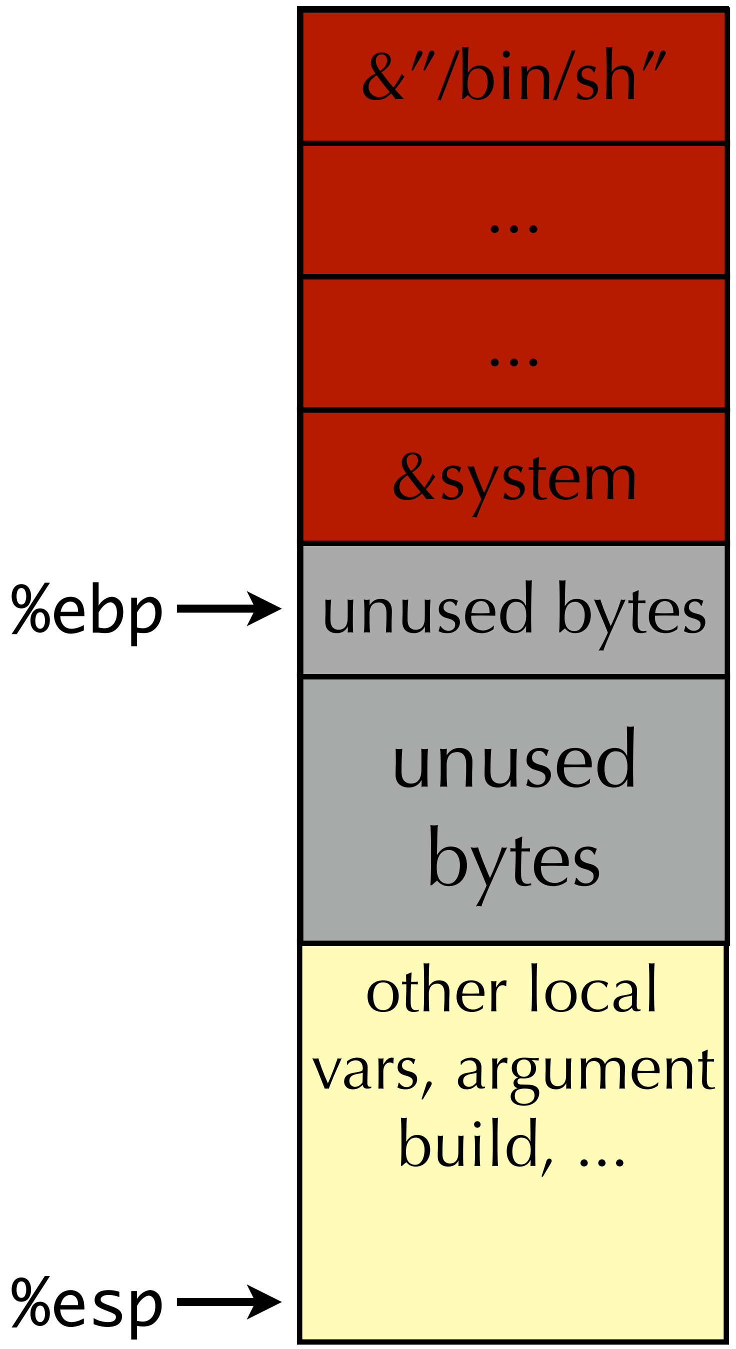
```
void vulnerable_function(char *input) {
    char buf[64];
    strcpy(buf, input);
}
```

- Classic buffer overflow attack:
  - Call vulnerable_function with input that puts x86 exploit code into buf and overwrites return address
  - Execute arbitrary code!
- Prevent it by ensuring non-executable stack



Return to libc attack

- Even if stack is non-executable, can make use of existing code
- libc is on most systems; address of libc code is guessable
- E.g., set up stack so that overwrite:
  - return address with address of system function in libc
  - overwrite argument build area with address of string "/bin/sh"
    - Maybe string is already in binary
    - Or maybe also put that string into the payload



It Gets Worse...

- Return-Oriented Programming (ROP)
  - A **gadget** is a short sequence of machine instructions that ends in a return instruction
    - Attackers can (automatically) identify gadgets that already exist in binaries
  - Key idea of ROP: chain gadgets together
    - Each gadget performs a small amount of computation, then return instruction jumps to the next gadget
    - i.e., overflow the stack to put the sequence of addresses of gadgets on the stack
- Gadgets perform operations but may also set up the machine for the next gadget
  - E.g., one gadget might load specific value into a register; next gadget will read the register

Weird Machines

- How do we formalize and think about these kinds of exploits?
  - Formal methods can help us understand and also possibly prevent entire class of exploits
- Recent work on **weird machines** presents a perspective on this
  - Dullien, 2020

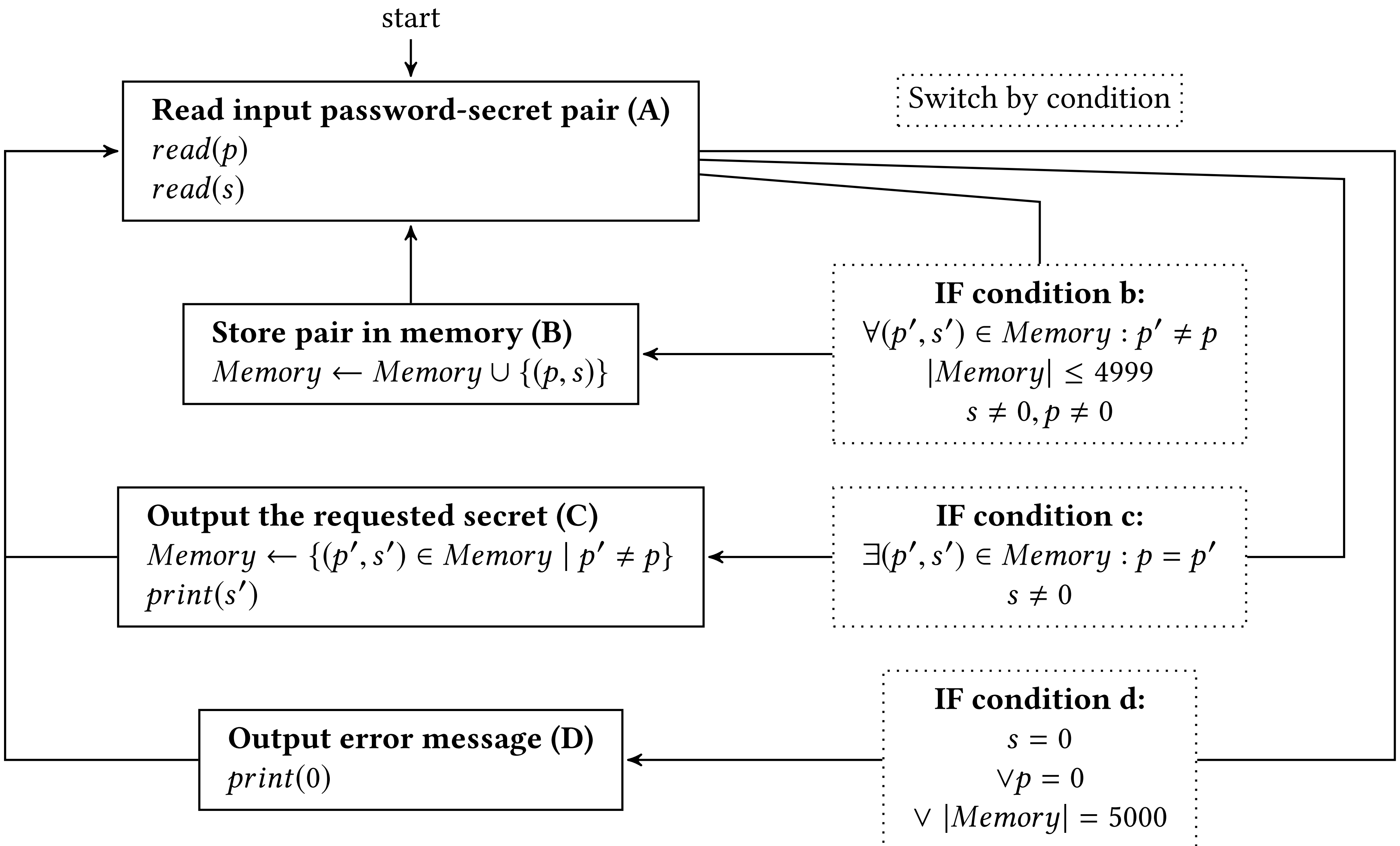
Intended Finite State Machine

- What the programmer intends to implement
- $\theta = (Q, i, F, \Sigma, \Delta, \delta, \sigma)$
  - Set of states Q
  - Initial state i
  - Final states F
  - Input alphabet Σ
  - Output alphabet Δ
  - State transition function $\delta : Q \times \Sigma \rightarrow Q$
  - Output function $\sigma : Q \times \Sigma \rightarrow \Delta$

Example: Tiny Secure Message-Passing Server

- Small, clearly-defined security boundary, complex enough to be interesting
- A machine that remembers password-secret pairs for later retrieval
  - Retrieval removes the pair
  - Arbitrary limit of 5000 password-secret pairs
- Security property: intuitively, you need to know (or guess) the right password to obtain the secret
  - Can express precisely using probabilities
- States of the FSM given by

$$\mathcal{M} := \left\{ \begin{array}{l|l} \emptyset, & \\ \{(p_1, s_1)\}, & p_i, s_i \in bits_{32} \backslash \{0\} \\ \ldots, & p_i \neq p_j \\ \{(p_1, s_1), \ldots, (p_{5000}, s_{5000})\} & \end{array} \right\}$$



$$Q := \{A_M, M \in \mathcal{M}\}, \qquad i := A_\emptyset, F := \emptyset$$

$$\Sigma := \{(p, s) | p, s \in bits_{32}\}, \qquad \Delta := \{s \in bits_{32}\}$$

$$\delta := A_M \times (p, s) \quad \rightarrow \quad \begin{cases} A_{M \cup (p,s)} \text{ if } & \begin{array}{l} (p, s) \notin M \\ \wedge |M| \leq 4999 \\ \wedge s \neq 0 \end{array} \\ A_{M \setminus (p,s)} \text{ if } (p, s) \in M \\ A_M \text{ otherwise} \end{cases}$$

$$\sigma := A_M \times (p, s) \quad \rightarrow \quad \begin{cases} s' \text{ if } (p, s') \in M \\ 0 \text{ if } s = 0 \vee |M| = 5000 \end{cases}$$

Security Property

- Set up a game (similar to cryptographic protocols)

(1) The attacker chooses a probability distribution $\mathcal{A}$ over finite-state transducers Θ_{exploit} that have an input alphabet $\Sigma_{\Theta_{\text{exploit}}} = \Delta$ and output alphabet $\Delta_{\Theta_{\text{exploit}}} = \Sigma$. This means that the attacker specifies one or more finite-state transducers that take as input the outputs of the IFSM, and output words that are the input for the IFSM.

(2) Once this is done, the defender draws two elements p, s from $bits_{32}$ according to the uniform distribution.

(3) The attacker draws a finite-state transducer from his distribution and is allowed to have it interact with the IFSM for an attacker-chosen number of steps n_{setup}.

(4) The defender sends his (p, s) to the IFSM.

(5) The attacker gets to have his Θ_{exploit} interact with the IFSM for a further attacker-chosen number of steps n_{exploit}.

- Probability for Θ_{exploit} to obtain secret is no better than guessing:

$$P[s \in o_{\text{IFSM}}] \leq \frac{n_{\text{setup}} + n_{\text{exploit}}}{|bits_{32}|} = \frac{|o_{\text{exploit}}|}{2^{32}}$$

Emulating the IFSM

- Programmer implements/emulates the IFSM
- Assume we have a simple machine (Cook-and-Reckhow RAM machine model)
  - Harvard architecture (i.e., code is not data)
  - 2^{16} 32-bit memory cells, treat first 6 as registers

$\mathrm{LOAD}(C, r_d)$	$: r_d \leftarrow C$	Load a constant
$\mathrm{ADD}(r_{s_1}, r_{s_2}, r_d)$	$: r_d \leftarrow r_{s_1} + r_{s_2}$	Add two registers or a register and constant
$\mathrm{SUB}(r_{s_1}, r_{s_2}, r_d)$	$: r_d \leftarrow r_{s_1} - r_{s_2}$	Subtract two registers or a register and constant
$\mathrm{ICOPY}(r_p, r_d)$	$: r_d \leftarrow r_{r_p}$	Indirect memory read
$\mathrm{DCOPY}(r_d, r_s)$	$: r_{r_d} \leftarrow r_s$	Indirect memory write
$\mathrm{JNZ/JZ}(r, I_z)$		Transfer control to I_z if r is nonzero, zero
$\mathrm{READ}(r_d)$	$: r_d \leftarrow \mathit{input}$	Read a value from input
$\mathrm{PRINT}(r_s)$	$: r_d \rightarrow \mathit{output}$	Write a value to output

Emulating the IFSM

- Variant 1:
  - Use registers/cells 0-5 as scratch
  - Use cells 6-10006 a simple flat array for storing pairs of values
  - No sophisticated data structures, just search through memory for empty pairs of memory cells
- Variant 2
  - Implement as two singly-linked lists
    - One to keep track of free space for password-secret pairs
    - One to keep track of currently active password-secret pairs

What is a bug?

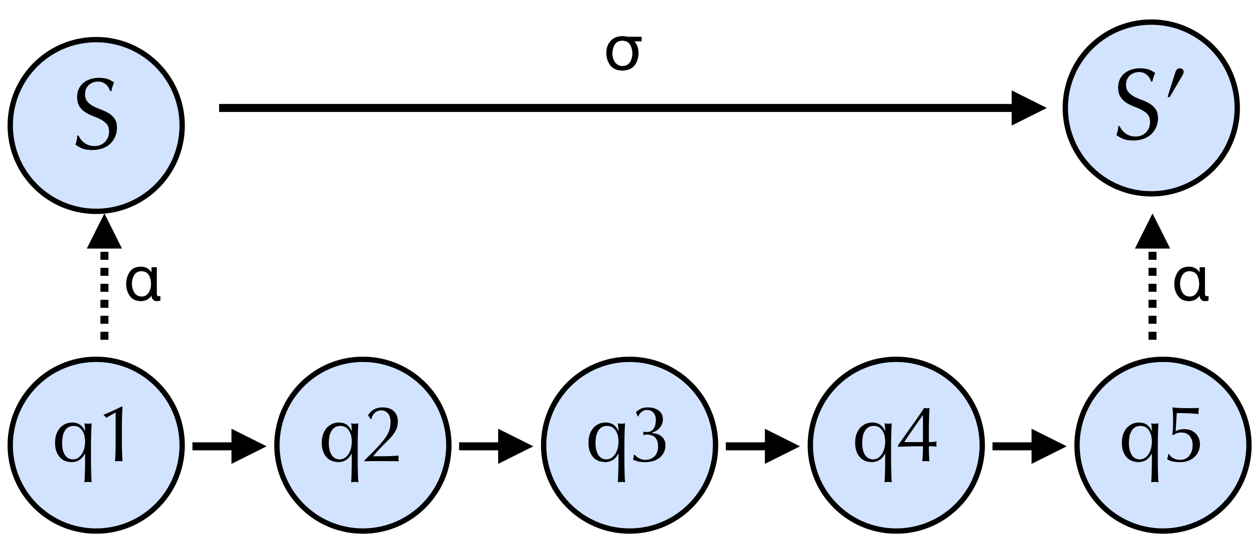
- Can explicitly define bugs in this setting
  - IFSM serves as intensional specification
- Call the implementation machine *cpu*
  - Let Q_{cpu} be the set of states of the implementation machine
- Bug has occurred in implementation when implementation state $q \in Q_{cpu}$ has no clean equivalent in IFSM

Sane and Transitory States

- Abstraction function from states Q_{cpu} to states Q_θ (of the IFSM)

$$\alpha_{\theta,cpu,\rho} : Q_{cpu} \to Q_\theta$$

- Set Q_{cpu}^{sane} are the states for which α is defined
  - i.e., the states that directly correspond to a state of the IFSM
- But *cpu* may take multiple steps to implement one step in the IFSM, i.e., may have some **transitory states**
  - legitimate states needed to reach a desired target state of the IFSM
  - Need to distinguish these from error states
  - Call them Q_{cpu}^{trans}



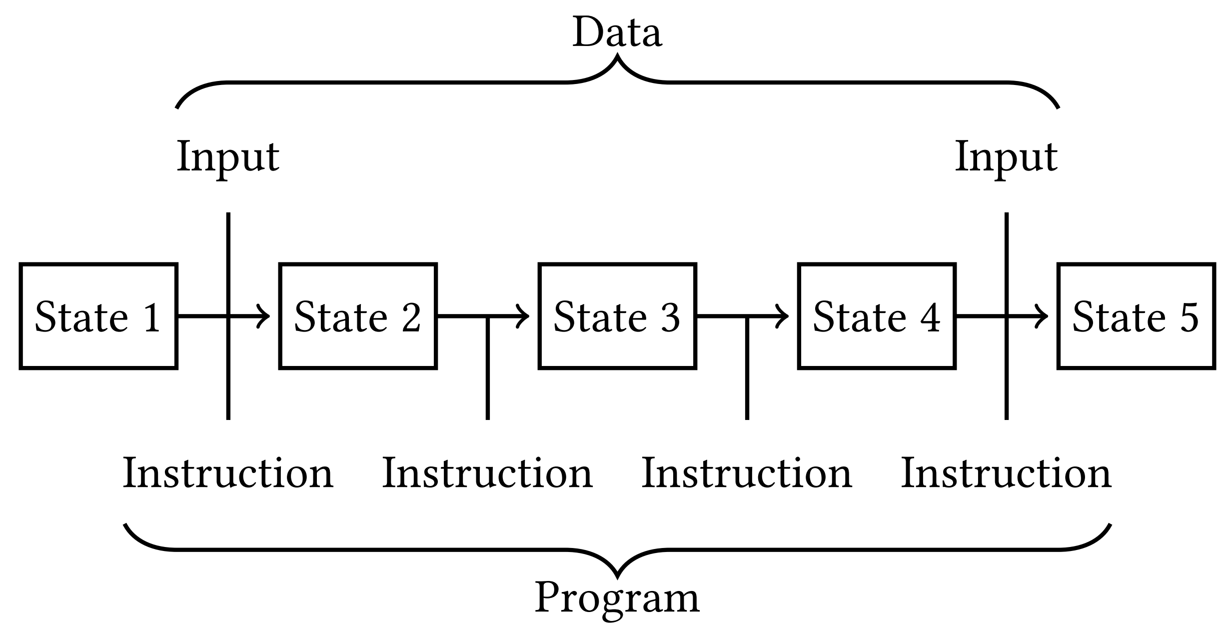
$\delta(S, \sigma) = S'$

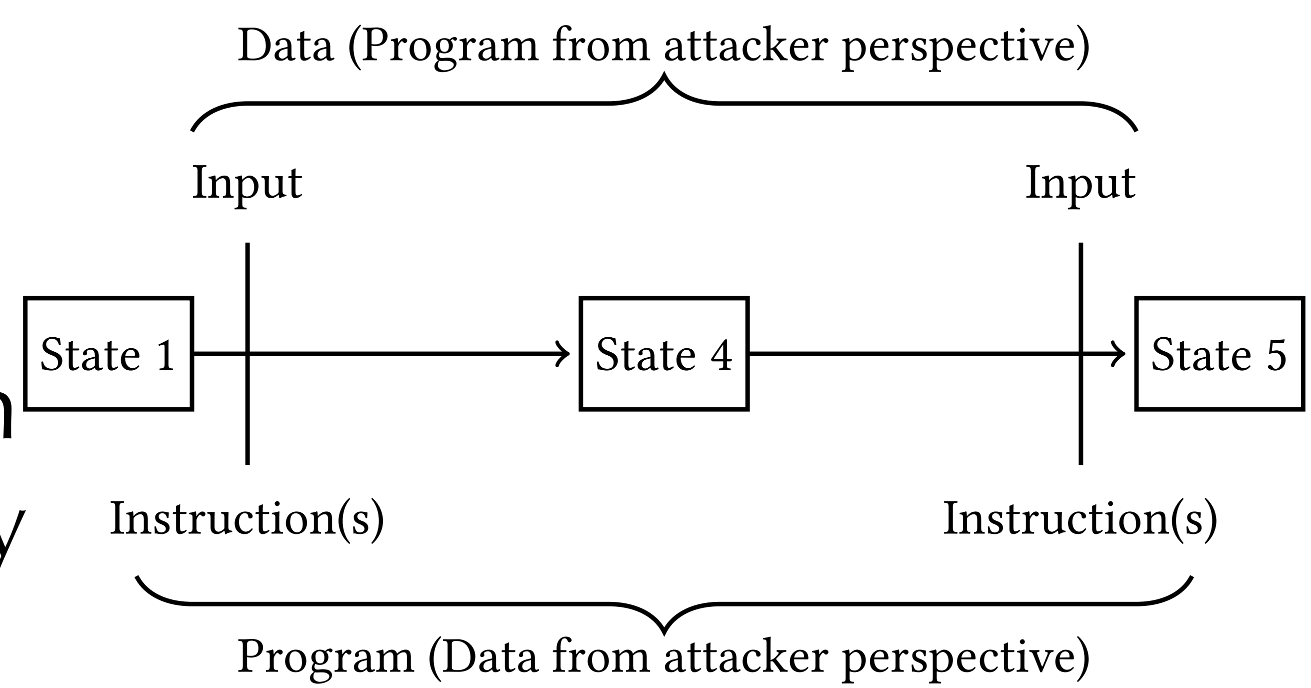
Weird States

- Weird states (Q_{cpu}^{weird}) are the states of Q_{cpu} that are neither sane nor transitory
  - $Q_{cpu} = Q_{cpu}^{sane} \dot\cup Q_{cpu}^{trans} \dot\cup Q_{cpu}^{weird}$
- Sources of weird states
  - Human error in writing program
    - Most common source! e.g., memory corruption bugs, buffer overflows, failed invariants, ...
  - Hardware faults during execution
    - Bit flips, from gamma rays or Rowhammer attacks, etc
  - Transcription errors
    - E.g., error in program transmission (over network, from disk, etc.)

Weird Machines

- Classical view of machine: runs program, accepts data as input
- Can summarize sequence of instructions, and intermediate states
- From attacker perspective: an unintended machine where the input data, combined with the code, operates on memory





Weird Machine

- Intended machine implementation
  - Emulates all state transitions of the IFSM so a state from Q_{cpu}^{sane} gets transformed to another state from Q_{cpu}^{sane}, maybe traversing some states from Q_{cpu}^{trans}
- There may be an unintended machine
  - Start in a weird state
  - "Instructions" in the form on input transform to other weird states
    - Transitions that were meant to transform valid states!

$$(Q_{cpu}^{\text{weird}}, q_{init}, Q_{cpu}^{\text{sane}} \cup Q_{cpu}^{trans}, \Sigma', \Delta', \delta', \sigma')$$

- Interesting properties:
  - Input as instruction stream
  - Unknown state space
  - Unknown computational power
  - Emergent instruction set

Attacker Models

- Given method of entering some initial q_{init} from some particular set of sane states $\{q_i\}_{i \in I} \subset Q_{cpu}^{sane}$
- Exploitation is the process of:
  - setup (choosing the right q_i)
  - instantiation (entering q_{init}) and
  - programming of the weird machine
- How to model the attacker? Some possibilities:
  - Arbitrary program-point, chosen-bitflip
    - Attacker gets to stop program execution, choose any bit to flip
  - Arbitrary program-point, chosen-bitflip, registers
    - Attacker gets to stop program execution, choose any bit (except for registers) to flip
  - Fixed-program point, chosen-bitflip, registers
    - At specific program point(s), attacker gets to choose any bit (except for registers) to flip
  - …

Exploitability

- Variant 1: Not exploitable!
  - Key idea: show that any bit-flip the attacker can do can be achieved by a finite number of legitimate transitions
  - i.e., bit-flipping stays within Q_{cpu}^{sane}
  - Show that the security property is achieved if staying only within Q_{cpu}^{sane}
- Variant 2: Exploitable
  - Key idea: attacker sets up data structure so that a bitflip corrupts a pointer, and a known value is treated as a password

Where to From Here?

- This provides a perspective on weird machines
  - Generalizes many kinds of vulnerabilities
- But does it provide insight in how to prevent these vulnerabilities?

Weird Machines as Insecure Compilation

- Paykin et al. (2019)
- Key idea: an exploit is behavior in the target that doesn't correspond to behavior in the source

Definition (Exploit). *An* exploit *of a vulnerable source program* V *is an attacker context* $\mathbf{A}$ *from an attack class* $\mathcal{A}$ *if* $\mathit{Behav}\,(C[V]) \neq \mathit{Behav}\,(\mathbf{A}[\llbracket V \rrbracket])$ *for every non-oblivious*[1] C.

$$\mathit{Exploit}^{\mathcal{A}}(V) \triangleq \left\{ \mathbf{A} \in \mathcal{A} \;\middle|\; \begin{array}{l} \forall\, C\,.\,\neg\mathit{oblivious}(C) \;\Rightarrow \\ \mathit{Behav}\,(C[V]) \neq \mathit{Behav}\,(\mathbf{A}[\llbracket V \rrbracket]) \end{array} \right\}$$

Definition (Weird Machine). *The* weird machine *of a vulnerable source program* V *for an attack class* $\mathcal{A}$ *is the collection of behaviors arising from exploits of* V.

$$\mathit{WM}^{\mathcal{A}}(V) \triangleq \left\{ \mathit{Behav}\,(\mathbf{A}[\llbracket V \rrbracket]) \;\middle|\; \mathbf{A} \in \mathit{Exploit}^{\mathcal{A}}(V) \right\}$$

Weird Machines as Insecure Compilation

- Generalizes Dullien's approach
- Uses vocabulary of language-based security
- Recall Robust Hyperproperty Preservation:

Theorem III.3 (Abate et al. [9])**.** *A compiler satisfies RHP if and only if for all source components U^S and target contexts C^T, there exists a back-translated source context C^S such that $B(C^S[U^S]) \mapsto B(C^T[U^S\downarrow])$.*

- Exactly characterizes no weird-machine exploits!

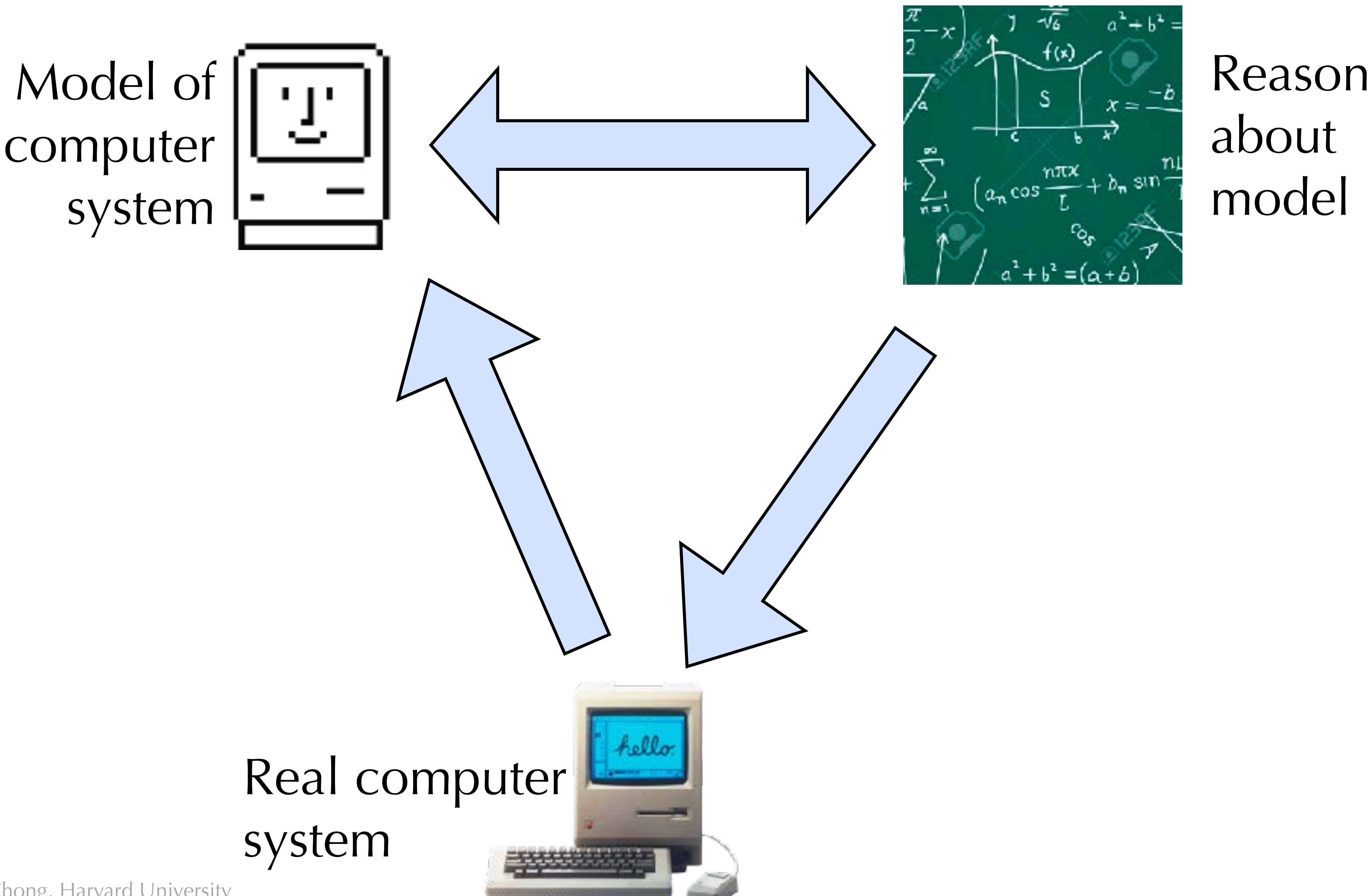
Theorem III.4. *A compiler satisfies RHP if and only if it has no exploits: for all source components U^S, $\mathit{Exploit}(U^S) = \varnothing$.*


KINDNESS
THE MORAL OF THE STORY
HHRDIENSTY

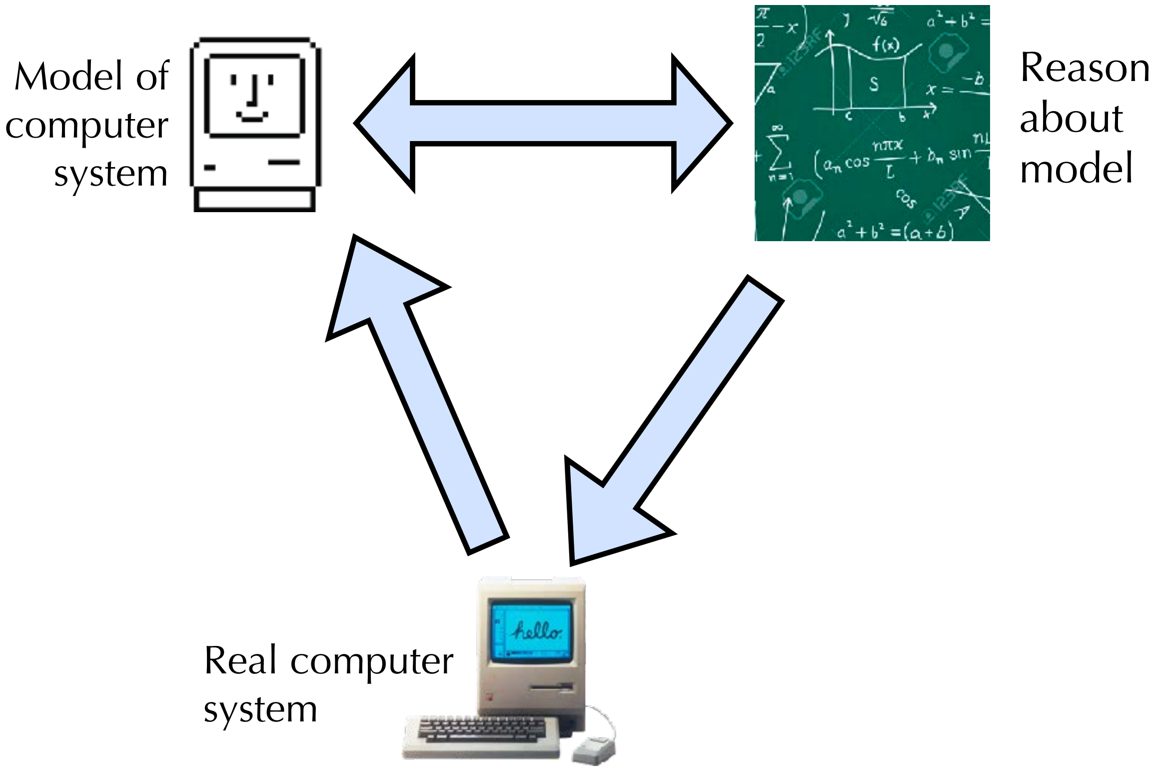
Moral of this Lecture

- Weird machines: application of formal methods to **understand** and **reason** about certain vulnerabilities
- Language-based security connected it with existing security definitions and on-going work on enforcing these security definitions

Moral of this Lecture Series



Moral of this Lecture Series



- PL techniques and ideas are a great fit for formal approaches to computer security
  - Useful models for systems, effective reasoning techniques, practical enforcement mechanisms, enforcing language abstractions preserves reasoning, ...
- Go forth and research!

Moral of this Lecture Series

References/Further Reading

- Dullien, T. (2020). Weird machines, exploitability, and provable unexploitability. IEEE Transactions on Emerging Topics in Computing 8(2), 391–403.
- Paykin, J., E. Mertens, M. Tullsen, L. Maurer, B. Razet, and S. Moore (2019). Weird machines as insecure compilation. In Workshop on Foundations of Computer Security.