PL and Crypto

Case studies in their combination

Two great tastes that go great together

Michael Hicks, Professor of CS @ UMD

MARYLAND CYBERSECURITY CENTER

PLUM

Cryptography

- Original focus: data security
- Expanded to computation security
  - Homomorphic encryption
  - Secure Multi-party computation
  - Authenticated data structures
  - Zero-knowledge proofs
  - Oblivious RAM
  - …

Programming Languages

- Originally just for writing programs
- Expanded to help ensure programs are
  - Correct
  - Efficient
  - Secure
  - Maintainable
  - …
- In a variety of domains

Crypto Reasoning

- Crypto has developed several techniques for reasoning about security in the presence of a (bounded) adversary

- Reasoning is often done per application (a.k.a. construction)
- Use a simple model of computation to make reasoning easier
  - Turing machines, boolean circuits, etc.

PL Reasoning

- PL focuses on per-language reasoning
  - Type system that ensures all programs P have property Q
  - Program analysis that ensures programs P have no flaws of class B

- PL mathematical semantics highly developed for expressing realistic and accurate models of behavior

Idea: Bring Them Together

- Start with a crypto idea
- Broaden and strengthen it by embedding it in a programming language
  - Prove security of *all* programs in the language, not just a particular one
  - Optimize the performance of *all* programs
    - Taking advantage of finer-grained, PL-style semantic reasoning

This talk: Two Examples

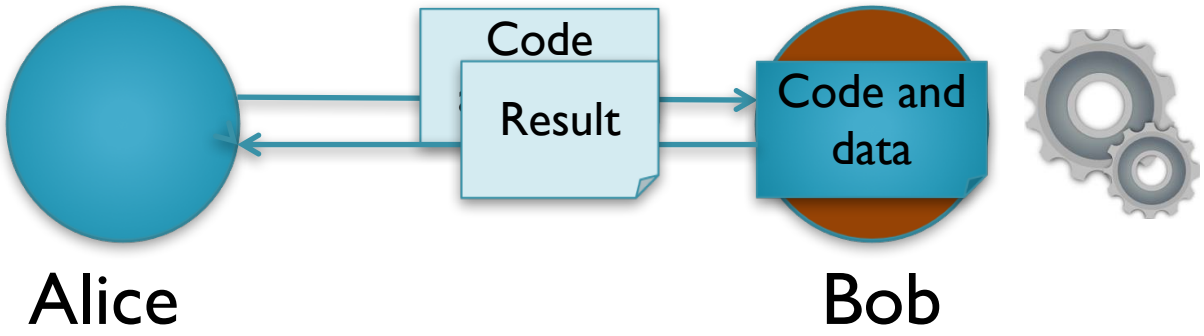
- Privacy-preserving outsourced computation
  - Generalize use of Oblivious RAM to strengthen security and (greatly) optimize performance

- Integrity-preserving outsourced data management
  - Generalize idea underlying Merkle trees to more optimized data structures

Memory-Trace Oblivious Program Execution

Joint work with **Chang Liu, Austin Harris, Martin Maas, Mohit Tiwari, Elaine Shi**

Outsourcing computation



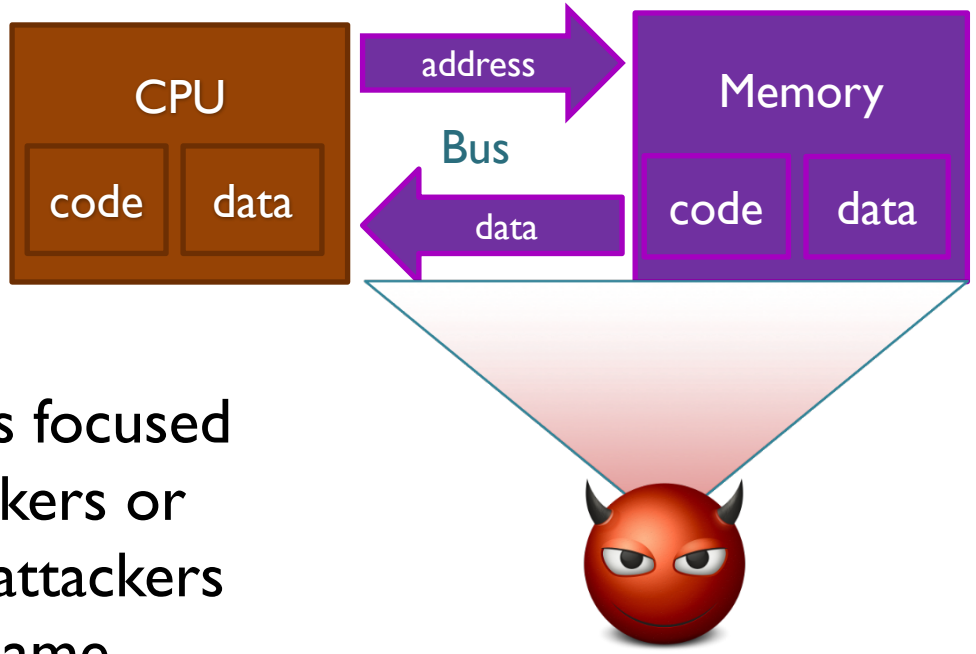
Security goal

- **Privacy**: Bob cannot learn about the input or output of the computation

Bob is **honest but curious** – will snoop but won't corrupt

Challenge: Physical access

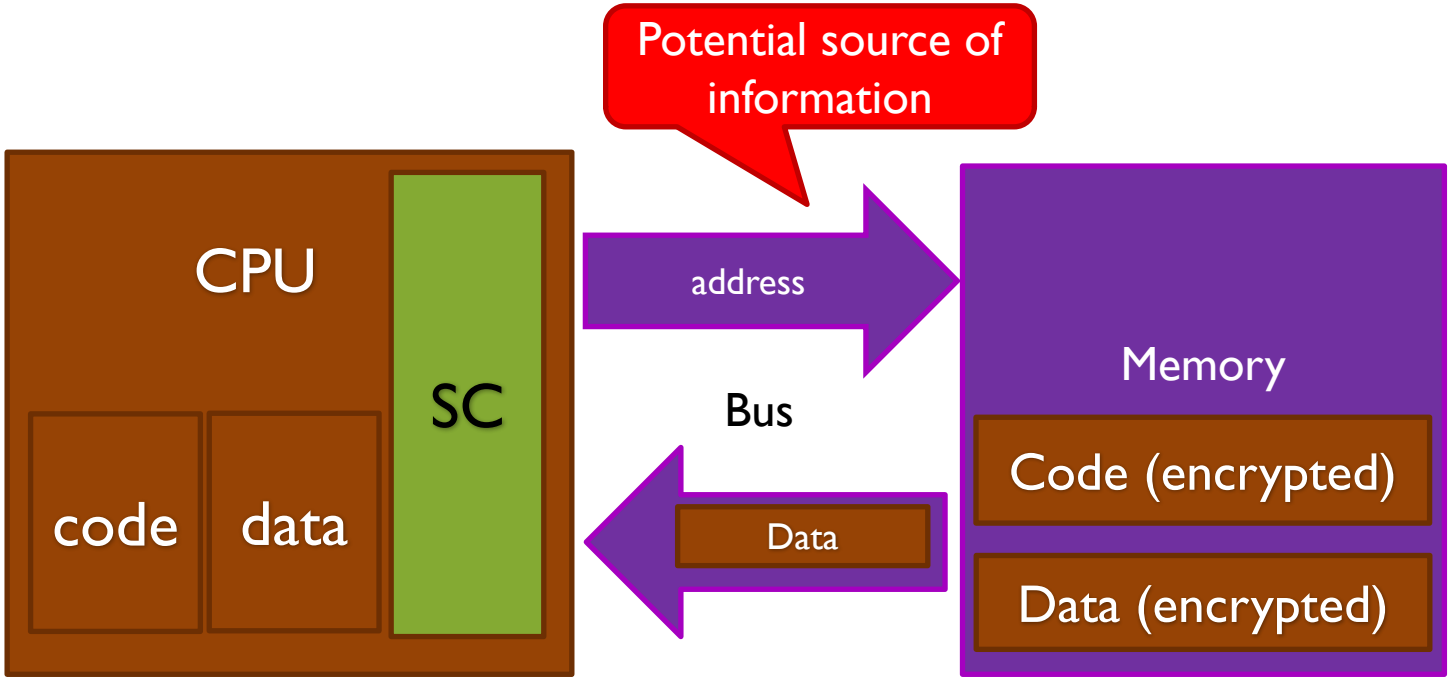
- The cloud provider has physical access to the machine running the computation



Recent work has focused on remote attackers or software-based attackers running on the same hardware

Solution(?): Encrypted data/code

- Use a secure co-processor (SC) to encrypt code and data



Access pattern is a side channel

- It turns out that the **address trace** alone **can leak private data**
  - For example, prior work has shown that the control flow graph can be inferred from it

- Two sources of leakage
  - The **pattern** of addresses
  - The **total number** of memory events

Access pattern is a side channel

- Program
- a[x]:=s

- Input: x=1, s=2
- Memory bus

  read(x, 1)

  read(s, 2)

  write(a+1, 2)

Purple variables are secret
Red values are encrypted

Variables accessed leak info

- Program

```
if(s) then
      x:=1
else
      y:=1
```

- Input: s=1
- Memory bus

```
    read(s, 1)
true branch:
    write (x, 1)
false branch:
    write (y, 1)
```

Purple variables are encrypted

Instructions fetched leak info

- Program

```
1: if(s) then
2:     x:=1
3: else
4:     x:=2
```

- Input: s=1
- Memory bus

  read(s, 1)

true branch:

  fetch line 2

false branch:

  fetch line 4

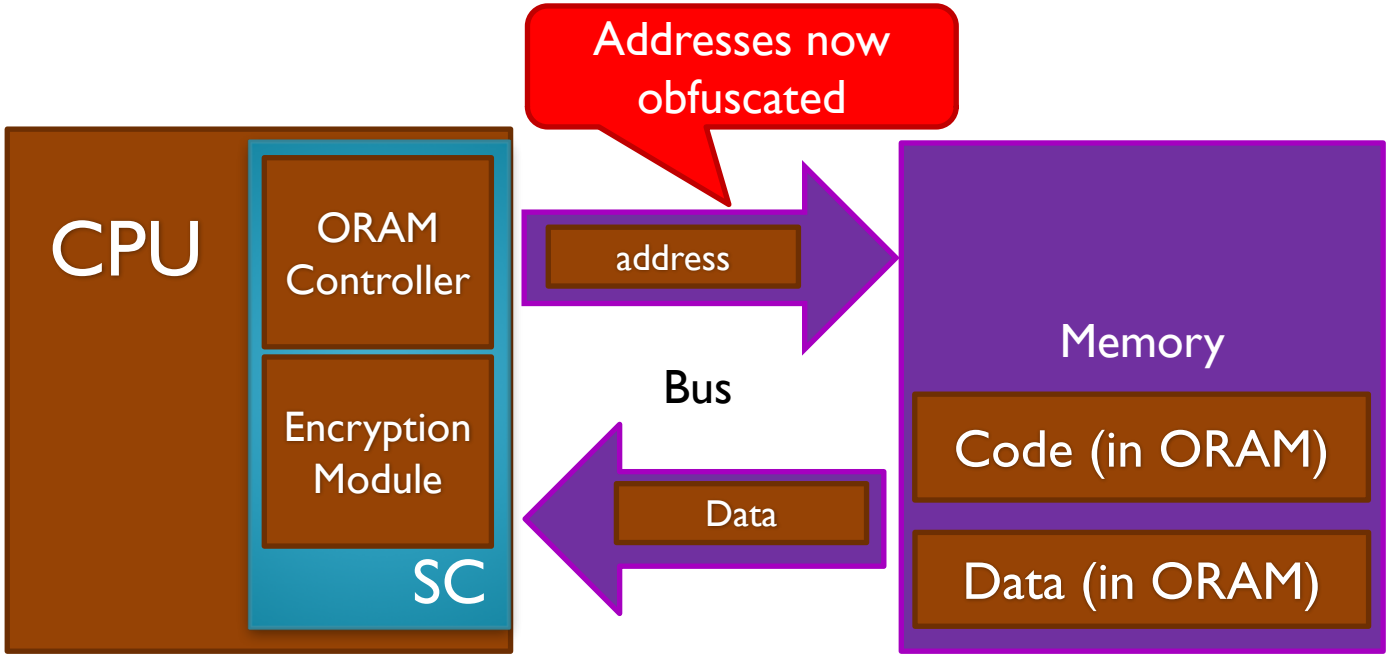
Purple variables are encrypted

Oblivious RAM (ORAM)

- A primitive to hide RAM access patterns
  - Due to Goldreich and Ostrovsky (STOC'87)
  - Reads cannot be distinguished from writes
  - The memory address cannot be distinguished

- Practical implementations now exist
  - E.g., Path ORAM (CCS'13) by Stefanov et al.

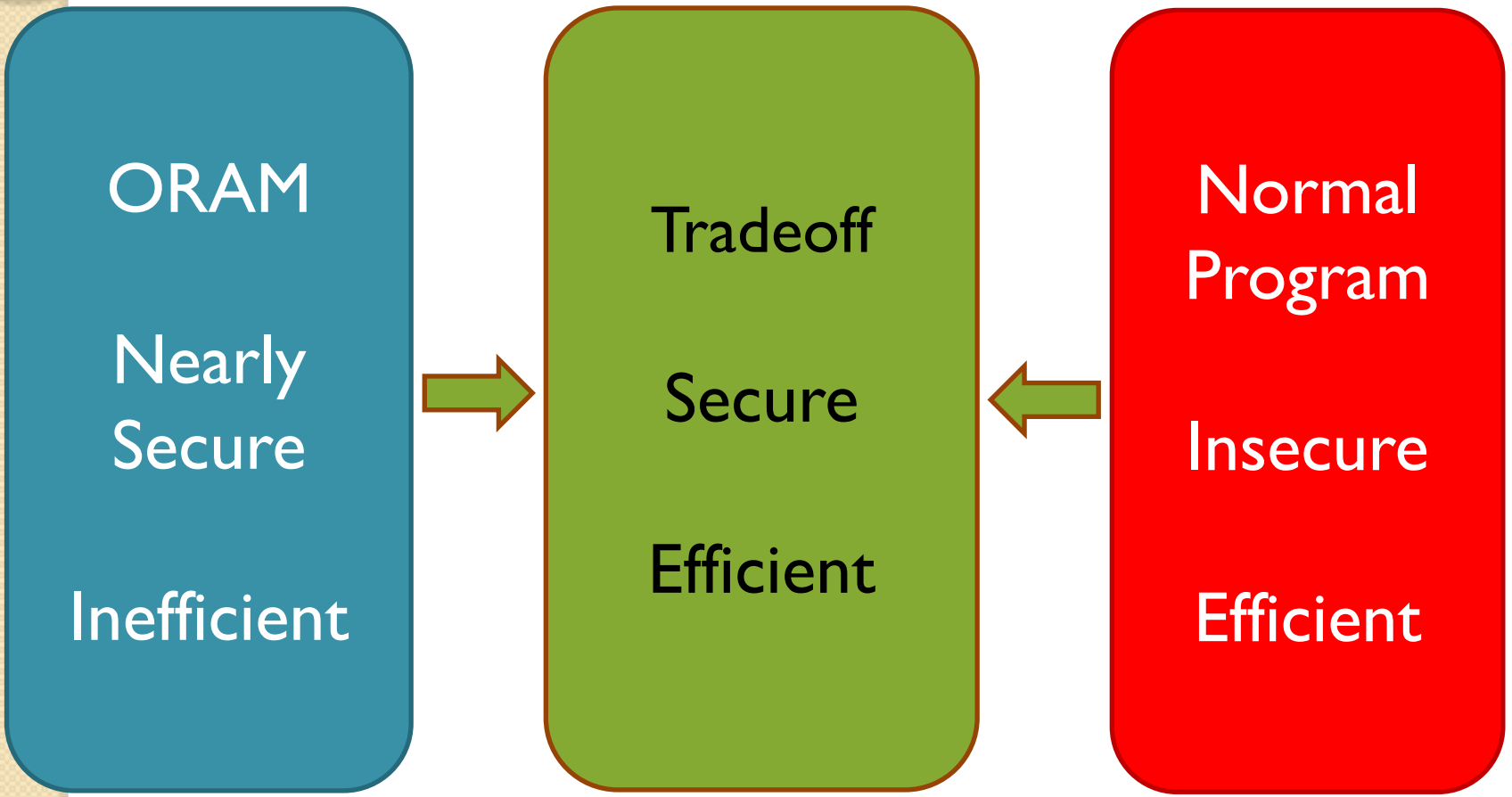
Architecture with ORAM



ORAM drawbacks

- **Performance**
  - ORAM accesses are O(poly-log N) where N is the size of the memory bank
    - Normal accesses O(1)
  - N can be large for many cloud computations
- **Security**
  - Does not protect leaks via the length of the address trace
    - "Padding out" to a fixed length is impractical

Can we do better?



PL to the rescue!

- Use program analysis (a type system) to
  - **Optimize** the use of ORAM based on the program's semantics
    - Use only where necessary, not by default
  - **Reason about** the influence of secret data on the **length of the address trace**

Results

- Formal notion of **memory-trace oblivious (MTO)** execution
  - Like noninterference, but applied to a physical attacker with access to the memory trace
- **Formal language and type system**
  - Proved that type-correct programs are MTO
- **Prototype compiler**
  - Performs transformations and type inference to establish MTO on secure programs
- **Hardware implementation**
  - Demonstrate significant speedups compared to naïve ORAM allocation

Memory Trace Oblivious Program Execution, *CSF 2013*, Winner of 2013 NSA Best Scientific Cybersecurity Paper Competition

GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation, *ASPLOS* 2015 (Best paper).

Assume: Program not secret

- Then memory access trace *may* reveal no addition information when …

- Certain data stored in DRAM
  - With encryption, or not
    - Eliminates ORAM overhead entirely
- Data stored in multiple ORAM banks
  - Reduces impact of logarithmic factor

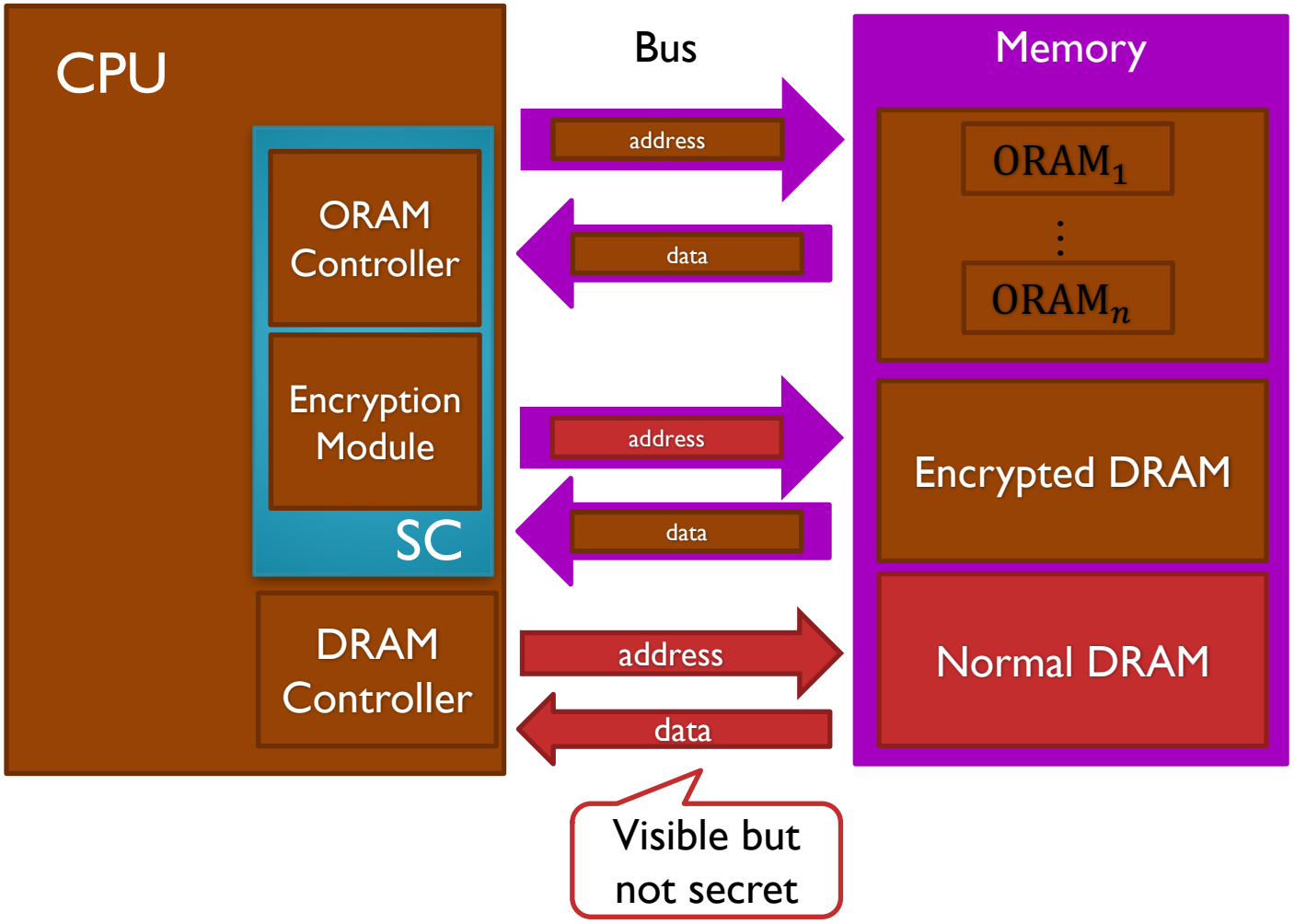
Examples

- Program 1
  a[3]:=0
- Program 2
  for i=1 to 10
  a[i]=s
- Program 3
  b[s]=1
  c[s]=2

- a[] can be stored in *normal* DRAM since it contains no secret data
- a[] stored in *encrypted* DRAM since access pattern is predictable (i is non-secret)
- b[] and c[] must be in *ORAM*, since s is secret, but can be stored in separate ORAM banks

Red variables must be stored in ORAM
Blue variables can be encrypted and stored in DRAM

Hybrid Architecture



Formal language

- Formalize simple imperative source language and its semantics

Variables	x, y, z	$\in$	**Vars**
Numbers	n	$\in$	**Nat**
ORAM bank	o	$\in$	**ORAMbanks**
Expressions	e	$::=$	$x \mid e\ op\ e \mid x[e] \mid n$
Statements	s	$::=$	$\textbf{skip} \mid x := e \mid x[e] := e \mid$ $\textbf{if}(e, S, S) \mid \textbf{while}\ (e, S)$
Programs	S	$::=$	$p{:}s \mid S; S$
Locations	p	$::=$	$n \mid o$

Big Step Semantics

$$\langle M, S \rangle \Downarrow_t M'$$

- *M* is a memory
- *S* is a program
- *t* is a trace event
- *M'* is the resulting memory

Big Step Semantics

- Semantics has two novelties
  - Variables in DRAM, and in ORAM banks
  - Execution steps may produce **events** *t*
    - *Read(x,n)* – read variable x, has value n
    - *Write(x,n)* – write n to variable x
    - *o* – read or write to ORAM bank o
    - *Fetch(p)* – fetch instruction at address p

Memory Trace Obliviousness

- *MTO:* Given a program S and two **indistinguishable** memories M1 and M2
  - Running S under M1 **produces the same trace** as running S under M2

- Def. of *memory indistinguishability*
  - All variables allocated in the same banks
  - **DRAM variables** have **same values** in both
  - **ORAM variables** may have **different values**

Type System

- Extends standard security type system with a *trace effect* T
  - Static abstraction of possible run-time trace
  - Trace effects have notion of equivalence ~
    - Only traces whose lengths we can statically reason about are potentially equivalent

- Judgment $\Gamma, I \vdash S;T$ states
  - Program S is type correct having trace effect *T* when variables have types given by Γ and in context *I* (the "pc label")

Type Rule for If

- **if (e) then S1 else S2** produces trace T under environment Γ only if
  - S1 produces trace T1
  - S2 produces trace T2
  - e mentions secret variables, or the context does, implies T1 ~ T2, in which case T = T1
    - Else T = T1+T2 ("T1 or T2")
  - (Plus some other details I'm skipping)

Type Rule for If

$$\textbf{T-Cond}\ \frac{\begin{array}{c}\Gamma \vdash e : \mathbf{Nat}\ l; T \qquad \Gamma, l \sqcup l_0 \vdash S_i; T_i\ \ (i = 1, 2) \\ l \sqcup l_0 \neq \mathbf{L} \Rightarrow T_1 \sim_L T_2 \qquad T' = \mathit{select}(T_1, T_2)\end{array}}{\Gamma, l_0 \vdash \mathbf{if}(e, S_1, S_2); T@T'}$$

- S1 produces trace T1
- S2 produces trace T2
- e mentions secret variables, or the context does, implies T1 ~ T2, in which case T = T1
  - Else T = T1+T2 ("T1 or T2")
- (Plus some other details I'm skipping)

Type Rule for Loops

- **while (e) do S** produces trace *loop(T1,T2)* under environment Γ only if
  - e produces trace T1
  - S produces trace T2
  - **e mentions no secret variables**, nor does the context

- Hence: No secret variables in loop guards
  - No loops in secure conditionals

Type Rule for Loops

$$\textbf{T-While}\ \frac{\Gamma \vdash e : \text{Nat}\ l; T_1 \qquad \Gamma, l_0 \vdash S; T_2 \qquad l \sqcup l_0 \sqsubseteq \mathrm{L}}{\Gamma, l_0 \vdash p{:}\textbf{while}(e, S); \textbf{Loop}(p, T_1, T_2)}$$

- e produces trace T1
- S produces trace T2
- **e mentions no secret variables**, nor does the context

- Hence: No secret variables in loop guards
  - No loops in secure conditionals

Controlling leaks

while (i < H) do S

- Can be rewritten to be

while (i < N) do

if (i < H) then S else *equiv(S)*

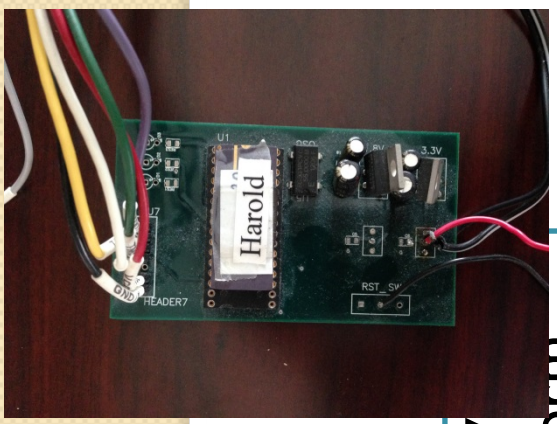
- Where
  - H is secret, but N is a public constant
  - *equiv(S)* is an inert code sequence that produces the same sequence as S

Security

- **Theorem**: If $\Gamma, I \vdash S;T$ then S is memory-trace oblivious

- Proof by standard techniques (induction on derivations)

Putting it all together



Source program

GhostRider Compiler

Optimizer

Secure Type Checker

Validate **Security**

Assembly Code

Convey Platform

Secure Processor

Scratchpad

DRAM Controller

ERAM Controller

ORAM 1 Controller

..

ORAM n Controller

RISC-V

XILINX

If rule, for assembly

$$\text{T-IF}\ \frac{\begin{array}{c} I = \iota_1; I_t; \iota_2; I_f \qquad |I_t| = n_1 - 2 \qquad |I_f| + 1 = n_2 \\ \iota_1 = \mathbf{br}\ r_1\ rop\ r_2 \hookrightarrow n_1 \qquad\qquad \iota_2 = \mathbf{jmp}\ n_2 \\ \ell' = \ell \sqcup \Upsilon(r_1) \sqcup \Upsilon(r_2) \\ \ell' \vdash I_t : \langle \Upsilon, Sym \rangle \to \langle \Upsilon', Sym' \rangle; T_1 \\ \ell' \vdash I_f : \langle \Upsilon, Sym \rangle \to \langle \Upsilon', Sym' \rangle; T_2 \\ \ell' = \mathtt{H} \Rightarrow \left\{ \begin{array}{l} T_1 @ \mathbf{F} \equiv T_2 \wedge \\ \ell = \mathtt{L} \Rightarrow \vdash_{const} Sym \wedge \\ \forall r.\, \Upsilon'(r) = \mathtt{L} \Rightarrow \vdash_{safe} Sym'(r) \end{array} \right\} \\ T = ite(\ell' = \mathtt{H},\ \mathbf{F}@T_1@\mathbf{F},\ \mathbf{F}@((T_1@\mathbf{F}) + T_2)) \end{array}}{\ell \vdash I : \langle \Upsilon, Sym \rangle \to \langle \Upsilon', Sym' \rangle; T}$$

While rule, for assembly

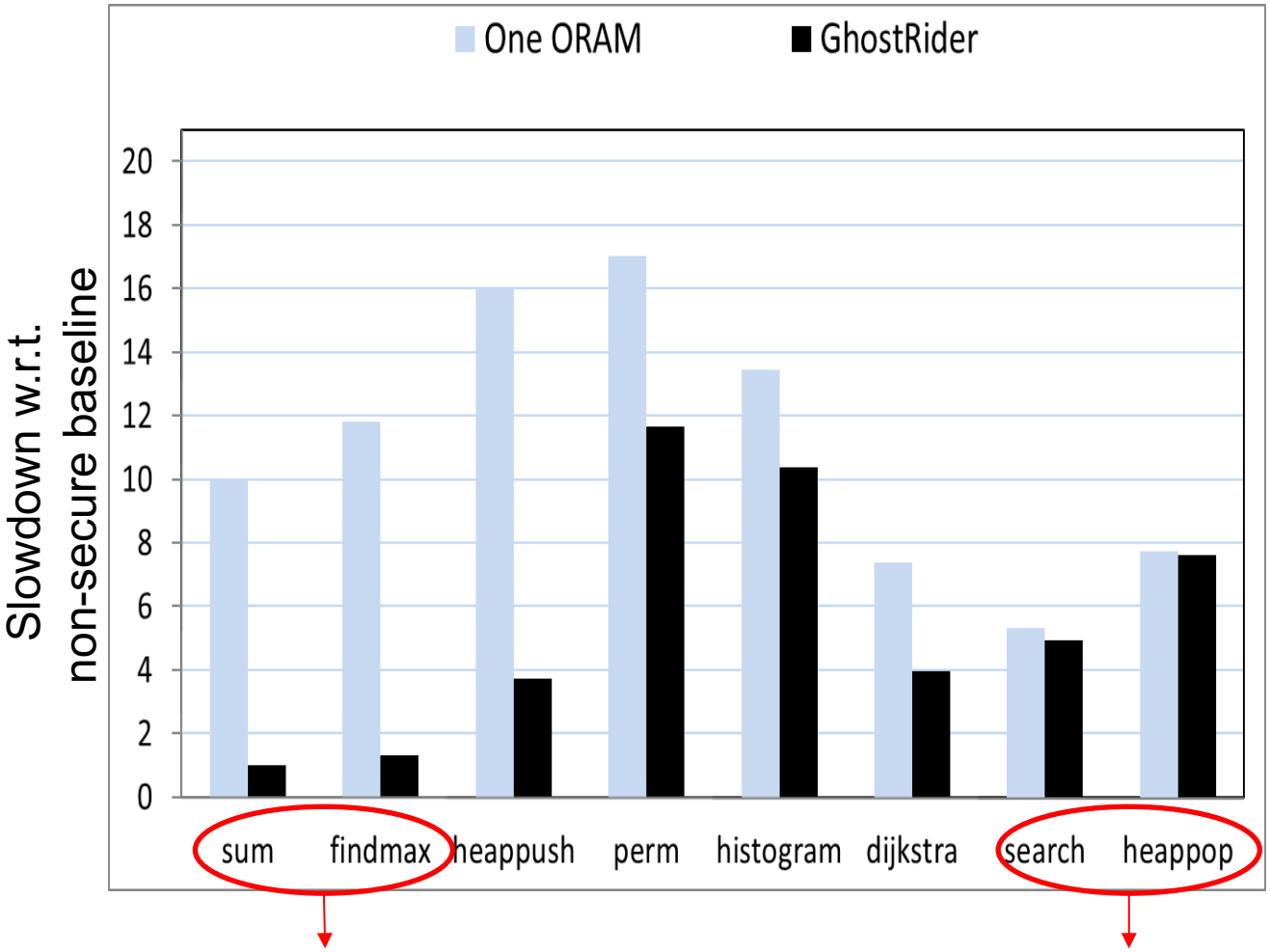
$$\textsc{T-LOOP}\ \frac{\begin{array}{c} I = I_c; \iota_1; I_b; \iota_2 \\ |I_b| = n_1 - 2 \qquad |I_c| + n_1 = 1 - n_2 \\ \iota_1 = \mathbf{br}\ r_1\ rop\ r_2 \hookrightarrow n_1 \qquad \iota_2 = \mathbf{jmp}\ n_2 \\ \ell \sqcup \Upsilon(r_1) \sqcup \Upsilon(r_2) \sqsubseteq \mathtt{L} \\ \ell \vdash I_c : \langle \Upsilon, Sym \rangle \to \langle \Upsilon', Sym' \rangle; T_1 \\ \ell \vdash I_b : \langle \Upsilon', Sym' \rangle \to \langle \Upsilon, Sym \rangle; T_2 \end{array}}{\ell \vdash I : \langle \Upsilon, Sym \rangle \to \langle \Upsilon', Sym' \rangle; \mathbf{loop}(T_1, T_2)}$$

FPGA-based evaluation

Non-Secure Baseline	All data in DRAM
One ORAM	All data in one ORAM
GhostRider	Optimized version utilizing the hybrid memory model

- We compare **One ORAM** and **GhostRider** to see the improvement from the compiler
  - measure the overhead of **One ORAM** and **GhostRider** over the non-secure baseline

Result: up to 10x faster than one ORAM



Lower bar ⇒ better performance

Little overhead over non-secure baseline for some programs

For programs whose memory trace patterns **heavily depend on the input**, speedup is small

Opening the Black Box

- Ghostrider treats ORAM as a primitive
  - Part of the trusted computing base (TCB)
  - Full flexibility costs some performance
    - Supports random access
- We are now working on a language in which ORAM can be implemented
  - Less to trust
  - Permits efficient oblivious data structures
  - Key feature: Proper use of randomness
  - http://www.cs.umd.edu/~mwh/papers/darais17obliv.html

Summary

- Goal: **Outsourced computation** that is private despite physical snooping
  - And gets good performance
- Solution: **Memory-trace Oblivious program execution** enabled by Program analysis and Oblivious RAM
  - HW/SW co-design
  - Designed/validated using formal semantics of PL: type systems and operational semantics

Next: ADS

- (switch decks)

Other PL/crypto connections

- Techniques (or PLs) for analyzing/verifying implementations of crypto
  - E.g., side-channel freeness
- Verified crypto-style proofs of security
  - Adopt PL ideas of contextual equivalent and full abstraction
  - Using PL-style automation (theorem proving)
- Real/ideal paradigm of security
  - Matches specification/implementation in PL

Further reading

- The Synergy between Programming Languages and Cryptography
  - http://www.pl-enthusiast.net/2014/12/17/synergy-programming-languages-cryptography/
- Formal Reasoning in PL and Crypto
  - http://www.pl-enthusiast.net/2014/12/23/formal-reasoning-pl-crypto/
- What is PL research and how is it useful?
  - http://www.pl-enthusiast.net/2015/05/27/what-is-pl-research-and-how-is-it-useful/

Broader Message

- Crypto is cool, PL is cool
- Each has something to offer the other:
  - Result is better than both!

http://www.dagstuhl.de/en/program/calendar/semhp/?semnr=14492

- Other related work from our group
  - PL-optimized secure multiparty computation
    - ASPLOS'15, IEEE S&P'14 x 2, PLAS'13, PLAS'12
    - and works in progress