

Memory Trace Oblivious Program Execution for Cloud Computing

Combining **PL**, **Crypto**, **Architecture** Research

Three great tastes that go great together

Chang Liu

With Michael Hicks, Elaine Shi,
Austin Harris, Martin Maas, and Mohit Tiwari



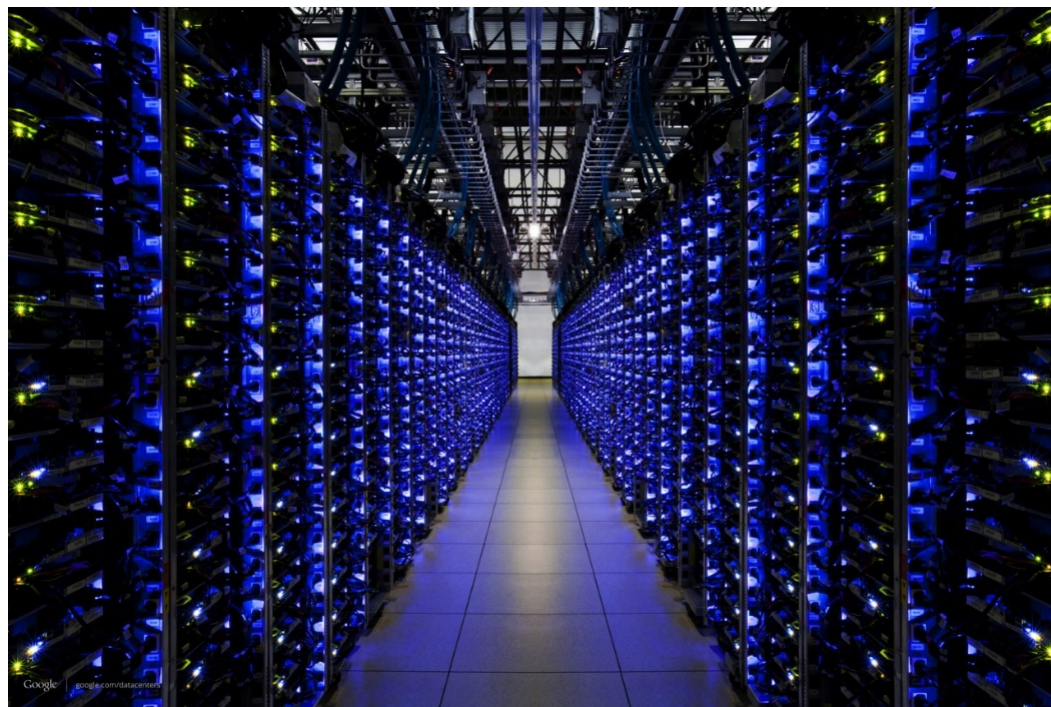


Cloud computing raises **privacy concerns** for sensitive data

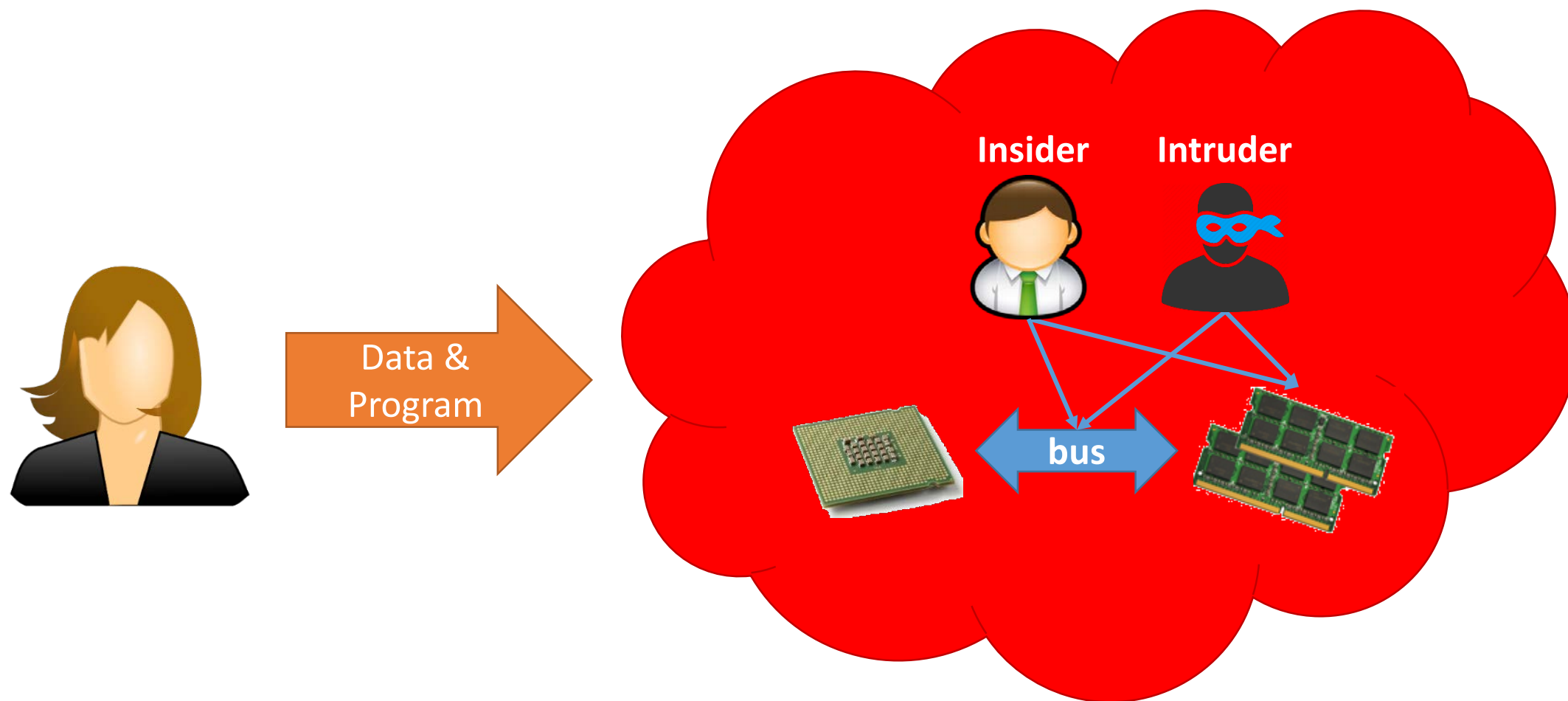
Financial
Medical
Government
etc.



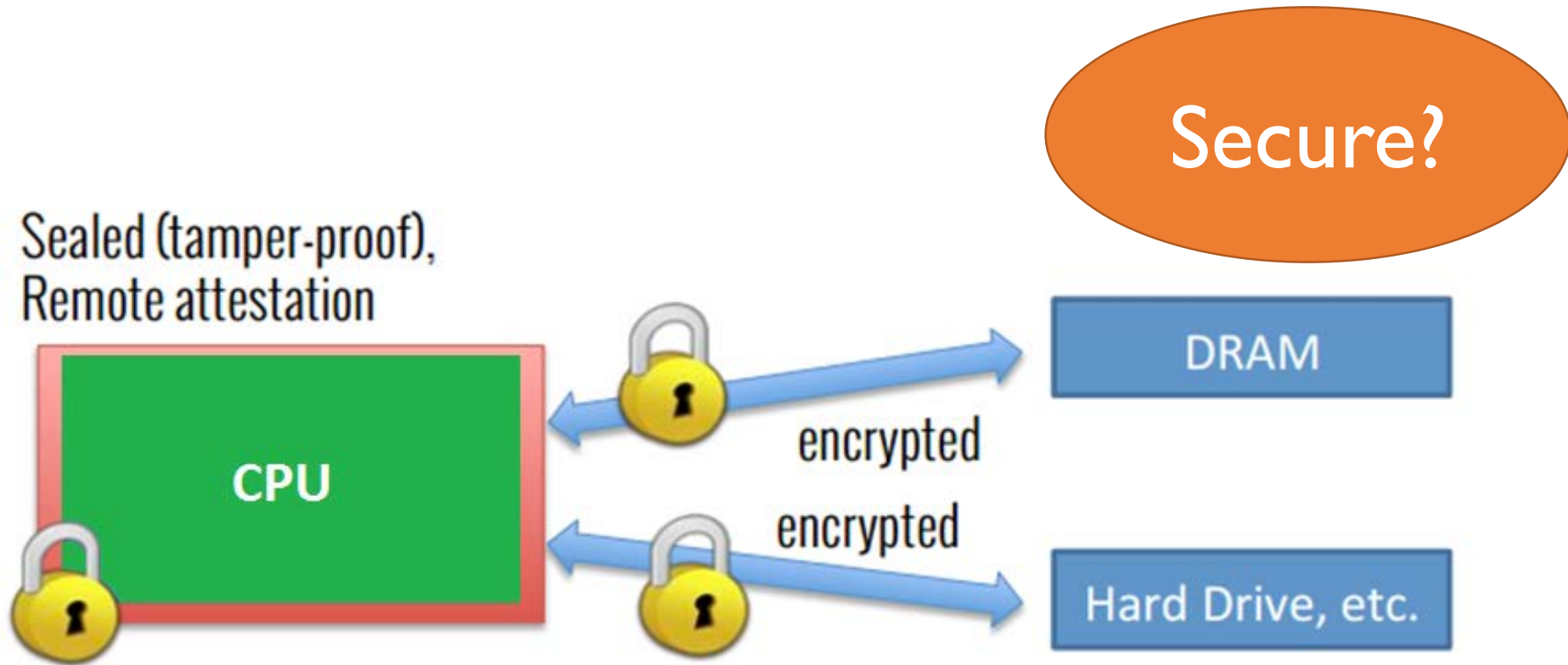
Run analysis
over the
sensitive data



Malicious insiders or intruders may perform **physical attacks** to snoop sensitive data

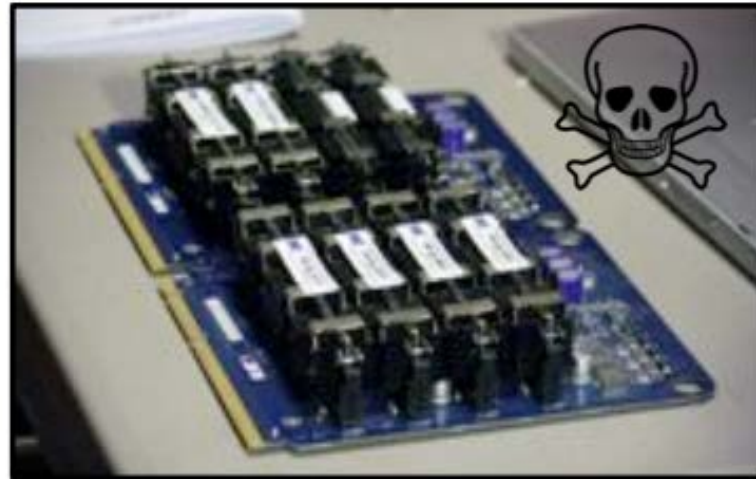


Solution 1: Secure processors **encrypt memory**



- e.g. Secure Processors (AEGIS, XOM, AISE-BMT), IBM Cryptographic Coprocessors, Intel SGX

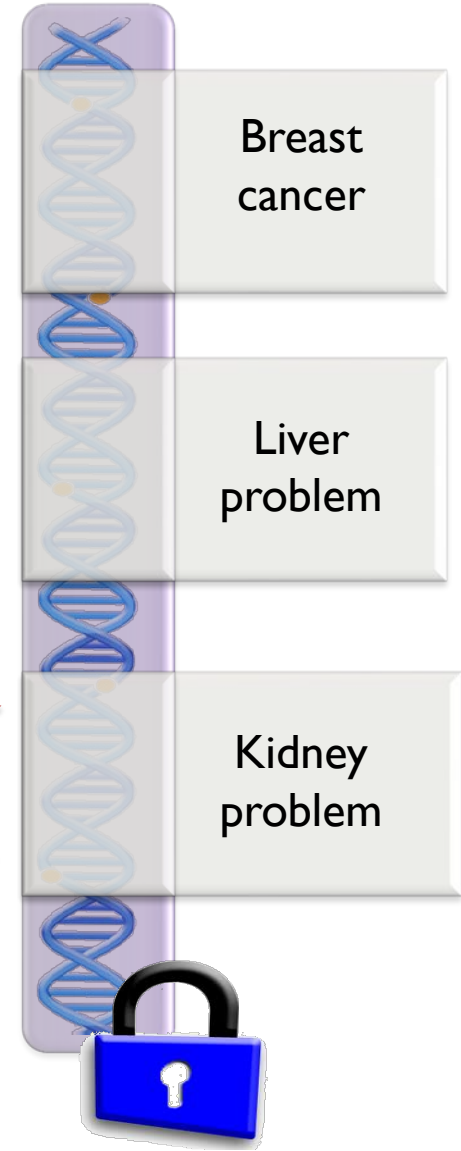
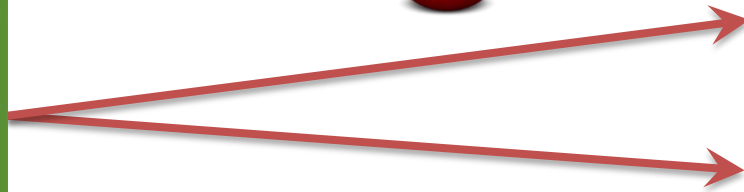
NO! It is easy to learn **memory access patterns** through **physical attacks**



- E.g. replace **DRAM DIMMs** with **NVDIMMs** that have non-volatile storage to record accesses

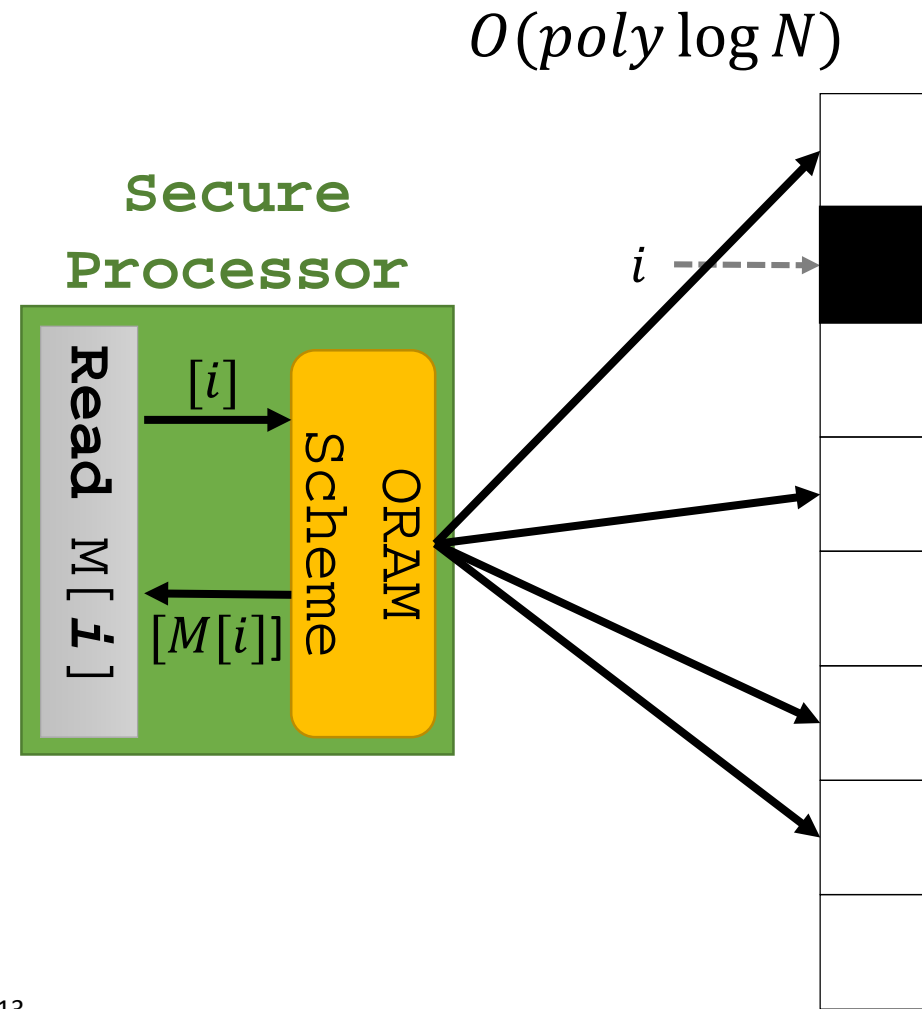
Problem: **Access patterns** to even encrypted data leak sensitive information.

Secure processor



Crypto tool: Oblivious RAM

- *Hide access patterns*
 - *Redundancy*
 - *Data Shuffling*
- *Poly-logarithmic cost per access*



[Shi, et al., 2011] Oblivious RAM with $O((\log N)^3)$ Worst-Case Cost. In ASIACRYPT 2011.

[Stefanov et al., 2013] Path ORAM: An extremely simple oblivious RAM protocol. In CCS 2013

[Maas, et al., 2013] Phantom: Practical oblivious computation in a secure processor. In CCS 2013.

ORAM-capable Secure Processor

1. Somewhat practical, but still **moderately expensive**
2. Timing and termination channels leak information



**Given a computation (C program),
what data (variables) do we place
inside an ORAM?**

Naïve answer: *all* of them

Key observation:

**Accesses that do not depend on secret
inputs need not be hidden**

Example: FindMax

```
int max(public int n, secret int h[]) {  
    public int i = 0;  
    secret int m = 0;  
    while (i < n) {  
        if (h[i] > m) then m = h[i];  
        i++;  
    }  
    return m;  
}
```

h[] need not be in ORAM.
Encryption suffices.

Dynamic Memory Accesses:

Main loop in Dijkstra

```
for(int i=1; i<n; ++i) {  
    int bestj = -1;  
    for(int j=0; j<n; ++j)  
        if(!vis[j] && (bestdis < 0 || dis[j] < bestdis))  
            bestdis = dis[j];  
  
    vis[bestj] = 1;  
    for(int j=0; j<n; ++j)  
        if(!vis[j] && (bestdis + e[bestj][j] < dis[j]))  
            dis[j] = bestdis + e[bestj][j];  
}
```

dis[]: Not in ORAM

vis[], e[][]: Inside ORAM



What programs leak information?

- $a[x] := s$

Array index leaks secret variable

- 1: $\text{if}(s)$ then

- 2: $x := 1$

- 3: else

- 4: $y := 2$

Secret ifs leak information through **variables accessed** and **instructions fetched**

How can PL help here?

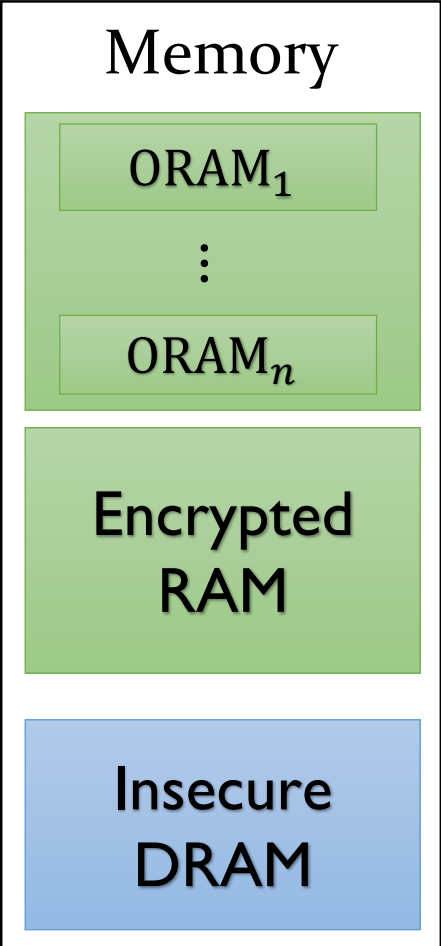
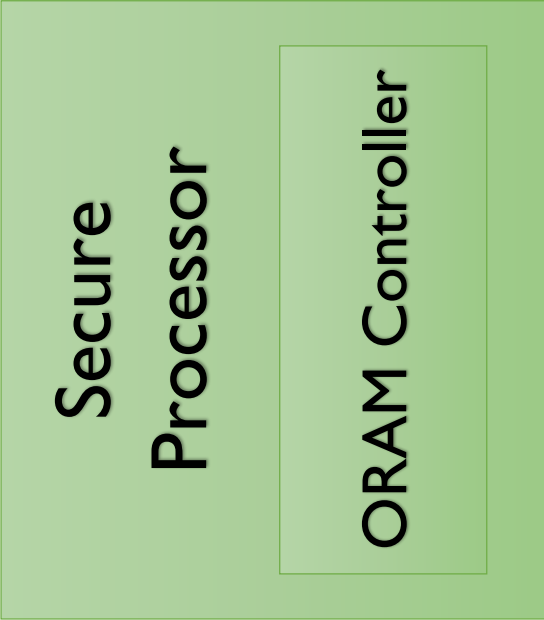
Our compiler **automates** this analysis

- Recognize code whose ***access patterns do not leak information***
- Minimize the usage of ORAM

Formal security

- **Memory-trace oblivious** type system

Hybrid Architecture



Observable trace

ORAM: Bank ID

ERAM: address

DRAM: address+data



Memory Trace Obliviousness

*How can we design a **type system** for enforcing MTO?*

Challenge: conditionals and loops

Type System: Rule for If

```
int findmax(public int n, secret int[] h) {  
  1: max:=h[0];  
  2: i:=1;  
  3: while(i<n)  
  4:   if(h[i]>max) then  
  5:     max:=h[i] → fetch line 5, read i, read h, write to max  
  6:   else  
  7:     skip → fetch line 6, do nothing  
  8:   i:=i+1;  
  9: return max  
}
```

if-guard mentions **secret variable**



both branches have **equivalent traces**

Type System: Padding for If Rule

```
int findmax(public int n, secret int[] h) {  
  1: max:=h[0];  
  2: i:=1;  
  3: while(i<n)  
  4:   if(h[i]>max) then  
  5:     max:=h[i]  
  6:   else  
  7:     dumm:=h[i]  
  8:   i:=i+1;  
  9: return max  
}
```

fetch *b*, read i, access h, access *a*

- Padding
 - dumm and max in the same ORAM *a*
- Place both instructions (Line 5 and Line 6) in the same ORAM *b*



Type System: Rule for Loops

```
int findmax(public int n, secret int[] h) {  
  1: max:=h[0];  
  2: i:=1;  
  3: while(i<n)  
  4:   if(h[i]>max) then  
  5:     max:=h[i]  
     else  
  6:     dumm:=h[i]  
  7:   i:=i+1;  
  8: return max  
}
```

To prevent information leakage through **the number of loop iterations**

No secret variables in loop guards



Controlling leaks

Given **secret H**, **public N**

while (i < H) do S

⇒

while (i < N) do

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

equiv(S): padding instructions that produce the same trace as S

Security

- **Theorem (informally):** If a program P type-checks, then P is memory-trace oblivious
- Proof by standard PL techniques (progress and preservation)

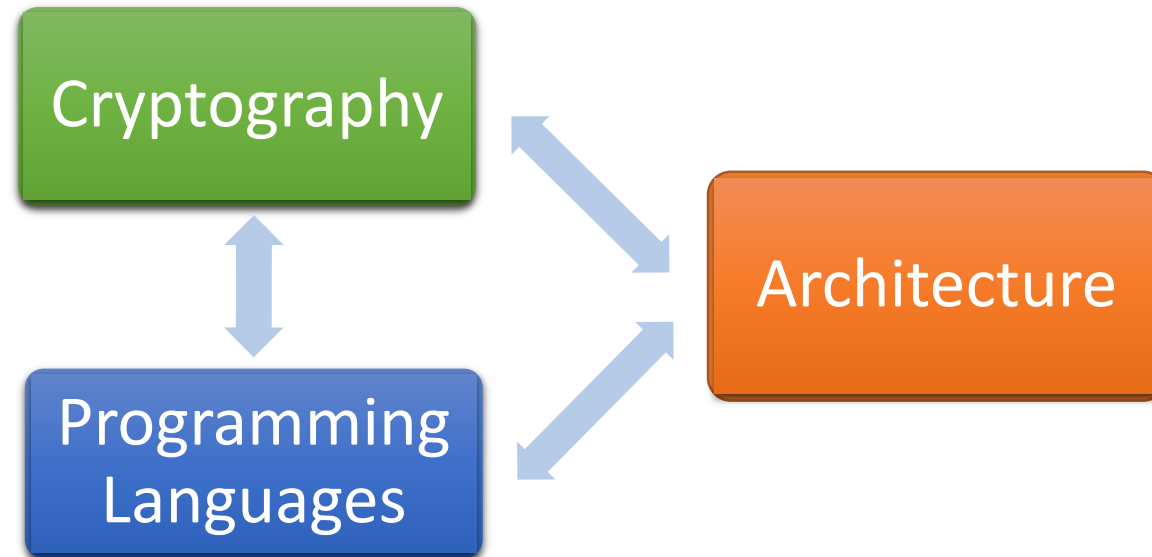


Additional Challenges

- Function calls inside secret ifs
 - Partially solved in our latest work [LWNHS-IEEE S&P '15]
- Pointers and memory allocations
 - Oblivious memory allocation algorithms proposed in [WNLCSH-CCS '14]

Roadmap

- So far: Memory-trace oblivious type system
- **Next: Implementation on a real processor**



[LHMHTS- ASPLOS 2015] **GhostRider**: A Hardware-Software System for Memory Trace Oblivious Computation. *Best Paper Award*.

Challenge I: Cache Channel

Implicit cache may make MTO programs **NOT MTO**

- Program

b[0] := 0

if(s) then

 a[0] := 1

b[0] := 2

else

 a[0] := 1

 b[1000] := 2

The true branch will have **only one memory accesses** because of the cache!



Problem: previous type system is not aware of cache!

Question: How to model cache behavior in the type system?

If hardware has **implicit caching** behavior \Rightarrow Very **HARD** to predict

Solution: **hardware-compiler co-design**

- 1) Modify hardware to expose knobs to control **scratchpad**
- 2) Explicitly model the scratchpad behavior in the type system

Not Too Slow After Using Scratchpad

- Program-implemented cache using scratchpad
- $y := a[i]$
 - a is placed in ERAM, and use scratchpad block k_1
- Compute the block id to be
$$t_1 \leftarrow \frac{r_i}{size_{blk}}$$
- If $t_1 = blk_{id}(k_1)$, then retrieve $k_1 \leftarrow ERAM[t_1]$
- Retrieve $k_1[r_i \bmod size_{blk}]$

Challenge II: Timing Channel

- Program

$b[0] := 0$

if(s) then

$a[0] := 1$

$b[0] := 2$

else

$a[0] := 1$

$b[1000] := 2$


The true branch runs **faster** than the false branch, since it makes less ORAM accesses

Challenge II: Timing Channel

- Program
if(*s*) then
 x := *y* + *z*;
else
 x := *y* * *z*;

The true branch runs **faster** than the false branch, since multiplication takes longer time than addition

Solution: Deterministic Timing



Challenge III: The type system need deal with *assembly code*

• SOLUTION

- The type system keeps track of *trace patterns*
- In trace patterns, instead of actual value, the type system keeps track of *symbolic values*
- To deal with *branching instructions*, the type system allows a limited form of code patterns containing branching
 - only allowed in *IF-code pattern* and *LOOP-code pattern*

MTO for L_T

- $y := a[x]$
 - a is placed in ERAM

$t_1 \leftarrow r_x \mathbf{div} \mathit{size}_{blk}$
 $t_1 \leftarrow t_1 + \mathit{startblk}_a$
 $t_2 \leftarrow r_x \mathbf{mod} \mathit{size}_{blk}$
ldb $k_1 \leftarrow E[t_1]$
ldw $r_y \leftarrow k_1[t_2]$

- Input: $x = 513$ (secret input)
 - Assume $\mathit{size}_{blk} = 512$

fetch
fetch
fetch
eread(1)

Depending on x !

MTO for L_T

- $y := a[x]$
 - a is placed in an ORAM o

$t_1 \leftarrow r_x \text{ div } size_{blk}$
 $t_1 \leftarrow t_1 + startblk_a$
 $t_2 \leftarrow r_x \text{ mod } size_{blk}$
ldb $k_1 \leftarrow o[t_1]$
ldw $r_y \leftarrow k_1[t_2]$

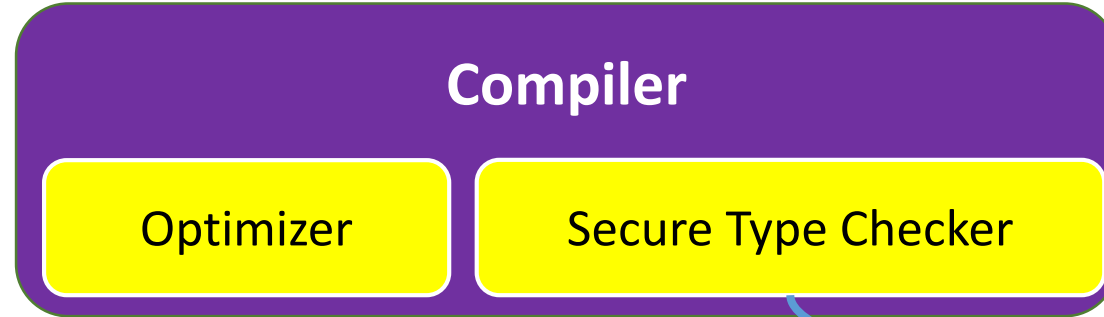
fetch
fetch
fetch
o
fetch

- Input: $x = 513$ (secret input)
 - Assume $size_{blk} = 512$

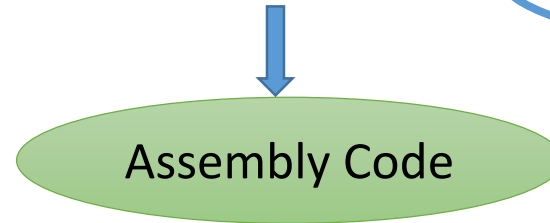
✓ **Memory Trace Oblivious**

GhostRider: Putting it all together

C
program

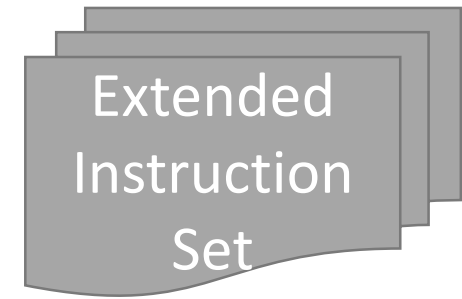
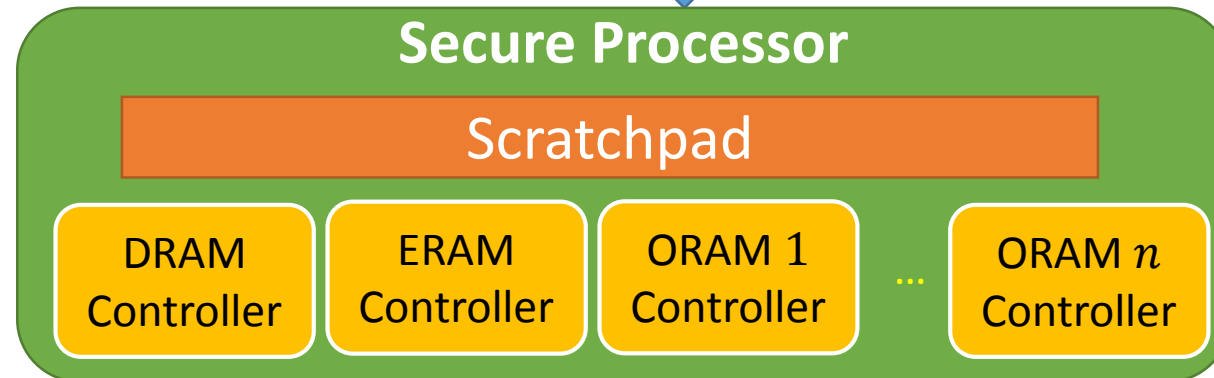


Formally Enforce **MTO**



Security guarantee

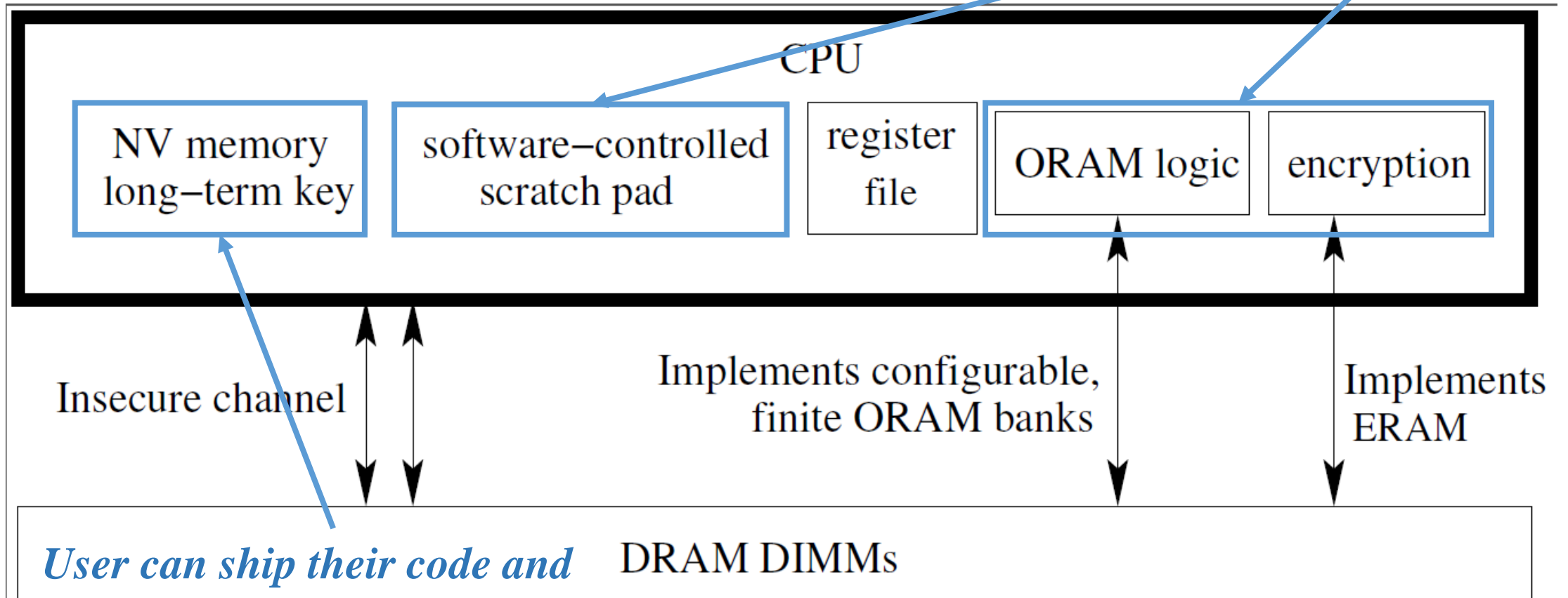
MTO ⇒ {
Cache Channel
Timing Channel
Termination Channel



Architecture Overview

Software-controlled scratchpad to replace ORAM-ERAM implicit cache memory system

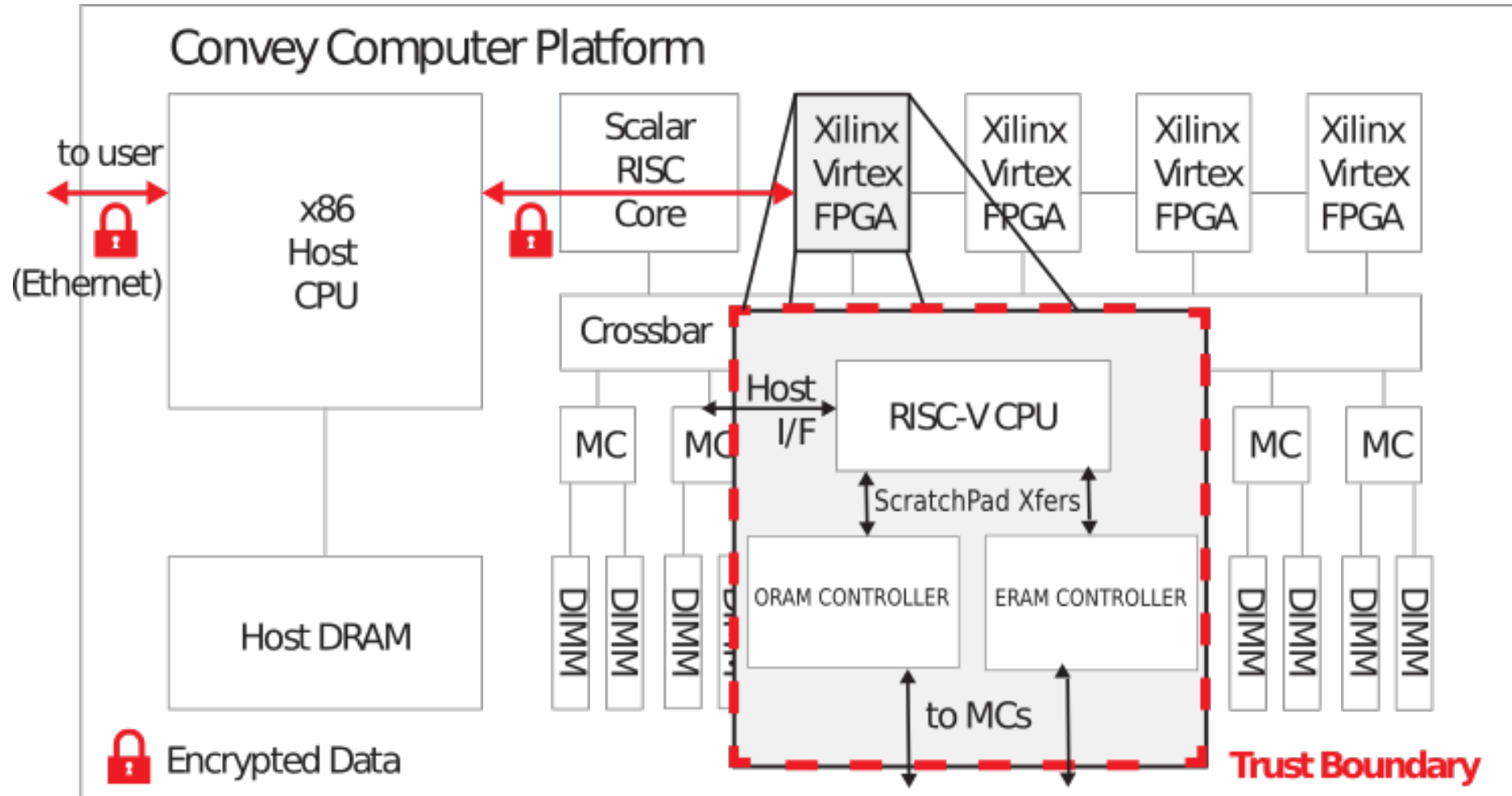
Instructions have deterministic timings



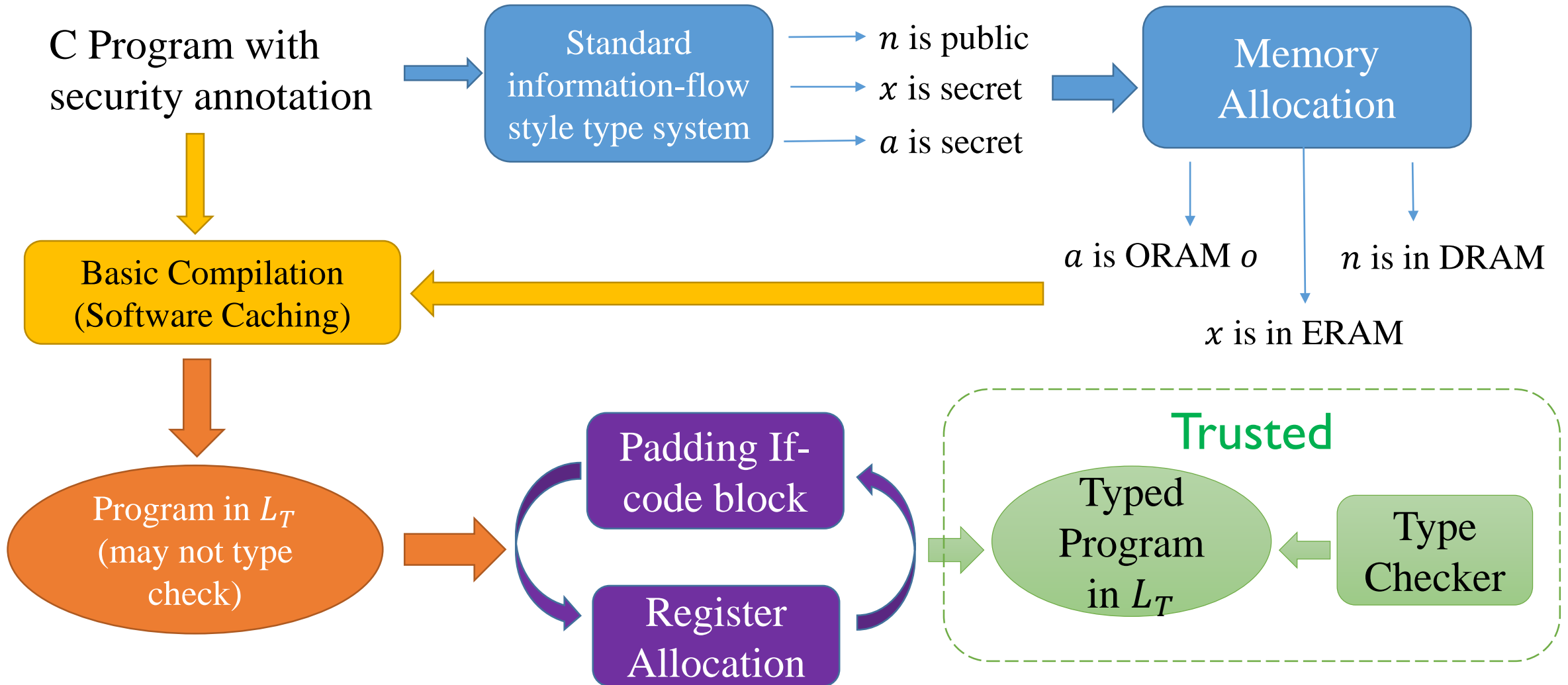
User can ship their code and data securely using standard method.

DRAM DIMMs

FPGA Implementation



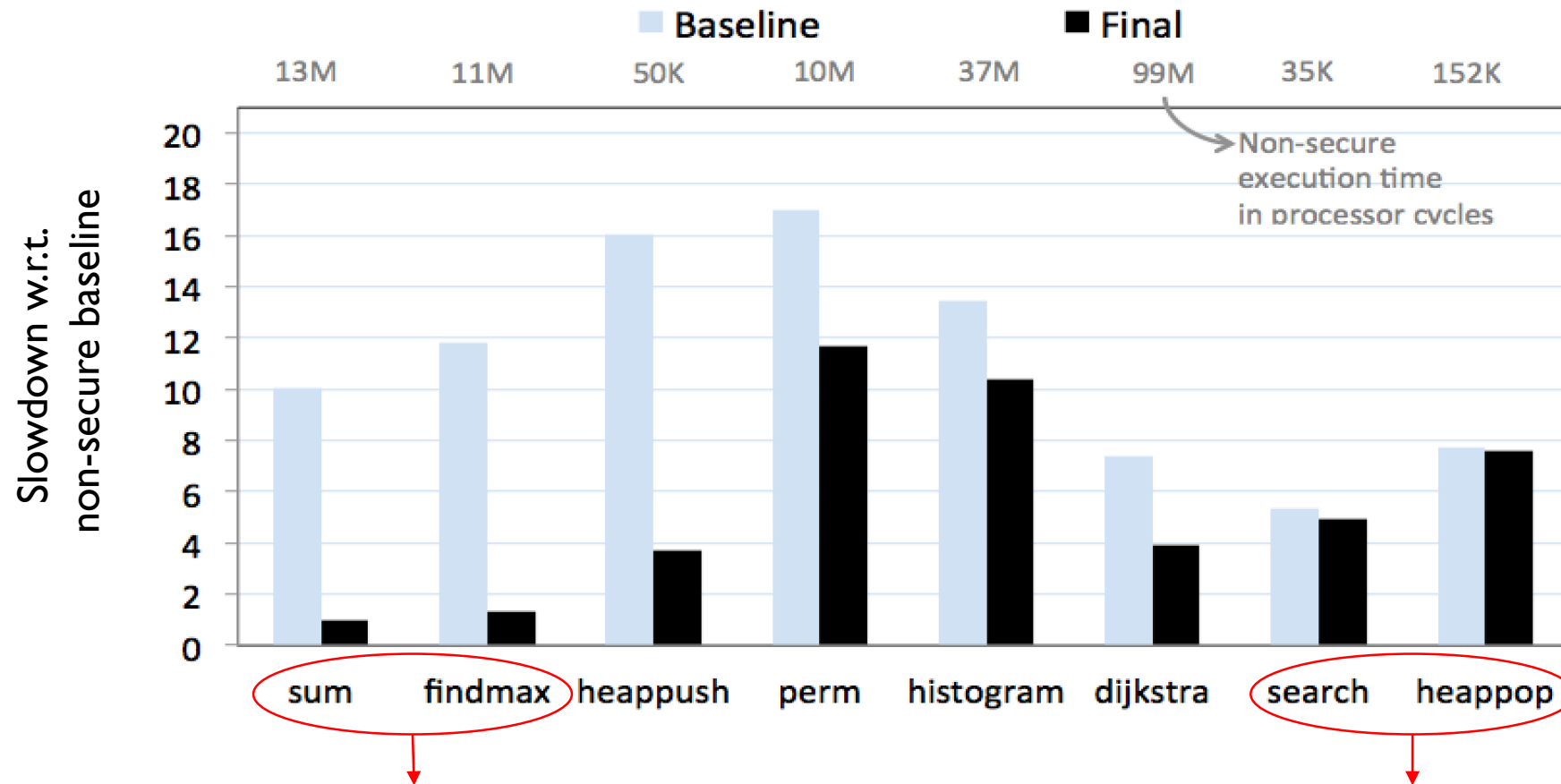
Compiler Implementation





FPGA Evaluation

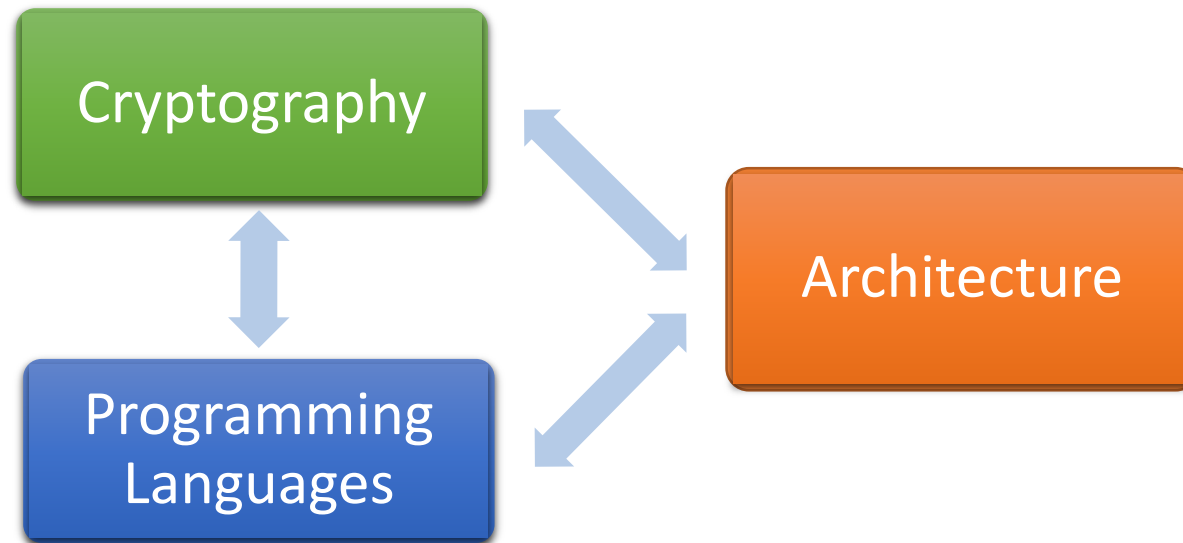
up to **8.94 × faster** than baseline



Little overhead over non-secure baseline for some programs

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

Memory-trace oblivious compiler + GhostRider processor enable practical outsourcing secure against physical attacks



- The **work continues**: relaxed adversary model, support larger programs

Other Applications of Trace Obliviousness

ObliVM: Trace Oblivious Program Execution for *Secure Computation*

- www.oblivm.com
- [LHSKH-IEEE S&P '14, LWNHS-IEEE S&P '15]

More in progress

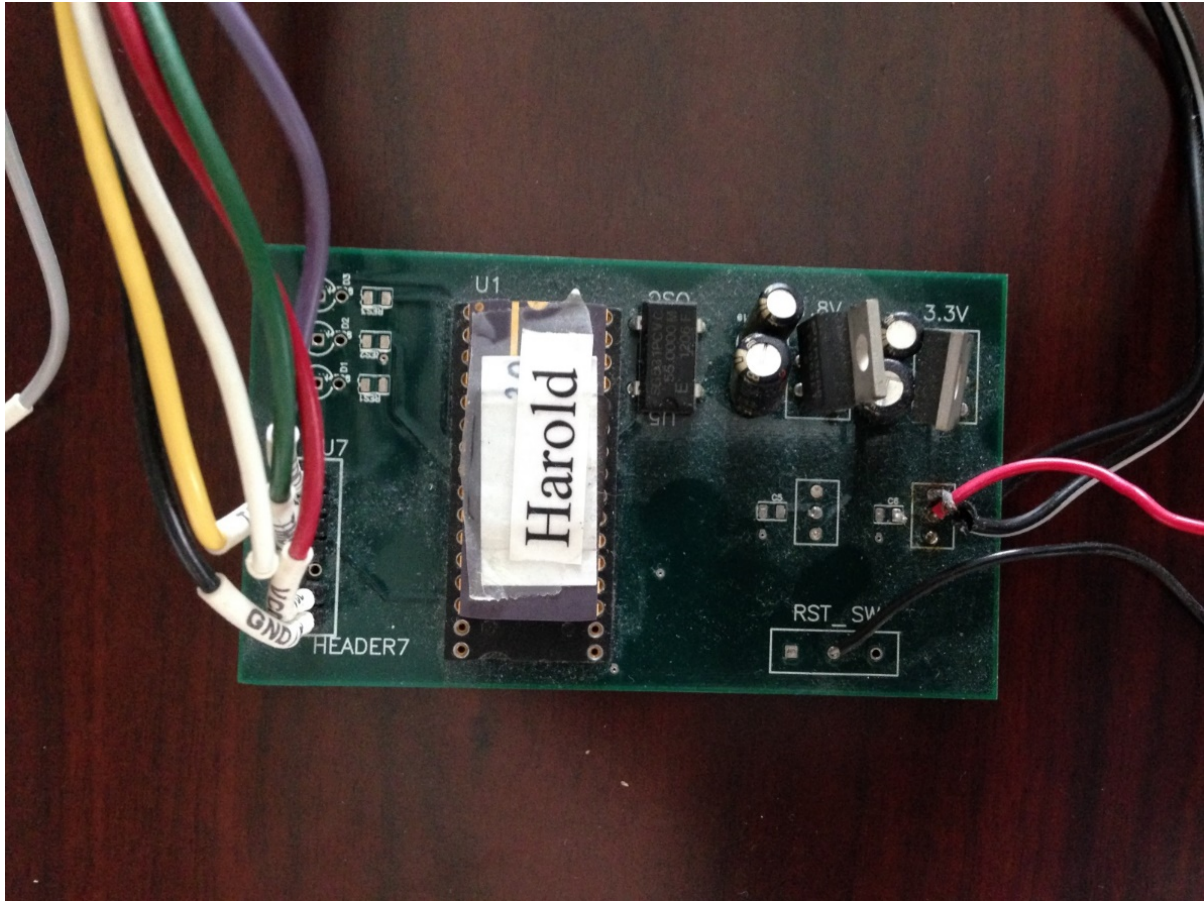
[LHSKH-IEEE S&P '14] Automating RAM-model Secure Computation, In IEEE S&P 2014

[LWNHS-IEEE S&P '15] ObliVM: A Programming Framework for Secure Computation, In IEEE S&P 2015

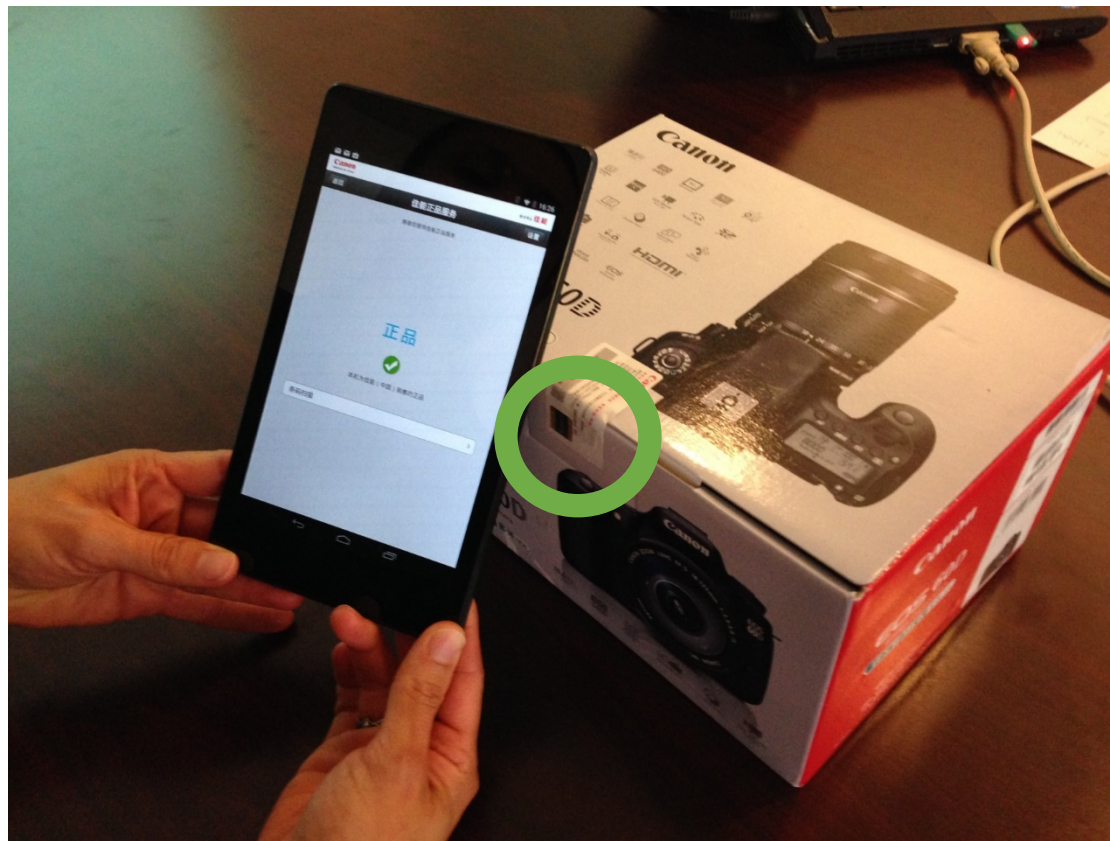


MIT, 2002, Devadas et al.

Success Story: PUF 13 Years Ago



Success Story: PUF Today





Looks like this



**ORAM-capable
secure processor
today**

Where will ORAM be in 2028?