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

Data & Program
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.**
Crypto tool: Oblivious RAM

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

\[ O(\text{poly log } N) \]

[Stefanov et al., 2013] Path ORAM: An extremely simple oblivious RAM protocol. 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
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];
}
```
What programs leak information?

- $a[x]:=s$
  - **Array index** leaks secret variable

- 1: 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

[**LHS-CSF 2013**] Memory Trace Oblivious Program Execution, In CSF 2013, NSA Best Scientific Cybersecurity Paper Award
Hybrid Architecture

- Secure Processor
- ORAM Controller
- Memory
  - ORAM\textsubscript{1}
  - \vdots
  - ORAM\textsubscript{n}
- Encrypted RAM
- Insecure DRAM

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
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:   skip;
        7:   i := i + 1;
    }
    8: return max
}
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:         dumm:=h[i]
  7:     else
  8:         dumm:=h[i]
  9:     i:=i+1;
10: return max
}
**Type System: Rule for Loops**

```c
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: dummy:=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 $S$

$\Rightarrow$

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 [WNLCSSH-CCS ’14]

[WNLCSSH-CCS ’14] Oblivious Data Structures, In CCS 2014
Roadmap

• So far: Memory-trace oblivious type system

• Next: Implementation on a real processor

Implicit cache may make MTO programs NOT MTO

• Program
  \[ b[0] := 0 \]
  \[
  \text{if}(s) \text{ then} \]
  \[ a[0] := 1 \]
  \[ b[0] := 2 \]
  
  \[
  \text{else} \]
  \[ a[0] := 1 \]
  \[ b[1000] := 2 \]

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 ⇒ 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

- \( y = a[i] \)  

- 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 \( k_1[r_i \mod size_{blk}] \)

- \( a \) is placed in ERAM, and use scratchpad block \( k_1 \)
Challenge II: Timing Channel

- Program

\[
\begin{align*}
&b[0] := 0 \\
&\text{if}(s) \text{ then} \\
&\quad a[0] := 1 \\
&\quad b[0] := 2 \\
&\text{else} \\
&\quad a[0] := 1 \\
&\quad b[1000] := 2
\end{align*}
\]

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
    \[ x := y \times z; \]

The true branch runs \textbf{faster} than the false branch, since multiplications 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 \ \text{div} \ size_{blk} \\
t_1 \leftarrow t_1 + startblk_a \\
t_2 \leftarrow r_x \ \text{mod} \ size_{blk} \\
\text{ldb} \ k_1 \leftarrow E[t_1] \\
\text{ldw} \ r_y \leftarrow k_1[t_2]
\]

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

- fetch
- fetch
- fetch
- eread(1)

Depending on \( x \)!
MTO for $L_T$

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

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

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

✓ Memory Trace Oblivious
GhostRider: Putting it all together

- Compiler
  - Optimizer
  - Secure Type Checker
- Assembly Code
- Secure Processor
  - Scratchpad
  - DRAM Controller
  - ERAM Controller
  - ORAM 1 Controller
  - ORAM n Controller
- Formally Enforce MTO

Security guarantee

MTO ⇒ Cache Channel, Timing Channel, Termination Channel

Extended Instruction Set
Joint ORAM-ERAM memory system

Software-controlled scratchpad to replace an implicit cache memory system

Instructions have deterministic timings

CPU

NV memory long-term key

software-controlled scratch pad

register file

ORAM logic

encryption

Insecure channel

Implements configurable, finite ORAM banks

Implements ERAM

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

DRAM DIMMs
FPGA Implementation
Compiler Implementation

C Program with security annotation

Basic Compilation (Software Caching)

Program in $L_T$ (may not type check)

Standard information-flow style type system

Padding If-code block

Register Allocation

Memory Allocation

Typed Program in $L_T$

Type Checker

trusted
FPGA Evaluation
up to $8.94 \times$ 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](http://www.oblivm.com)

More in progress

Success Story: PUF 13 Years Ago

MIT, 2002, Devadas et al.
Success Story: PUF Today
Where will ORAM be in 2028?

 Looks like this

 ORAM-capable secure processor today