HOP: Hardware makes Obfuscation Practical

Kartik Nayak

With Chris Fletcher, Ling Ren, Nishanth Chandran, Satya Lokam, Elaine Shi and Vipul Goyal
Used by everyone, perhaps license it

No one should “learn” the algorithm - VBB Obfuscation

Another scenario: Release patches without disclosing vulnerabilities
Known Results

Heuristic approaches to obfuscation [KKNVT’15, SK’11, ZZP’04]

Impossible to achieve program obfuscation in general [BGIRSVY’01]
Weaker Notion of Obfuscation

Indistinguishability Obfuscation (iO) is Achievable [BGIRSVY’01]
  Construction via multilinear maps [GGHRSW’13]
    - Not strong enough for practical applications
    - Non-standard assumptions
    - Inefficient

16-bit point function [AHKM’14]
  Obfuscation: ~6.5 hours
  Evaluation: ~11 minutes
  32-core machine, 41 GB RAM
  52 bits of security

point_func(x) {
  if x == secret
    return 1;
  else return 0;
}
Using Trusted Hardware Token

Program obfuscation, Functional encryption using stateless tokens
[GISVW’10, DMMN’11, CKZ’13]

- Boolean Circuits
- Token functionality program dependent
- Inefficient - using FHE, NIZKs
- Sending many tokens
Work on Secure Processors

Intel SGX, AEGIS [SCGDD’03], XOM [LTMLBMH’00]: encrypts memory, verifies integrity
  - reveals memory access patterns
  - notion of obfuscation against software only adversaries

Ascend [FDD’12], GhostRider [LHMHTS’15]
  - assume public programs; do not obfuscate programs
Key Contributions

1. Efficient obfuscation of RAM programs using stateless trusted hardware token
   - Design and implement hardware system called HOP
   - FHE, NIZKs, Boolean circuits

2. Scheme Optimizations
   - Challenges in using stateless token
   - 5x-238x better than a baseline scheme
   - 8x-76x slower than an insecure system
   - Security under UC framework
Using Trusted Hardware Token

Sender (honest)

Store Key

Obfuscate

Receiver (malicious)

Execute

Input2

Output2
Ideal Functionality for Obfuscation

Trusted third party

Sender

 prog id

output

Receiver

(prog id, inp)
Stateful Token

Maintain state between invocations

Authenticate memory
Run for a fixed time $T$

Oblivious RAM

- load $a5, 0(s0)$
- add $a5, a4, a5$
- add $a5, a5, a5$
A scheme with stateless tokens is more challenging.

Enables context switching.

Given a scheme with stateless tokens, using stateful tokens can be viewed as an optimization.
Stateless Token

Does not maintain state between invocations

Authenticated Encryption

Oblivious RAM

<table>
<thead>
<tr>
<th>load a5, 0(s0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>add a5, a4, a5</td>
</tr>
<tr>
<td>add a5, a5, a5</td>
</tr>
</tbody>
</table>
Stateless Token - Rewinding

Time 0: load a5, 0(s0)
Time 1: add a5, a4 a5

Rewind!

Time 0: load a5, 0(s0)
Time 1: add a5, a4 a5

Oblivious RAM

<table>
<thead>
<tr>
<th>Operation</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>load a5, 0(s0)</td>
<td></td>
</tr>
<tr>
<td>add a5, a4 a5</td>
<td></td>
</tr>
<tr>
<td>add a5, a5 a5</td>
<td></td>
</tr>
</tbody>
</table>
Oblivious RAMs are generally not secure against rewinding adversaries [SCSL’11, PathORAM’13]
Binary-tree Paradigm for Oblivious RAMs

Path identified by leaf node $\ell$

Memory

Token State

Position map
Block x Must Now Relocate!

Memory

Token State

Position map
Data-access Write Back

```
<table>
<thead>
<tr>
<th>Memory</th>
<th>Token State</th>
<th>Position map</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>r</td>
</tr>
<tr>
<td>Update position</td>
<td></td>
<td></td>
</tr>
<tr>
<td>map</td>
<td></td>
<td></td>
</tr>
<tr>
<td>New designated</td>
<td></td>
<td></td>
</tr>
<tr>
<td>leaf node</td>
<td>r</td>
<td></td>
</tr>
</tbody>
</table>
```

Update position map

New designated leaf node
A Rewinding Attack!

Access Pattern: 3, 3

T = 0: leaf 4, reassigned 2
T = 1: leaf 2, reassigned ...

Rewind!

T = 0: leaf 4, reassigned 7
T = 1: leaf 7, reassigned ...

Access Pattern: 3, 4

Time 0: leaf 4, reassigned ...
Time 1: leaf 1, reassigned ...

Rewind!

Time 0: leaf 4, reassigned ...
Time 1: leaf 1, reassigned ...
For rewinding attacks, ORAM uses $\text{PRF}_K(\text{program digest, input digest})$.
Stateless Token – Rewinding on inputs

| Inp 1 = 20 |
| Inp 2 = 10 |
| Inp 3 = 40 |

Oblivious RAM

| Inp 1 = 20 |
| Inp 2 = 10 |
| Inp 3 = 30 |
For rewinding on inputs, adversary commits input digest during initialization.
Main Theorem: Informal

Our scheme UC realizes the ideal functionality in the $F_{\text{token}}$-hybrid model assuming:
- ORAM satisfies obliviousness
- sstore adopts a semantically secure encryption scheme and a collision resistant Merkle hash tree scheme and
- Assuming the security of PRFs

Proof in the paper.
Efficient obfuscation of RAM programs using *stateless* trusted hardware token

Next:

**1. Interleaving arithmetic and memory instructions**

**2. Using a scratchpad**

**Scheme**

**Optimizations**

**Design and implement hardware system called HOP**
Optimizations to the Scheme – 1. \(A^N M\) Scheduling

Types of instructions – Arithmetic and Memory

1 cycle \(\sim 3000\) cycles

Naïve schedule: \(A M A M A M \ldots\)

Memory accesses visible to the adversary

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Value</th>
<th>Access Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1170: load</td>
<td>a5,0(a0)</td>
<td>M</td>
</tr>
<tr>
<td>1174: addi</td>
<td>a4,sp,64</td>
<td>+ dummy memory access</td>
</tr>
<tr>
<td>1178: addi</td>
<td>a0,a0,4</td>
<td>A</td>
</tr>
<tr>
<td>117c: slli</td>
<td>a5,a5,0x2</td>
<td>A</td>
</tr>
<tr>
<td>1180: add</td>
<td>a5,a4,a5</td>
<td>A</td>
</tr>
<tr>
<td>1184: load</td>
<td>a4,-64(a5)</td>
<td>M</td>
</tr>
<tr>
<td>1188: addi</td>
<td>a4,a4,1</td>
<td>A</td>
</tr>
<tr>
<td>118c: bne</td>
<td>a3,a0,1170</td>
<td>A</td>
</tr>
</tbody>
</table>

Histogram – main loop
Optimizations to the Scheme - 1. $A^N M$ Scheduling

What if a memory access is performed after “few” arithmetic instructions?

Naïve scheduling: 12000 extra cycles

$A A A A M A A M \rightarrow A M A M A M A M A M A M$

$A^4 M$ schedule

$A^4 M$ scheduling: 2 extra cycles
Optimizations to the Scheme - 1. $A^N M$ Scheduling

Ideally, $N$ should be program independent

\[
N = \frac{\text{Memory Access Latency}}{\text{Arithmetic Access Latency}} = \frac{3000}{1}
\]

A A A A M A A M

2996 2998

6006 cycles of actual work

< 6000 cycles of dummy work
Amount of dummy work < 50% of the total work

In other words, our scheme is $2x$-competitive, i.e., in the worst case, it incurs $\leq 2x$-overhead relative to best schedule with no dummy work
Optimizations to the Scheme – 2. Using a Scratchpad

Program
void bwt-rle(char *a) {
    bwt(a, LEN);
    rle(a, LEN);
}
void main() {
    char *inp = readInput();
    for (i=0; i < len(inp); i+=LEN)
        len = bwt-rle(inp + i);
}

Why does a scratchpad help?
Memory accesses served by scratchpad

Why not use regular hardware caches?
Cache hit/miss reveals information as they are program independent
HOP Architecture

- 512 KB
- Variant of Path ORAM
  - Freecursive ORAM
  - PMMAC
  - 64 byte block,
  - 4 GB memory

For efficiency, use stateful tokens

1. single stage 32b integer base
2. spld

16 KB
Evaluation – Speed-up over Baseline Scheme

Scratchpad with $A^N M$
- 3x – 238x better than baseline scheme

$A^N M$ scheme only
- 1.5x – 18x better than baseline scheme
Slowdown Relative to Insecure Schemes

- Slowdown to Insecure: 8x-76x
- Slowdown to GhostRider: 2x-41x
Case Study: bzip2

bzip2: Compression algorithm

Performance does not vary much based on input, so perhaps “easy” to determine running time T

Two highly compressible strings

String S1
106x speedup wrt baseline
17x slowdown wrt insecure

String S2
234x speedup wrt baseline
8x slowdown wrt insecure
### Time for Context Switching

<table>
<thead>
<tr>
<th></th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program State: program params</td>
<td>&lt; 1 KB</td>
</tr>
<tr>
<td>Memory State: ORAM state, auth</td>
<td>~264 KB</td>
</tr>
<tr>
<td>Execution State: cpustate, time</td>
<td>&lt; 1 KB</td>
</tr>
<tr>
<td>Scratchpads: Instruction, Data</td>
<td>~528 KB</td>
</tr>
</tbody>
</table>

Data stored by token: ~800 KB

Assuming 10 GB/s, will require ~160μs to swap state
Conclusion

We are among the first to design and implement a secure processor with a matching cryptographically sound formal abstraction (in the UC framework)

Paper will be on eprint soon.
Code will be open sourced.

kartik@cs.umd.edu