HOP: Hardware makes Obfuscation Practical

Kartik Nayak

With Christopher W. Fletcher, Ling Ren, Nishanth Chandran, Satya Lokam, Elaine Shi and Vipul Goyal
Compression

1 MB → 1 KB

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]
Approaches

Cryptography

1. Indistinguishability Obfuscation
   [BGIRSVY’01, GGRSW’13]
   - Not strong enough in practice
   - Non standard assumptions
   - Inefficient [AHKM’14]

2. Using Trusted Hardware Tokens
   [GISVW’10, DMMN’11, CKZ’13]
   - Boolean circuits
   - Inefficient (FHE, NIZKs)

Secure Processors

1. Intel SGX, AEGIS, XOM [SCGDD’03, LTMLBMH’00]
   - Reveal access patterns
   - Obfuscation against s/w only adversaries

2. Ascend, GhostRider [FDD’12, LHMHTS’15]
   - Assume public programs
Key Contributions

1. Efficient obfuscation of RAM programs using *stateless* trusted hardware token.

2. Design and implement hardware system called HOP using *stateful* tokens.

3. Scheme Optimizations:
   - 5x-238x better than a baseline scheme
   - 8x-76x slower than an insecure system
Using Trusted Hardware Token

Sender (honest)
- Store Key

Receiver (malicious)
- Execute

Obfuscate
- Input2 → Output2
Stateful Token

Maintain state between invocations

Authenticate memory

Run for a fixed time $T$

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>
A scheme with stateless tokens is more challenging

Advantage: 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>
Oblivious RAMs are generally not secure against rewinding adversaries.
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

<table>
<thead>
<tr>
<th>Inp 1 = 20</th>
<th>Inp 2 = 10</th>
<th>Inp 3 = 40</th>
</tr>
</thead>
</table>

Oblivious RAM

- Inp 1 = 20
- Inp 2 = 10
- Inp 3 = 30
For rewinding on inputs, adversary commits input digest during initialization
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:

**Scheme**

1. Interleaving arithmetic and memory instructions
2. Using a scratchpad

**Optimizations**

Design and implement hardware system called HOP
Optimizations to the Scheme – 1. A^NM Scheduling

Types of instructions – Arithmetic and Memory

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>a5,0(a0)</td>
</tr>
<tr>
<td>addi</td>
<td>a4,sp,64</td>
</tr>
<tr>
<td>addi</td>
<td>a0,a0,4</td>
</tr>
<tr>
<td>slli</td>
<td>a5,a5,0x2</td>
</tr>
<tr>
<td>add</td>
<td>a5,a4,a5</td>
</tr>
<tr>
<td>load</td>
<td>a4,-64(a5)</td>
</tr>
<tr>
<td>addi</td>
<td>a4,a4,1</td>
</tr>
<tr>
<td>bne</td>
<td>a3,a0,1170</td>
</tr>
</tbody>
</table>

Memory accesses visible to the adversary

Naïve schedule:

A M A M A M ...

12000 extra cycles

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

Types of instructions – Arithmetic and Memory

- Arithmetic: 1 cycle
- Memory: ~3000 cycles

Memory accesses visible to the adversary

Naïve schedule:

A M A M A M ... 12000 extra cycles

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

$A^4 M$ schedule:

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

Ideally, N should be program independent

$N = \text{Memory Access Latency} / \text{Arithmetic Access Latency} = 3000 / 1$

A A A A M A A M

2996A 2998A

6006 cycles of actual work

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

Our schedule incurs ≤ 2x- overhead relative to best schedule with no dummy work
Optimizations to the Scheme – 2. Using a Scratchpad

Program
```c
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

For efficiency, use stateful tokens
Slowdown Relative to Insecure Schemes

Slowdown to Insecure 8x-76x
Conclusion

We are the first to design and prototype a secure processor with a matching cryptographically sound formal abstraction in the UC framework.

Thank You!

kartik@cs.umd.edu