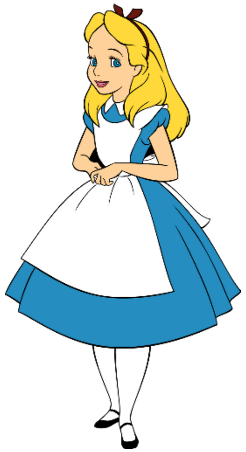
HOP: Hardware makes Obfuscation Practical

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]

```
#include<stdio.h> #include<string.h> main(){char*O,l[999]=
"'`acgo\177~|xp .-\0R^8)NJ6%K4O+A2M(*0ID57$3G1FBL";while(O=
fgets(l+45,954,stdin)){*l=O[strlen(O)[O-1]=0,strspn(O,l+11)];
while(*O)switch((*l&&isalnum(*O))-!*l){case-1:{char*I=(O+=
strspn(O,l+12)+1)-2,O=34;while(*I&3&&(O=(O-16<<1)+*I---'-')<80);
putchar(O&93?*I&8||!(  I=memchr( l , O , 44 ) ) ?'?':I-l+47:32);
break;case 1: ;}*l=(*O&31)[l-15+(*O>61)*32];while(putchar(45+*l%2),
(*l=*l+32>>1)>35);case 0:putchar((++O,32));}putchar(10);}}
```

Impossible to achieve program obfuscation in general [BGIRSVY'01]

Approaches

Cryptography

1. Indistinguishability Obfuscation [BGIRSVY'01, GGHRSW'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

~~FHE, NIZKs~~ ~~Boolean circuits~~

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

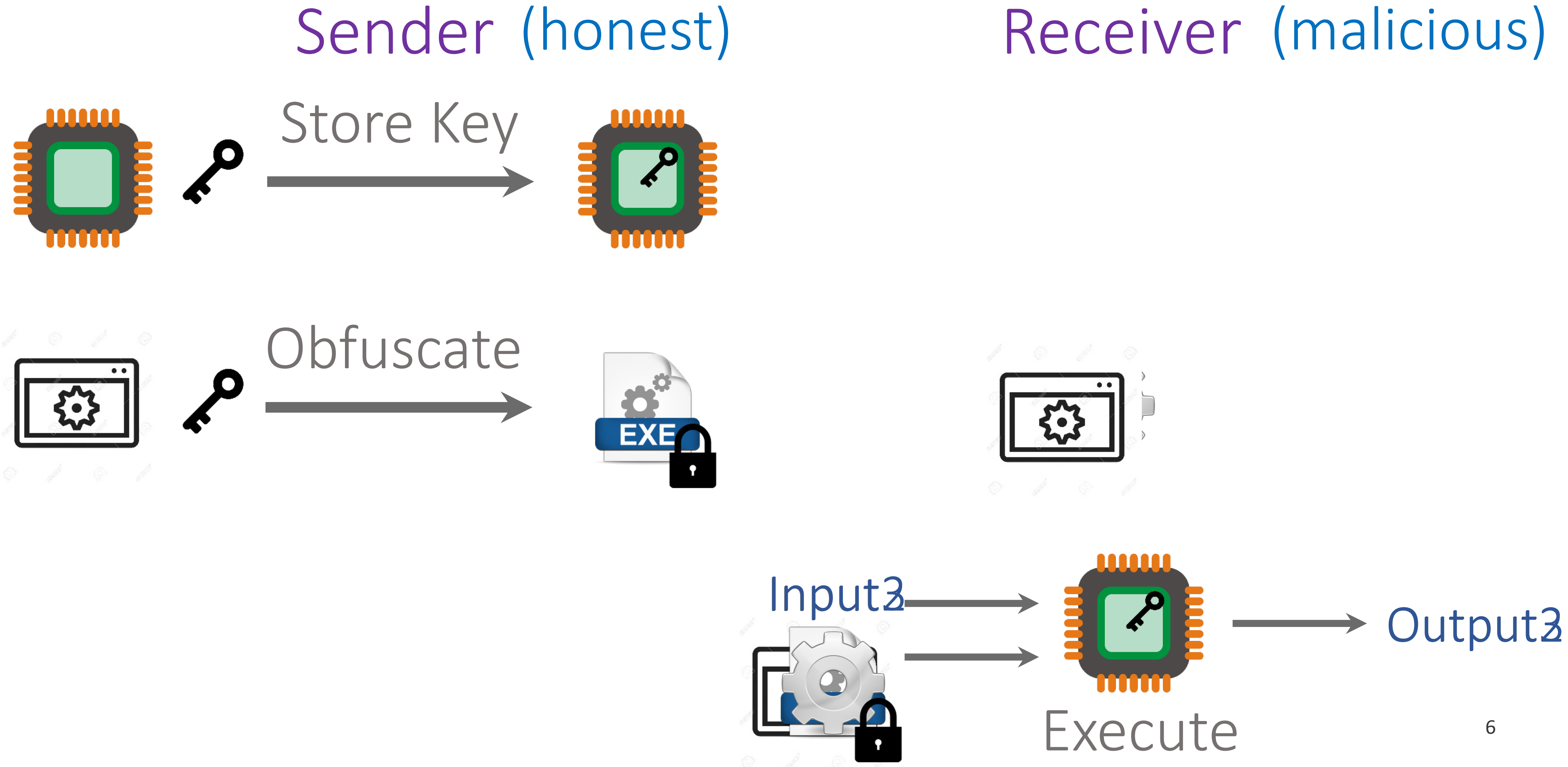
2♪ Design and implement hardware system called HOP using stateful tokens

3♪ Scheme Optimizations

5x-238x better than a baseline scheme

8x-76x slower than an insecure system

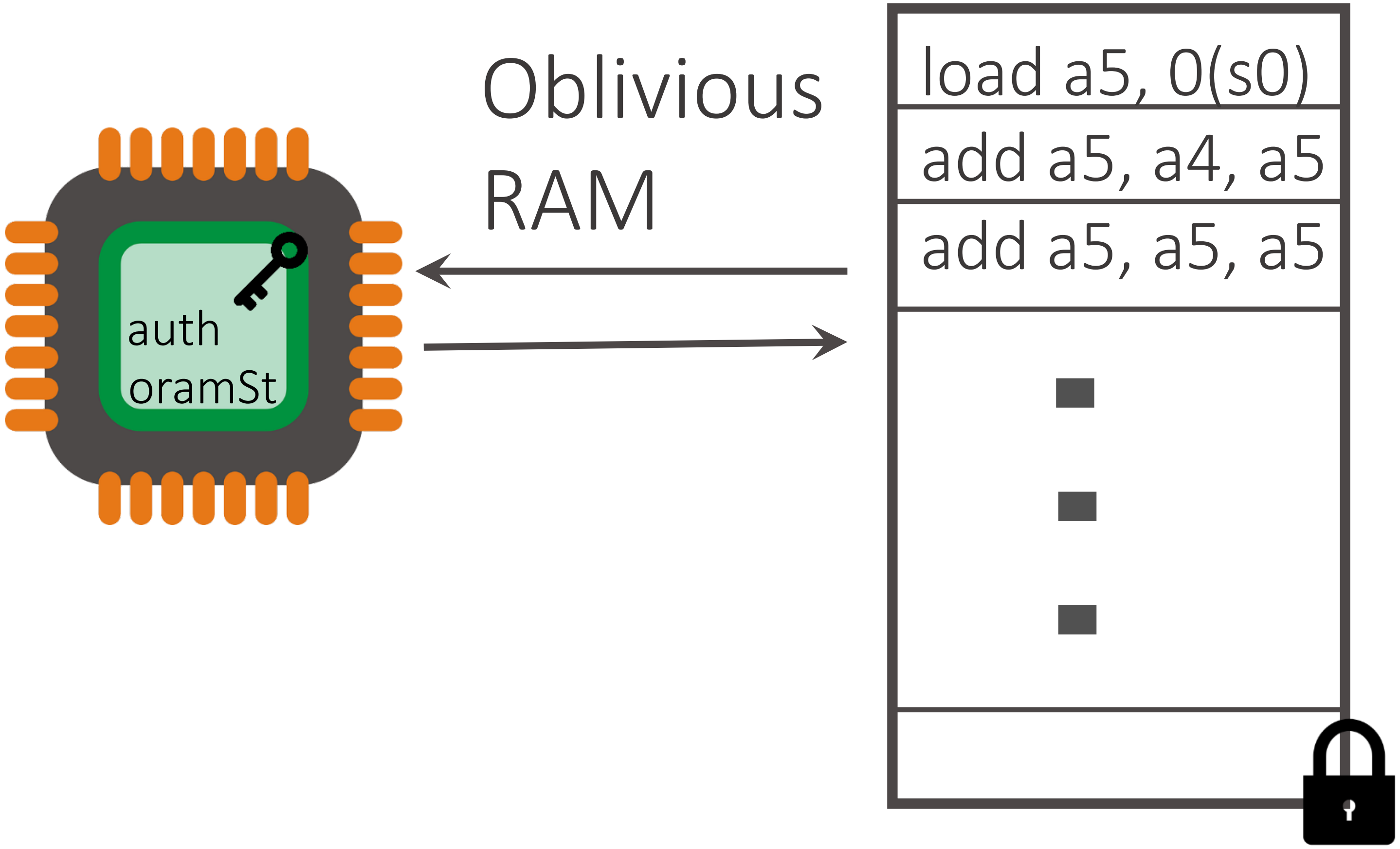
Using Trusted Hardware Token



Stateful Token

Maintain state between invocations

Authenticate memory
Run for a fixed time T



A scheme with stateless tokens is more challenging

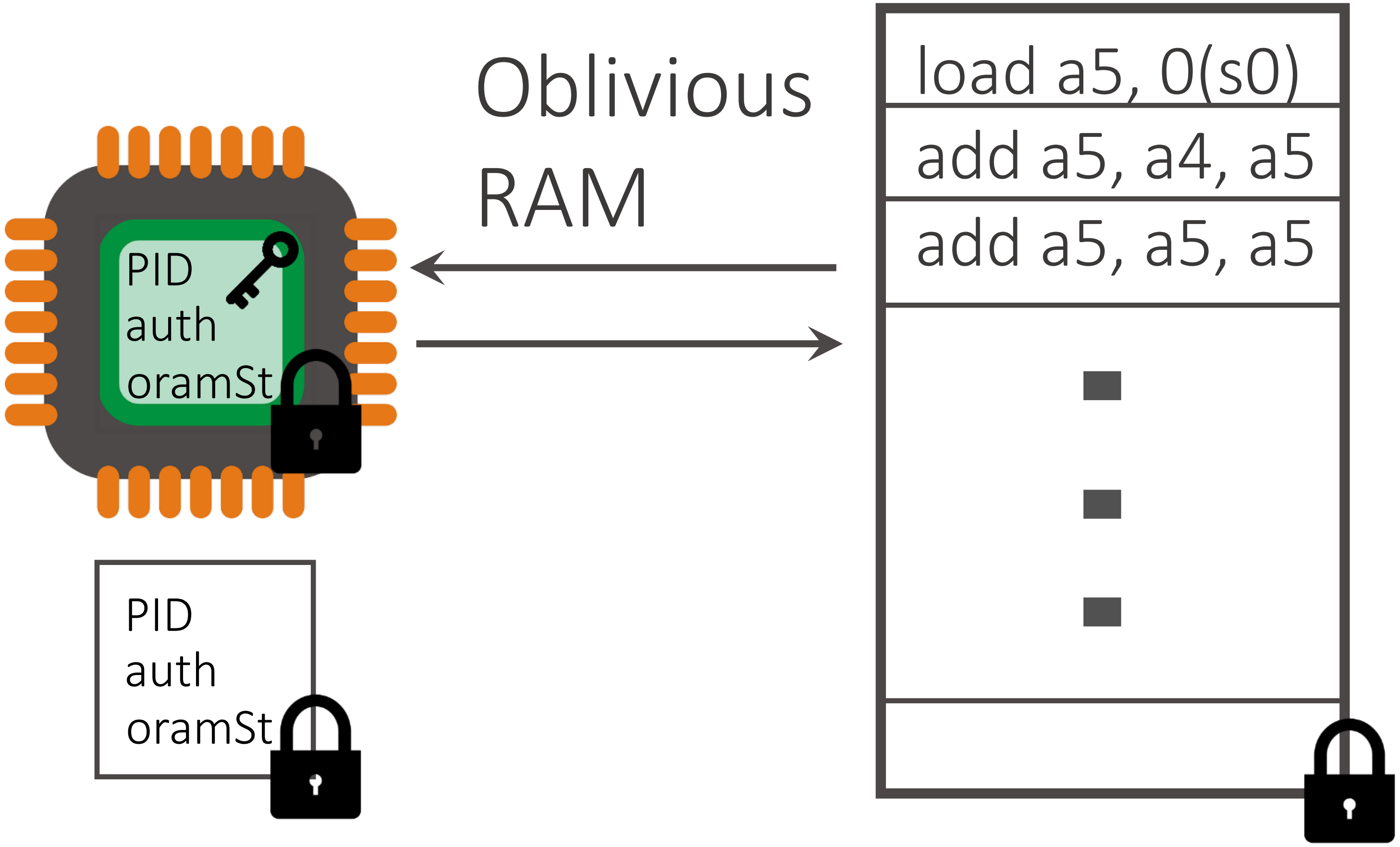
Advantage: Enables context switching

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

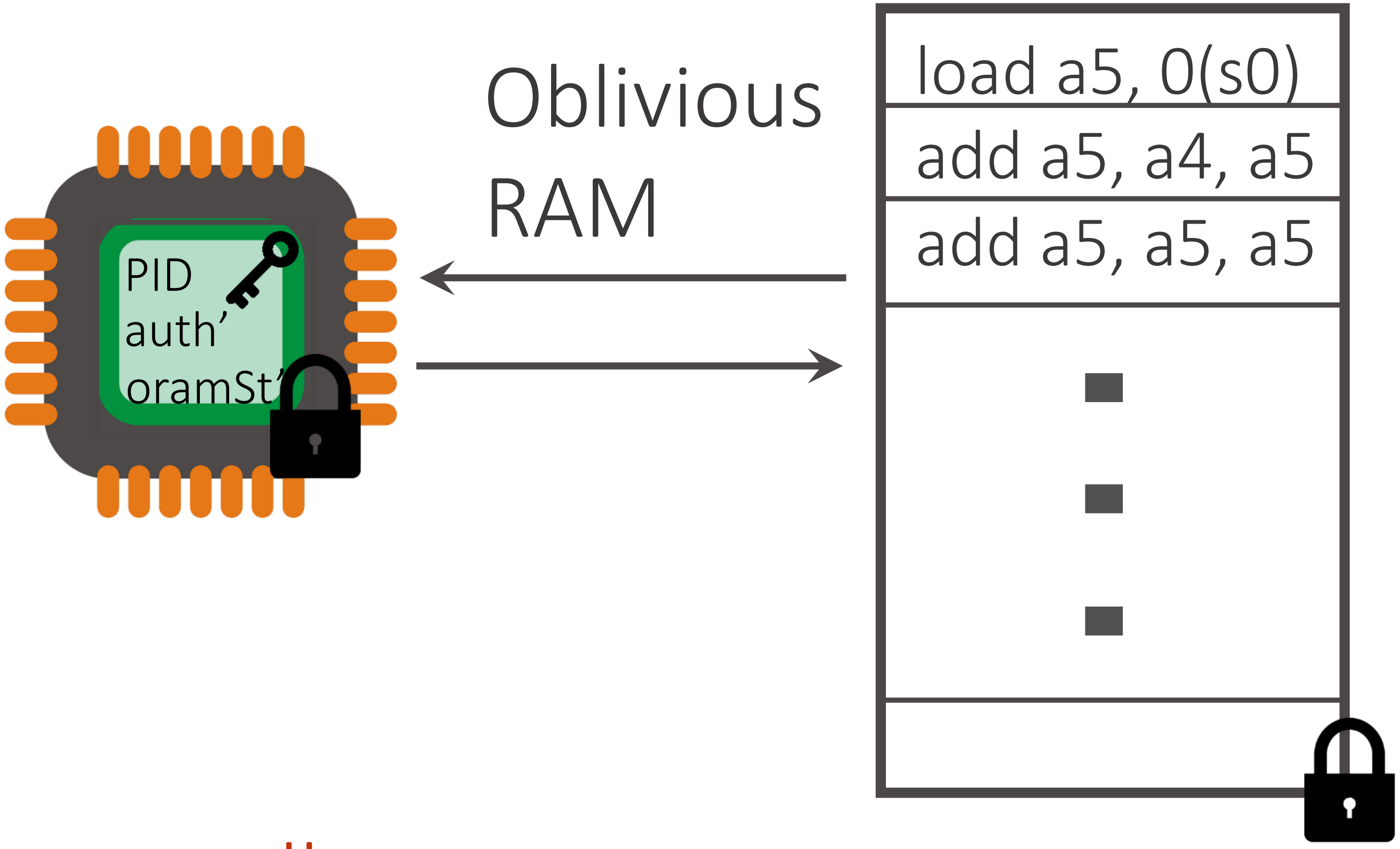


Stateless Token - Rewinding

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

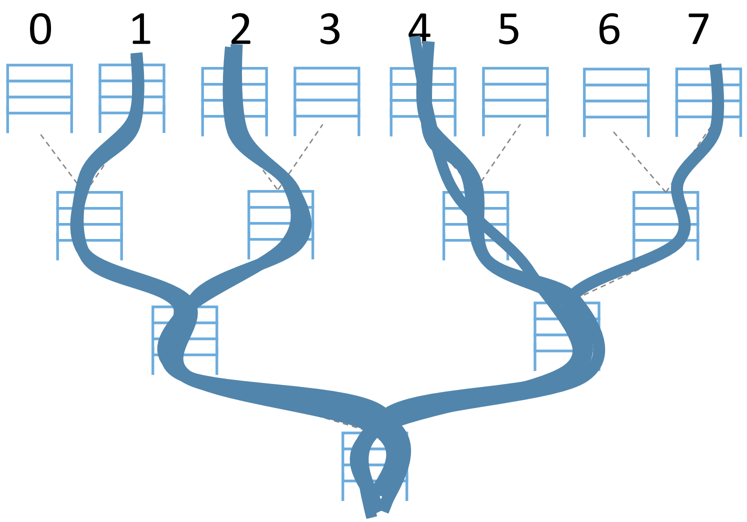
Rewind!

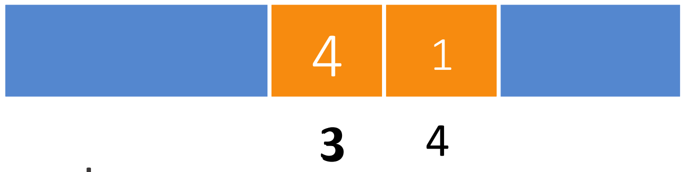
Time 0: load a5, 0(s0)
Time 1: add a5, a4 a5



Oblivious 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!

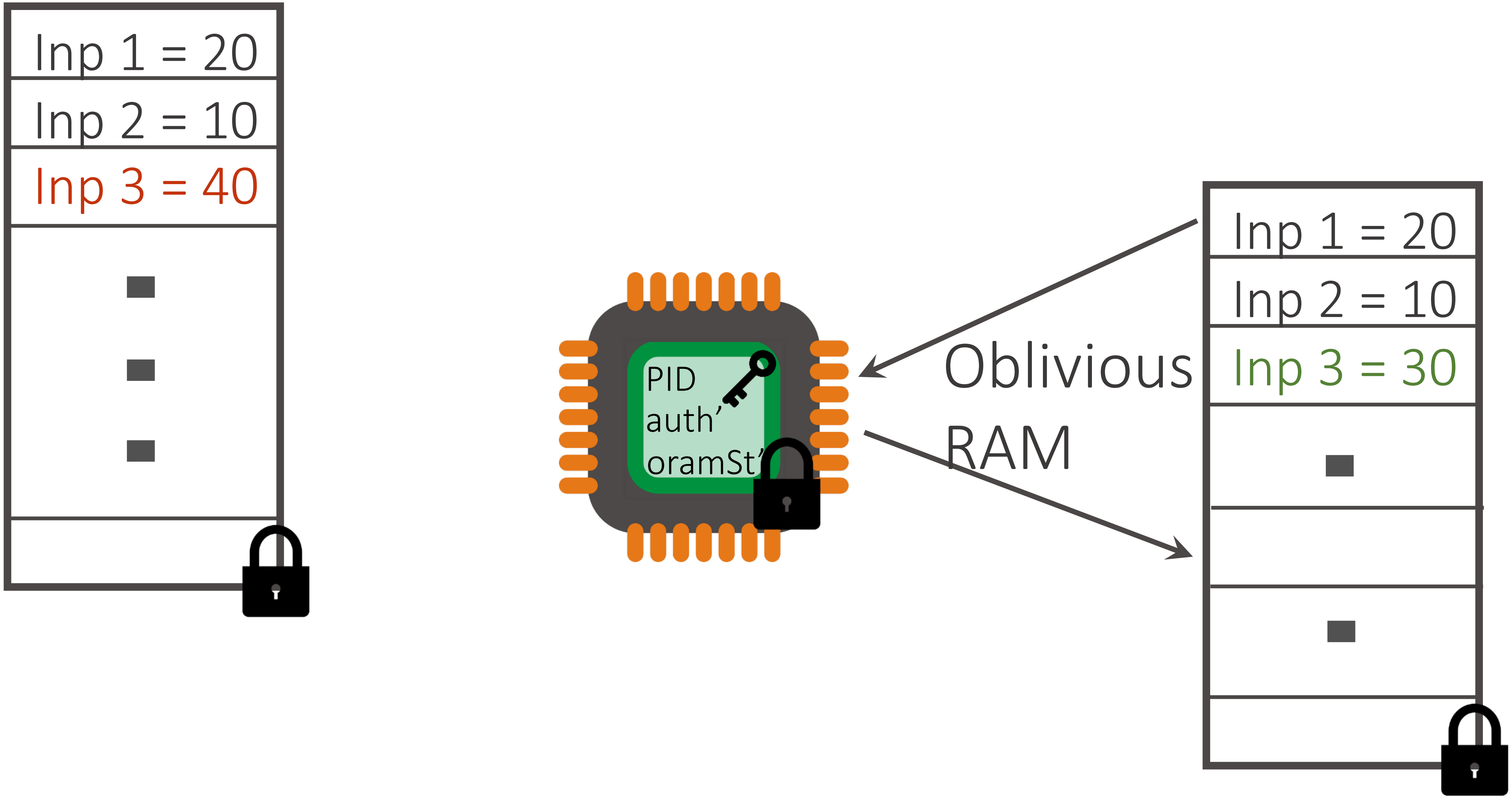
Time 0: leaf 4, reassigned …

Time 1: leaf 1, reassigned …

For rewinding attacks, ORAM uses

PRF_K(program digest, input digest)

Stateless Token – Rewinding on inputs



For rewinding on inputs, adversary commits input digest during initialization

Main Theorem: Informal

Our scheme UC realizes the ideal functionality in the F_{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.

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

Next:

2♪ Scheme Optimizations

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

3♪ Design and implement hardware system called HOP

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

Types of instructions – Arithmetic and Memory

1 cycle ~3000 cycles

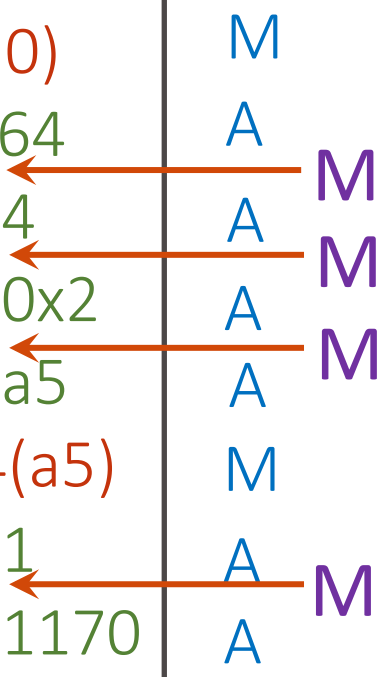
Memory accesses visible to the adversary

Naïve schedule:

A M A M A M …

12000 extra cycles

```
1170: load    a5,0(a0)
1174: addi    a4,sp,64
1178: addi    a0,a0,4
117c: slli    a5,a5,0x2
1180: add     a5,a4,a5
1184: load    a4,-64(a5)
1188: addi    a4,a4,1
118c: bne     a3,a0,1170
```



Histogram – main loop

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

Types of instructions – Arithmetic and Memory

1 cycle ~3000 cycles

Memory accesses visible to the adversary

Naïve schedule:

A M A M A M …

12000 extra cycles

```
1170: load   a5,0(a0)       M
1174: addi   a4,sp,64       A
1178: addi   a0,a0,4        A
117c: slli   a5,a5,0x2      A
1180: add    a5,a4,a5       A
1184: load   a4,-64(a5)     M
1188: addi   a4,a4,1        A   <- AA
118c: bne    a3,a0,1170     A
```

Histogram – main loop

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

```
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

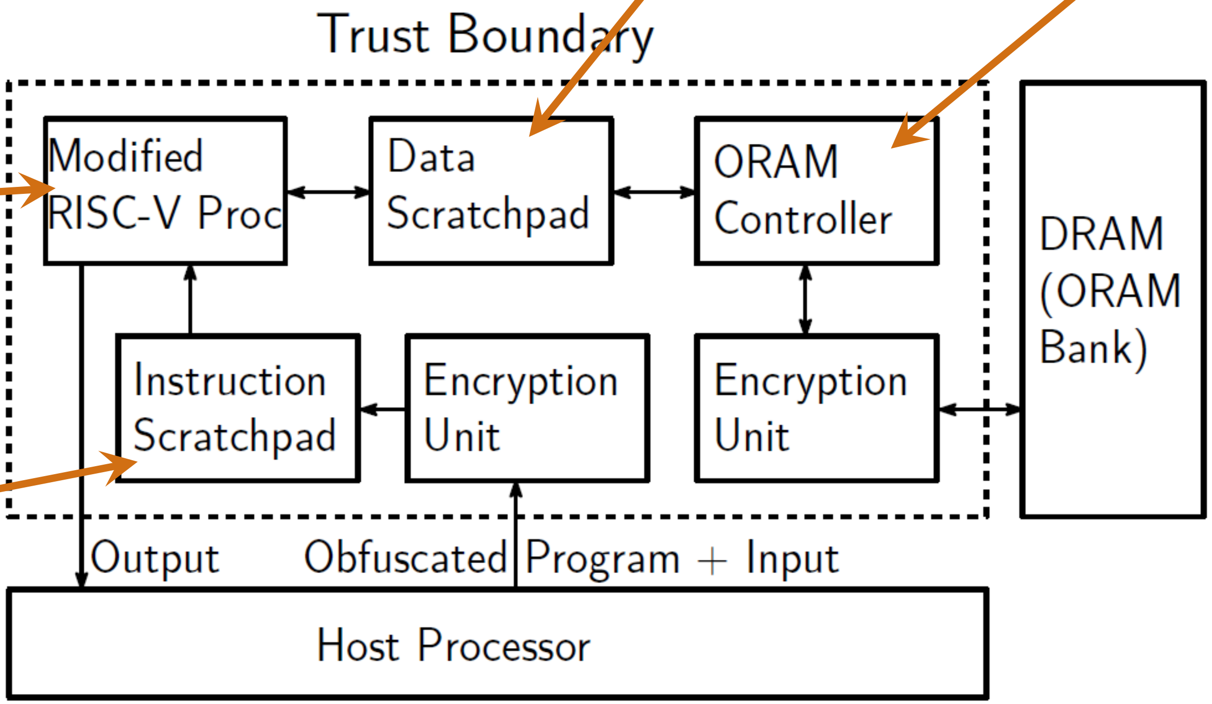
512 KB

Variant of Path ORAM

- Freecursive ORAM
- PMMAC
- 64 byte block,
- 4 GB memory

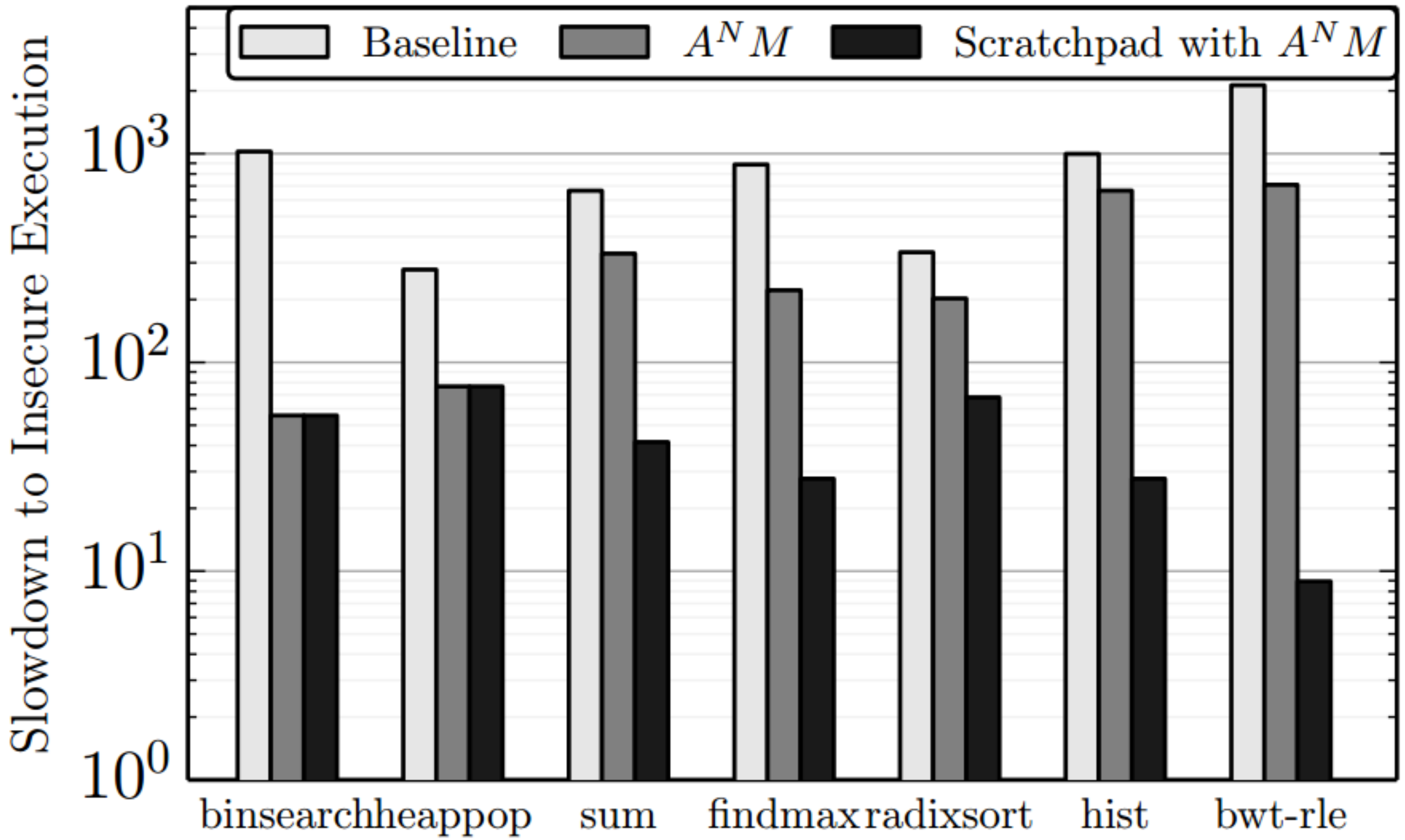
1. single stage 32b integer base
2. spld

16 KB



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