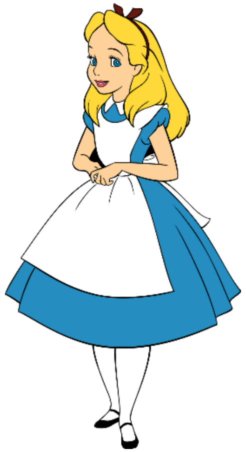
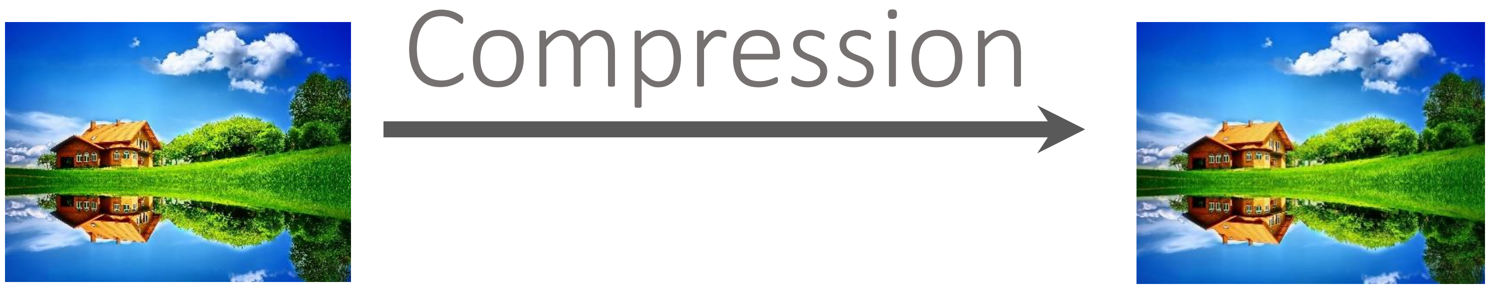
HOP:
Hardware makes Obfuscation Practical

Kartik Nayak

With Chris Fletcher, Ling Ren, Nishanth Chandran, Satya Lokam, Elaine Shi
and Vipul Goyal





Compression

1 MB

1 KB

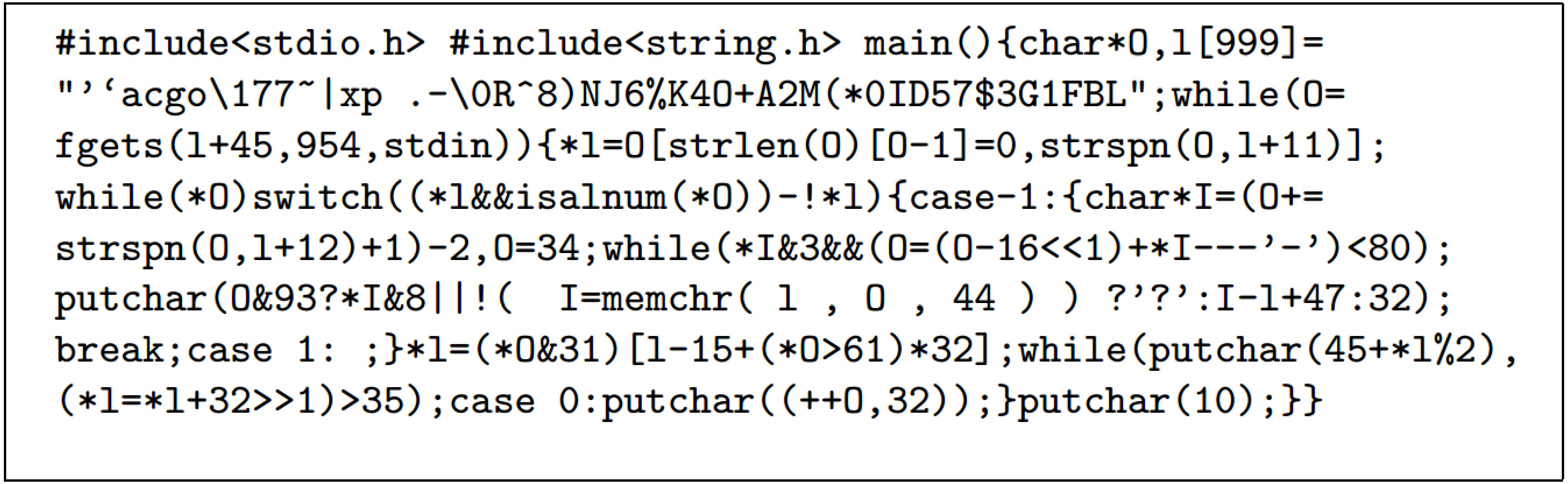
Used by everyone, perhaps license it

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]

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

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

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

2 Design and implement hardware system called HOP

3 Scheme Optimizations

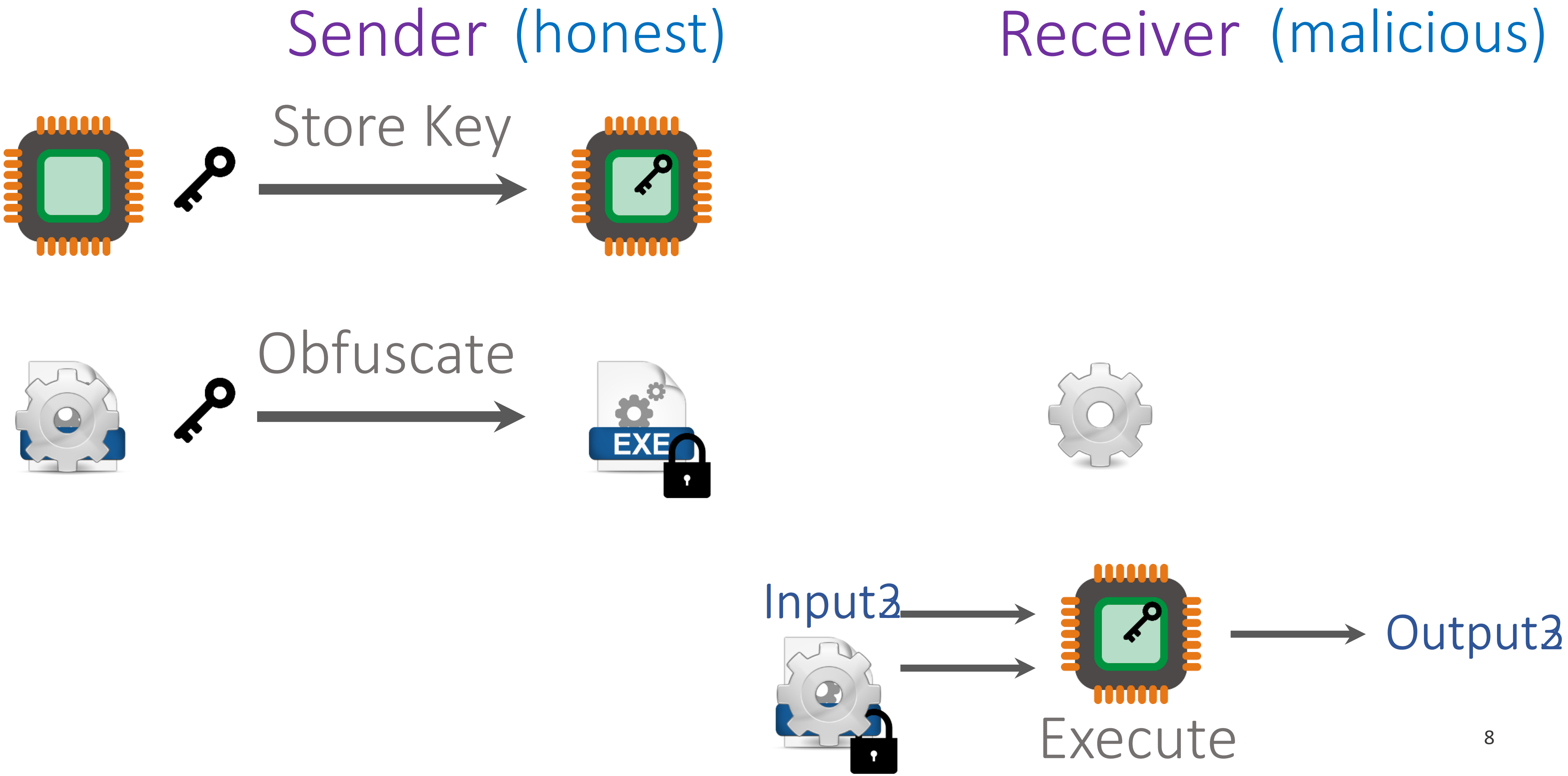
Challenges in using stateless token

Security under UC framework

5x-238x better than a baseline scheme

8x-76x slower than an insecure system

Using Trusted Hardware Token



Sender (honest)

Receiver (malicious)

Store Key

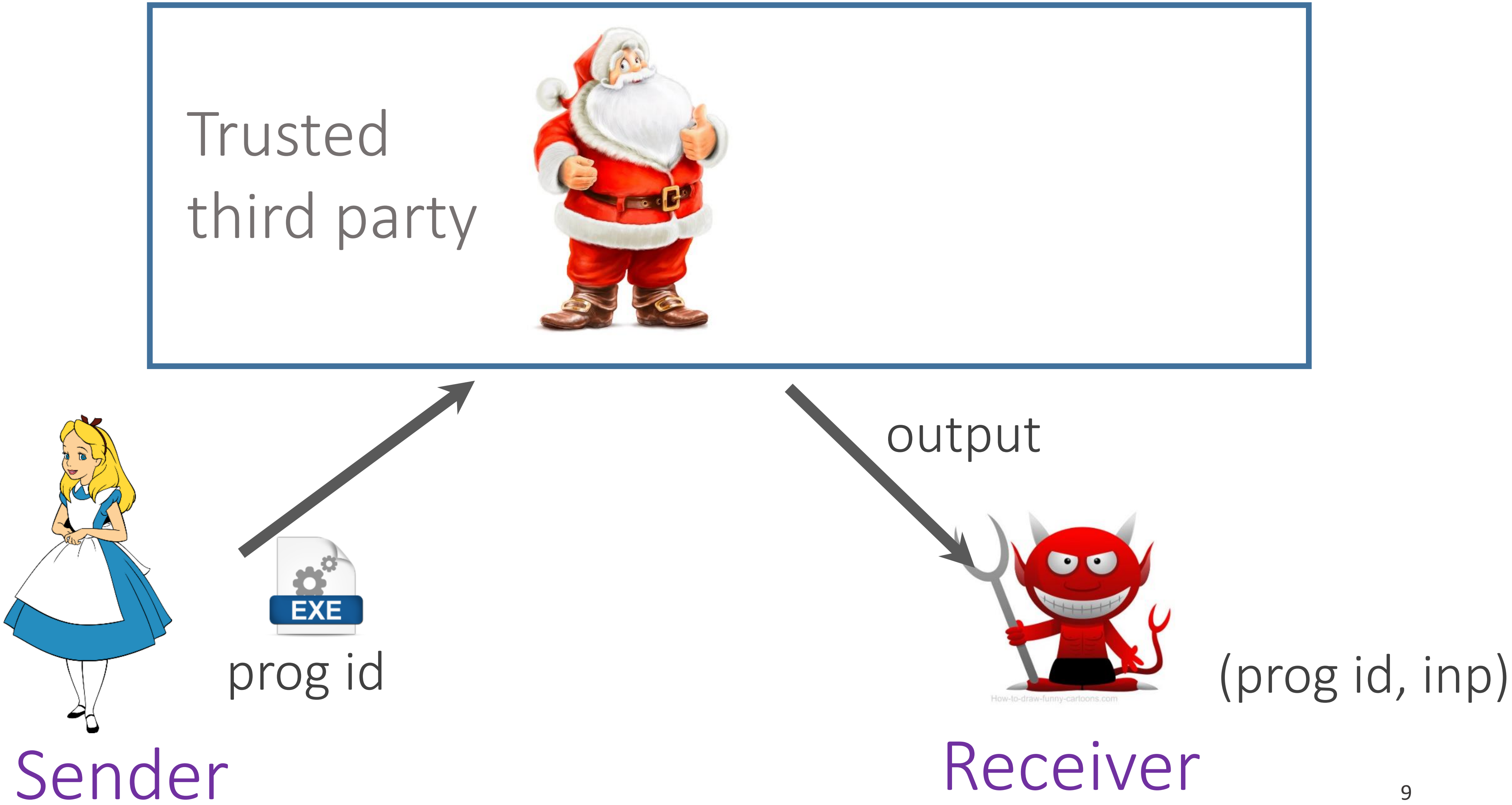
Obfuscate

Input3

Execute

Output3

Ideal Functionality for Obfuscation



Trusted third party

prog id

output

(prog id, inp)

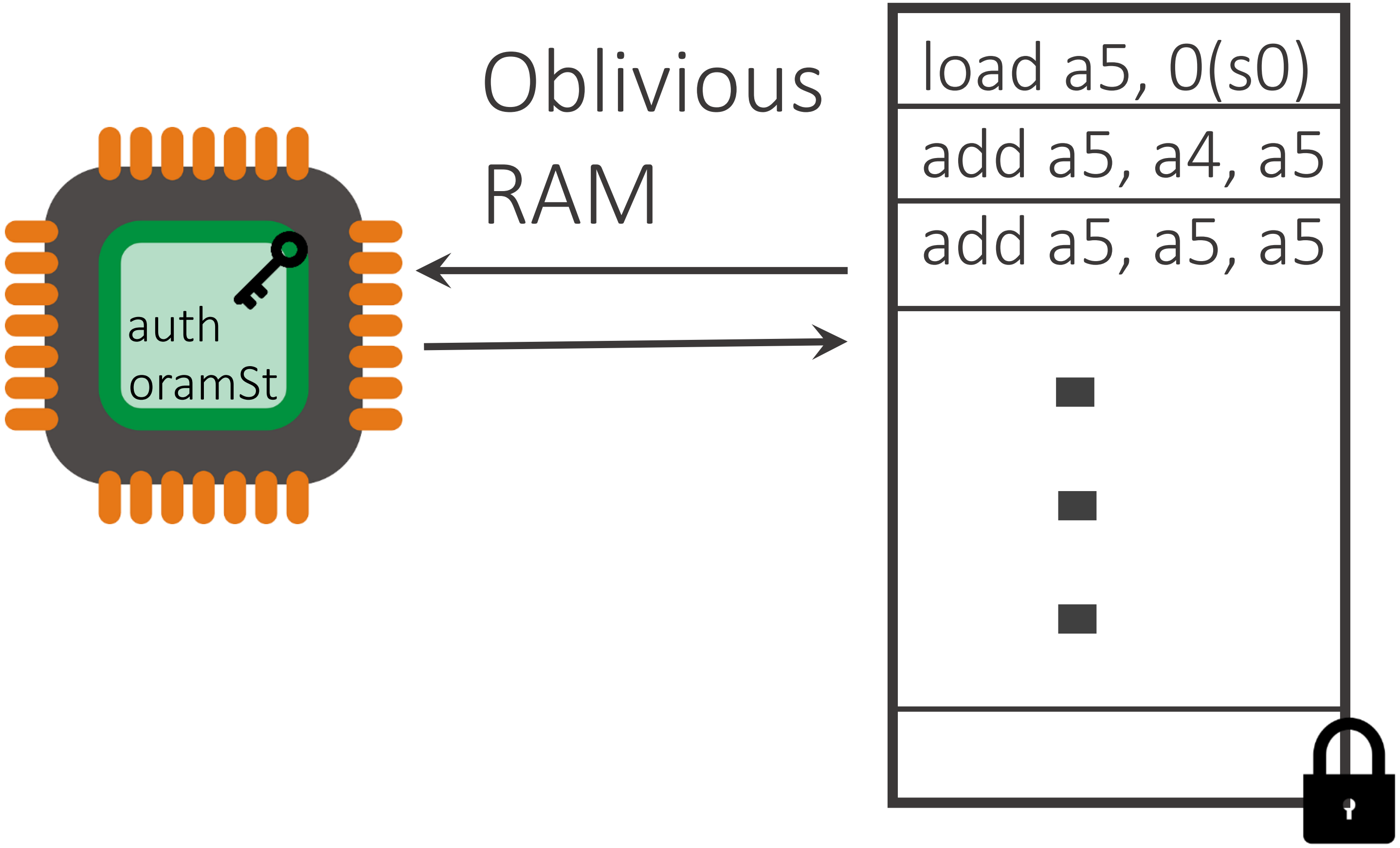
Sender

Receiver

Stateful Token

Maintain state between invocations

Authenticate memory
Run for a fixed time T



```
load a5, 0(s0)
add a5, a4, a5
add a5, a5, a5
```

A scheme with stateless tokens is more challenging

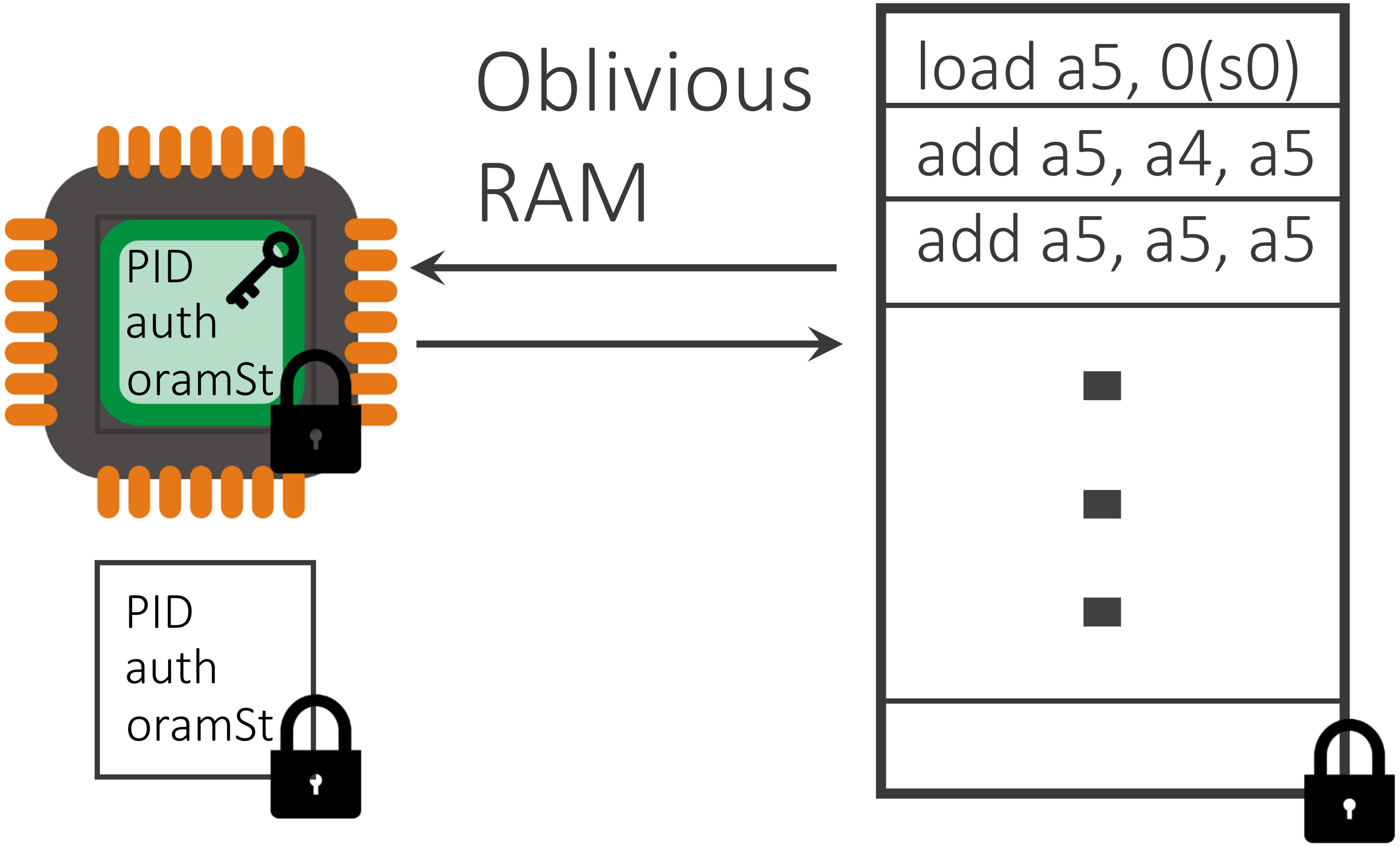
Enables context switching

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



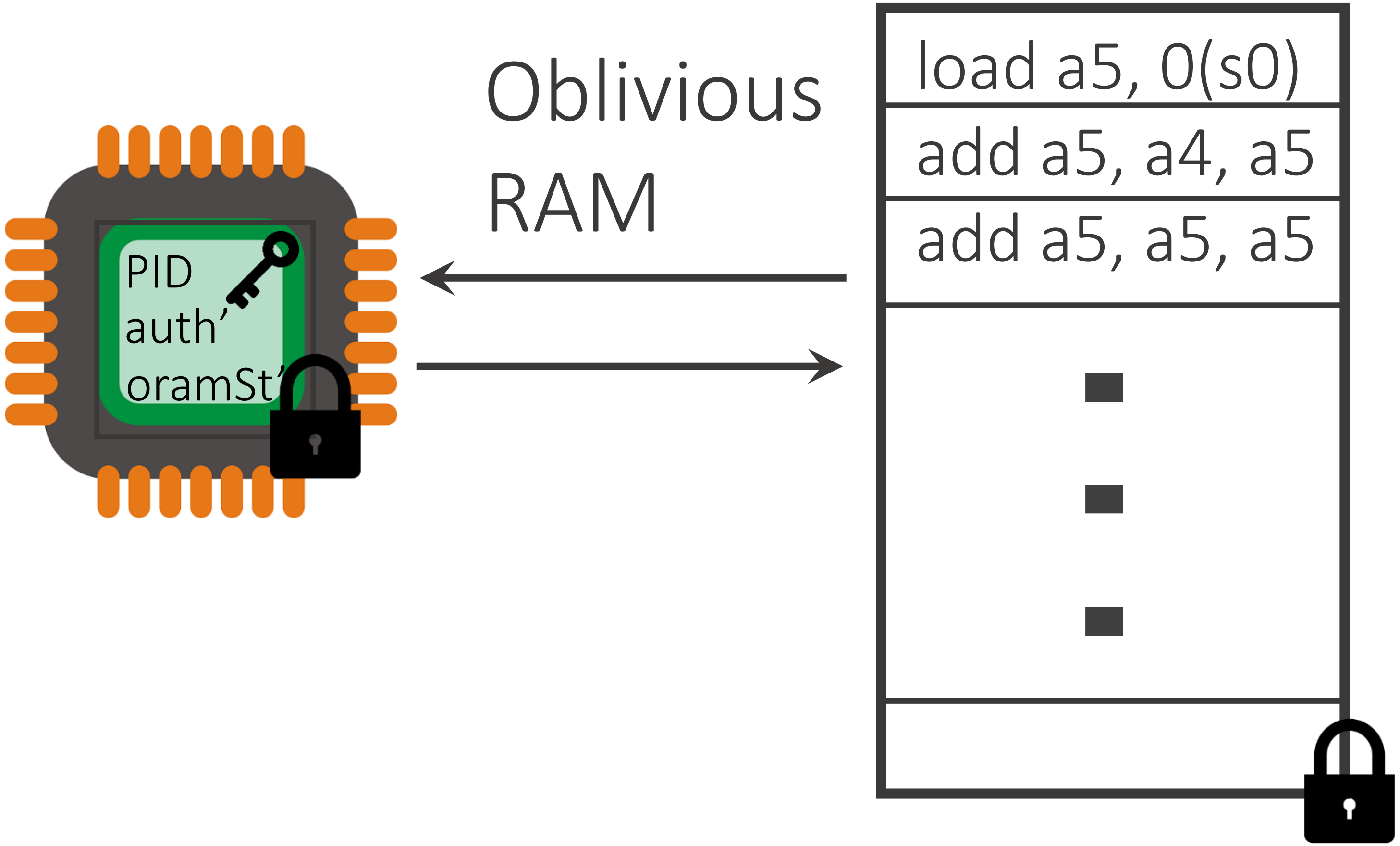
load a5, 0(s0)
add a5, a4, a5
add a5, a5, a5

Stateless Token - Rewinding

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

Rewind!

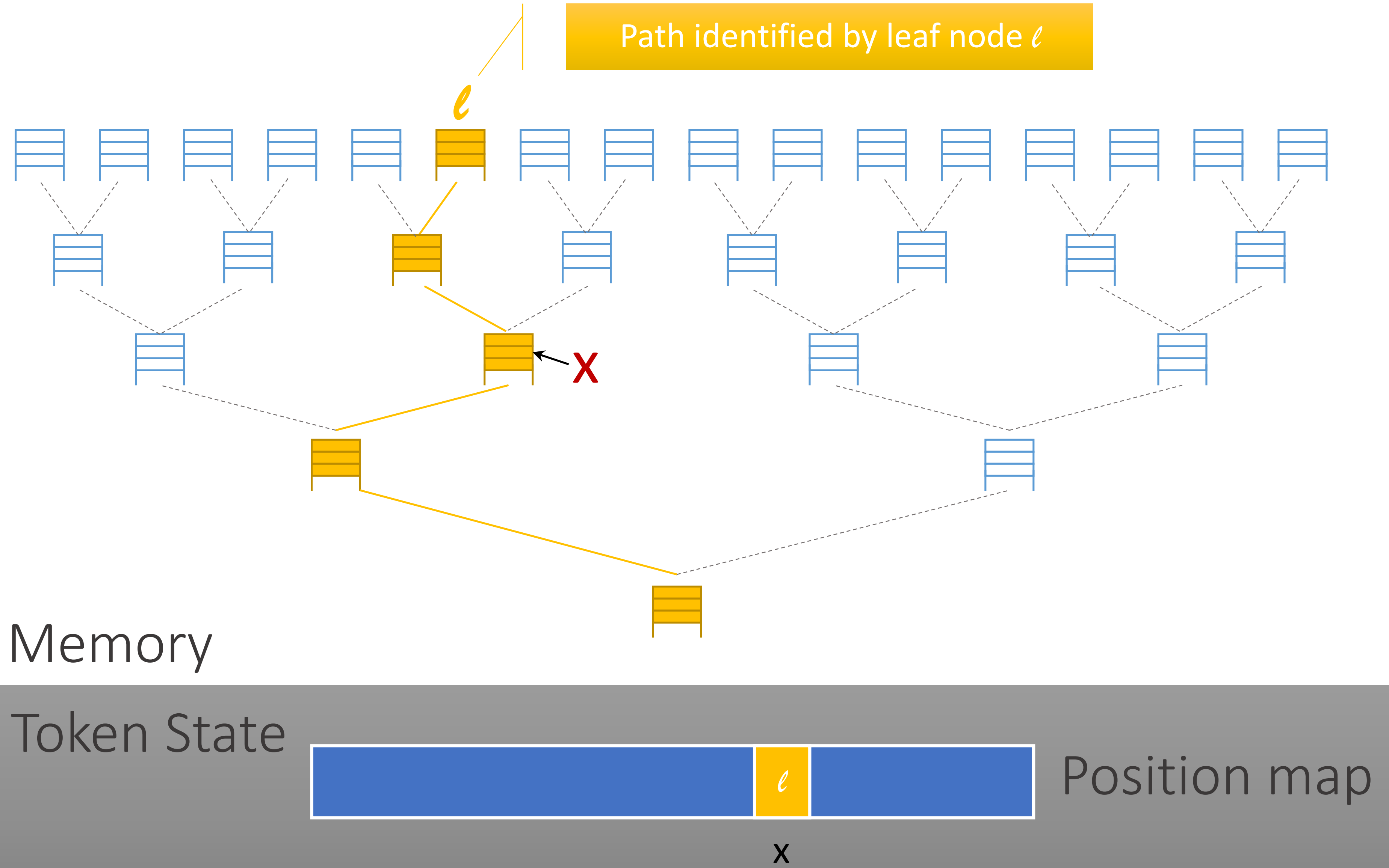
Time 0: load a5, 0(s0)
Time 1: add a5, a4 a5



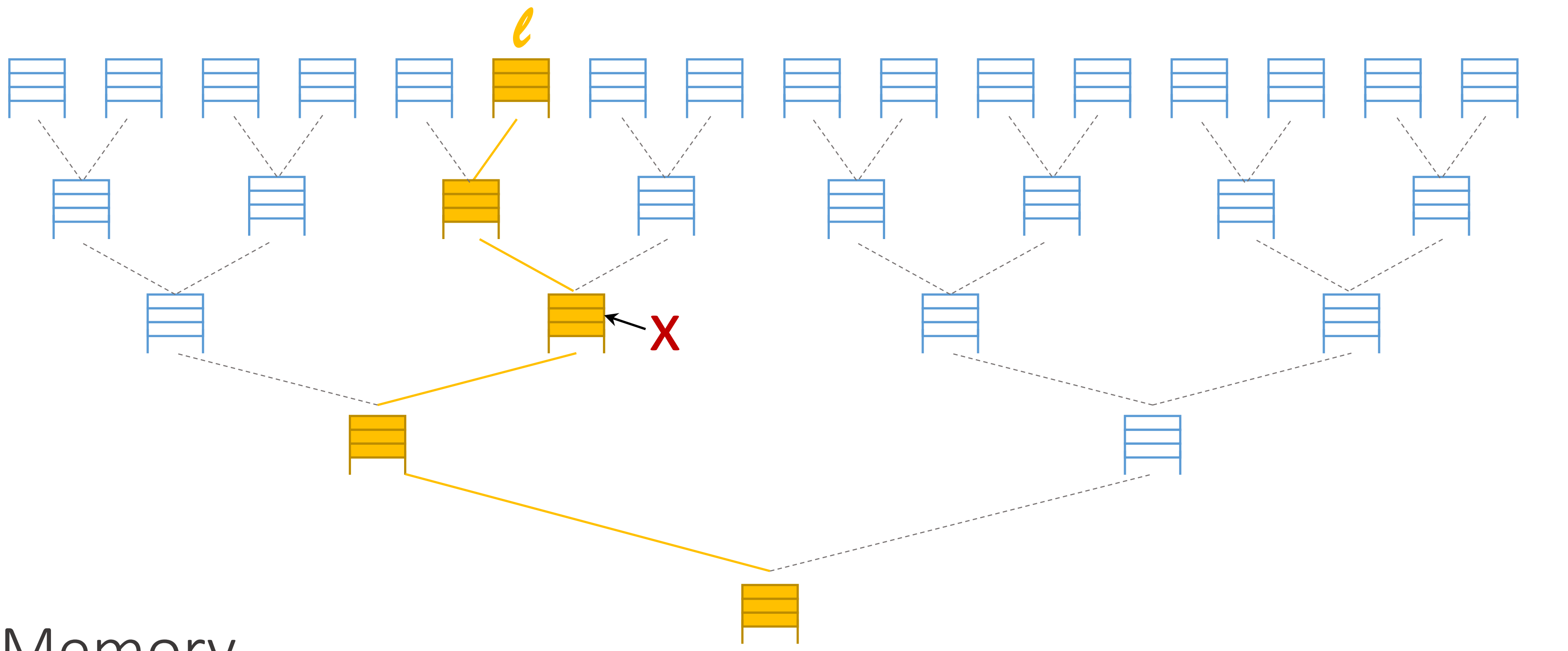
load a5, 0(s0)
add a5, a4, a5
add a5, a5, a5
⋮

Oblivious RAMs are generally not secure against rewinding adversaries [SCSL’11, PathORAM’13]

Binary-tree Paradigm for Oblivious RAMs

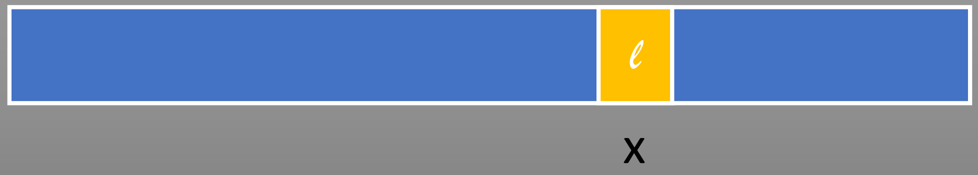


Block x Must Now Relocate!



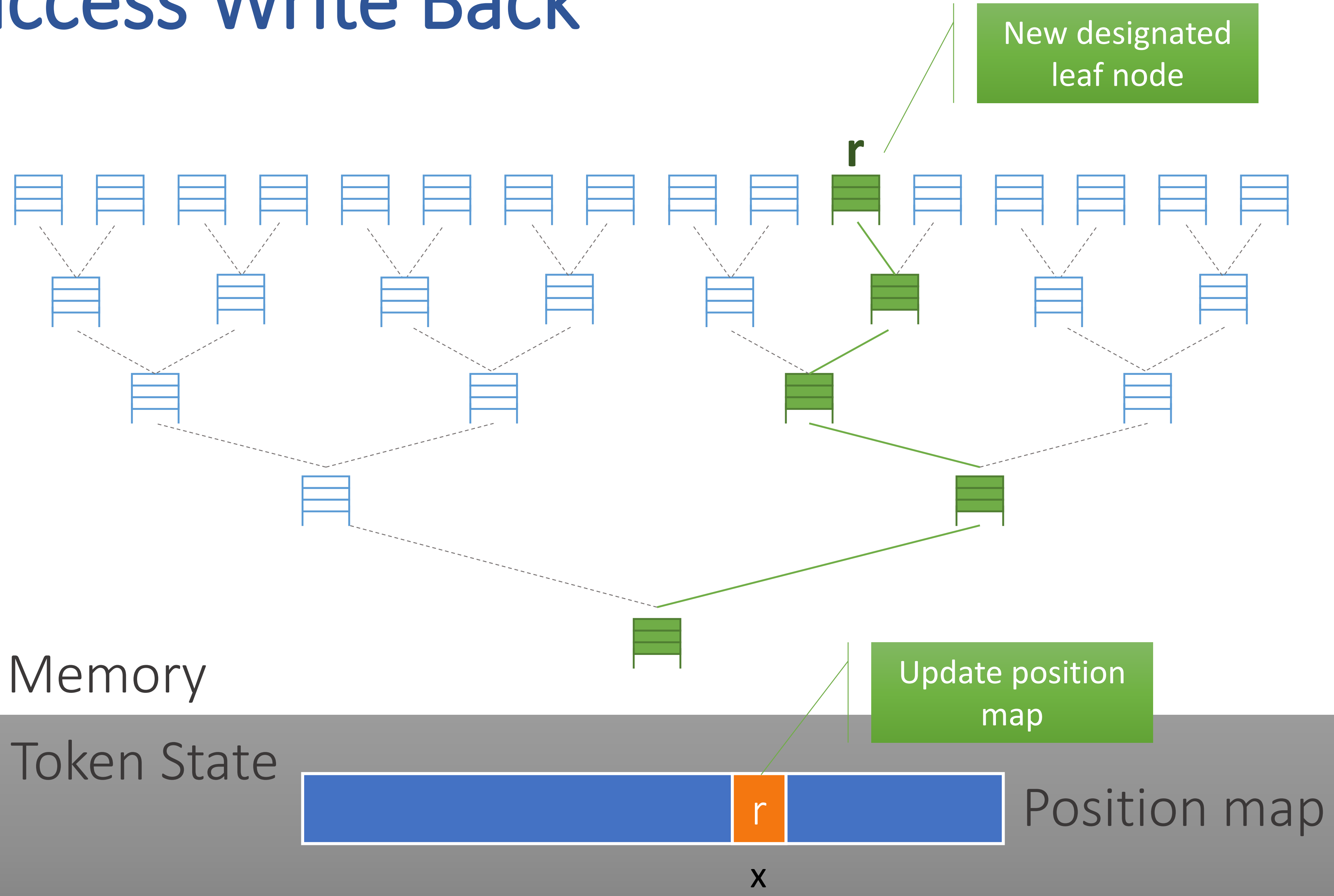
Memory

Token State

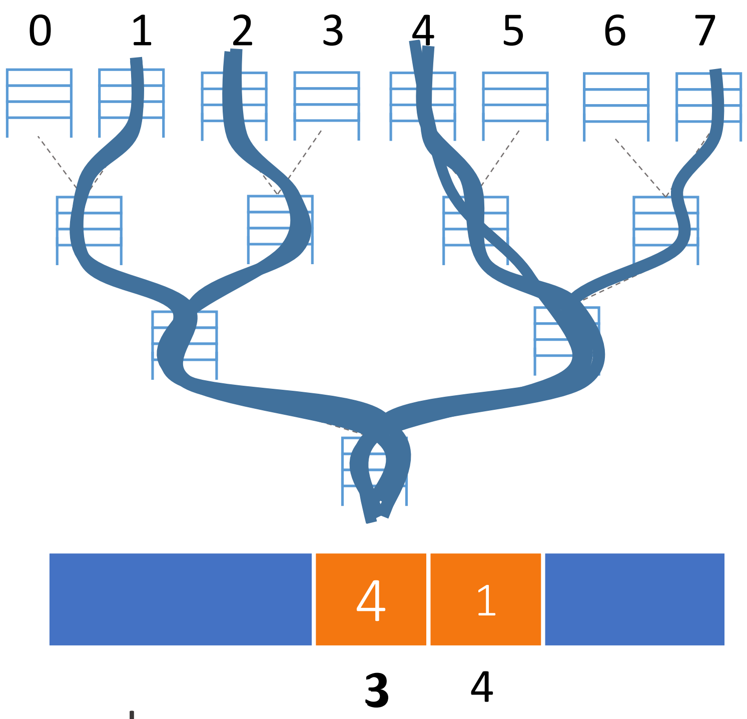


Position map

Data-access Write Back



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 …

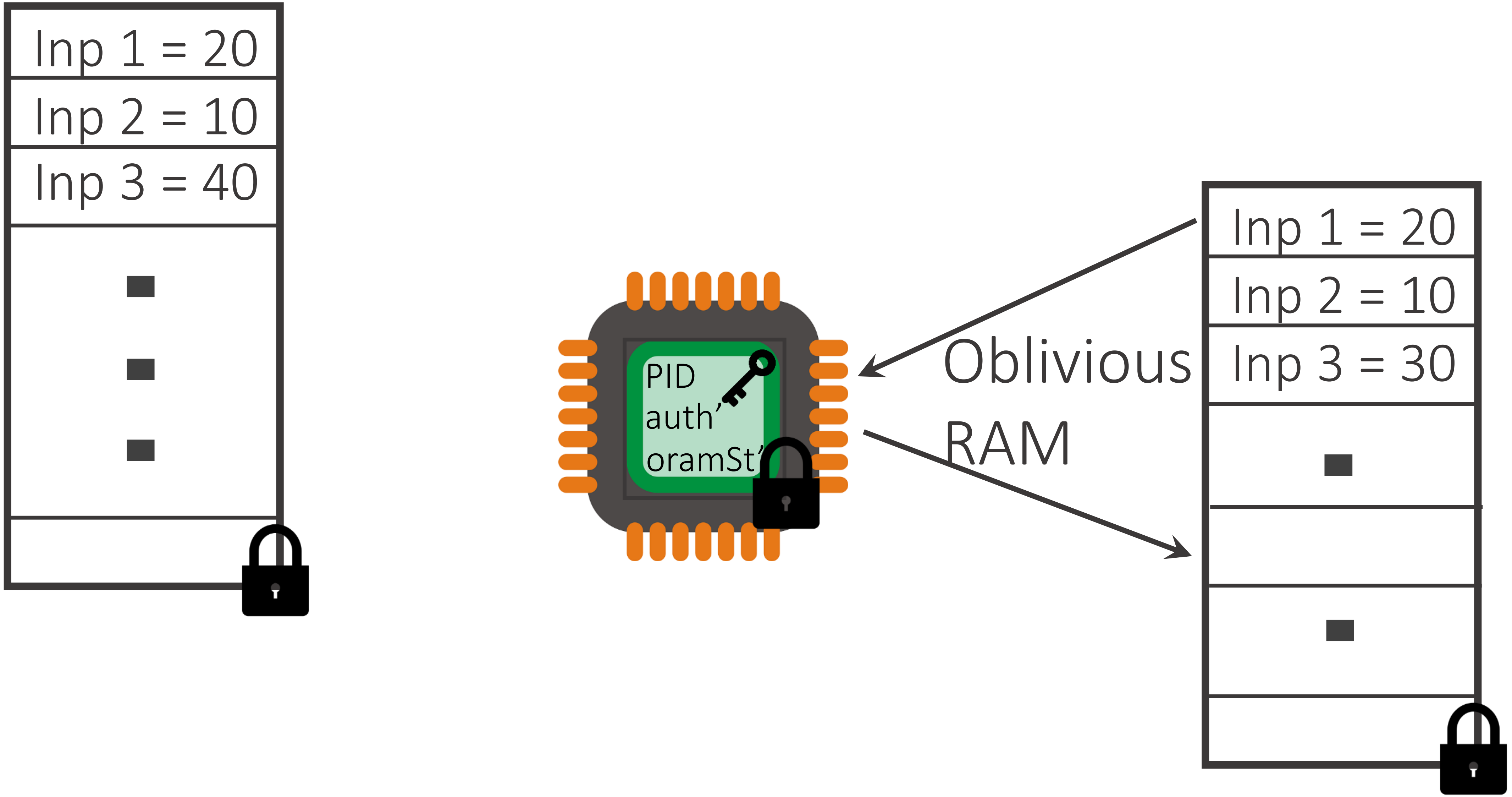
Rewind!

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



Inp 1 = 20

Inp 2 = 10

Inp 3 = 40

⋮

PID

auth’

oramSt’

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_{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

2 Next:

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 A M …

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

Histogram – main loop

Optimizations to the Scheme - 1. A^{N}M Scheduling

A A A A M A A M → A **M** A **M** A **M** A M A **M** A M

Naïve scheduling: 12000 extra cycles

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

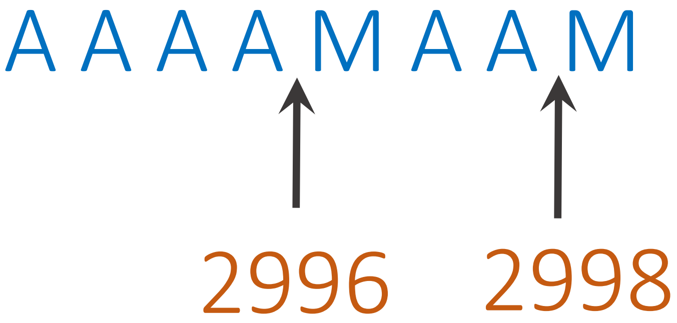
A A A A M A A M → A A A A M A A **A** **A** M (A^{4}M schedule)

A^{4}M scheduling: 2 extra cycles

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

Ideally, N should be program independent

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



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 ≤ 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);
}
```

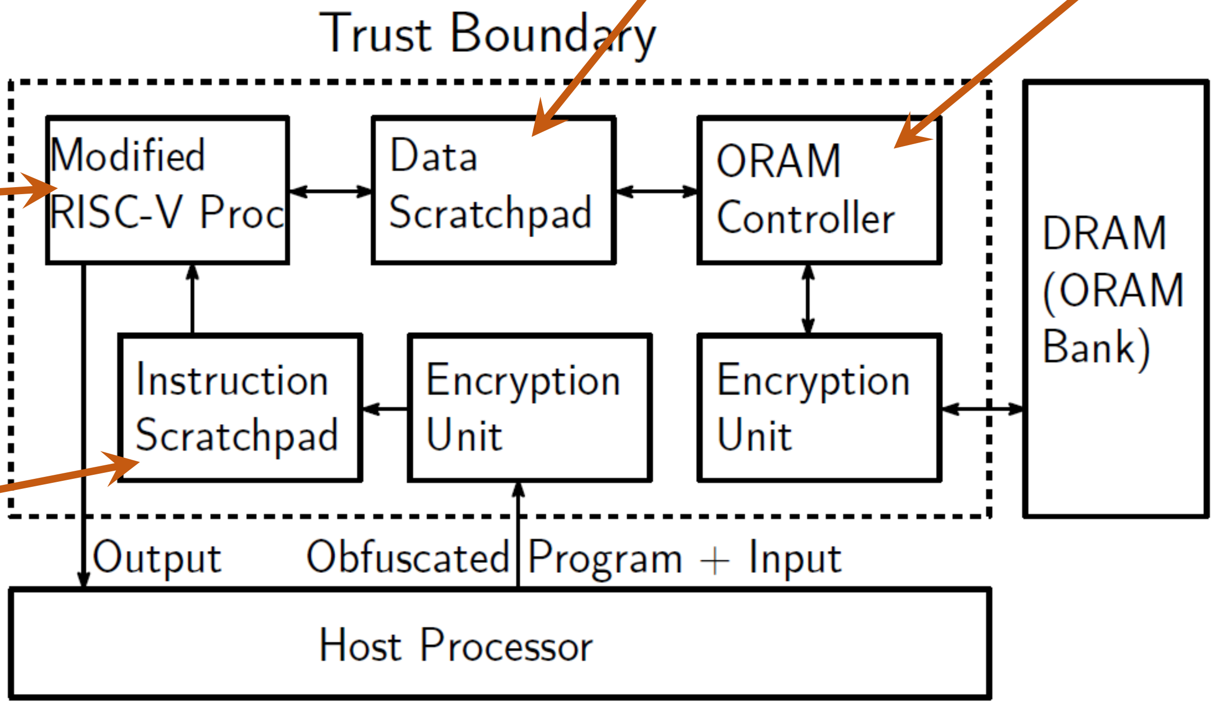
Why does a scratchpad help?

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

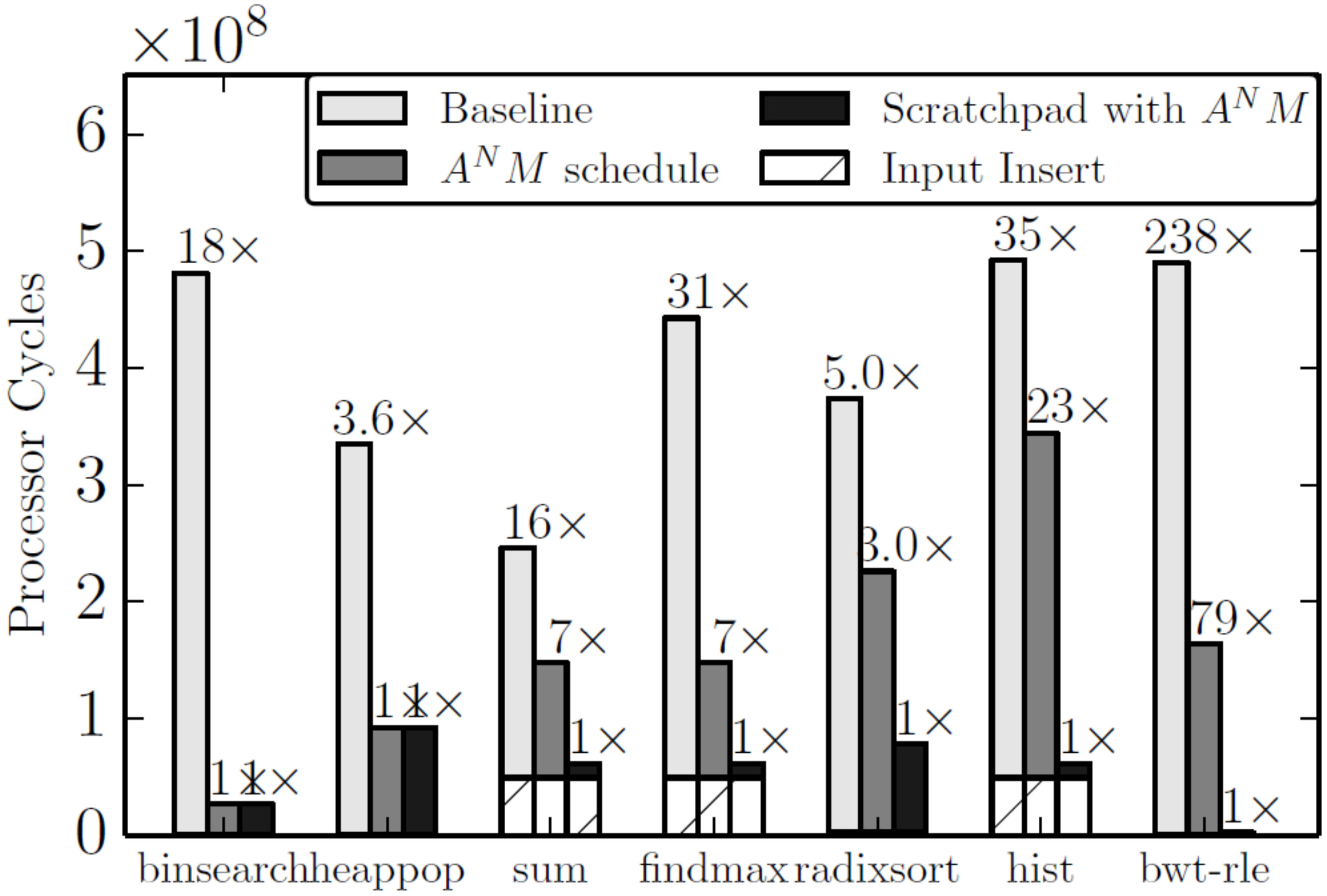
- Freecursive ORAM
- PMMAC
- 64 byte block,
- 4 GB memory

1. single stage 32b integer base
2. spld

16 KB

For efficiency, use stateful tokens

Evaluation – Speed-up over Baseline Scheme



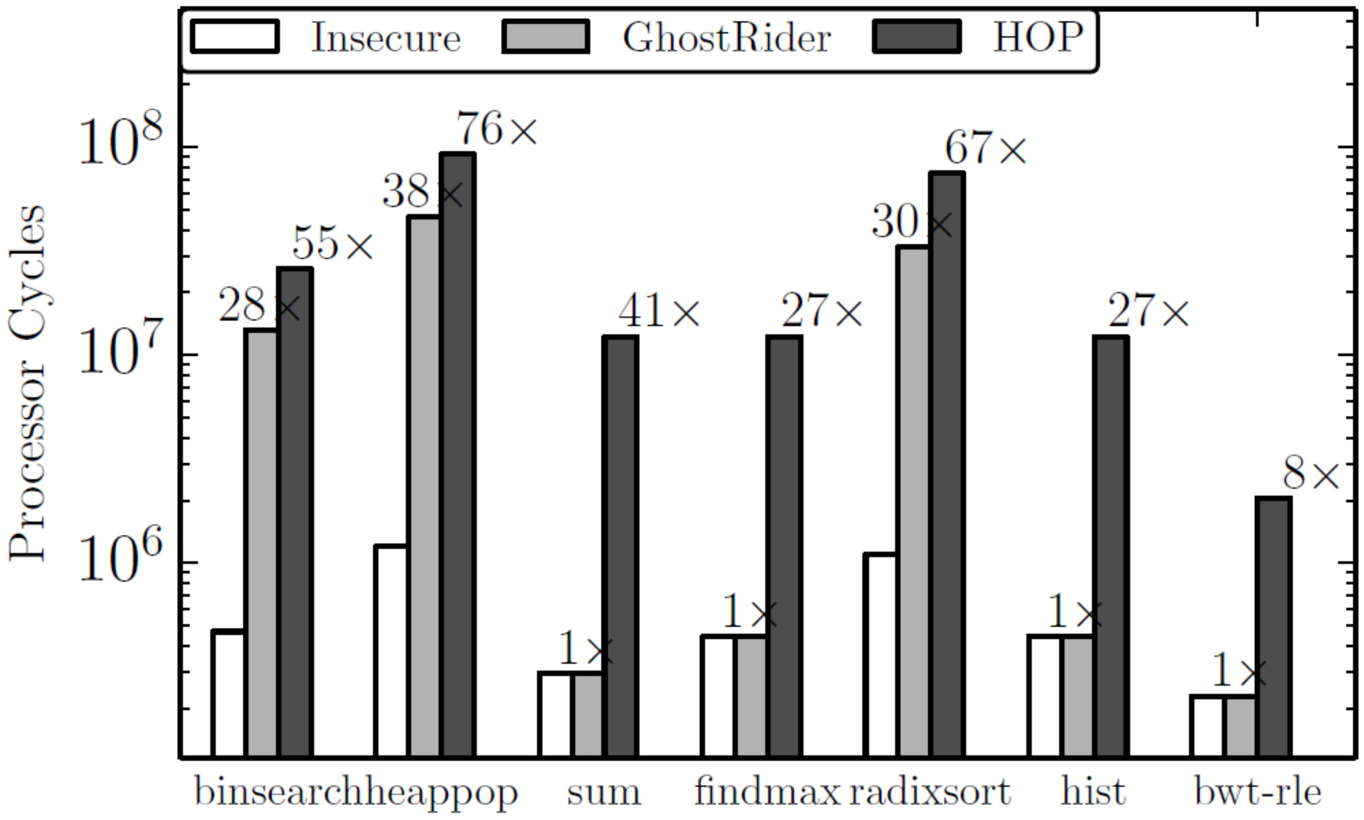
Scratchpad with A^NM

3x – 238x better than baseline scheme

A^NM 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

Program State: program params	< 1 KB
Memory State: ORAM state, auth	~264 KB
Execution State: cpustate, time	< 1 KB
Scratchpads: Instruction, Data	~528 KB

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