Adversarial Symbolic Execution for Detecting Timing Side-Channel Leaks

Chao Wang

USC University of Southern California

KLEE Workshop 2021

Collaborators

Shengjian (Daniel) Guo, **Meng Wu, Jingbo Wang, Chungha Sung, Mukund Raghothaman, Brandon Paulsen, Hassan Eldib, Jun Zhang, Pengfei Gao, Fu Song, Mostafa Taha, Patrick Schaumont**

Outline

01

1. Background

Cache timing side channel

02

2. Motivation

Concurrency and side channel

03

3. Method

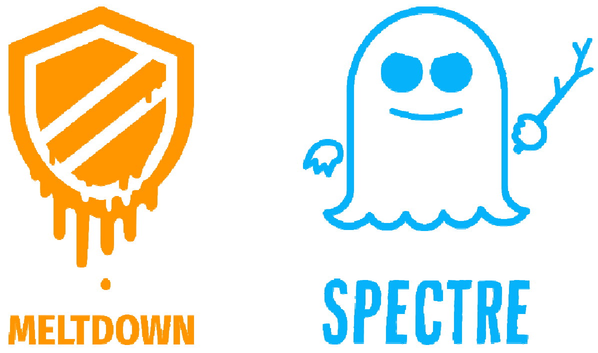
Adversarial symbolic execution

04

4. Evaluation

Benchmarks and experiments

Side Channel: *January 2018 – News*



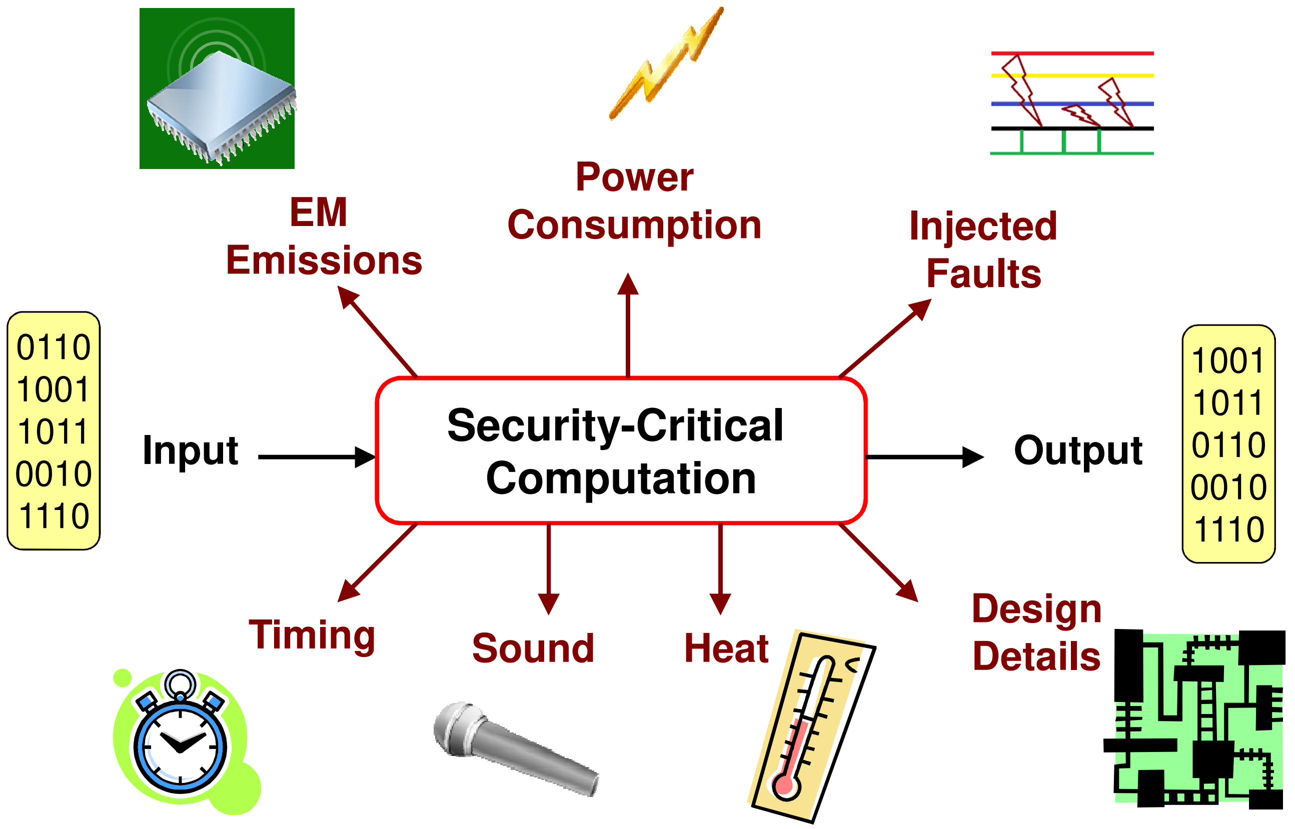
Affected a wide range of x86 processors produced since 1995

Pictures are taken from https://meltdownattack.com/

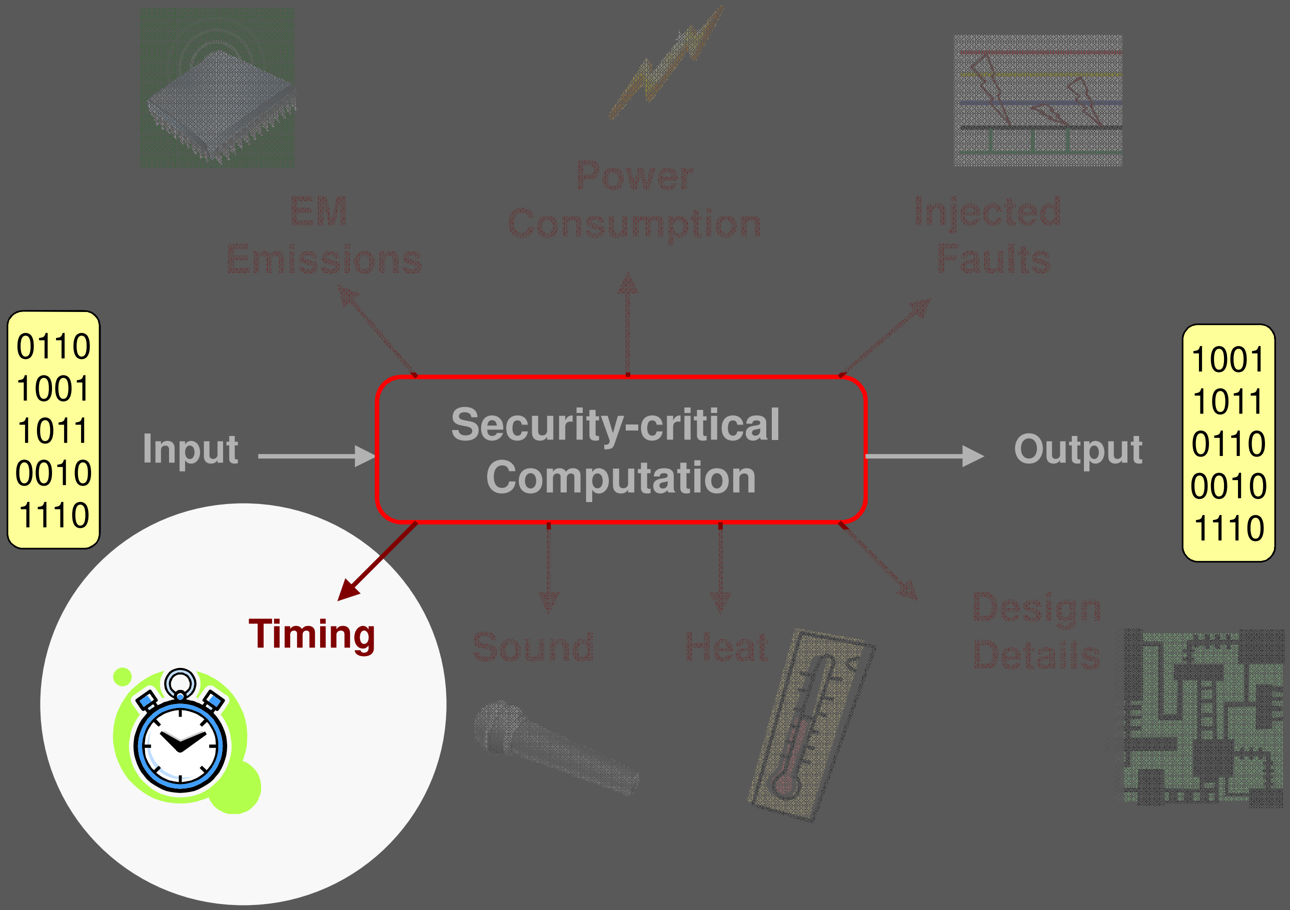
Side Channel: *The black magic behind it…*

Cache timing side channel leak
due to *speculative execution*

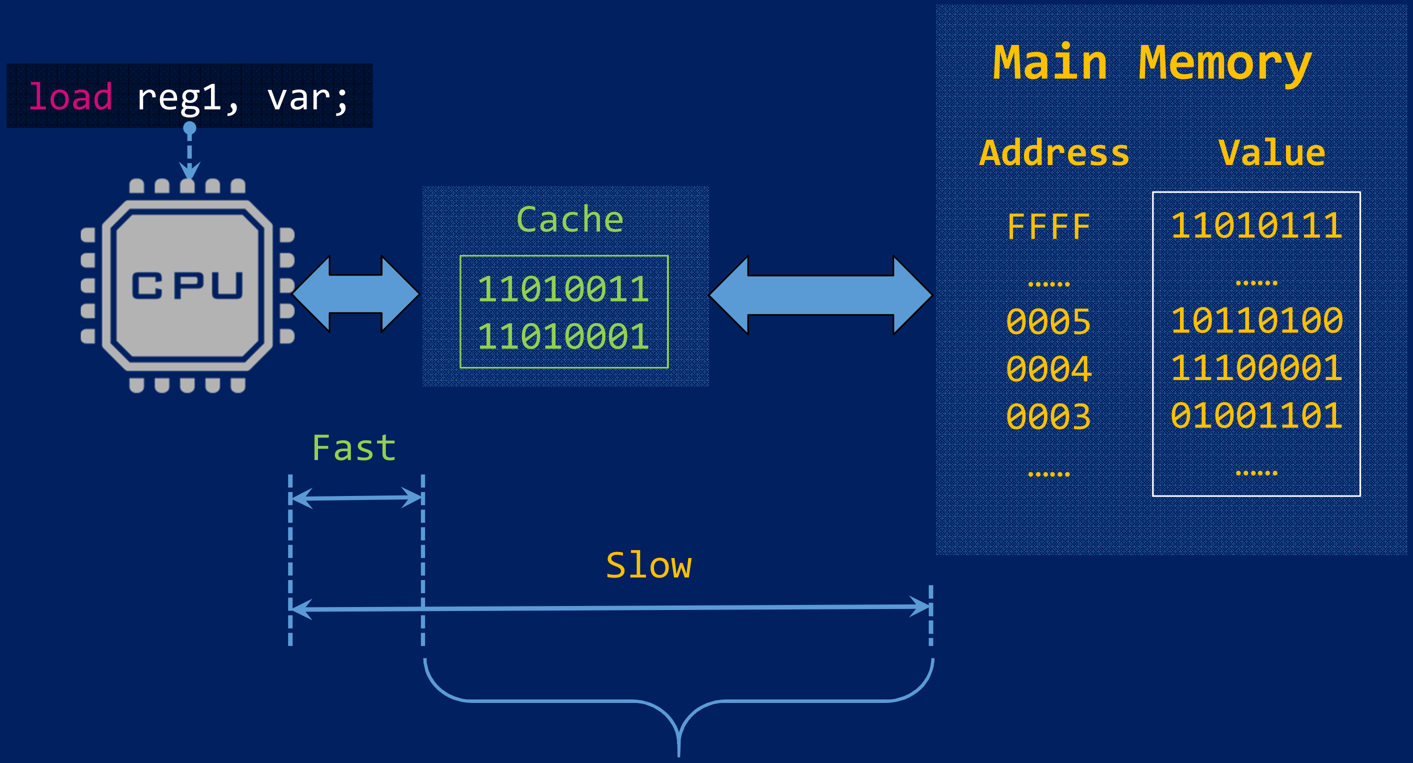
Other Side Channels



Timing Side Channel



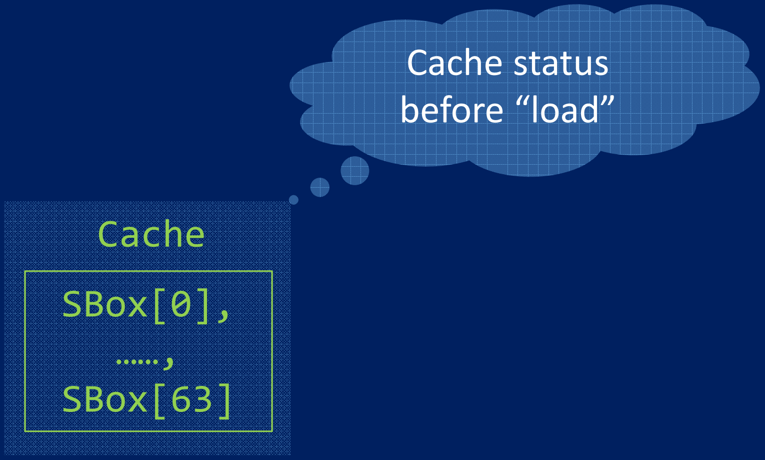
Cache Timing

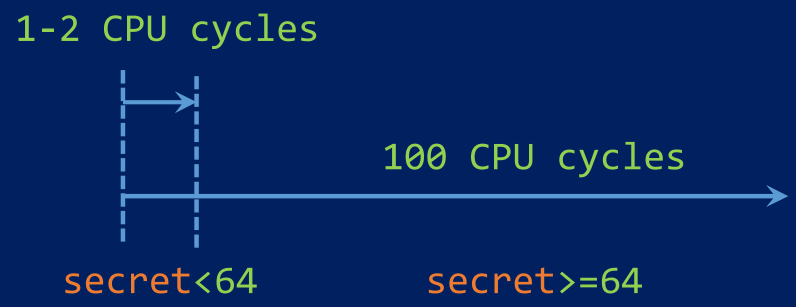


The timing difference may leak sensitive information

Cache Timing Leaks

```
void leak(uint8_t secret){
      uint8_t SBox[256]={…};
      ……
      load reg1, SBox[secret];
      ……
}
```





Measure the execution time of the load

Related Work

Leakage-free Verification
- *Zhang et.al [CAV’2018]*
- *Antonopoulos et.al [PLDI’2017]*
- *Chen et.al [CCS’2017]*
- ……

Static Analysis & Trace Analysis
- *Wu et.al [ISSTA’2018]*
- *Sung et.al [ASE’2018]*
- *Wang et.al [Usenix Sec’2017]*
- ……

Worst-case Execution Time
- *Basu et.al [ICST’2017]*
- *Chattopadhyay et.al [TACAS’2017]*
- *Touzeau et.al [CAV’2017]*
- ……

Degree of Leaked Information
- *Brennan et.al [ICSE’2018]*
- *Bang et.al [FSE’2016]*
- *Pasareanu et.al [CSF’2016]*
- ……

Self-leaks in the sequential program when running alone

Our Key Insight

Timing-leak-freedom

is **NOT** a compositional property.

A leak-free program (when running alone) may still leak sensitive timing information when interleaved with other threads, if they share the same CPU and memory subsystem.

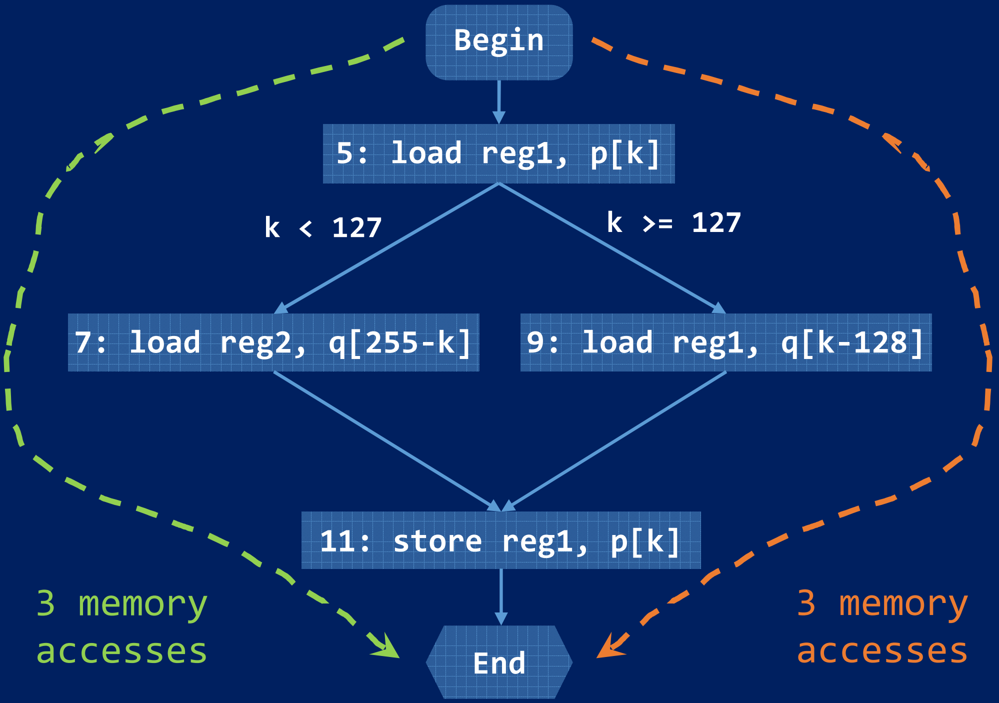
Outline

A Self-leaking Program

[Chattopadhyay et.al, MEMOCODE’2017]

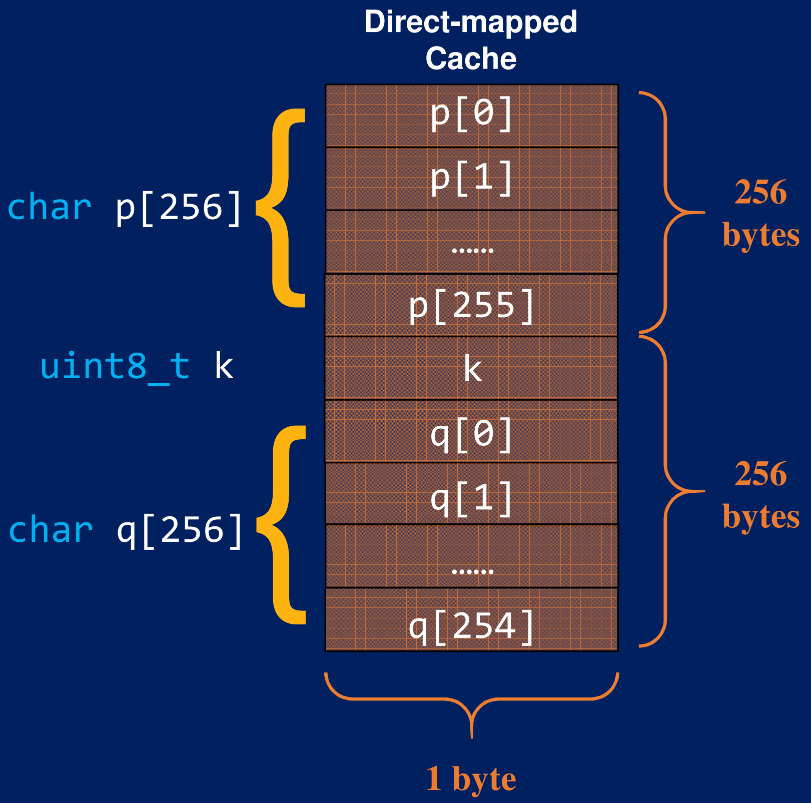
```
/* k is the sensitive input */
1:   char p[256];
2:   uint8_t k;
3:   char q[256];
4:
5:   load reg1, p[k];
6:
7:   if (k < 127)
8:       load reg2, q[255-k];
9:   else
10:      load reg2, q[128-k];
11:
12:  add reg1, reg2;
13:  store reg1, p[k];
```

Control Flow



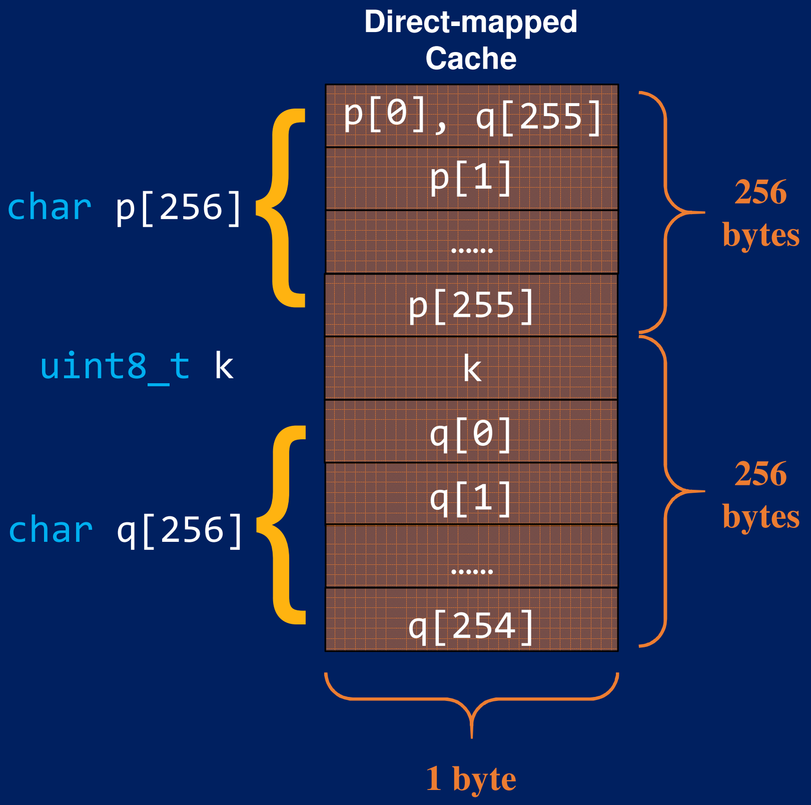
Cache Mapping

```
/* k is sensitive input */
char p[256];
uint8_t k;
char q[256];

load reg1, p[k];

if (k < 127)
    load reg2, q[255-k];
else
    load reg2, q[128-k];

add reg1, reg2;
store reg1, p[k];
```

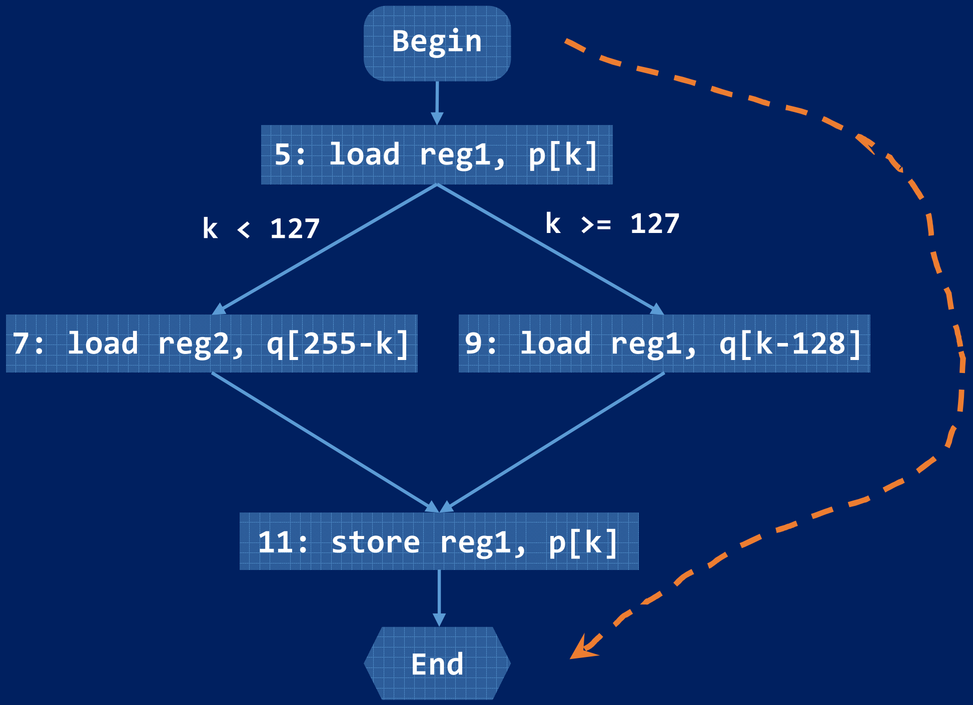


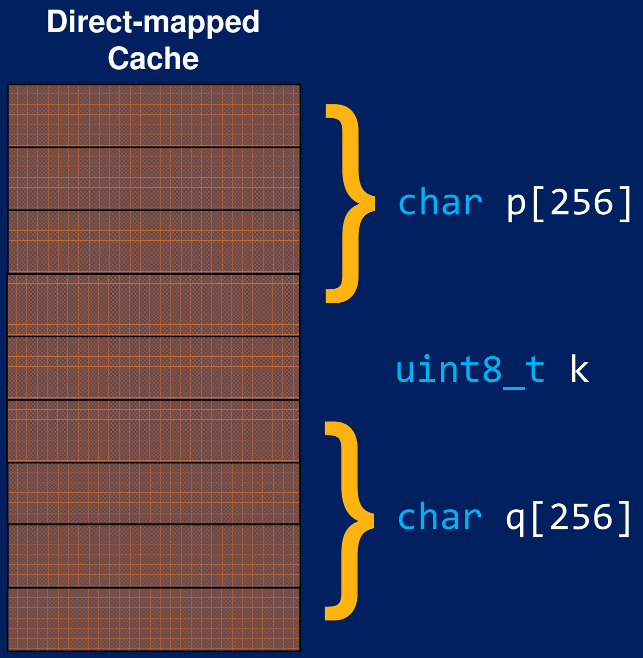
Cache Mapping

```
/* k is sensitive input */
char p[256];
uint8_t k;
char q[256];

load reg1, p[k];

if (k < 127)
    load reg2, q[255-k];
else
    load reg2, q[128-k];

add reg1, reg2;
store reg1, p[k];
```



Constant Cache Timing in Path 1

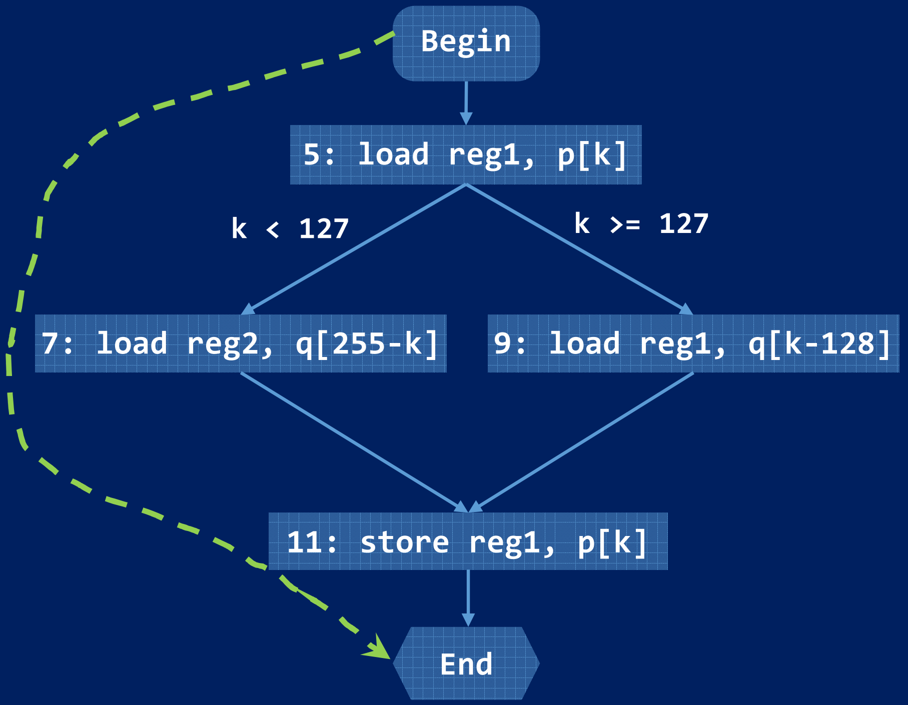


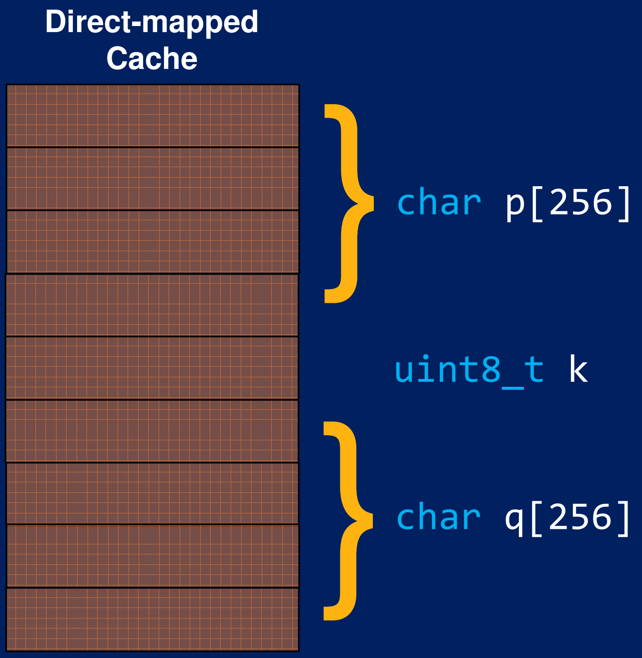


Path	Sensitive input k	Cache behavior
Right	127<= k <=255	

Same cache behavior for all $k \in [127, 255]$

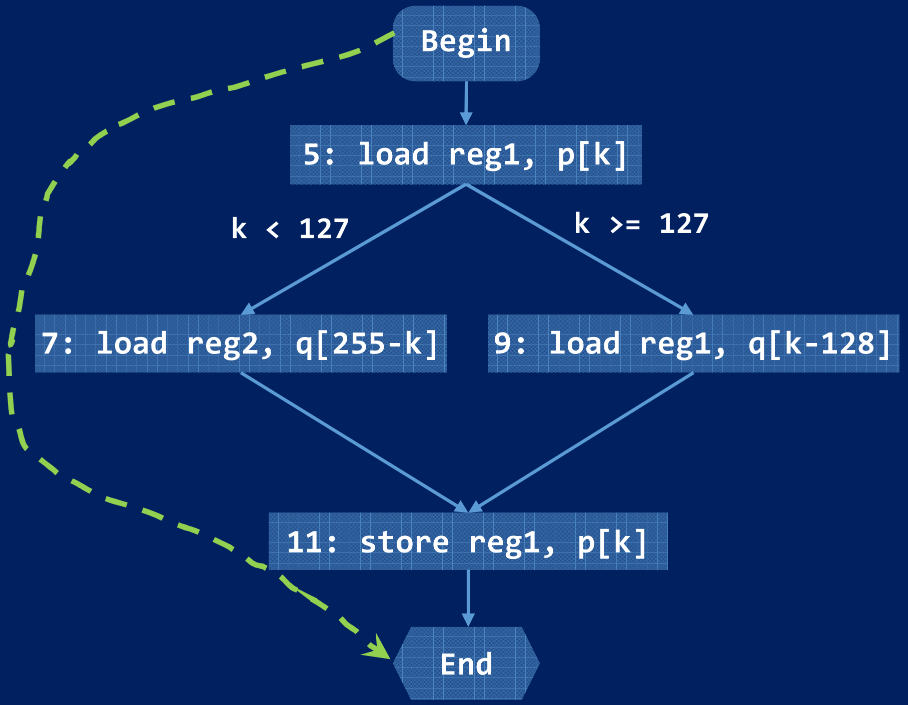
Constant Cache Timing in Path 2

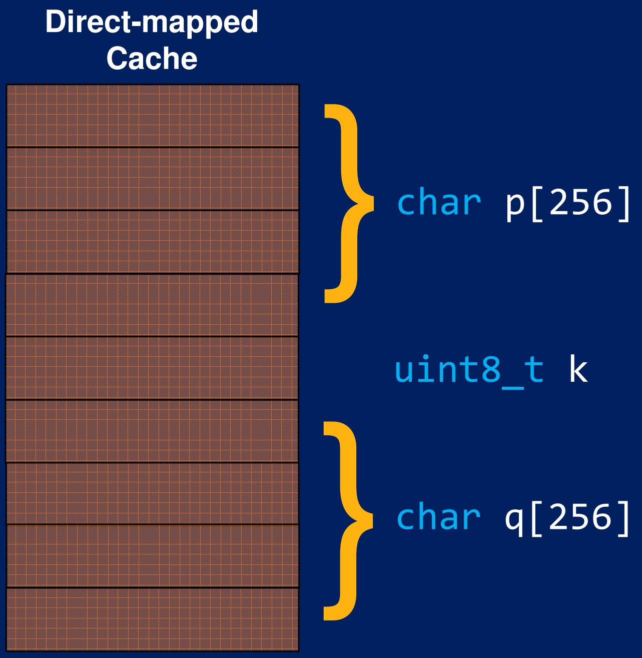




Path	Sensitive input k	Cache behavior
Right	127<= k <=255	miss miss hit
Left	0< k <127	

Divergent Cache Timing in Path 2

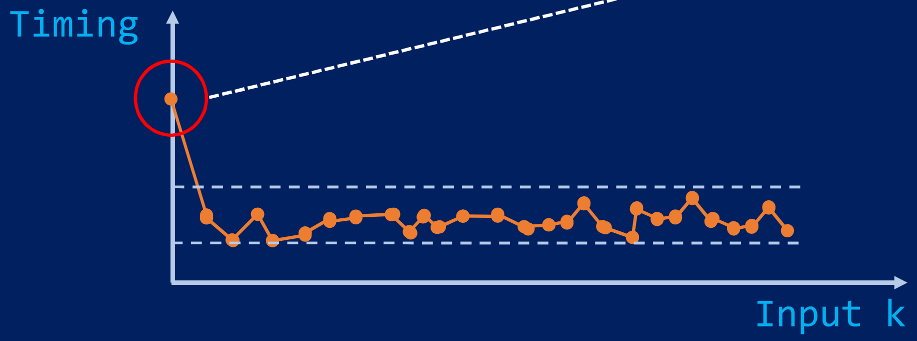




Path	Sensitive input k	Cache behavior
Right	127<= k <=255	miss miss hit
Left	0< k <127	miss miss hit
Left	k==0	

Timing Leak

Path	Sensitive input k	Cache behavior		
Right	127<= k <=255	miss	miss	hit
Left	0< k <127	miss	miss	hit
Left	k==0	miss	miss	miss



LEAK similar execution time ≈ k != 0
obviously more time ≈ k = 0

Leak Mitigation

Before Mitigation

```
/* k is sensitive input */
char p[256];
uint8_t k;
char q[256];

load reg1, p[k];

if (k < 127)
    load reg2, q[255-k];
else
    load reg2, q[128-k];

add reg1, reg2;
store reg1, p[k];
```

```
Path1: P[k], q[255-k], p[k]
Path2: P[k], q[k-128], p[k]
```

[Chattopadhyay et.al, MEMOCODE’2017]

Leak Mitigation

Mitigation --- Moving line 5 to line 10

```
/* k is sensitive input */
1:  char p[256];
2:  uint8_t k;
3:  char q[256];
4:
5:  if (k < 127)
6:      load reg2, q[255-k];
7:  else
8:      load reg2, q[128-k];
9:
10: load reg1, p[k];
11:
12: add reg1, reg2;
13: store reg1, p[k];
```

```
Path1: q[255-k], P[k], p[k]
Path2: q[k-128], P[k], p[k]
```

No matter what the value (k) is

- **Always miss – miss – hit**
- **Constant cache timing**

[Chattopadhyay et.al, MEMOCODE'2017]

Our Question

What if executing the program with a second thread?

Really Secure ?

A New Two-thread Program

The mitigated program

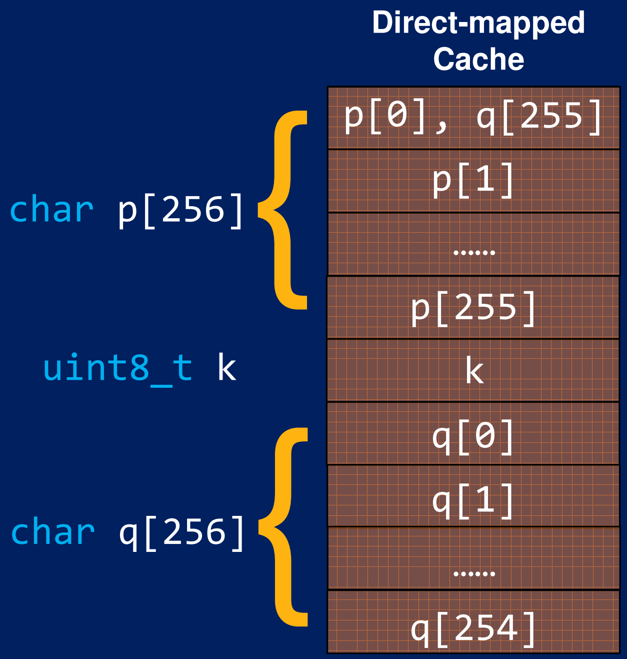
Thread T1

```
char p[256];
uint8_t k;
char q[256];

if (k < 127)
    load reg2, q[255-k];
else
    load reg2, q[128-k];

load reg1, p[k];

add reg1, reg2;
store reg1, p[k];
```

The new thread

Thread T2

```
uint8_t tmp;
load reg3, tmp
……
```



Really Secure ?

A New Two-thread Program

The mitigated program

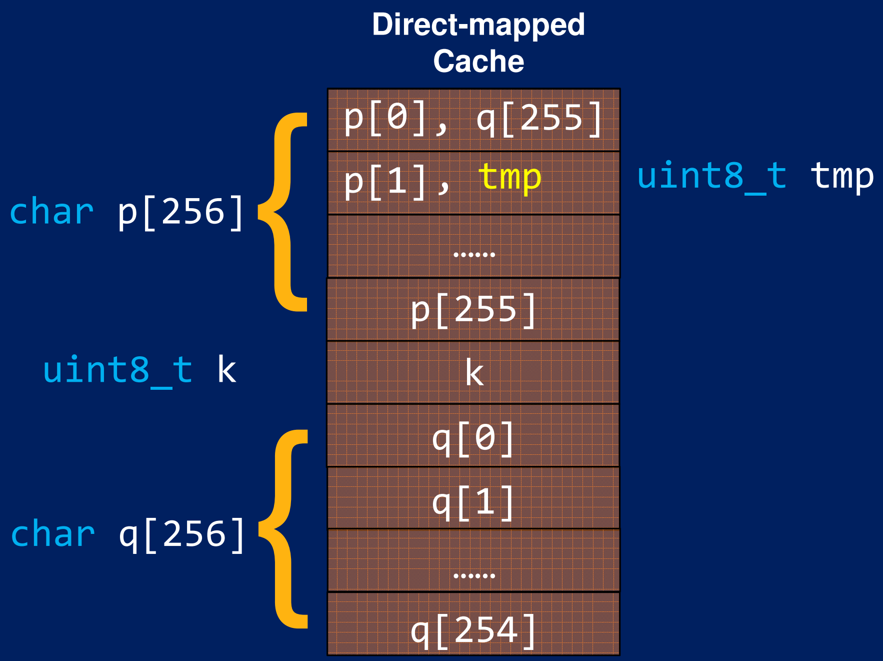
Thread T1

```
char p[256];
uint8_t k;
char q[256];

if (k < 127)
    load reg2, q[255-k];
else
    load reg2, q[128-k];

load reg1, p[k];

add reg1, reg2;
store reg1, p[k];
```

The new thread

Thread T2

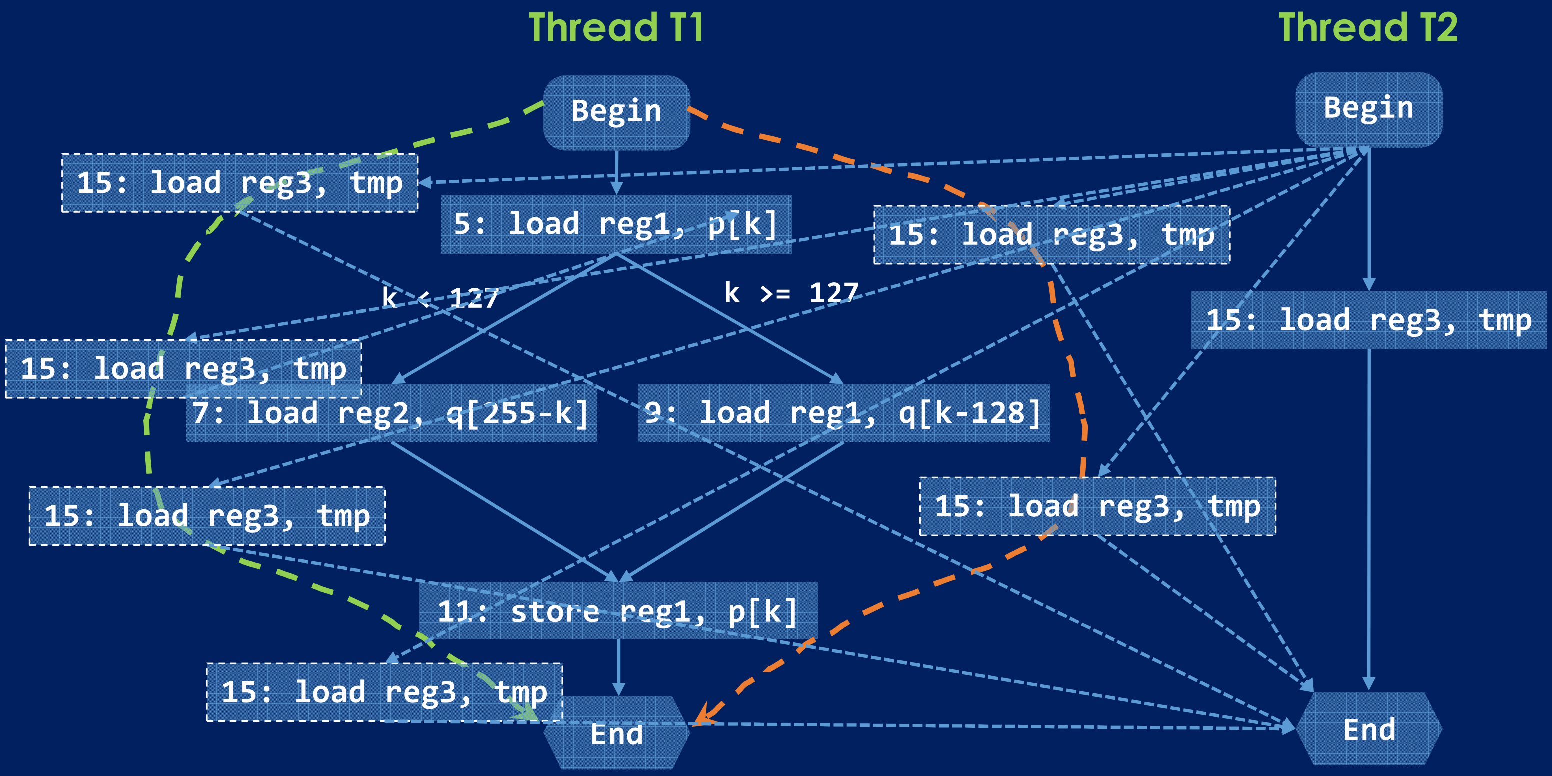
```
uint8_t tmp;
load reg3, tmp
……
```



Question

Can we still infer k by measuring the cache timing?

Interleaved Executions



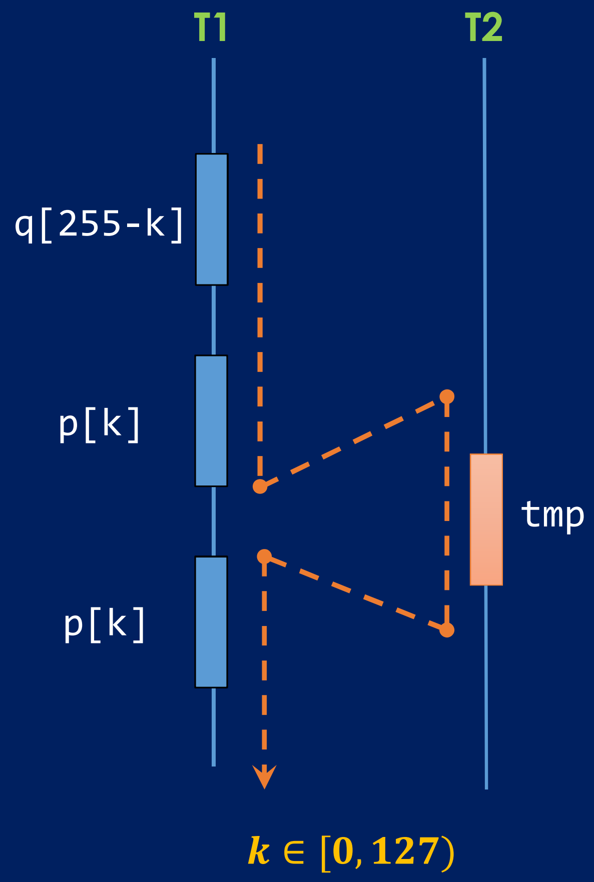
A Specific Interleaving

Thread T1

```
char p[256];
uint8_t k;
char q[256];

if (k < 127)
    load reg2, q[255-k];
else
    load reg2, q[128-k];

load reg1, p[k];

add reg1, reg2;
store reg1, p[k];
```

Thread T2

```
uint8_t tmp;
load reg3, tmp
……
```



A Specific Interleaving

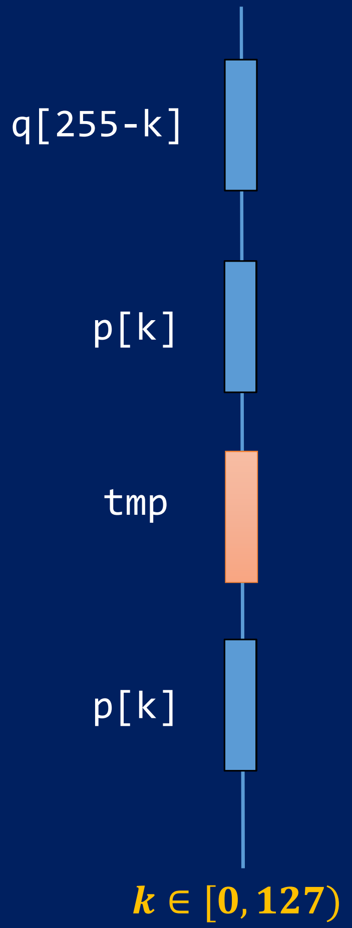
Thread T1

```
char p[256];
uint8_t k;
char q[256];

if (k < 127)
    load reg2, q[255-k];
else
    load reg2, q[128-k];

load reg1, p[k];

add reg1, reg2;
store reg1, p[k];
```

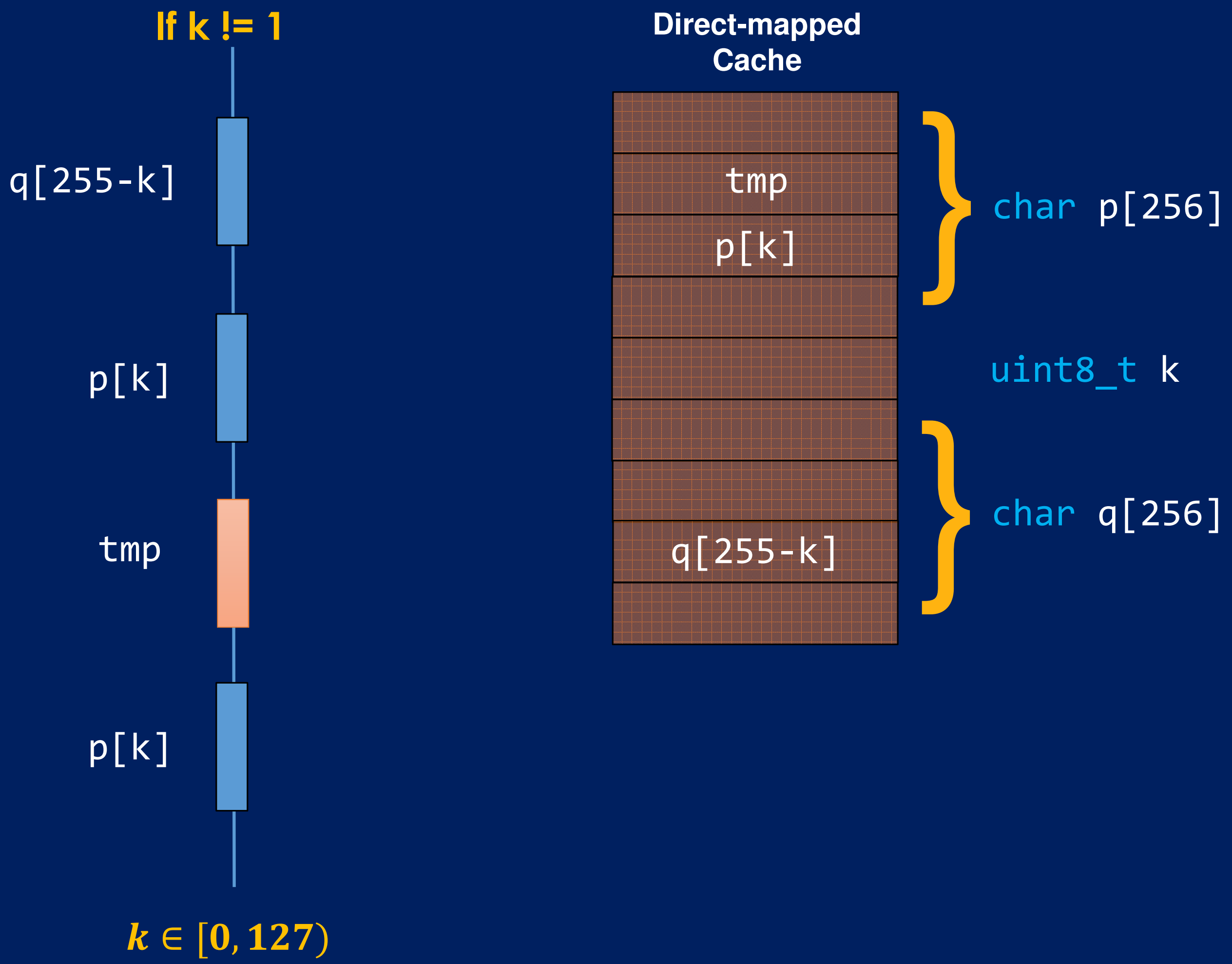
Thread T2

```
uint8_t tmp;
load reg3, tmp
……
```

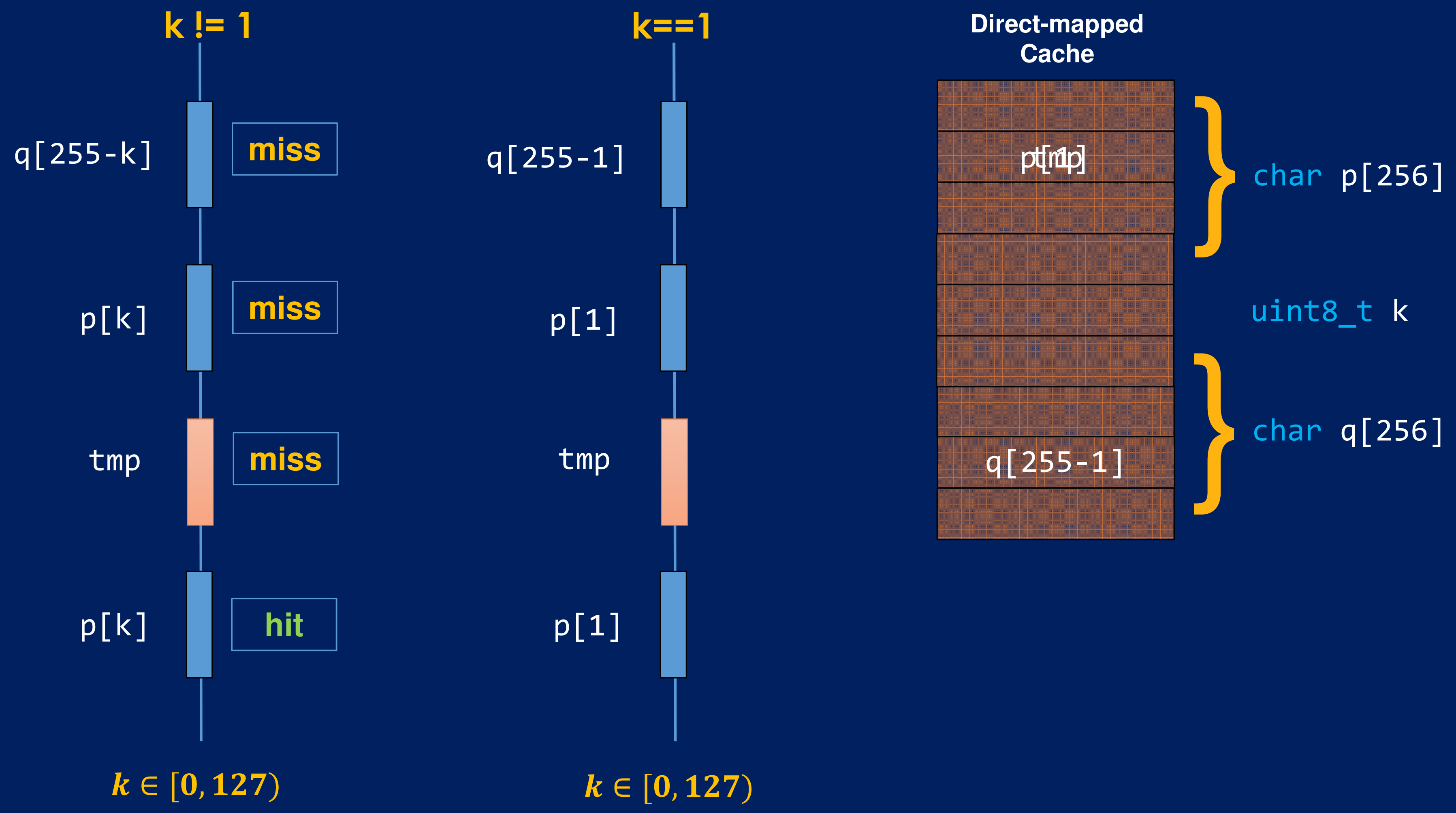
Interleaving



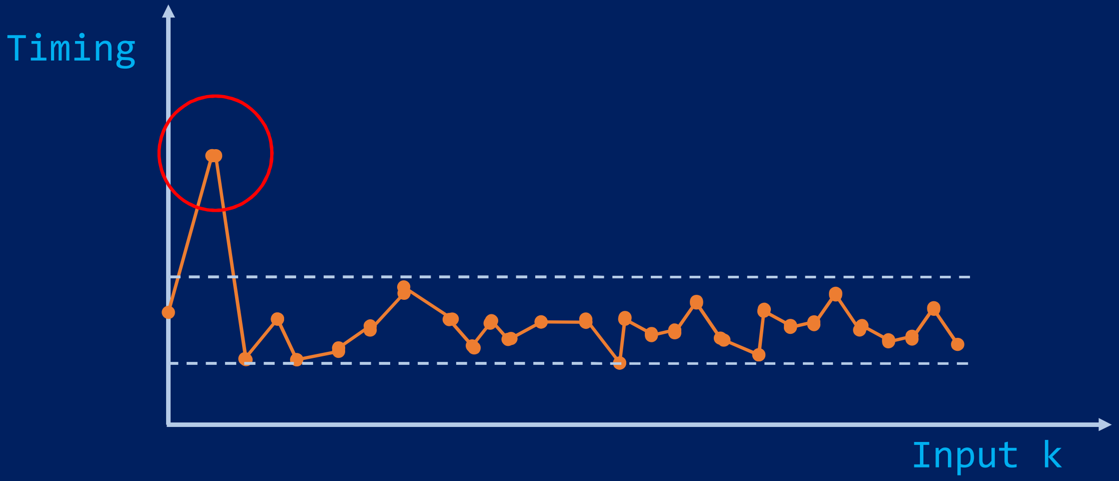
A Specific Interleaving



A Specific Interleaving



Timing Leak Again



LEAK

similar execution time ≈ k != 1

obviously more time ≈ k = 1

Challenges

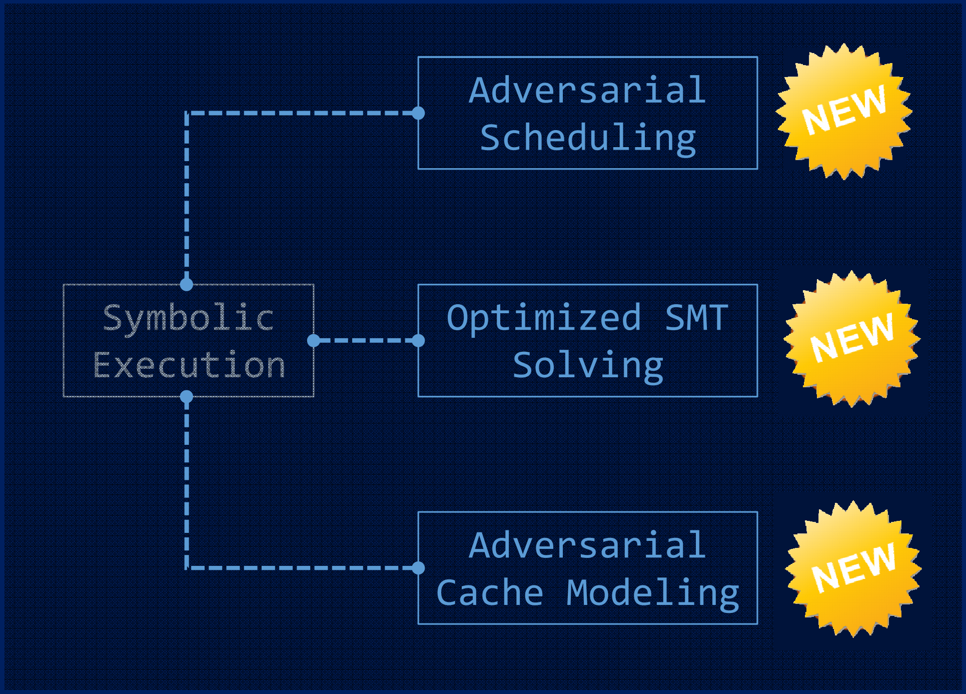
How to find the exact interleaving?

How to analyze the cache behavior?

How to obtain the right value of k?

Outline

In a Nutshell



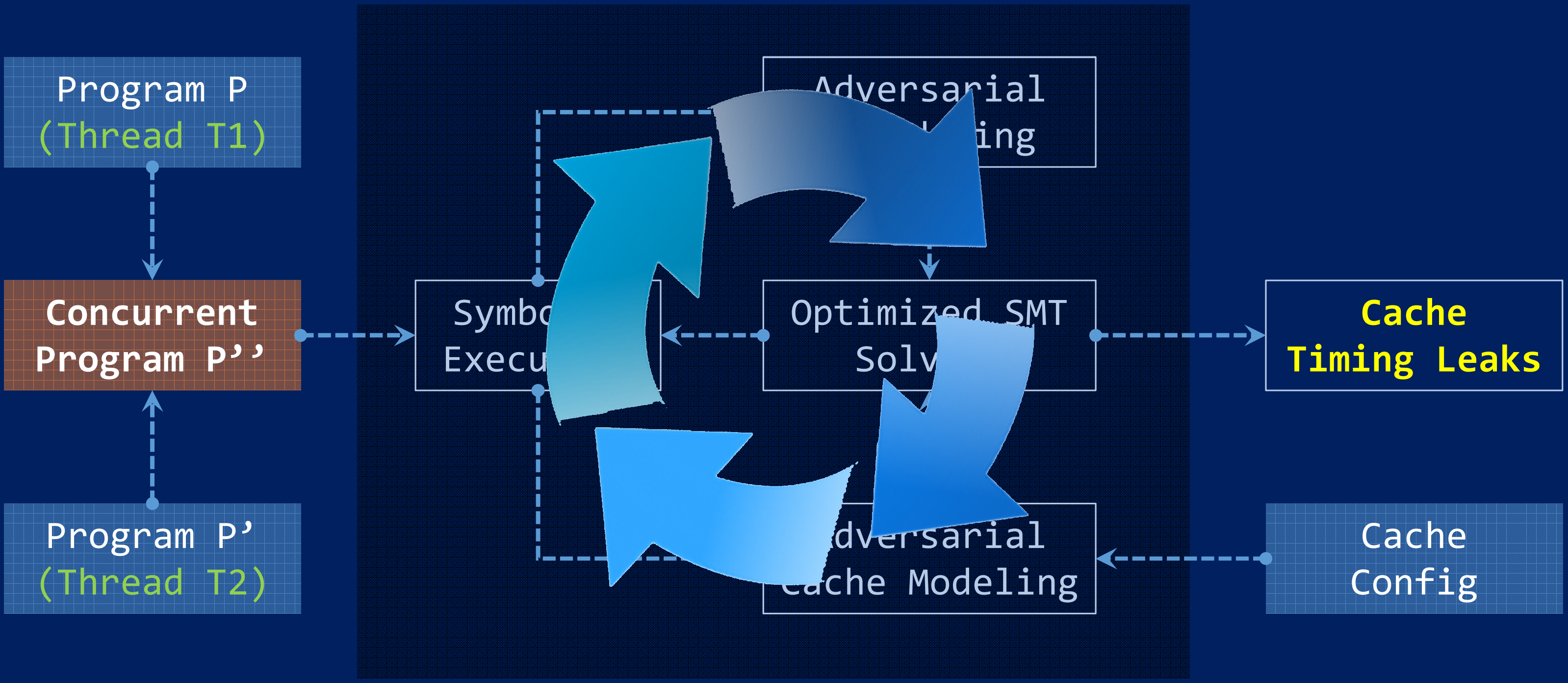
Find Interleaving?

Generate Input ?

Analyze Cache ?

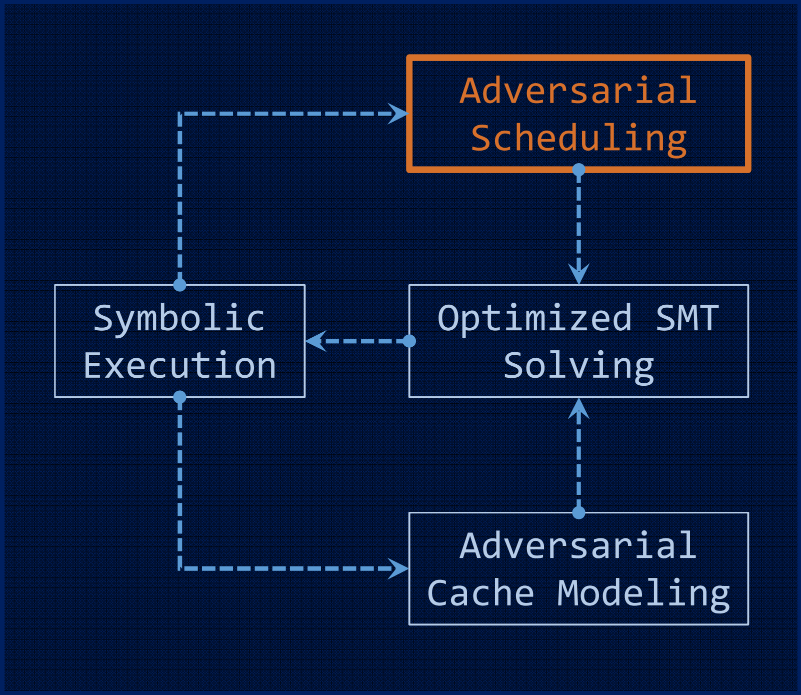
Our Adversarial Symbolic Execution

Overall Flow



Our Adversarial Symbolic Execution

Adversarial Scheduling

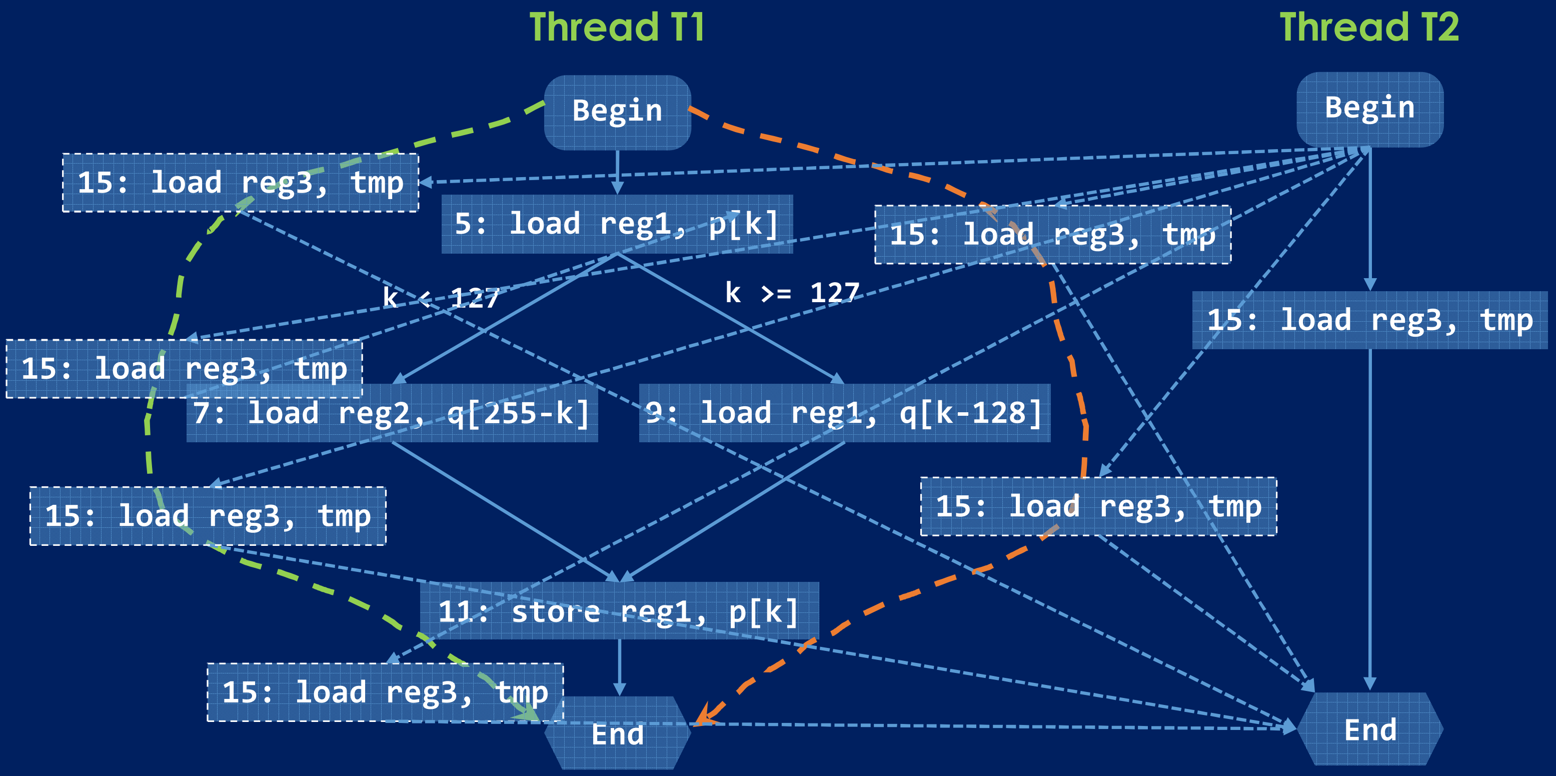


Our Adversarial Symbolic Execution

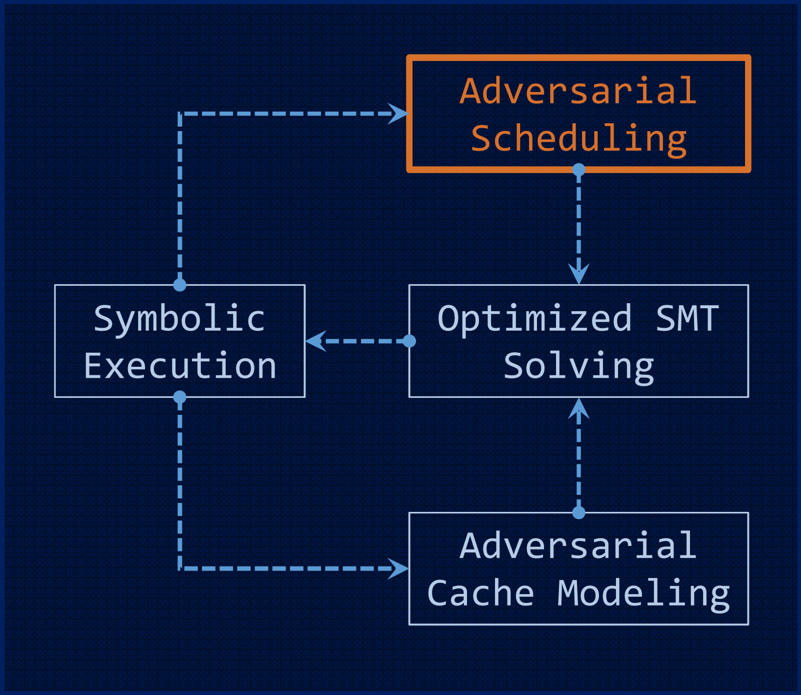
1. State search space can be large
2. Many interleavings are redundant

The State Space

8 Different Interleavings --- only 1 causes timing leak

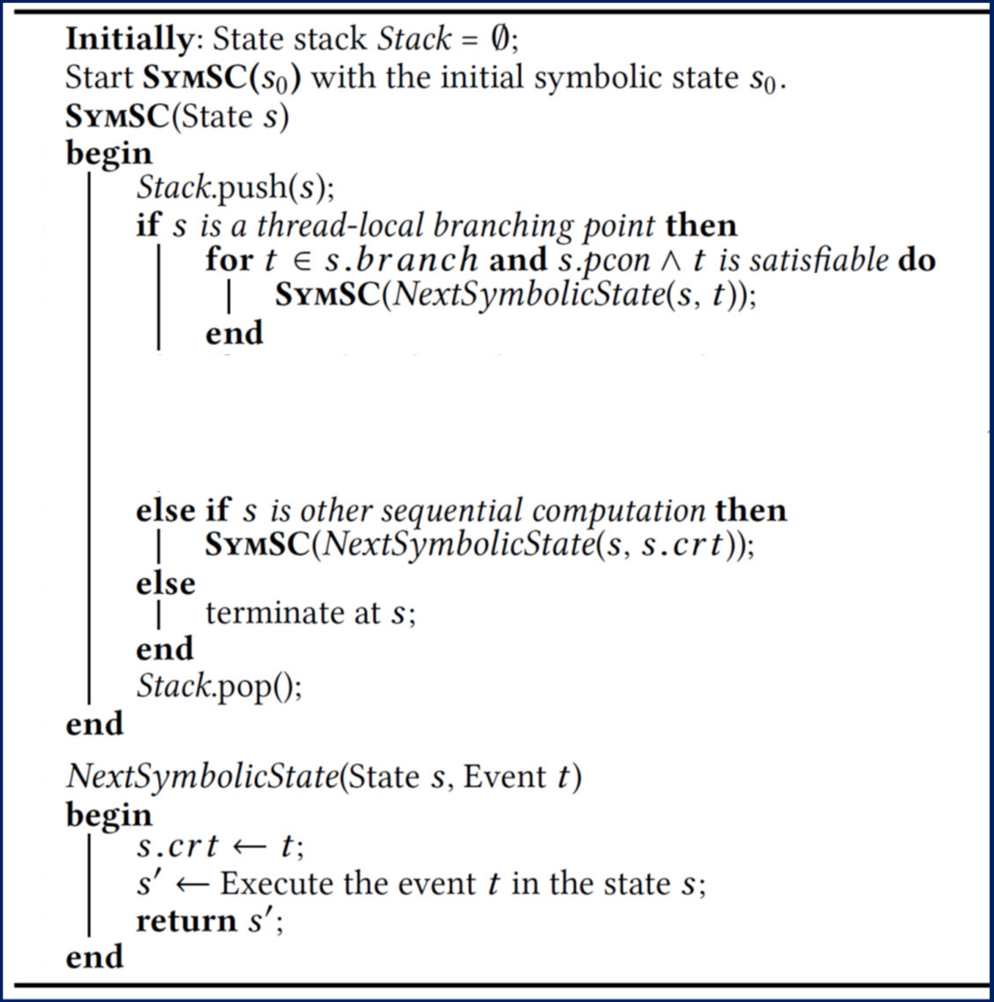


Adversarial Scheduling



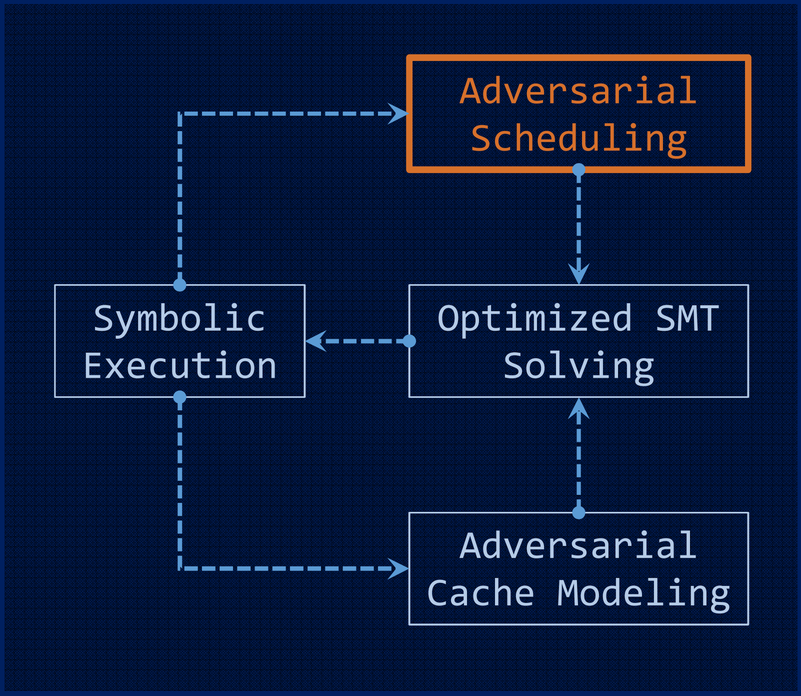
Our Adversarial Symbolic Execution

Symbolic execution (sequential)



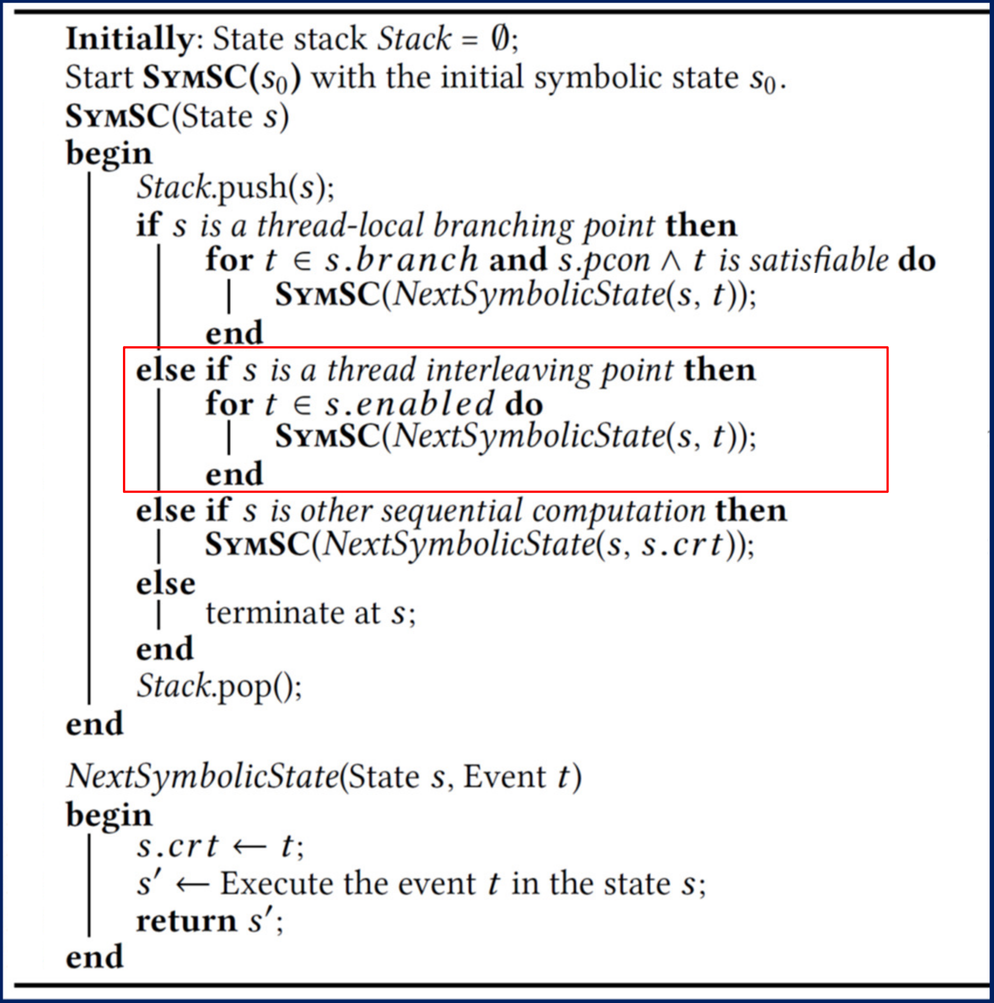
```
Initially: State stack Stack = ∅;
Start SymSC($s_0$) with the initial symbolic state $s_0$.
SymSC(State $s$)
begin
    Stack.push($s$);
    if $s$ is a thread-local branching point then
        for $t \in s.branch$ and $s.pcon \wedge t$ is satisfiable do
            SymSC(NextSymbolicState($s$, $t$));
        end
    else if $s$ is other sequential computation then
        SymSC(NextSymbolicState($s$, $s.crt$));
    else
        terminate at $s$;
    end
    Stack.pop();
end

NextSymbolicState(State $s$, Event $t$)
begin
    $s.crt \leftarrow t$;
    $s' \leftarrow$ Execute the event $t$ in the state $s$;
    return $s'$;
end
```

Adversarial Scheduling



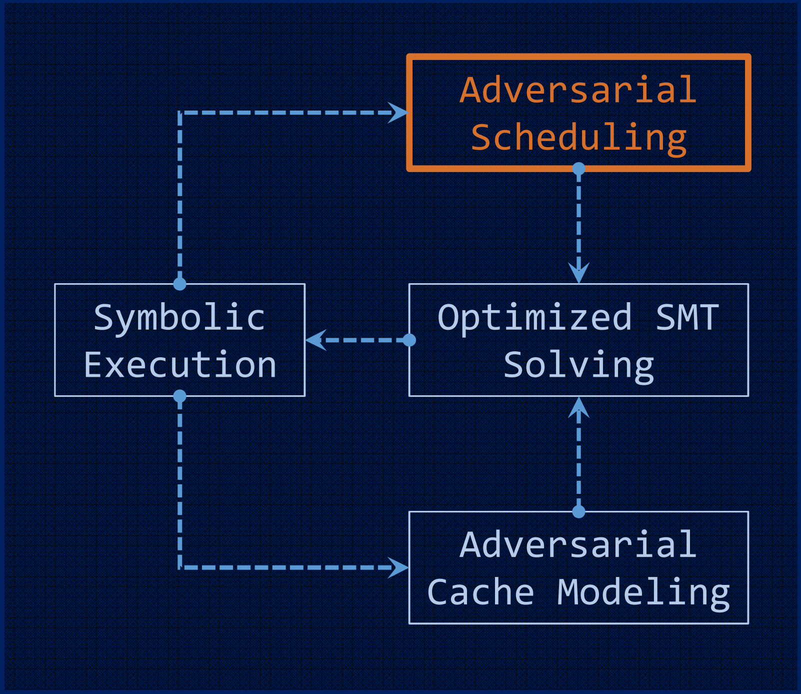
Our Adversarial Symbolic Execution

Symbolic execution (concurrent)



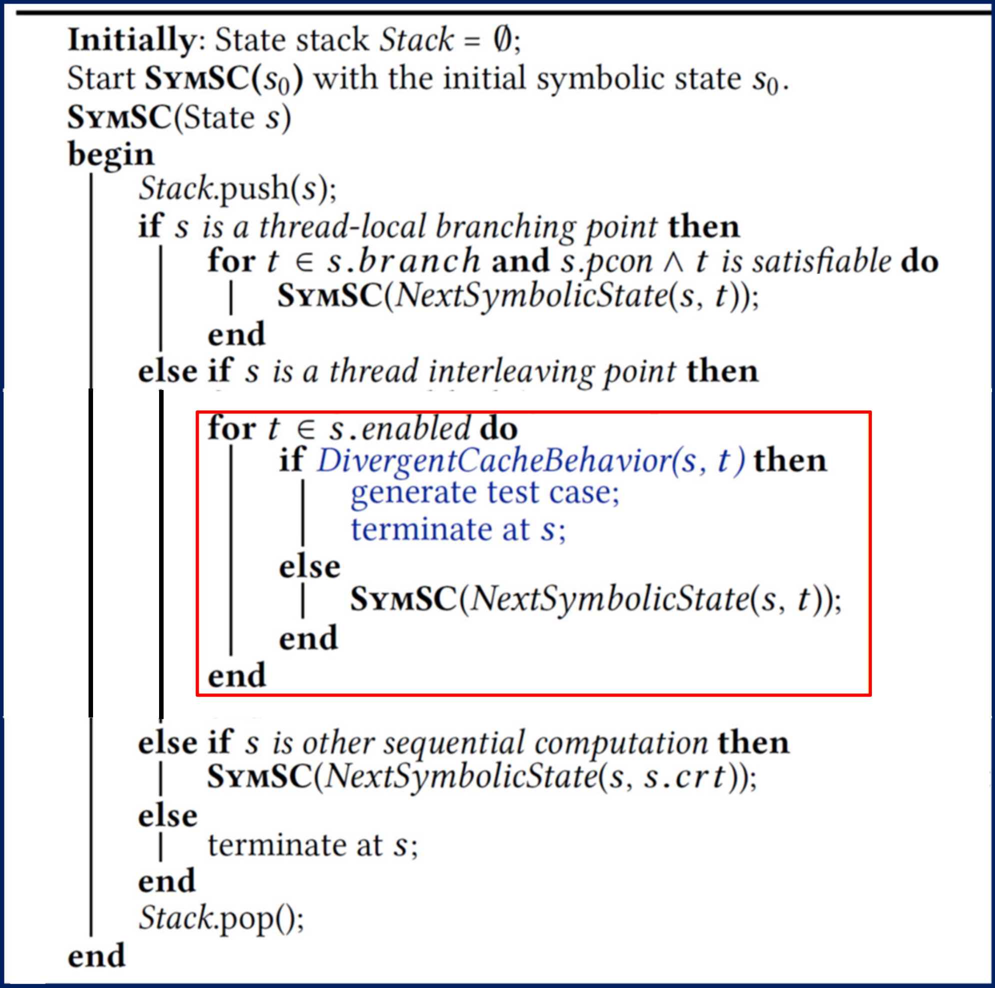
```
Initially: State stack $Stack = \emptyset$;
Start SymSC($s_0$) with the initial symbolic state $s_0$.
SymSC(State $s$)
begin
    $Stack$.push($s$);
    if $s$ is a thread-local branching point then
        for $t \in s.branch$ and $s.pcon \wedge t$ is satisfiable do
            SymSC(NextSymbolicState($s$, $t$));
        end
    else if $s$ is a thread interleaving point then
        for $t \in s.enabled$ do
            SymSC(NextSymbolicState($s$, $t$));
        end
    else if $s$ is other sequential computation then
        SymSC(NextSymbolicState($s$, $s.crt$));
    else
        terminate at $s$;
    end
    $Stack$.pop();
end

NextSymbolicState(State $s$, Event $t$)
begin
    $s.crt \leftarrow t$;
    $s' \leftarrow$ Execute the event $t$ in the state $s$;
    return $s'$;
end
```

Adversarial Scheduling



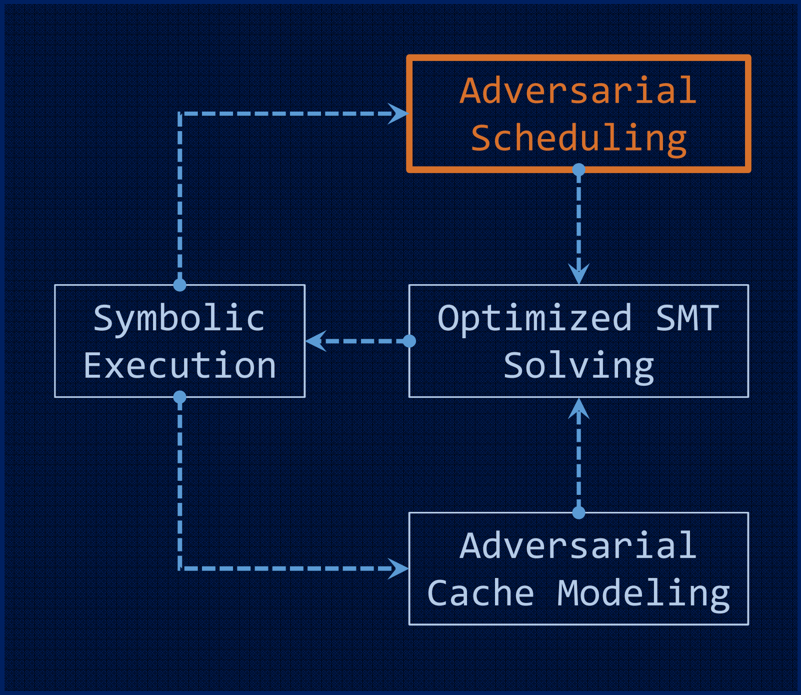
Our Adversarial Symbolic Execution

Symbolic execution (side channel)



```
Initially: State stack $Stack = \emptyset$;
Start SYMSC($s_0$) with the initial symbolic state $s_0$.
SYMSC(State $s$)
begin
    Stack.push($s$);
    if $s$ is a thread-local branching point then
        for $t \in s.branch$ and $s.pcon \wedge t$ is satisfiable do
            SYMSC(NextSymbolicState($s$, $t$));
        end
    else if $s$ is a thread interleaving point then
        for $t \in s.enabled$ do
            if DivergentCacheBehavior($s$, $t$) then
                generate test case;
                terminate at $s$;
            else
                SYMSC(NextSymbolicState($s$, $t$));
            end
        end
    else if $s$ is other sequential computation then
        SYMSC(NextSymbolicState($s$, $s.crt$));
    else
        terminate at $s$;
    end
    Stack.pop();
end
```

Adversarial Scheduling

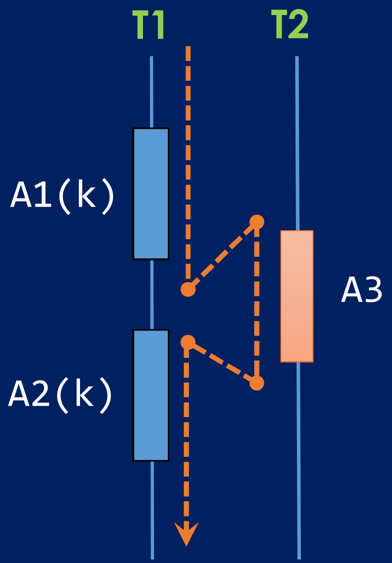


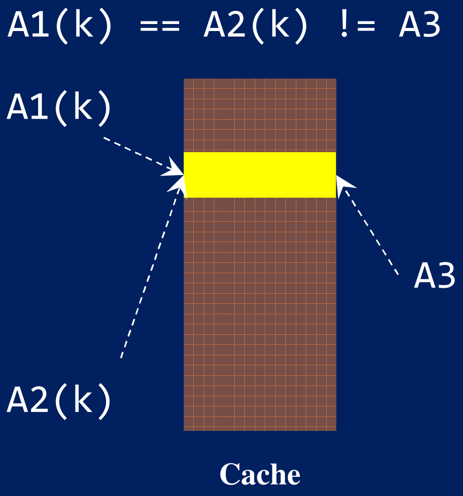
Our Adversarial Symbolic Execution

Only Create New Schedules

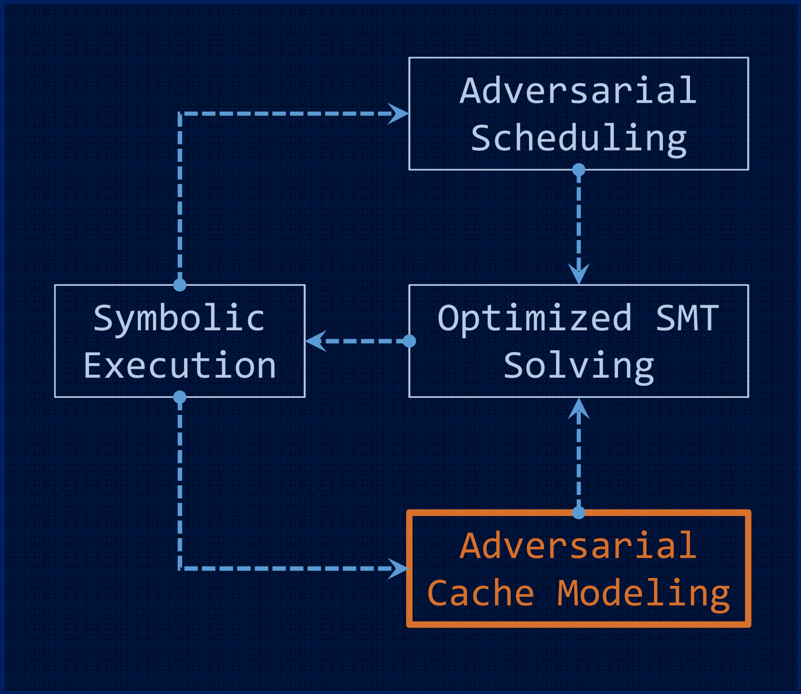
- At symbolic memory accesses
- Two memory accesses can cause cache hit
- An adversarial access may cause cache miss

$input\ k = *$

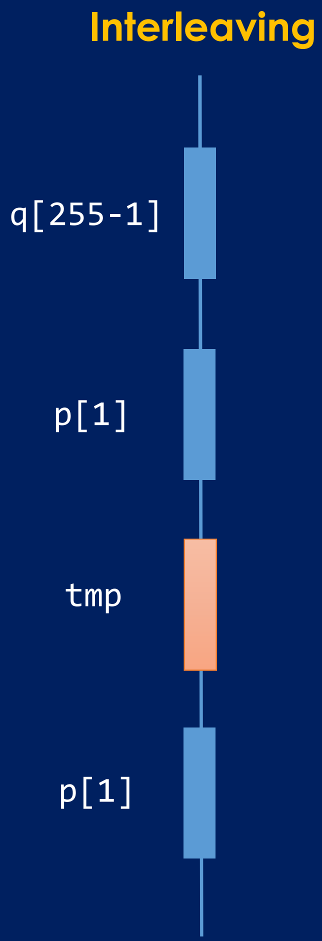




Cache Modeling



Our Adversarial Symbolic Execution



An ordered sequence of memory events:

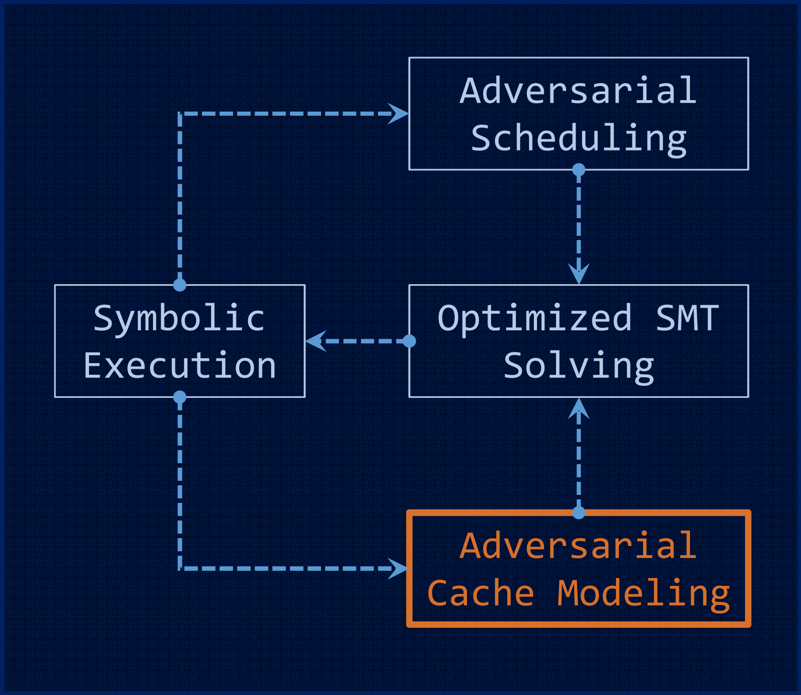
- A_0 : load q[254]
- A_1 : load p[1]
- A_2 : load &tmp
- A_3 : store p[1]

$\boldsymbol{addr_i}$:
the memory address accessed by A_i

$\boldsymbol{tag\ (addr_i)}$:
the unique tag of $addr_i$

$\boldsymbol{line\ (addr_i)}$:
the cache line that $addr_i$ maps to

Cache Modeling



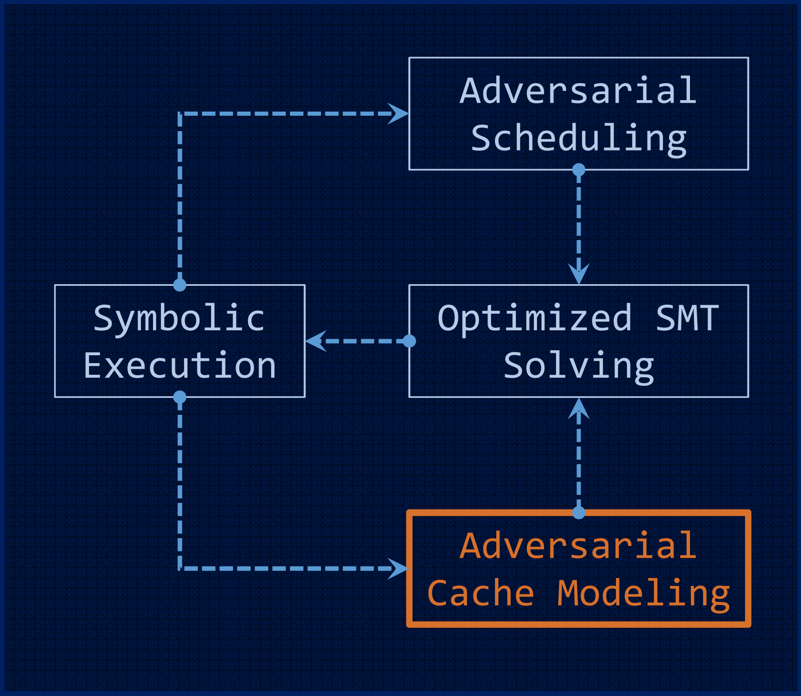
Our Adversarial Symbolic Execution

The ***Cache-hit*** **condition** of A_i on sensitive input k :

$$\tau_i(k) \equiv \bigcup_{0 \le j < i} \left(\begin{array}{c} tag(addr_j) = tag(addr_i) \wedge \\ \forall l \in [j+1, i-1]\ line(addr_l) \neq line(addr_i) \end{array} \right)$$

- An event A_j happed before A_i
- *A_j and A_i access the same address*
- *A_j still in cache when reaching A_i*

Leak Definition



Our Adversarial Symbolic Execution

Program P: a function $f(k, x) = c$

- k is sensitive input
- x is insensitive input

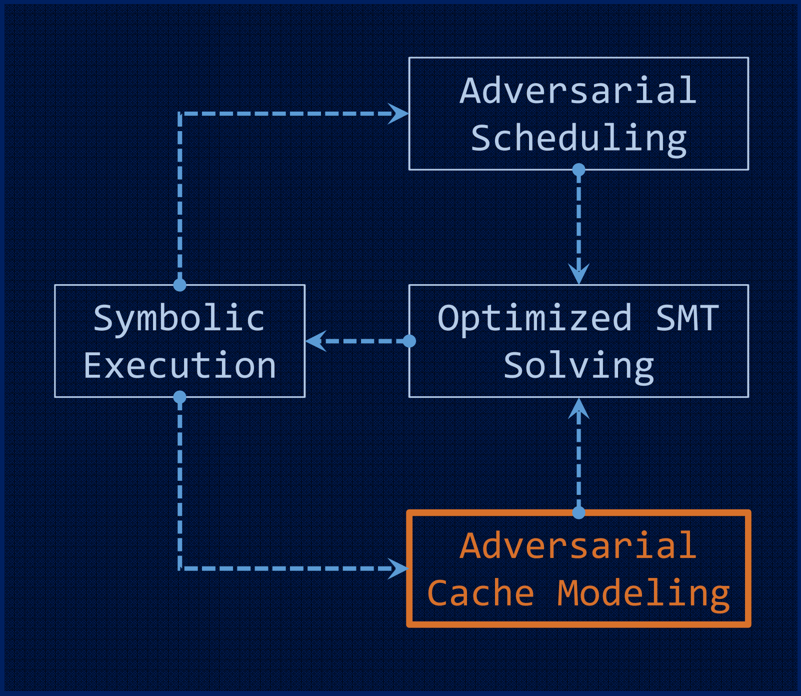
P is leakage-free if

- $\tau_i(k, x)$ is the same for all sensitive inputs

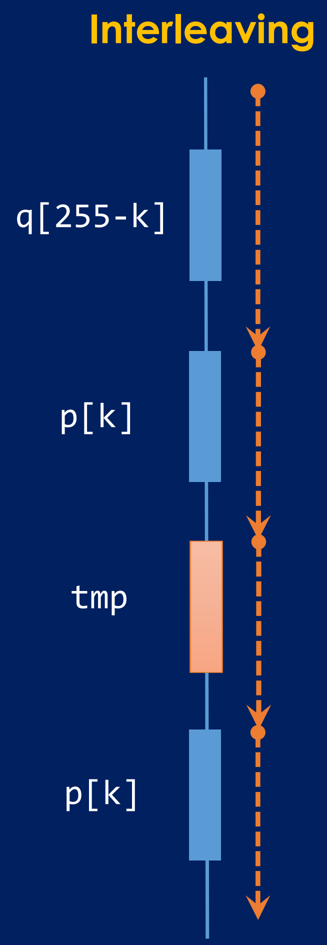
P has leakage if

- Exists two sensitive inputs k_1 and k_2
- $\boxed{\tau_i(k_1, x) \neq \tau_i(k_2, x)}$

Constraint Construction



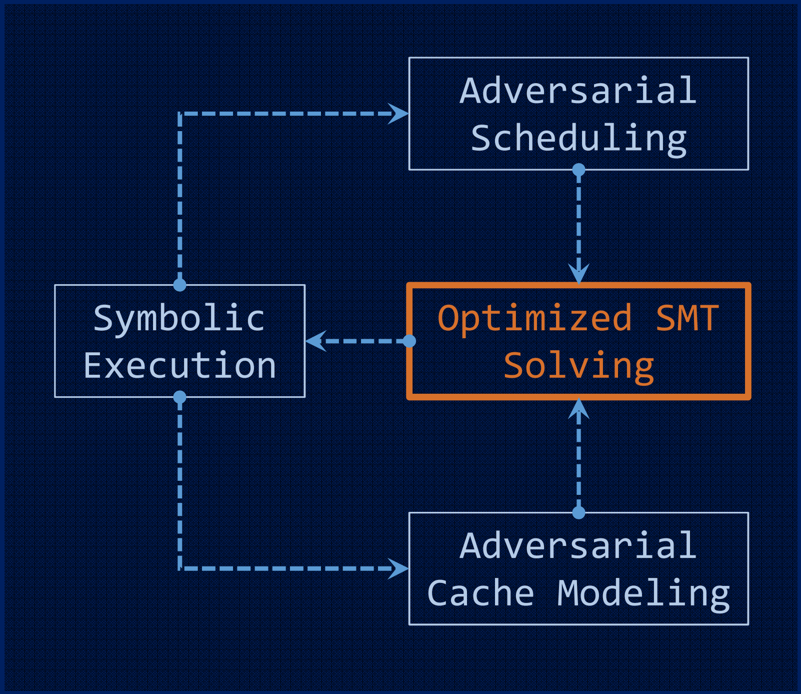
Our Adversarial Symbolic Execution



i	Hit condition	Cache
0	**False**	miss
1	tag(p[k])=tag(q[255-k])	miss
2	tag(tmp) = tag(p[k]) ∨ (tag(tmp) = tag(q[255-k]) ∧ line(tmp) ≠ line(p[k]))	miss
3	tag(p[k]) = tag(tmp) ∨ (tag(p[k]) = tag(p[k]) ∧ line(p[k]) ≠ line(tmp))	miss or hit

LEAK!

SMT Solving



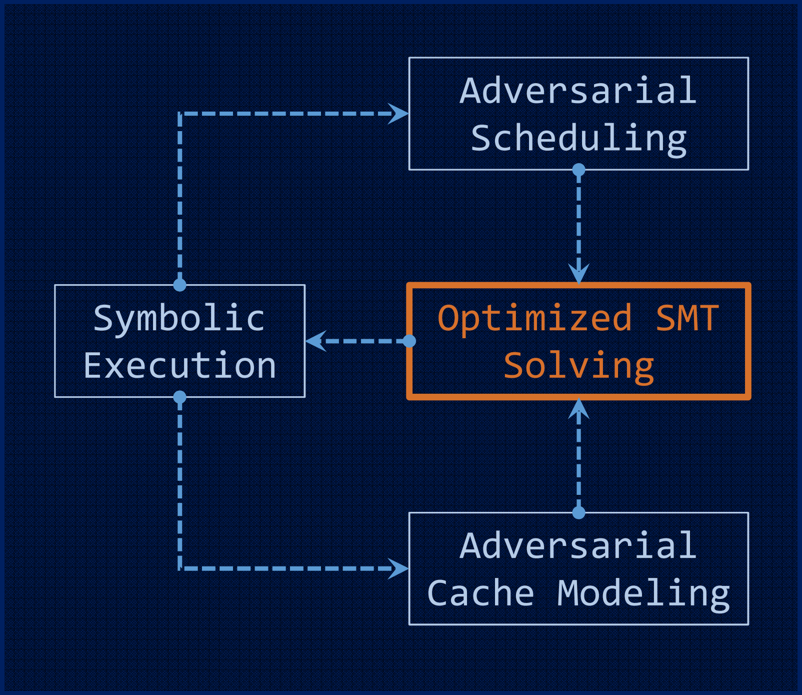
Our Adversarial Symbolic Execution

Precise Solution
Two-step Approximation

P has leakage if

$$\exists k_1, k_2 \mid \tau_i(k_1, x) \neq \tau_i(k_2, x)$$

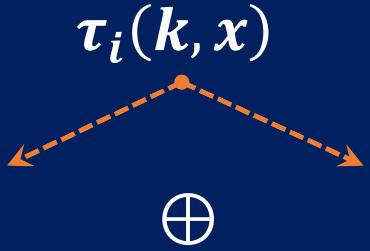
SMT Solving



Our Adversarial Symbolic Execution

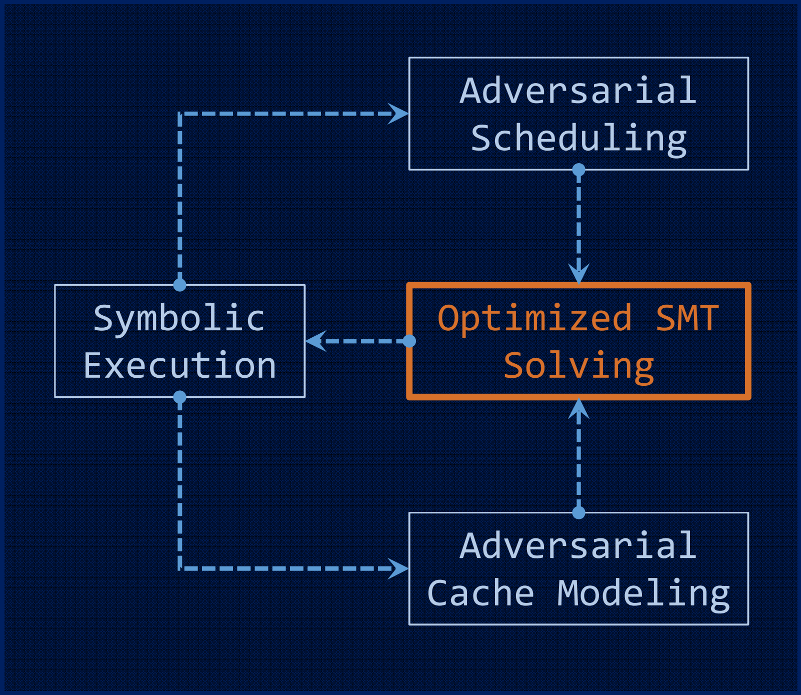
Precise Solution
Two-step Approximation

$$\exists k_1, k_2 \mid \tau_i(k_1, x) \neq \tau_i(k_2, x)$$



- Two fresh copies of $\tau_i(k, x)$
- Double-sized formula
- Precise, but also expensive

SMT Solving



Our Adversarial Symbolic Execution

Precise Solution
Two-step Approximation

$$\exists k_1, k_2 \mid \tau_i(k_1, x) \neq \tau_i(k_2, x)$$

$$1.\ \tau_i(k_1, x)$$

$$2.\ \neg\tau_i(k_2, x) \wedge (k_1' \neq k_2)$$

- Cheaper, step 1 returns if unsatisfiable
- Faster, a solution is promised to be valid
- May miss solution

Outline

Benchmarks

A Diverse Set of Real-world Cipher Programs

Source	Cipher Name
FELICS	Chaskey, Lblock, Piccolo, PRESENT, TWINE
Chronos	AES, CAST5, DES, Khazad, FCrypt
Libgcrpt	DES, rfc2268, Seed, Twofish
LibTomcrpt	Camellia, CAST5, Seed, Twofish
OpenSSL	AES
CHALICE	KV_Name

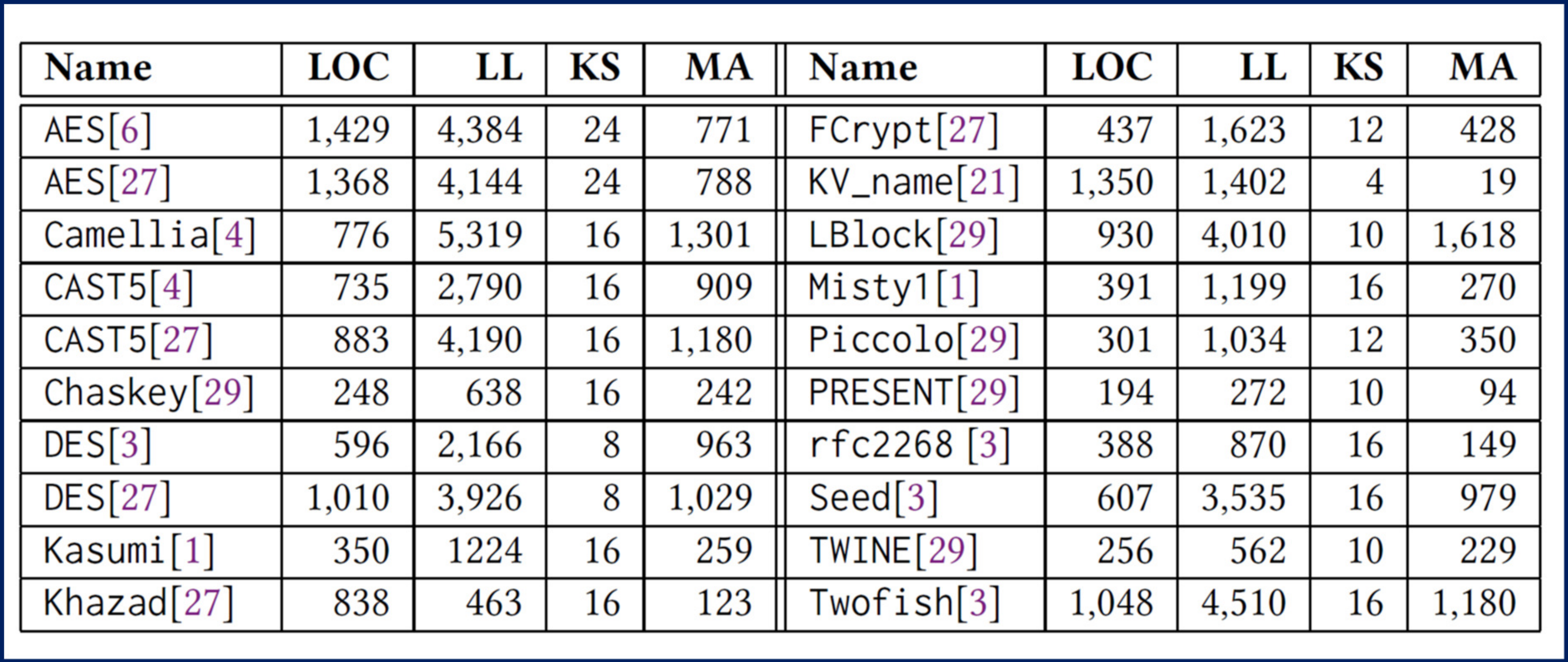
Benchmarks

Benchmark Statistics

Item	Statistics
Programs	20
Sources	6
Lines of Code	194 -- 1429
Sensitive Input Size	8 bytes -- 24 bytes
Memory Accesses	19 -- 1618

Benchmarks

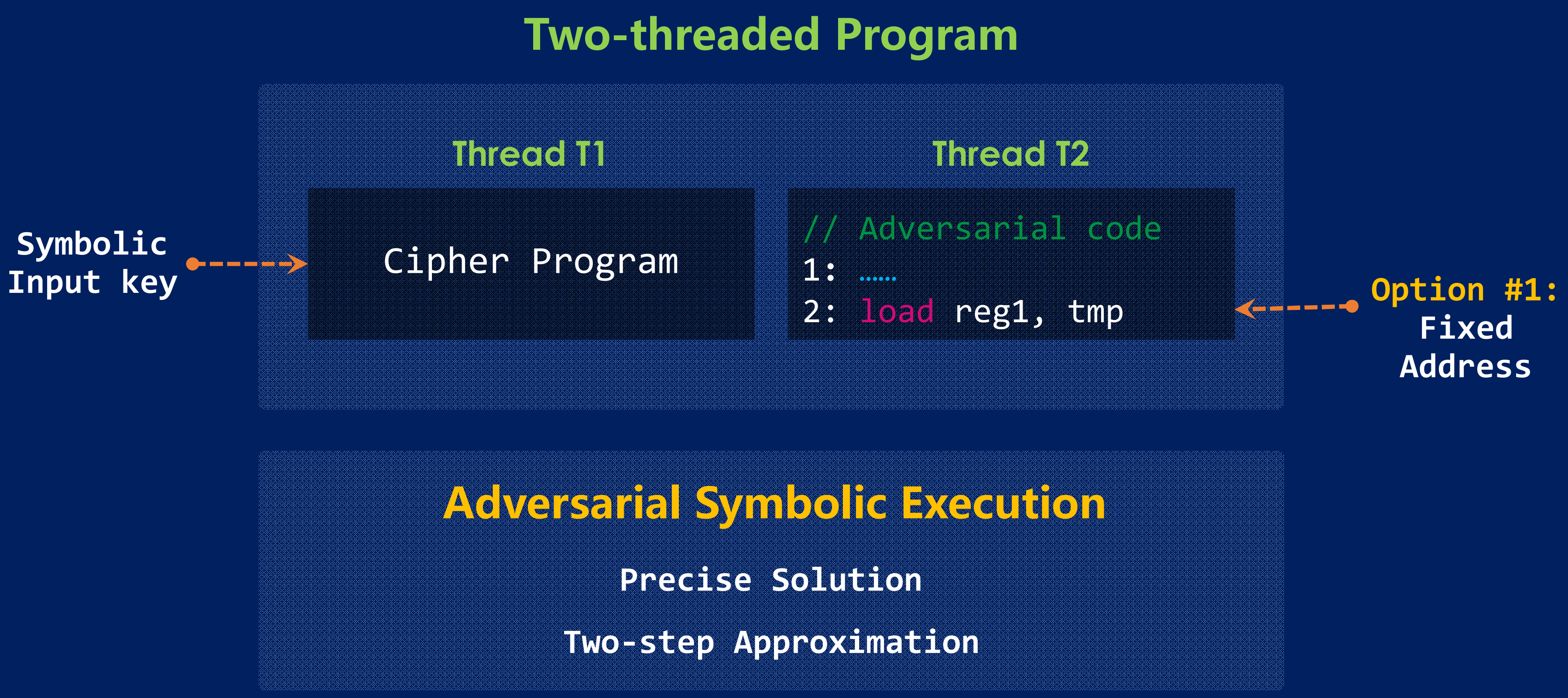
Benchmark Statistics



Name	LOC	LL	KS	MA	Name	LOC	LL	KS	MA
AES[6]	1,429	4,384	24	771	FCrypt[27]	437	1,623	12	428
AES[27]	1,368	4,144	24	788	KV_name[21]	1,350	1,402	4	19
Camellia[4]	776	5,319	16	1,301	LBlock[29]	930	4,010	10	1,618
CAST5[4]	735	2,790	16	909	Misty1[1]	391	1,199	16	270
CAST5[27]	883	4,190	16	1,180	Piccolo[29]	301	1,034	12	350
Chaskey[29]	248	638	16	242	PRESENT[29]	194	272	10	94
DES[3]	596	2,166	8	963	rfc2268 [3]	388	870	16	149
DES[27]	1,010	3,926	8	1,029	Seed[3]	607	3,535	16	979
Kasumi[1]	350	1224	16	259	TWINE[29]	256	562	10	229
Khazad[27]	838	463	16	123	Twofish[3]	1,048	4,510	16	1,180

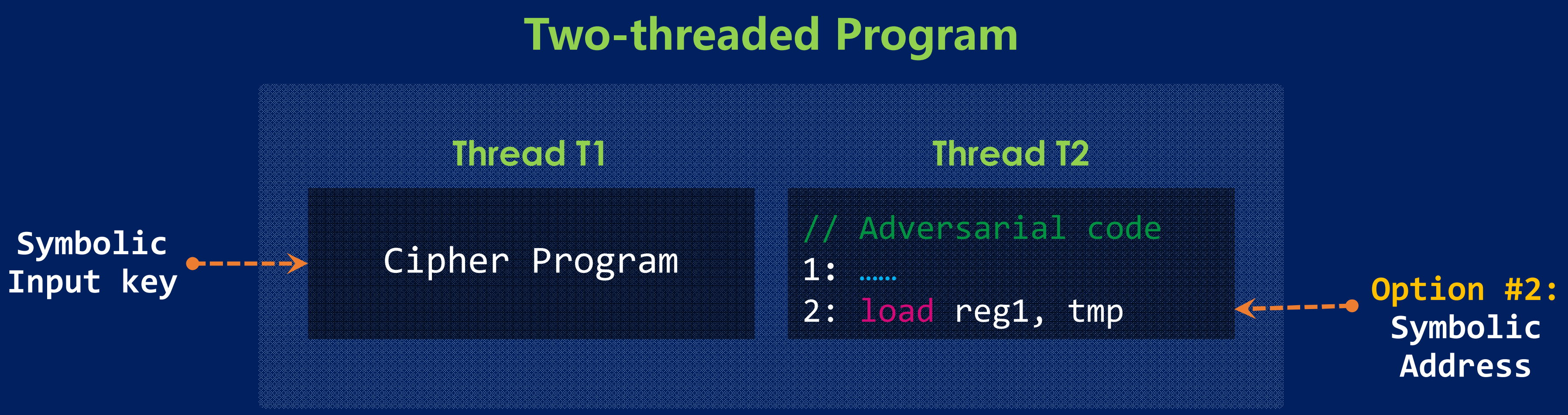
LOC: Line of C code	LL: Line of LLVM bit code
KS: Sensitive input size in bytes	MA: Number of memory accesses

Methodology



```
// Adversarial code
1: ......
2: load reg1, tmp
```

Methodology



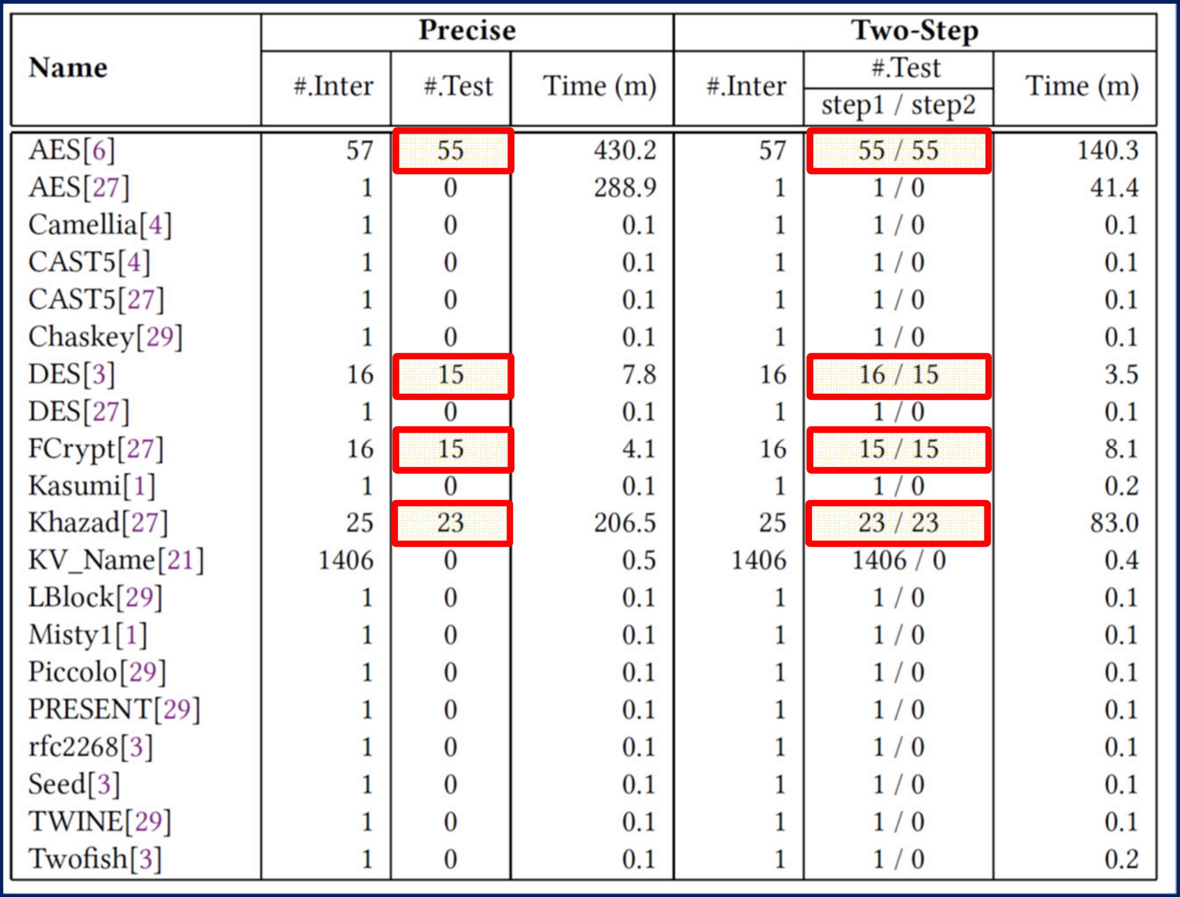
```
// Adversarial code
1: ......
2: load reg1, tmp
```

Adversarial Symbolic Execution

Precise Solution

Two-step Approximation

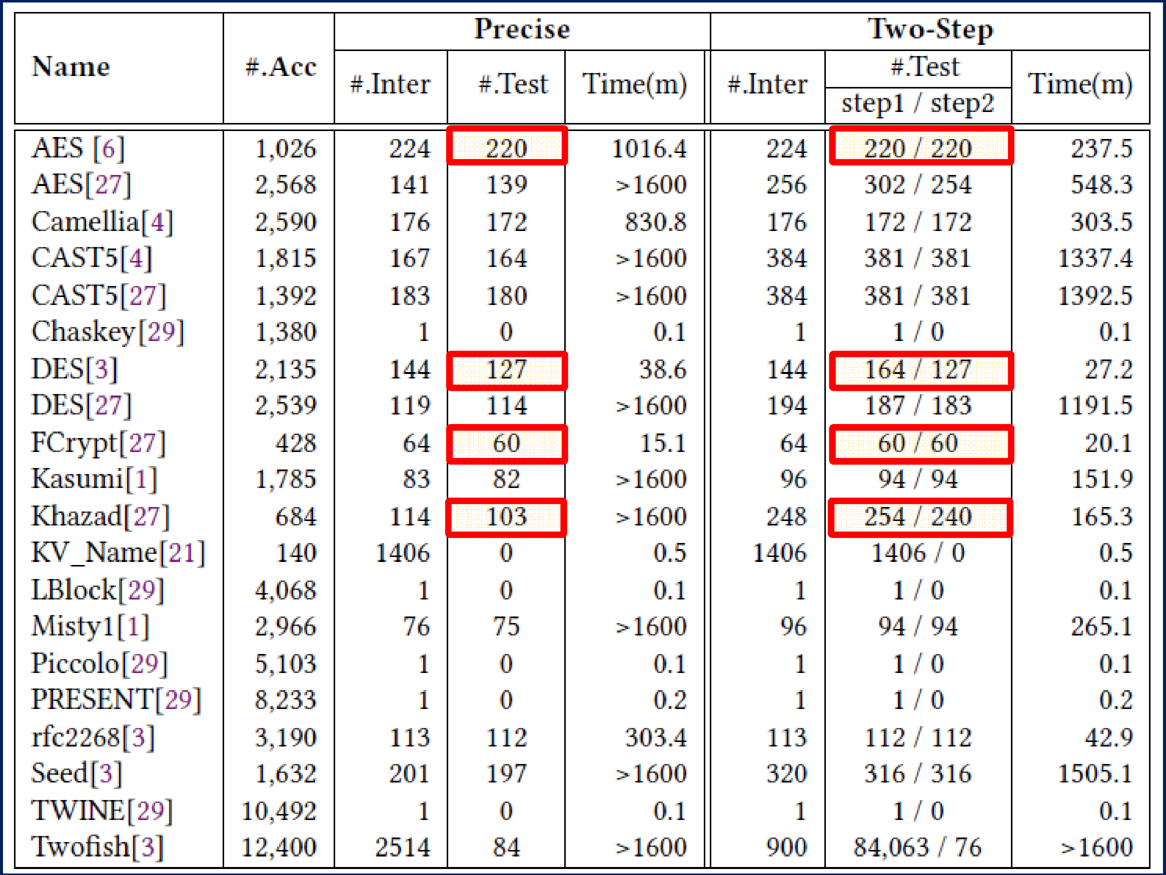
Leakage Detection Result (option #1: fixed address)



Name	Precise			Two-Step		
	#.Inter	#.Test	Time (m)	#.Inter	#.Test step1 / step2	Time (m)
AES[6]	57	55	430.2	57	55 / 55	140.3
AES[27]	1	0	288.9	1	1 / 0	41.4
Camellia[4]	1	0	0.1	1	1 / 0	0.1
CAST5[4]	1	0	0.1	1	1 / 0	0.1
CAST5[27]	1	0	0.1	1	1 / 0	0.1
Chaskey[29]	1	0	0.1	1	1 / 0	0.1
DES[3]	16	15	7.8	16	16 / 15	3.5
DES[27]	1	0	0.1	1	1 / 0	0.1
FCrypt[27]	16	15	4.1	16	15 / 15	8.1
Kasumi[1]	1	0	0.1	1	1 / 0	0.2
Khazad[27]	25	23	206.5	25	23 / 23	83.0
KV_Name[21]	1406	0	0.5	1406	1406 / 0	0.4
LBlock[29]	1	0	0.1	1	1 / 0	0.1
Misty1[1]	1	0	0.1	1	1 / 0	0.1
Piccolo[29]	1	0	0.1	1	1 / 0	0.1
PRESENT[29]	1	0	0.1	1	1 / 0	0.1
rfc2268[3]	1	0	0.1	1	1 / 0	0.1
Seed[3]	1	0	0.1	1	1 / 0	0.1
TWINE[29]	1	0	0.1	1	1 / 0	0.1
Twofish[3]	1	0	0.1	1	1 / 0	0.2

- Leaks found in fewer programs
  - 4 out of 20 programs have leaks
- Two-Step is as good as Precise
  - Found the same leaks
  - In shorter time

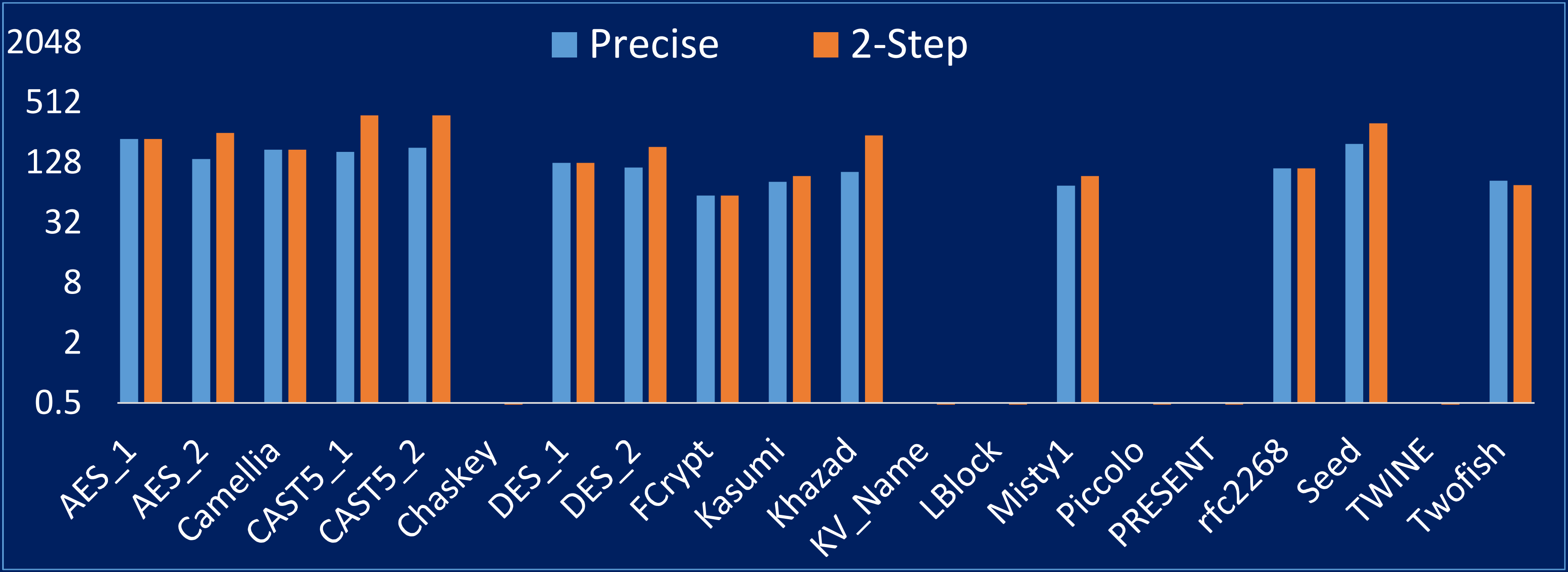
Leakage Detection Result (option #2: symbolic address)



Name	#.Acc	Precise			Two-Step		
		#.Inter	#.Test	Time(m)	#.Inter	#.Test step1 / step2	Time(m)
AES [6]	1,026	224	220	1016.4	224	220 / 220	237.5
AES[27]	2,568	141	139	>1600	256	302 / 254	548.3
Camellia[4]	2,590	176	172	830.8	176	172 / 172	303.5
CAST5[4]	1,815	167	164	>1600	384	381 / 381	1337.4
CAST5[27]	1,392	183	180	>1600	384	381 / 381	1392.5
Chaskey[29]	1,380	1	0	0.1	1	1 / 0	0.1
DES[3]	2,135	144	127	38.6	144	164 / 127	27.2
DES[27]	2,539	119	114	>1600	194	187 / 183	1191.5
FCrypt[27]	428	64	60	15.1	64	60 / 60	20.1
Kasumi[1]	1,785	83	82	>1600	96	94 / 94	151.9
Khazad[27]	684	114	103	>1600	248	254 / 240	165.3
KV_Name[21]	140	1406	0	0.5	1406	1406 / 0	0.5
LBlock[29]	4,068	1	0	0.1	1	1 / 0	0.1
Misty1[1]	2,966	76	75	>1600	96	94 / 94	265.1
Piccolo[29]	5,103	1	0	0.1	1	1 / 0	0.1
PRESENT[29]	8,233	1	0	0.2	1	1 / 0	0.2
rfc2268[3]	3,190	113	112	303.4	113	112 / 112	42.9
Seed[3]	1,632	201	197	>1600	320	316 / 316	1505.1
TWINE[29]	10,492	1	0	0.1	1	1 / 0	0.1
Twofish[3]	12,400	2514	84	>1600	900	84,063 / 76	>1600

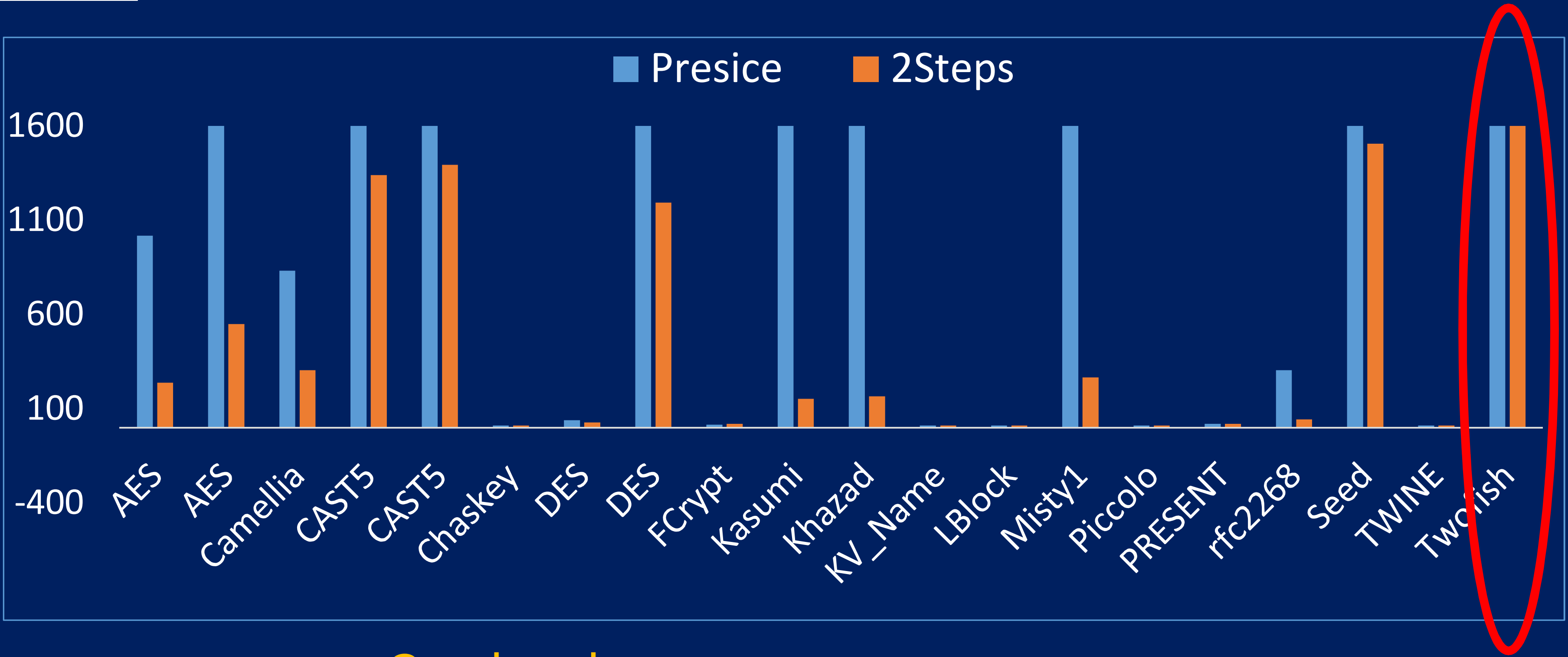
- Leaks found in more programs
  - 14 out of 20 programs have leaks
- Two-Step is as good as Precise
  - Found more leaks
  - Timed out in 1/20 programs (vs. 9/20 programs)

Detected Leak Points



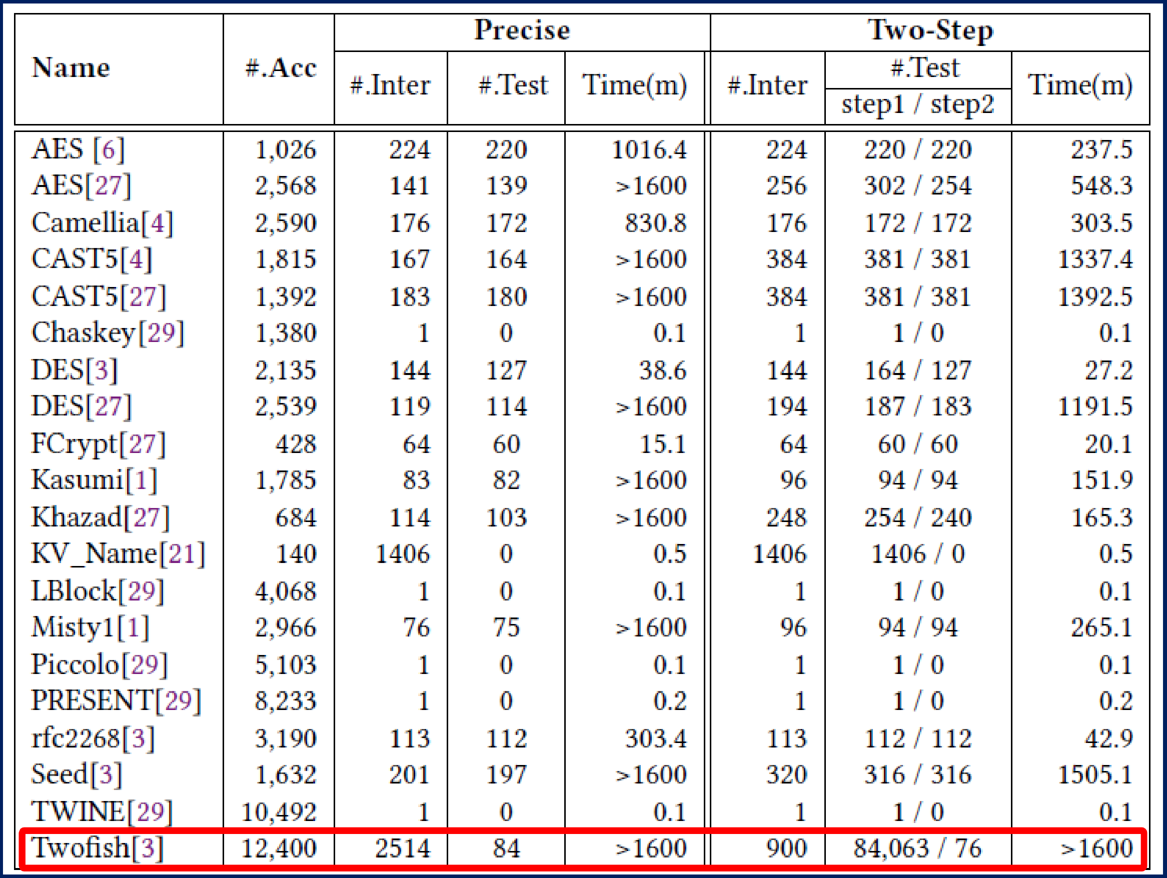
- Leakage Detection
  - Both detected leakage in 14 programs
  - Similar accuracy in leak points

Overhead



- Overhead
  - Precise: 45% time-out rate
  - Two-step: finished 19/20, much faster

The Twofish Case



Name	#.Acc	Precise			Two-Step		
		#.Inter	#.Test	Time(m)	#.Inter	#.Test step1 / step2	Time(m)
AES [6]	1,026	224	220	1016.4	224	220 / 220	237.5
AES[27]	2,568	141	139	>1600	256	302 / 254	548.3
Camellia[4]	2,590	176	172	830.8	176	172 / 172	303.5
CAST5[4]	1,815	167	164	>1600	384	381 / 381	1337.4
CAST5[27]	1,392	183	180	>1600	384	381 / 381	1392.5
Chaskey[29]	1,380	1	0	0.1	1	1 / 0	0.1
DES[3]	2,135	144	127	38.6	144	164 / 127	27.2
DES[27]	2,539	119	114	>1600	194	187 / 183	1191.5
FCrypt[27]	428	64	60	15.1	64	60 / 60	20.1
Kasumi[1]	1,785	83	82	>1600	96	94 / 94	151.9
Khazad[27]	684	114	103	>1600	248	254 / 240	165.3
KV_Name[21]	140	1406	0	0.5	1406	1406 / 0	0.5
LBlock[29]	4,068	1	0	0.1	1	1 / 0	0.1
Misty1[1]	2,966	76	75	>1600	96	94 / 94	265.1
Piccolo[29]	5,103	1	0	0.1	1	1 / 0	0.1
PRESENT[29]	8,233	1	0	0.2	1	1 / 0	0.2
rfc2268[3]	3,190	113	112	303.4	113	112 / 112	42.9
Seed[3]	1,632	201	197	>1600	320	316 / 316	1505.1
TWINE[29]	10,492	1	0	0.1	1	1 / 0	0.1
Twofish[3]	12,400	2514	84	>1600	900	84,063 / 76	>1600

Twofish

- Time
  - Both run time-out
- Leak points
  - Precise: 84
  - Two-step: 76
- Interleaving
  - Precise: 2514
  - Two-step: 900
- 1st step in two-step method
  - 84063 tries !

Conclusions

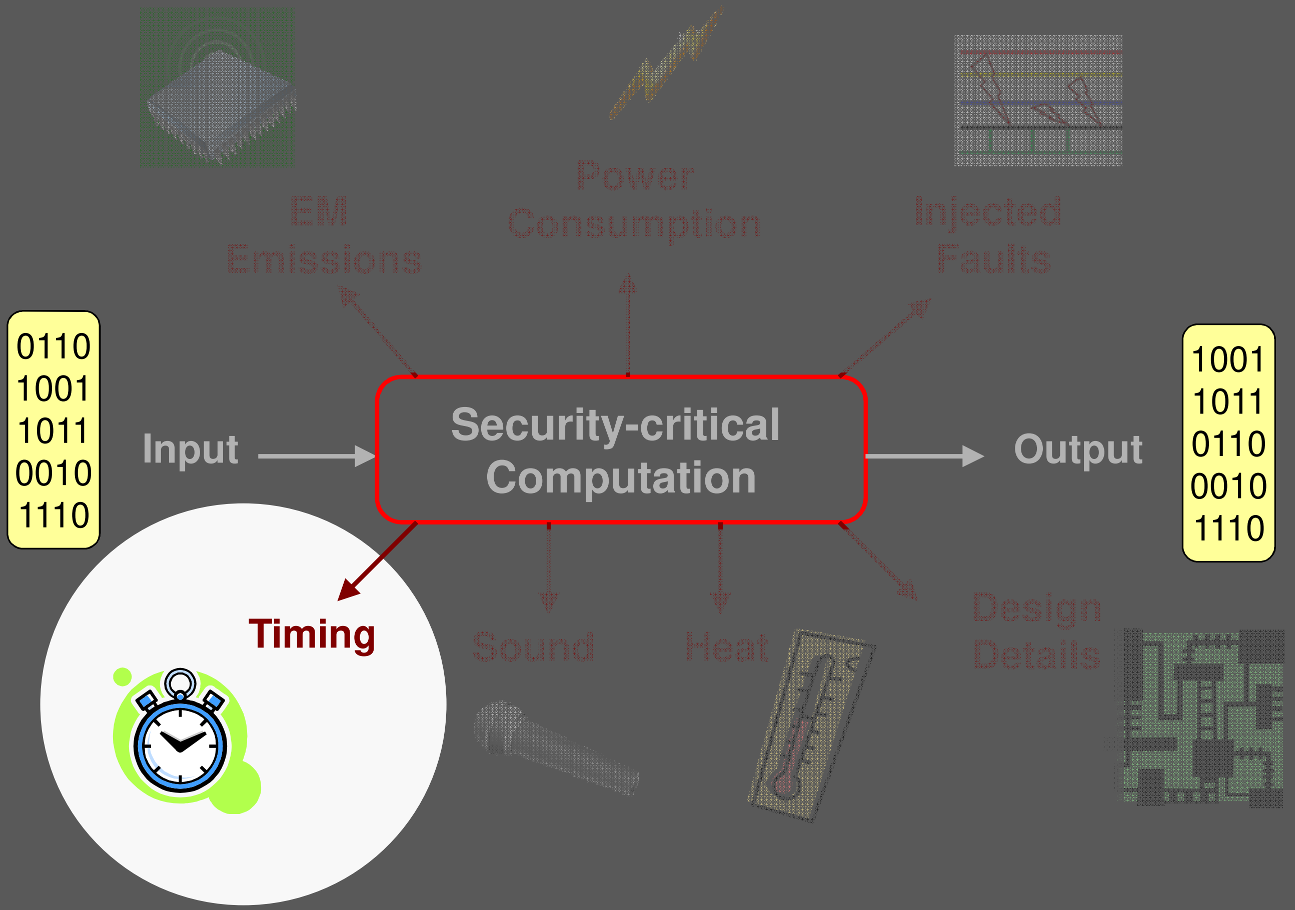
Symbolic execution method

- Detecting cache timing leaks due to concurrency.
- Adversarial scheduling and cache modeling.

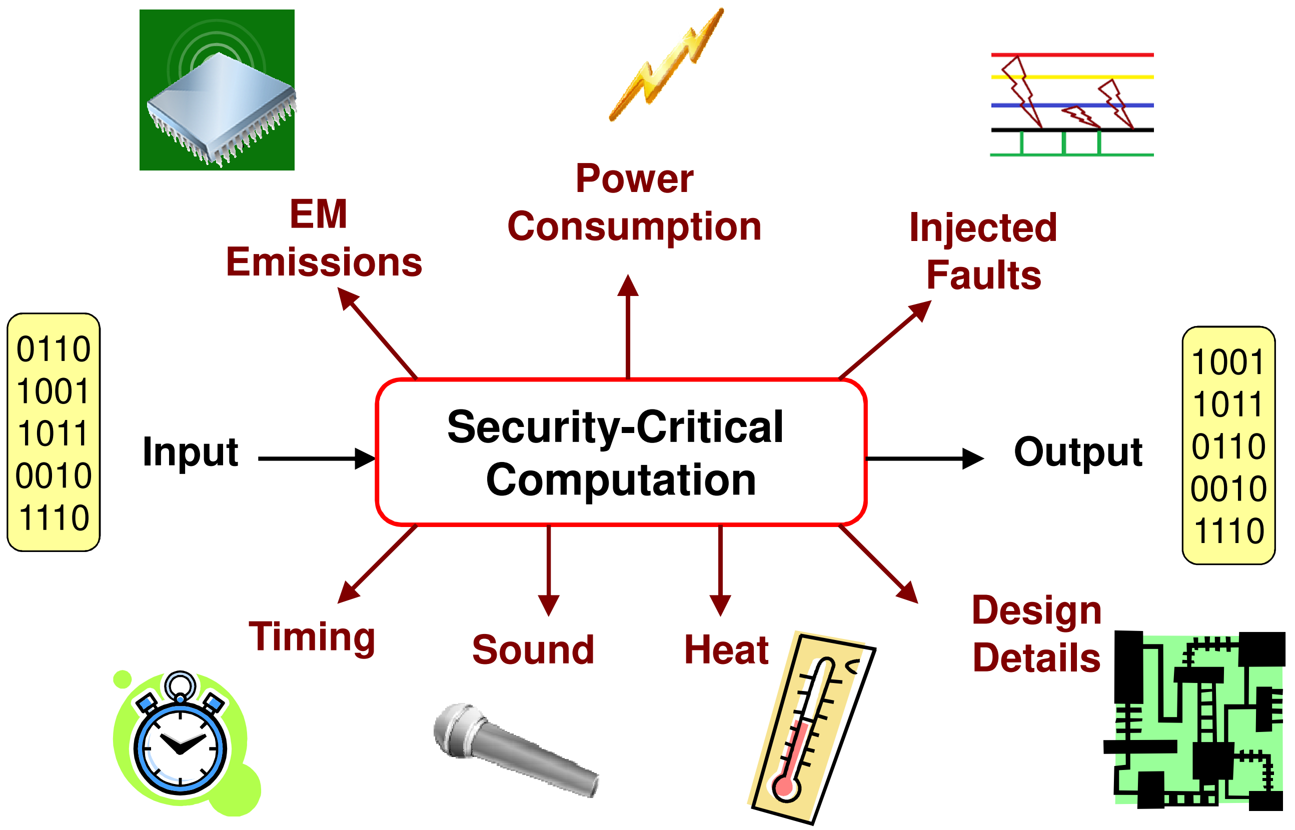
Real ciphers do have such cache timing side-channel leaks

- More efforts needed to exploit them in practice.
- Concrete “data inputs and interleaving schedules” to show the leaks.

Timing Side Channel

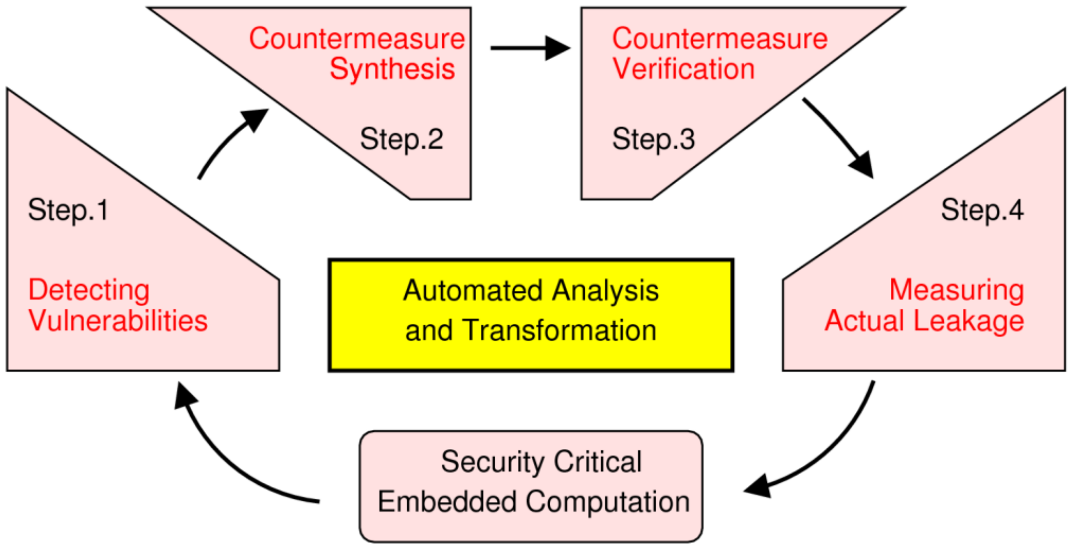


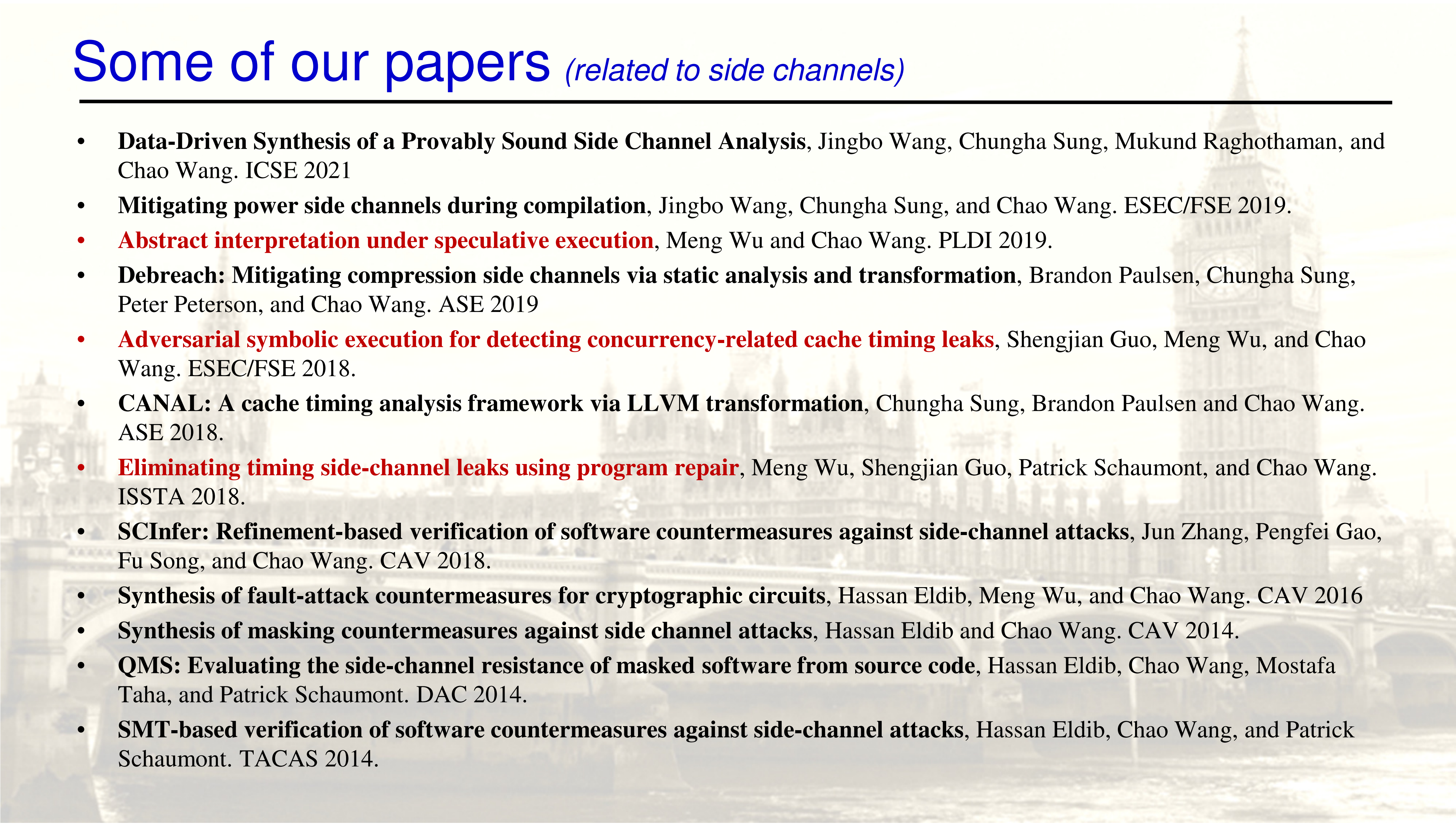
Other Side Channels



The Big Picture

manually verified/secured systems ➔ **automatically** verified/secured systems

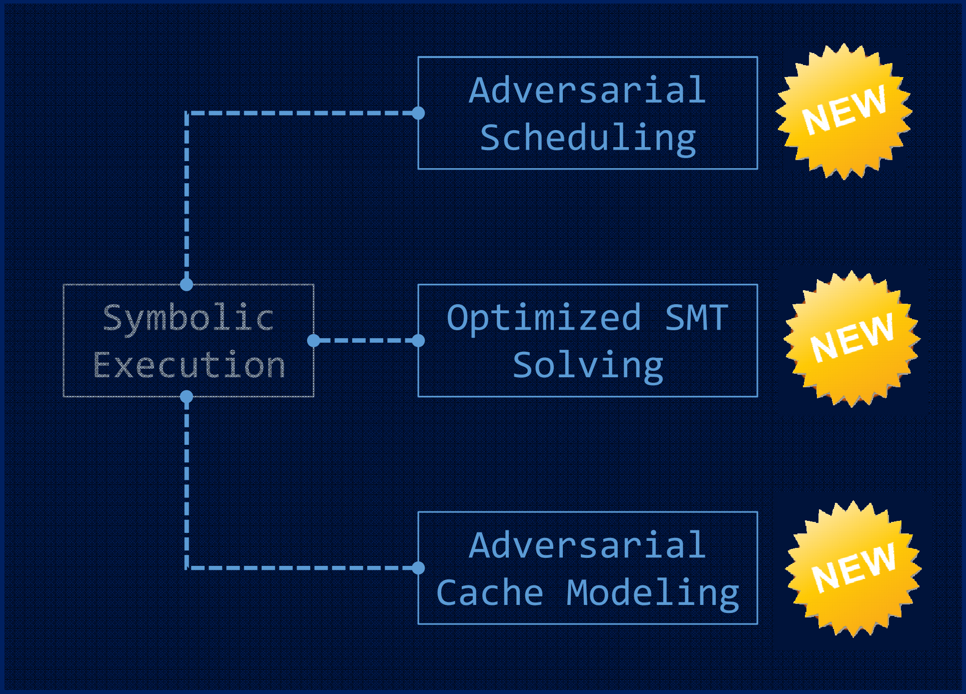

Countermeasure Synthesis
Step.2
Countermeasure Verification
Step.3
Step.1
Detecting Vulnerabilities
Automated Analysis and Transformation
Step.4
Measuring Actual Leakage
Security Critical Embedded Computation



Some of our papers *(related to side channels)*

- **Data-Driven Synthesis of a Provably Sound Side Channel Analysis**, Jingbo Wang, Chungha Sung, Mukund Raghothaman, and Chao Wang. ICSE 2021
- **Mitigating power side channels during compilation**, Jingbo Wang, Chungha Sung, and Chao Wang. ESEC/FSE 2019.
- **Abstract interpretation under speculative execution**, Meng Wu and Chao Wang. PLDI 2019.
- **Debreach: Mitigating compression side channels via static analysis and transformation**, Brandon Paulsen, Chungha Sung, Peter Peterson, and Chao Wang. ASE 2019
- **Adversarial symbolic execution for detecting concurrency-related cache timing leaks**, Shengjian Guo, Meng Wu, and Chao Wang. ESEC/FSE 2018.
- **CANAL: A cache timing analysis framework via LLVM transformation**, Chungha Sung, Brandon Paulsen and Chao Wang. ASE 2018.
- **Eliminating timing side-channel leaks using program repair**, Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. ISSTA 2018.
- **SCInfer: Refinement-based verification of software countermeasures against side-channel attacks**, Jun Zhang, Pengfei Gao, Fu Song, and Chao Wang. CAV 2018.
- **Synthesis of fault-attack countermeasures for cryptographic circuits**, Hassan Eldib, Meng Wu, and Chao Wang. CAV 2016
- **Synthesis of masking countermeasures against side channel attacks**, Hassan Eldib and Chao Wang. CAV 2014.
- **QMS: Evaluating the side-channel resistance of masked software from source code**, Hassan Eldib, Chao Wang, Mostafa Taha, and Patrick Schaumont. DAC 2014.
- **SMT-based verification of software countermeasures against side-channel attacks**, Hassan Eldib, Chao Wang, and Patrick Schaumont. TACAS 2014.

In a Nutshell



Find Interleaving?

Generate Input ?

Analyze Cache ?

Our Adversarial Symbolic Execution

Collaborators

Shengjian (Daniel) Guo, Meng Wu, Jingbo Wang, Chungha Sung, Mukund Raghothaman, Brandon Paulsen, Hassan Eldib, Jun Zhang, Pengfei Gao, Fu Song, Mostafa Taha, Patrick Schaumont

"**Exposing Cache Timing Leaks through Out-of-Order Symbolic Execution**". Shengjian Guo, Yueqi Cheng, Jiyong Yu, Meng Wu, Zhiqiang Zuo, Peng Li, Yueqiang Cheng, and Huibo Wang.
(OOPSLA 2020)

"**SpecuSym: Speculative Symbolic Execution for Cache Timing Leak Detection**". Shengjian Guo, Yueqi Chen, Peng Li, Yueqiang Cheng, Huibo Wang, Meng Wu, and Zhiqiang Zuo.
(ICSE 2020)

Collaborators

Shengjian (Daniel) Guo, **Meng Wu**, Jingbo Wang, Chungha Sung, Mukund Raghothaman, Brandon Paulsen, Hassan Eldib, Jun Zhang, Pengfei Gao, Fu Song, Mostafa Taha, Patrick Schaumont

Abstract interpretation under speculative execution,
Meng Wu and Chao Wang.
(PLDI 2019)

Eliminating timing side-channel leaks using program repair,
Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang.
(ISSTA 2018)