Eliminating Timing Side-Channel Leaks using Program Repair

Meng Wu, Shengjian Guo, Patrick Schaumont, Chao Wang

Presented by: Cristina Noujaim, Diego Rojas Salvador, Jacob Hage

Overview

- What Are Side-Channel Attacks?
- Proposed Approach
- Results
- Paper Evaluation

Side Channel Attacks

Background and Motivation

What Are Side-Channel Attacks?

- Side Channel - Anything that transmits information other than code
  - Time it takes to execute a program
  - Memory Accesses
  - Energy output
  - Sound output
- Side Channel Attacks - Obtaining secret information through side channels

Focus - Timing Side Channels!!!

Timing Side-Channels

Example: Instruction-Timing Side Channel

- Branching on Secret Keys

K = SECRET KEY

Let $s_0 = 1$.
For $k = 0$ upto $w - 1$:
    If (bit k of x) is 1 then
        Let $R_k = (s_k \cdot y) \bmod n$.
    Else
        Let $R_k = s_k$.
    Let $s_{k+1} = R_k^2 \bmod n$.
EndFor.
Return (R_{w-1}).

Branch on secret key!

Example: Instruction-Timing Side Channel

- Branching on Secret Keys

Let $s_0 = 1$.
For $k = 0$ upto $w - 1$:
    If (bit k of x) is 1 then
        Let $R_k = (s_k \cdot y) \bmod n$.
    Else
        Let $R_k = s_k$.
    Let $s_{k+1} = R_k^2 \bmod n$.
EndFor.
Return (R_{w-1}).

Modulo is expensive

Move is cheap

Example: Instruction-Timing Side Channel

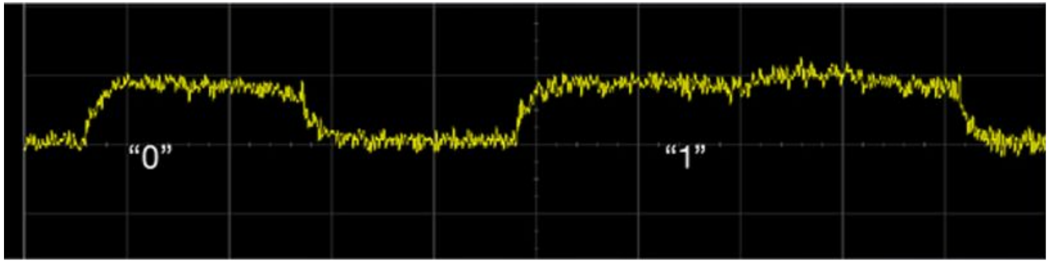
- Branching on Secret Keys

Let $s_0 = 1$.
For $k = 0$ upto $w - 1$:
  If (bit k of x) is 1 then
    Let $R_k = (s_k \cdot y) \bmod n$.
  Else
    Let $R_k = s_k$.
  Let $s_{k+1} = R_k^2 \bmod n$.
EndFor.
Return (R_{w-1}).

Modulo is expensive

Move is cheap

Not taken case takes more time than taken case - side channel!

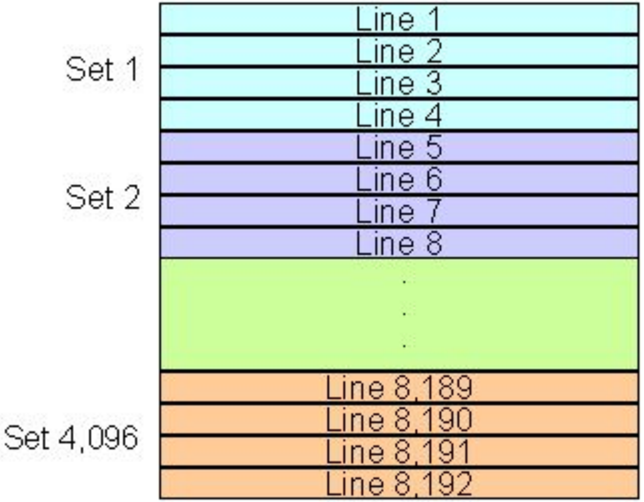


Example: Cache-Timing Side Channel

- Memory Lookup on Secret Keys

```
const uint8_t sbox[256] =
  0x30, 0x01, 0x67, 0x2b, 0xfe, ...};
void subBytes(uint8_t *block) {
  uint8_t i;
  for (i = 0; i < 16; ++i) {
    block[i] = sbox[block[i]];
  }
}
```

CACHE



512 KB L2 Memory Cache

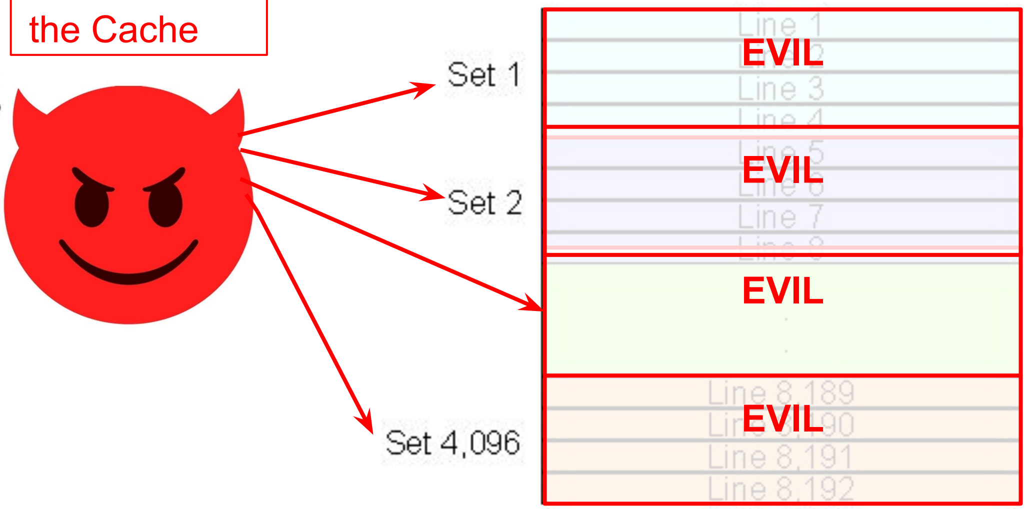
Example: Cache-Timing Side Channel

- Memory Lookup on Secret Keys

```
const uint8_t sbox[256] =
  0x30, 0x01, 0x67, 0x2b, 0xfe,
void subBytes(uint8_t *block) {
  uint8_t i;
  for (i = 0; i < 16; ++i) {
    block[i] = sbox[block[i]];
  }
}
```

Attacker Fills the Cache

CACHE



512 KB L2 Memory Cache

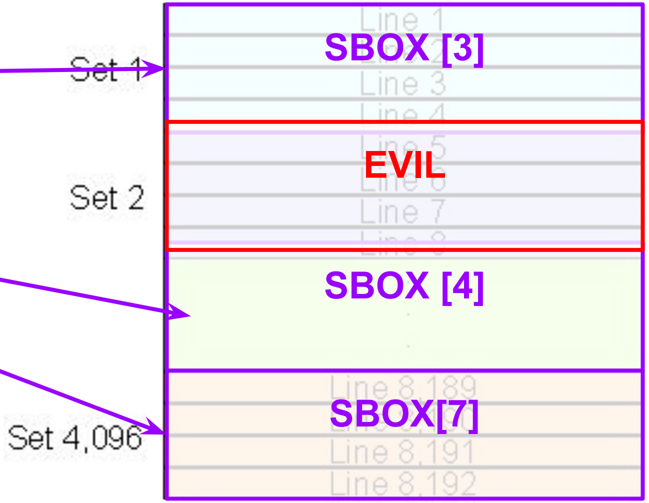
Example: Cache-Timing Side Channel

- Memory Lookup on Secret Keys

```
const uint8_t sbox[256] =
  0x30, 0x01, 0x67, 0x2b, 0xfe, ...};
void subBytes(uint8_t *block) {
  uint8_t i;
  for (i = 0; i < 16; ++i) {
    block[i] = sbox[block[i]];
  }
}
```

User Runs the Program

CACHE

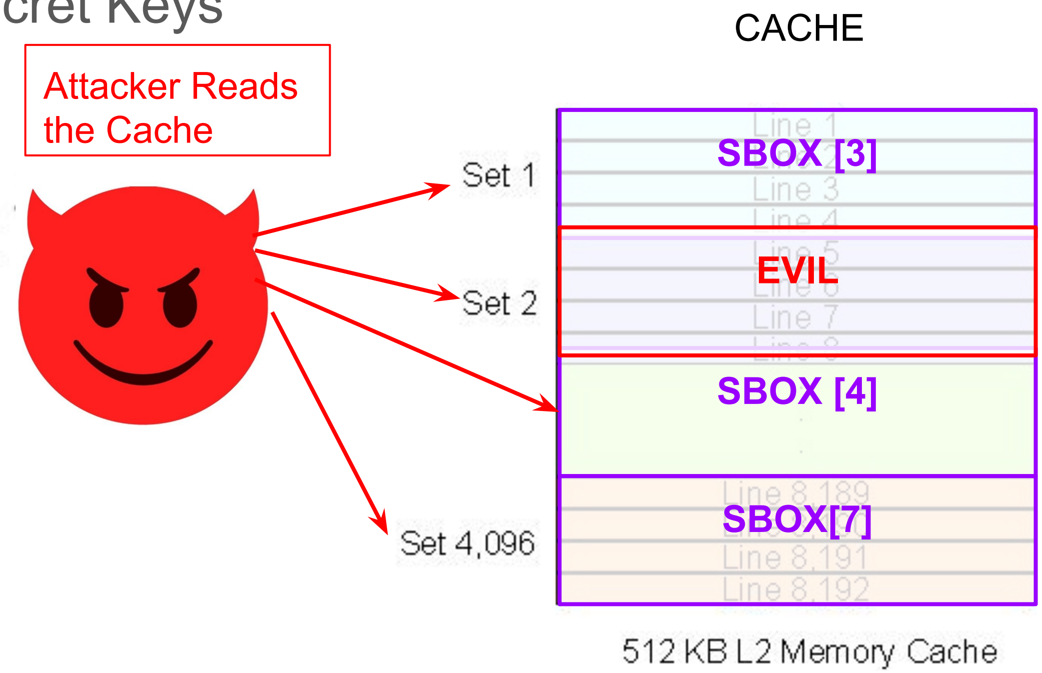


512 KB L2 Memory Cache

Example: Cache-Timing Side Channel

- Memory Lookup on Secret Keys

```
const uint8_t sbox[256] =
  0x30, 0x01, 0x67, 0x2b, 0xfe,
void subBytes(uint8_t *block) {
  uint8_t i;
  for (i = 0; i < 16; ++i) {
    block[i] = sbox[block[i]];
  }
}
```



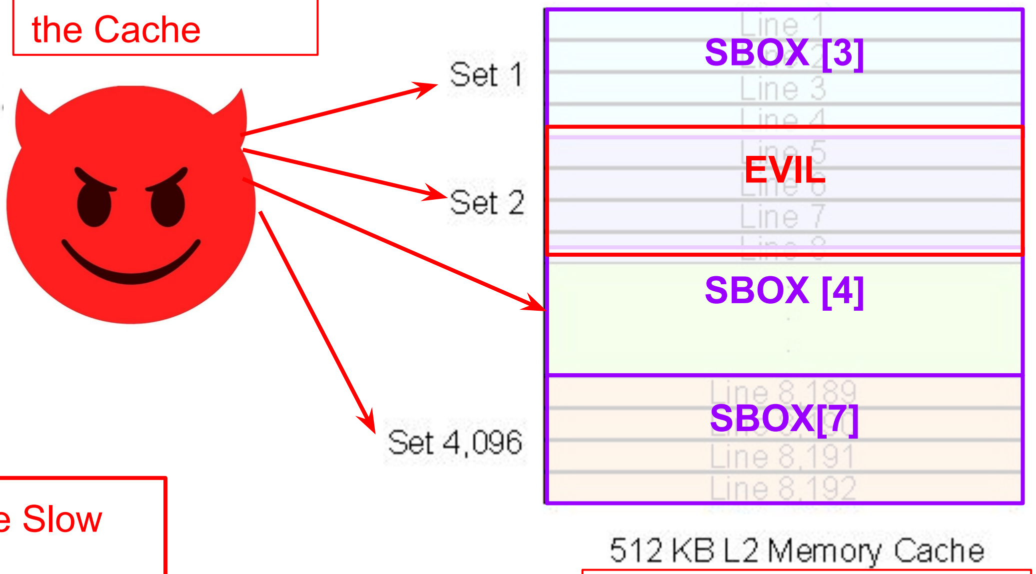
Example: Cache-Timing Side Channel

- Memory Lookup on Secret Keys

```
const uint8_t sbox[256] =
  0x30, 0x01, 0x67, 0x2b, 0xfe, ...
void subBytes(uint8_t *block) {
  uint8_t i;
  for (i = 0; i < 16; ++i) {
    block[i] = sbox[block[i]];
  }
}
```

Attacker Reads the Cache

CACHE



Set1, Set 3, and Set 4096 are Slow

AKA - Block 0, 1, and 2 are 3, 4, & 7!

EVIL blocks are FAST
SBOX blocks are SLOW

Threat Model

Assume less-capable attacker:

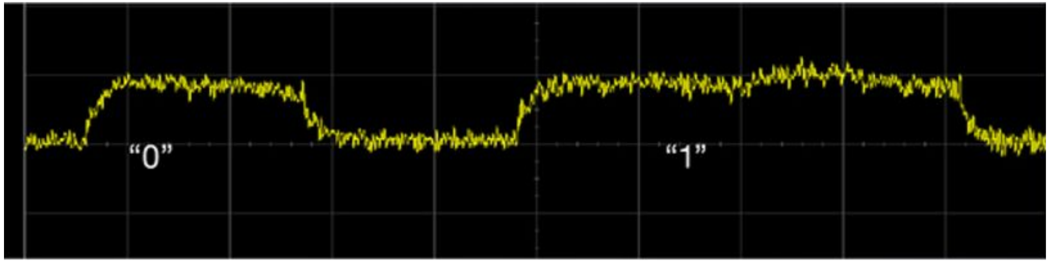
- **Observe variation of the total execution time of the victim's program**

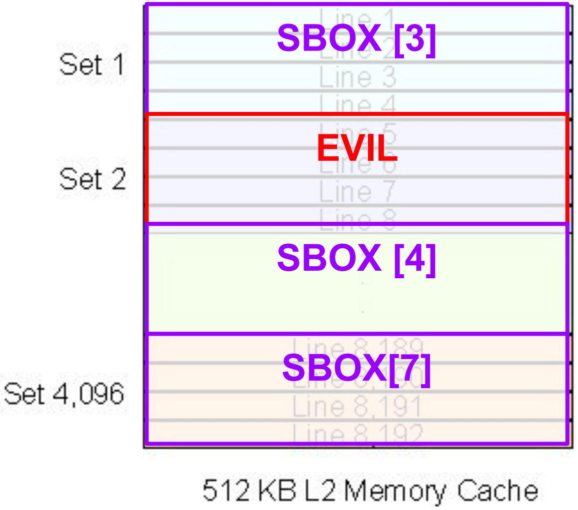
NOT *more-capable attacker:*

- directly access the victim's computer
- observe hidden states of the CPU at the micro-architectural levels

Goal

- **Want to eliminate:**
  - 1. *Instruction*-timing side channel attacks
  - 2. *Cache*-timing side channel attacks





Proposed Mitigation

SC-Eliminator

- **SC-Eliminator**
  - SC = “Side Channel”
  - Computer program that **Detects** and **Mitigates** both:
    - *Instruction*-timing side channel attacks
    - *Cache*-timing side channel attacks
  - Input: LLVM bit-code file
  - Output: Machine code

(p. 2)

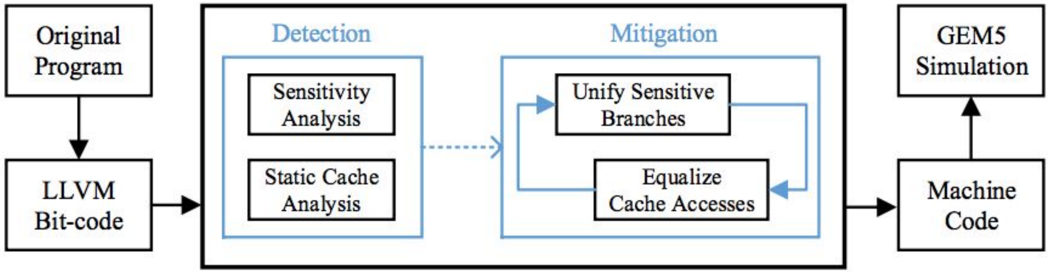


Figure 1: SC-Eliminator: a tool for detecting and mitigating both *instruction*- and *cache*-timing side channels.

SC-Eliminator

(p. 2)

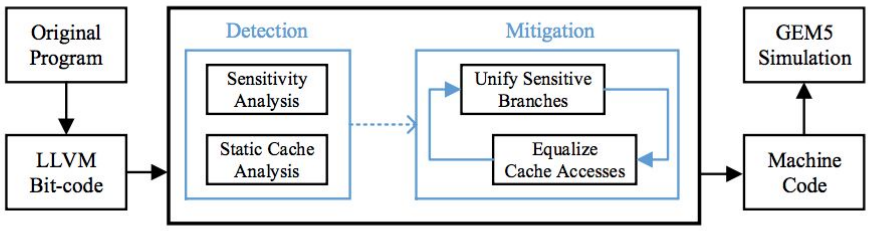


Figure 1: SC-Eliminator: a tool for detecting and mitigating both *instruction*- and *cache*-timing side channels.

- **Intuition: Detect & Mitigate**
  - *Conceptually*: If the execution time of both sensitive conditional statements and sensitive memory accesses are equalized, there will be no *instruction-* or *cache-* timing leaks
  - Two phases:
    1. <u>Detect</u> - Static analyses identify *sensitive* variables in LLVM bit-code
    2. <u>Mitigate</u> - Eliminate the identified *sensitive* variables:
       1. **Unify Sensitive branches**
       2. **Equalize cache accesses**

SC-Eliminator

(p. 2)

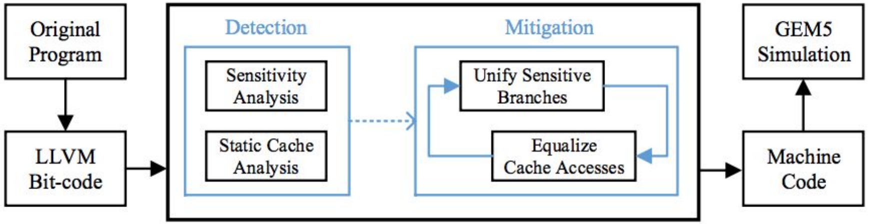


Figure 1: SC-Eliminator: a tool for detecting and mitigating both *instruction*- and *cache*-timing side channels.

- **Approach**: **Detect & Mitigate**
- Detect:
  - Static analysis - Identify, for a program and a list of secret inputs, the set of variables whose values depend on the secret inputs.
  - Sensitivity analysis - Decide if sensitive program variables lead to timing leaks by checking if they affect:
    - Unbalanced conditional jumps (*Instruction*-timing side channel)
    - Accesses of memory blocks across multiple cache lines (*Cache*-timing side channel).

Static & Sensitivity Analysis

- Initial set of sensitive variables labeled by user
  - Secret input == cryptographic key
  - Plaintext == public.
- Sensitivity tag - Attribute to be propagated from the secret source to other program variables following either data- or control-dependency transitively.
  - All variables whose values depend on sensitive variables

  - Data dependency:
    - the def-use relation in {b = a & 0x80;}
  - Control-dependency:
    - if (a == 0x10) { b = 1; } else { b = 0; }

SC-Eliminator

(p. 2)

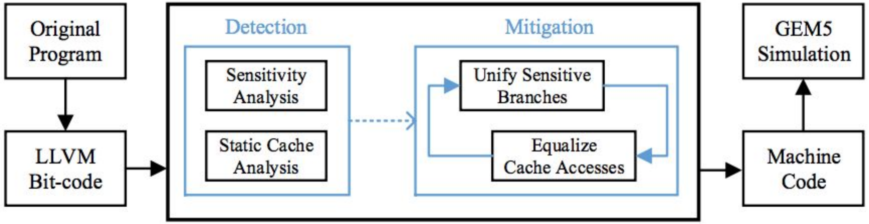


Figure 1: SC-Eliminator: a tool for detecting and mitigating both *instruction*- and *cache*-timing side channels.

- **Approach**: **Detect & Mitigate**
- Mitigate:
  - <u>Unify sensitive branches</u> - Eliminate differences in execution time caused by unbalanced conditional jumps
  - <u>Equalize cache accesses</u> - Eliminate differences in the number of cache hits/misses during the accesses of Lookup tables (LUTs).

Unifying Sensitive Branches

- Execute both the taken and not-taken basic blocks.
- Both blocks must:
  - Have *unique entry and exit blocks*
  - Be executed whenever either is executed
- *Optimization*: CTSEL(cond, val1, val2)
  - Gets [predicated] source operand of a store inst. in constant-time.

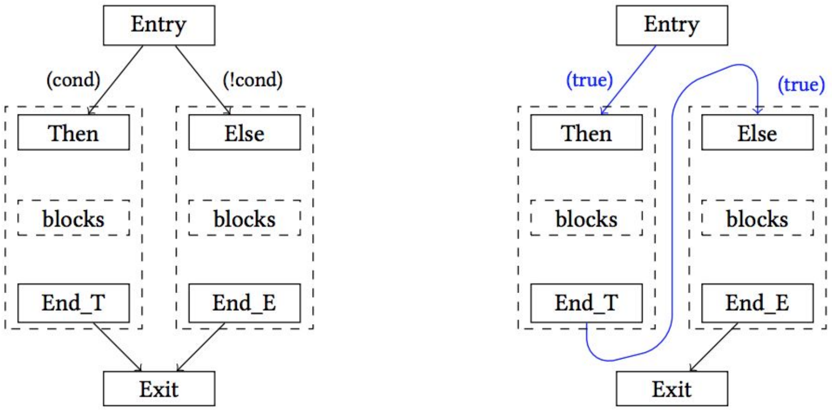


Figure 11: Removing the conditional jumps.

(p. 5)

If (cond) { *addr=valT; } else { *addr=valE ; } ⇨ *addr = CTSEL(cond, valT, valE);

Equalizing Cache Accesses

- Mitigate Lookup Table (LUT) accesses that depend on secret data
  - i-th bit of secret exponent is used to access the i-th index of some array.

while $e \geq 0$ **do**
    for $i \leftarrow 1$ **to** 4 **do**
        $s \leftarrow s \cdot s \bmod p$
    $u \leftarrow x_e \cdots x_{e-3}$
    $s \leftarrow t[u] \cdot s \bmod p$
    $e \leftarrow e - 4$
return s

```
  const uint8_t sbox[256] =
  0x30, 0x01, 0x67, 0x2b, 0xfe, ...};
void subBytes(uint8_t *block) {
  uint8_t i;
  for (i = 0; i < 16; ++i) {
    block[i] = sbox[block[i]];
  }
}
```

- Solution: Ensure that every element in the table is accessed, any time that any one element is accessed.
  - Naive; super slow!
- *Optimization*: MUST-HIT Analysis
  - Determine which LUT variables are already in the cache to prevent redundant cache accesses

Results & Evaluation

Validity to Threat Model

- Total-time-aware threat model
- Less-capable attacker
- Require *list* of known secret variables!

Unrealistic Biases!

Results

- Exec time original program:
  - varies!
- Exec time mitigated program:
  - constant!

Table 5: Results of GEM5 simulation with 2 random inputs.

Name	Before Mitigation				Mitigation w/o opt		Mitigation w/ opt	
	# CPU cycle (in_1,in_2)		# Miss (in_1,in_2)		# CPU cycle	# Miss	# CPU cycle	# Miss
aes	100,554	101,496	261	269	204,260	303	112,004	303
des	95,630	90,394	254	211	346,170	280	100,694	280
des3	118,362	111,610	271	211	865,656	280	124,176	280
anubis	128,602	127,514	276	275	512,452	276	134,606	276
cast5	102,426	102,070	282	279	266,156	304	108,068	304
cast6	96,992	97,492	238	245	233,774	245	100,914	245
fcrypt	84,616	83,198	224	218	114,576	240	88,236	240
khazad	101,844	100,724	332	322	366,756	432	130,682	432
aes	89,968	90,160	234	235	174,904	240	94,364	240
cast	117,936	117,544	345	342	520,336	436	136,052	435
aes_key*	243,256	243,256	329	329	254,262	329	245,584	328
cast128	161,954	161,694	298	296	305,514	321	167,626	321
des	118,848	119,038	269	270	182,830	317	127,374	316
kasumi	113,362	113,654	204	206	137,914	206	115,060	206
seed	106,518	106,364	239	238	165,546	249	110,486	249
twofish	309,160	299,956	336	334	1,060,832	340	315,018	339
3way	87,834	87,444	181	181	90,844	182	90,844	182
des	152,808	147,344	224	222	181,074	225	168,938	225
loki91	768,064	768,296	181	181	2,170,626	181	2,170,626	181
camellia	84,208	84,020	205	203	102,100	244	91,180	244
des	100,396	100,100	212	211	112,992	213	100,500	213
seed	83,256	83,372	228	230	107,318	240	96,266	239
twofish	230,838	229,948	334	327	982,258	338	295,268	338

Results:

- Reduction in:
  - **Code size!**
  - **Execution time!**

Table 4: Results of leak mitigation. Runtime overhead is based on average of 1000 simulations with random keys.

Name	Mitigation w/o opt				Mitigation w/ opt			
	# LUT-a	Time(s)	Prog-size	Ex-time	# LUT-a	Time(s)	Prog-size	Ex-time
aes	416	0.61	5.40x	2.70x	20	0.28	1.22x	1.11x
des	640	1.17	19.50x	5.68x	22	0.13	1.23x	1.07x
des3	1,152	1.80	12.90x	12.40x	22	0.46	1.13x	1.07x
anubis	868	3.12	9.08x	6.90x	10	0.75	1.10x	1.07x
cast5	448	0.79	7.24x	3.84x	12	0.22	1.18x	1.07x
cast6	384	0.72	7.35x	3.48x	12	0.25	1.16x	1.08x
fcrypt	128	0.07	5.70x	1.59x	8	0.03	1.34x	1.05x
khazad	248	0.45	8.60x	4.94x	16	0.07	1.49x	1.35x
aes	696	0.96	9.52x	2.39x	18	0.22	1.21x	1.06x
cast	448	1.42	13.40x	6.50x	12	0.30	1.35x	1.20x
aes_key	184	0.27	1.35x	1.19x	1	0.23	1.00x	1.00x
cast128	448	0.42	3.62x	2.48x	12	0.10	1.09x	1.06x
des	256	0.21	3.69x	1.86x	16	0.06	1.17x	1.08x
kasumi	192	0.18	2.27x	1.37x	4	0.11	1.03x	1.01x
seed	512	0.57	6.18x	1.94x	12	0.15	1.12x	1.03x
twofish	2,512	29.70	5.69x	4.77x	8	10.6	1.02x	1.03x
3way	0	0.01	1.01x	1.03x	0	0.01	1.01x	1.03x
des	128	0.05	2.21x	1.22x	8	0.03	1.09x	1.11x
loki91	0	0.01	1.01x	2.83x	0	0.01	1.01x	2.83x
camellia	32	0.04	2.21x	1.35x	4	0.03	1.20x	1.09x
des	128	0.06	2.30x	1.20x	8	0.03	1.10x	1.02x
seed	200	0.01	1.38x	1.36x	8	0.01	1.20x	1.18x
twofish	2,576	32.40	6.85x	6.59x	136	11.90	1.41x	1.46x

Strength & Weaknesses

+ Strong proof of concept:
    + Use LLVM to eliminate timing side-channels at instruction and cache-level.
- Very strong claims – hidden assumptions
- Does not work for real attacks:
    - Meltdown/Spectre
    - Similar cache attacks (Evict+Time, Prime+Probe, and Flush+Reload)
- Does not address leaks exploitable by probing the hardware
    - Instruction pipelines
    - Data buses.

:(

Conclusion

- Developed a method for mitigating side-channel leaks via program repair.
  - LLVM, targeting cryptographic software in C/C++
- Evaluated on a large number of real world applications:
  - Static analysis took only a few seconds
  - Transformation took less than a minute.
- Mitigated software moderate increase in code size and runtime overhead.
- Strong assumptions make their solution non-applicable to most real attacks

Questions?