TAMPERE UNIVERSITY OF TECHNOLOGY

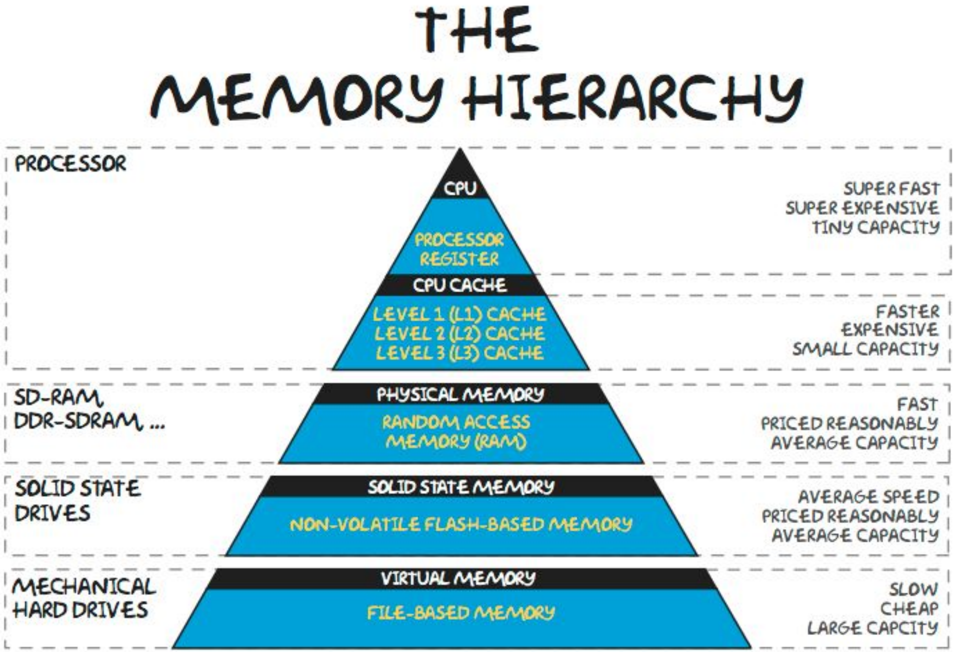
# Constant-Time Callees with Variable-Time Callers

Cesar Pereida García
Billy Bob Brumley
**Tampere University of Technology Finland**

# Outline

- **Enabling Cache-Timing Attacks**
- **Motivation**
  - Brief History of Cache-Timing Attacks
- **Recipe for Side-Channel Attacks**
  - Step 1, 2, 3, 4 and 5
- **End-to-End Cache-Attack**
  - TLS & SSH
  - Crypto libraries
- **Conclusions**

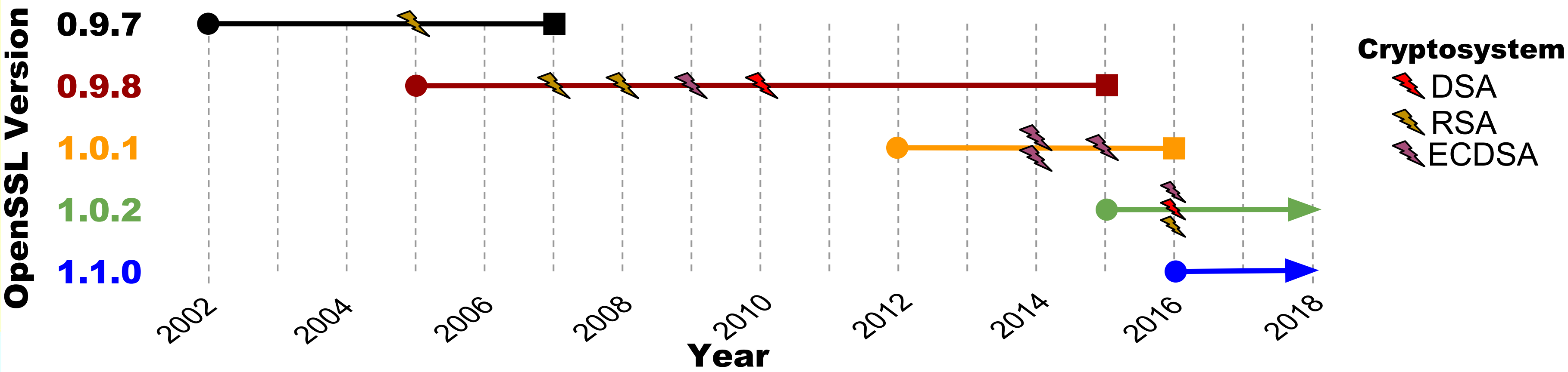
# Enabling Cache-Timing Attacks




https://source.ggy.bris.ac.uk/mediawiki/index.php?title=File:Memory-Hierarchy.jpg&limit=500

# Brief History of Cache-Timing Attacks for Public Key Cryptography in OpenSSL

# Cache-Timing Attacks for Public Key Cryptography




**ECDSA**

2009 - Brumley & Hakala (P+P/LI-I)

2014 - Benger et al. (F+R/LLC/secp256)

2014 - Yarom & Benger (F+R/LLC/Binary Field)

2015 - van de Pol et al. (F+R/LLC)

2016 - Allan et al.(F+R/Perf. Deg./LLC)

**RSA**

2005 - Percival (E+R/L1-D)

2007 - Aciicmez et al. (SBPA/L1-I)

2008 - Aciicmez & Schindler (SBPA/L1-D)

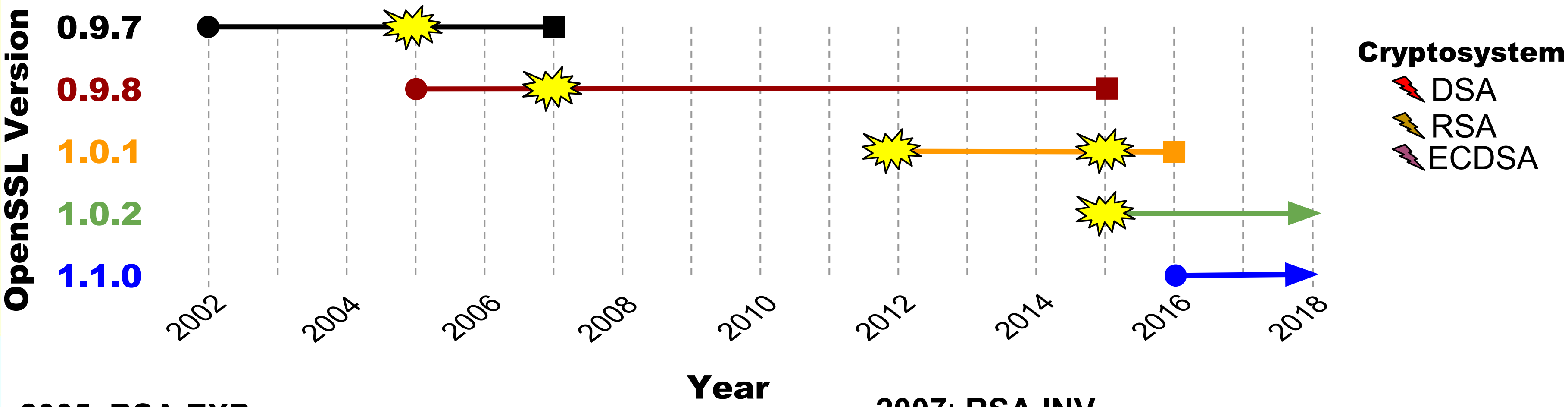
2016 - Yarom et al. (Cache-Bank Collision L1)

**DSA**

2010 - Aciicmez et al. (P+P/L1-I)

2016 - Pereida García et al. (F+R/Perf. Deg./LLC)

# Relevant Changes Introduced due to Cache-Timing Attacks




**2005**: **RSA EXP**

- BN_FLG_EXP_CONSTTIME
- BN_mod_exp_mont_consttime

**2012**: **ECDSA POINT MULT**

- EC_GFp_nistp256_method: Constant-time scalar multiplication (fixed window & masking)
- Research shifts to *secp256k1* (*wNAF*)

**2007**: **RSA INV**

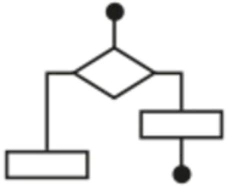
- BN_mod_inverse_no_branch
- BN_div
- BN_FLG_CONSTTIME

**2015**: **ECDSA FAST & MOD INV**

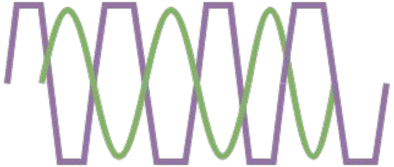
- EC_GFp_nistz256_method
- BN_mod_exp_mont_consttime + FLT

# Recipe for Side-Channel Attacks on Digital Signatures

# Recipe for a Side-Channel Attack



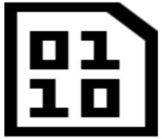
1) Take an algorithm that uses confidential data.



2) Measure the side-channel leakage.

SLSLLSLL...

3) Run the leaked data through a signal processing machine.



4) Convert sequences to bits and combine with message and signature.

5) Let it rest in a lattice for some time.

Et voilà, you have a private key.

# Step 1
# Take a primitive and an algorithm that uses confidential data

# ECDSA

## Given:

$$E : y^2 = x^3 + ax + b$$

$$(CURVE, h, G, n, \alpha_A)$$

## Signing:

**Note: Nonce** k **is recoverable if at least 3 bits are leaked for each signature.**

$$r = ([k]G)_x \bmod n$$

Constant-Time Scalar by Point Multiplication

$$s = k^{-1}(h(m) + \alpha_A r) \bmod n$$

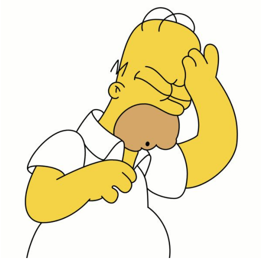
Modular Inversion?

$$(m, r, s)$$

# Modular Inversion (OpenSSL 1.0.1)

```
+--bn_gcd.c-----------------------------------------------------------------+
   |226    BIGNUM *BN_mod_inverse(BIGNUM *in,                                   |
   |227                           const BIGNUM *a, const BIGNUM *n, BN_CTX *ctx) |
   |228    {                                                                     |
B+ |229        BIGNUM *A, *B, *X, *Y, *M, *D, *T, *R = NULL;                     |
   |230        BIGNUM *ret = NULL;                                               |
   |231        int sign;                                                         |
   |232                                                                          |
   |233        if ((BN_get_flags(a, BN_FLG_CONSTTIME) != 0)                      |
  >|234            || (BN_get_flags(n, BN_FLG_CONSTTIME) != 0)) {                |
   |235            return BN_mod_inverse_no_branch(in, a, n, ctx);               |
   |236        }                                                                 |
+---------------------------------------------------------------------------+
   |0x7ffff77da1c7 <BN_mod_inverse+56>  mov   -0x90(%rbp),%rax                |
   |0x7ffff77da1ce <BN_mod_inverse+63>  mov   0x14(%rax),%eax                 |
   |0x7ffff77da1d1 <BN_mod_inverse+66>  and   $0x4,%eax                       |
   |0x7ffff77da1d4 <BN_mod_inverse+69>  test  %eax,%eax                       |
   |0x7ffff77da1d6 <BN_mod_inverse+71>  jne   0x7ffff77da1e9 <BN_mod_inverse+90> |
   |0x7ffff77da1d8 <BN_mod_inverse+73>  mov   -0x98(%rbp),%rax                |
   |0x7ffff77da1df <BN_mod_inverse+80>  mov   0x14(%rax),%eax                 |
   |0x7ffff77da1e2 <BN_mod_inverse+83>  and   $0x4,%eax                       |
   |0x7ffff77da1e5 <BN_mod_inverse+86>  test  %eax,%eax                       |
  >|0x7ffff77da1e7 <BN_mod_inverse+88>  je    0x7ffff77da212 <BN_mod_inverse+131> |
+---------------------------------------------------------------------------+
```

```
native process 3399 In: BN_mod_inverse                        L234  PC: 0x7ffff77da1e7
(gdb) run dgst -sha256 -sign prime256v1.pem -out lsb-release.sig /etc/lsb-release
Starting program: /usr/local/ssl/bin/openssl dgst -sha256 -sign prime256v1.pem ...
Breakpoint 1, BN_mod_inverse (...) at bn_gcd.c:229
(gdb) backtrace
#0  BN_mod_inverse (...) at bn_gcd.c:229
#1  0x00007ffff782aed9 in ecdsa_sign_setup (...) at ecs_ossl.c:182
#2  0x00007ffff782bc35 in ECDSA_sign_setup (...) at ecs_sign.c:105
#3  0x00007ffff782b29a in ecdsa_do_sign (...) at ecs_ossl.c:269
#4  0x00007ffff782bafd in ECDSA_do_sign_ex (...) at ecs_sign.c:74
#5  0x00007ffff782bb97 in ECDSA_sign_ex (...) at ecs_sign.c:89
#6  0x00007ffff782bb44 in ECDSA_sign (...) at ecs_sign.c:80 ...
(gdb) stepi
(gdb) macro expand BN_get_flags(a, BN_FLG_CONSTTIME)
expands to: ((a)->flags&(0x04))
(gdb) print BN_get_flags(a, BN_FLG_CONSTTIME)
$1 = 0
(gdb) print BN_get_flags(n, BN_FLG_CONSTTIME)
$2 = 0
```



# Binary Extended Euclidean Algorithm

**Input:** Integers $k$ and $p$ such that $gcd(k, p) = 1$.
**Output:** $k^{-1}$ mod $p$.

$v \leftarrow p,\ u \leftarrow k,\ X \leftarrow 1,\ Y \leftarrow 0$
**while** $u \neq 0$ **do**
  **while** $even(u)$ **do**
    $u \leftarrow u/2$     /* u loop */
    **if** $odd(X)$ **then** $X \leftarrow X + p$
    $X \leftarrow X/2$
  **while** $even(v)$ **do**
    $v \leftarrow v/2$     /* v loop */
    **if** $odd(Y)$ **then** $Y \leftarrow Y + p$
    $Y \leftarrow Y/2$
  **if** $u \geq v$ **then**
    $u \leftarrow u - v$
    $X \leftarrow X - Y$
  **else**
    $v \leftarrow v - u$
    $Y \leftarrow Y - X$
**return** $Y$ mod $p$

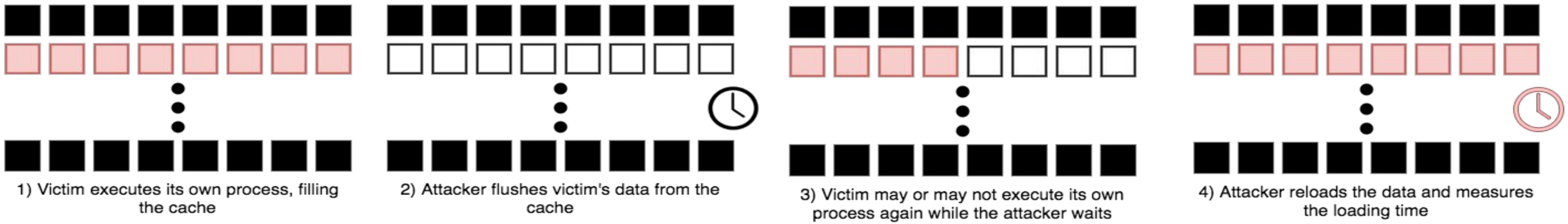
BN_rshift1

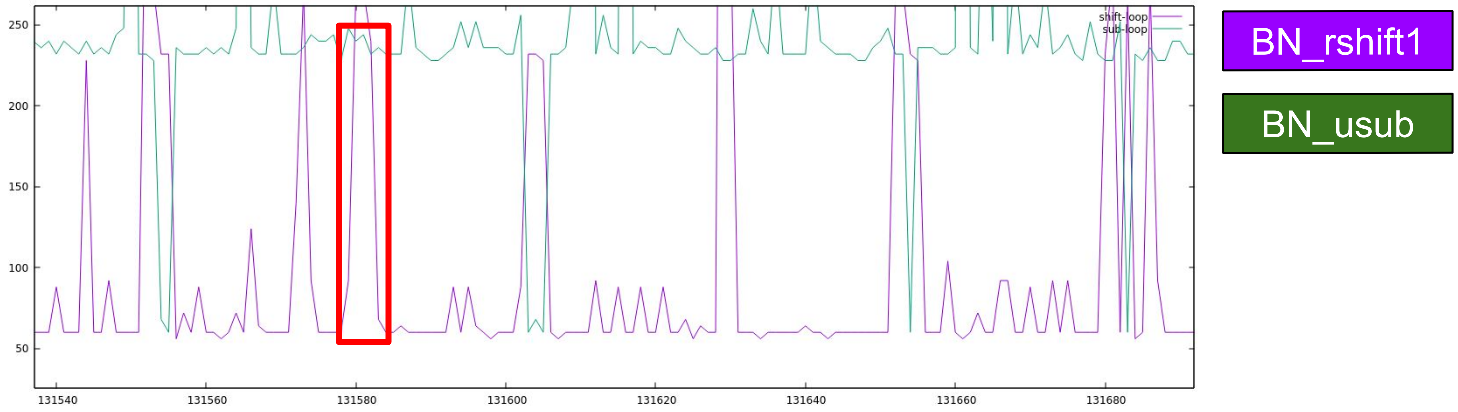
BN_usub

| Fact | Cache-Attack OpenSSL BBEA |
|---|---|
| Number of right-shifts on v | ✓ |
| Number of right-shifts on u | ✓ |
| Number and order of subtractions on v | ✗ |
| Number and order of subtractions on u | ✗ |
| Only one loop per iteration | ✓ |
| `U loop` is the only loop that can be executed during the first iteration | ✓ |
| k is protected, i.e. padded with modulus n | ✓ ✗ |

# Step 2
# Measure the Side-Channel Leakage

# Flush+Reload[1] on the BEEA




[1] Yarom, Yuval, and Katrina Falkner. "FLUSH+ RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack." *USENIX*. 2014.

# Improved Performance Degradation

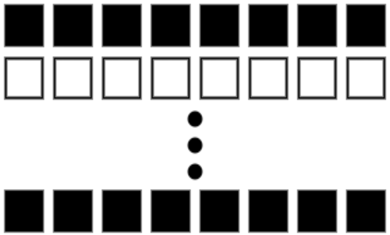
**Objective: Identify the addresses with the highest impact**

- Better probing
- Better degradation

1) Identify the candidate methods and their memory addresses.

```
BN_mod_inverse → 0xE7940
BN_rshift1     → 0xE48E0
BN_usub        → 0xD7B00
BN_uadd        → 0xD7800
BN_rshift      → 0xDDFC0
```

2) Degrade one memory address at a time.



Attacker flushes victim's data

3) Count cache-misses and CPU cycles using performance counters (`perf`).

| Target | Cache misses (*CM*) | Clock cycles (*CC*) |
|---|---|---|
| Baseline (*BL*) | 13 | 211,324 |
| BN_rshift1 | 2,396 | 947,925 |
| BN_usub | 489 | 364,399 |
| BN_mod_inverse | 956 | 540,357 |
| BN_uadd | 855 | 485,088 |
| bn_add_words | 1,124 | 558,839 |
| BN_rshift | 514 | 367,929 |

# Setup and Attack Scenario

**Setup**

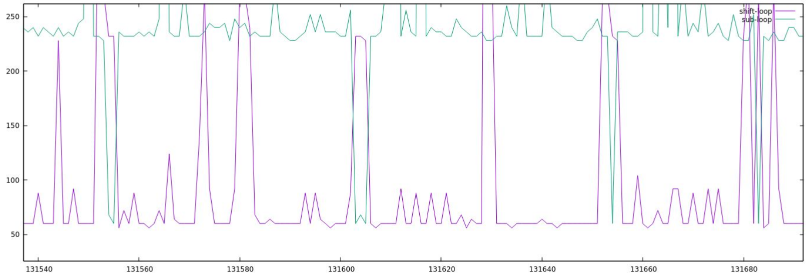
- Intel Core i5-2400 Sandy Bridge 3.10 GHz
- 8 GB memory
- Ubuntu 16.04 LTS “Xenial” 64-bits
- OpenSSL 1.0.1u

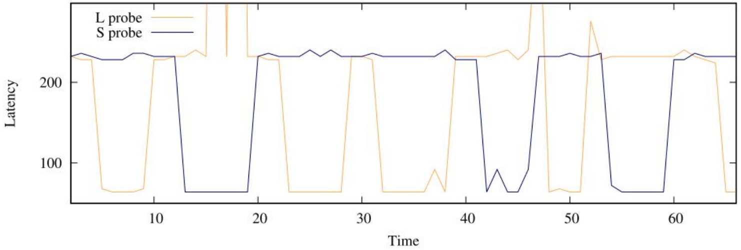
# Step 3
# Apply Signal Processing

# Signal Processing

**Trace**

- Template & Cross-correlation
- Apply moving average.
- Raw → Clean
- Translate to LS sequence

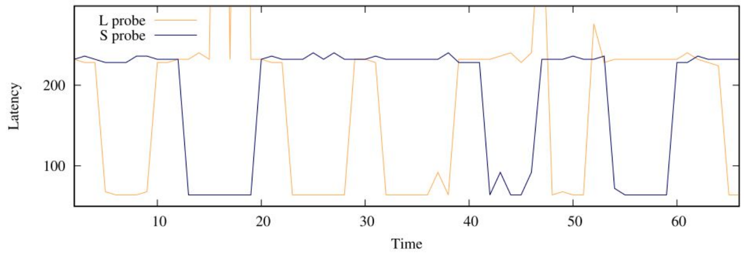


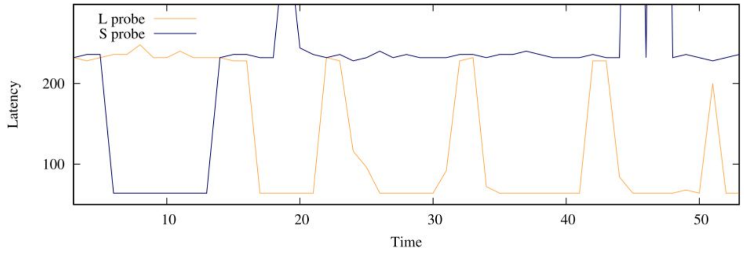

LSLLSLSL...

# Step 4
# Recover Bits

# Bit Recovery




LSLLSLSL... ≠ 01001010...




SLLLLL... ≠ 100000...

# Bit Recovery

$2^{26}$

Sequences

→

Bits >= 3

Length

L=5

→

| Pattern | $\ell_i$ | $a_i$ |
|---|---|---|
| LLLLL | 5 | 0 |
| SLLLL | 4 | 1 |
| LSLLL | 4 | 2 |
| SLSLL | 3 | 3 |
| LLSLL | 4 | 4 |
| SLLSL | 3 | 5 |
| LSLSL | 3 | 6 |
| SLSLS | 3 | 7 |
| LLLSL | 4 | 8 |
| SLLLS | 4 | 9 |
| LSLLS | 4 | 10 |
| LLSLS | 4 | 12 |
| LLLLS | 5 | 16 |

# Bit Recovery

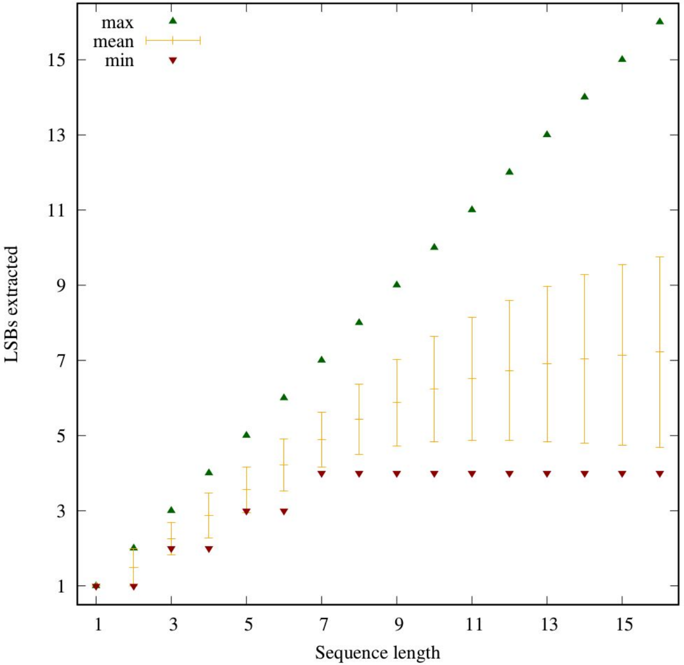



Figure 6: Empirical number of extracted bits for various sequence lengths. Each sequence length consisted of $2^{26}$ trials, over which we calculated the mean (with deviation), maximum, and minimum number of recovered LSBs. Error bars are one standard deviation on each side.

| Pattern | $\ell_i$ | $a_i$ |
|---|---|---|
| LLLLLLL | 7 | 0 |
| SLLLLSL | 5 | 1 |
| LSLLLLS | 6 | 2 |
| SLSLLSL | 4 | 3 |
| LLSLLLL | 6 | 4 |
| SLLSLSL | 4 | 5 |
| LSLSLLS | 5 | 6 |
| SLSLSLL | 4 | 7 |
| LLLSLLL | 6 | 8 |
| SLLLSLL | 5 | 9 |
| LSLLSLS | 5 | 10 |
| SLSLLLS | 5 | 11 |
| LLSLSLL | 5 | 12 |
| SLLSLLS | 5 | 13 |
| LSLSLSL | 4 | 14 |
| SLSLSLS | 4 | 15 |
| LLLLSLL | 6 | 16 |
| SLLLLLL | 6 | 17 |
| LSLLLSL | 5 | 18 |
| LLSLLSL | 5 | 20 |
| LSLSLLL | 5 | 22 |
| LLLSLSL | 5 | 24 |
| SLLLSLS | 5 | 25 |
| LSLLSLL | 5 | 26 |
| SLSLLLL | 5 | 27 |
| LLSLSLS | 5 | 28 |
| SLLSLLL | 5 | 29 |
| LLLLLSL | 6 | 32 |
| LSLLLLL | 6 | 34 |
| LLSLLLS | 6 | 36 |
| LLLSLLS | 6 | 40 |
| LLLLSLS | 6 | 48 |
| SLLLLLS | 6 | 49 |
| LLLLLLS | 7 | 64 |

TAMPERE UNIVERSITY OF TECHNOLOGY

# Step 5
# Lattice Attack

# Lattice Attack

**Input** parameters to **Lattice:**

- `Bits recovered`
- `Messages`
- `Signatures`

**Lattice** information:

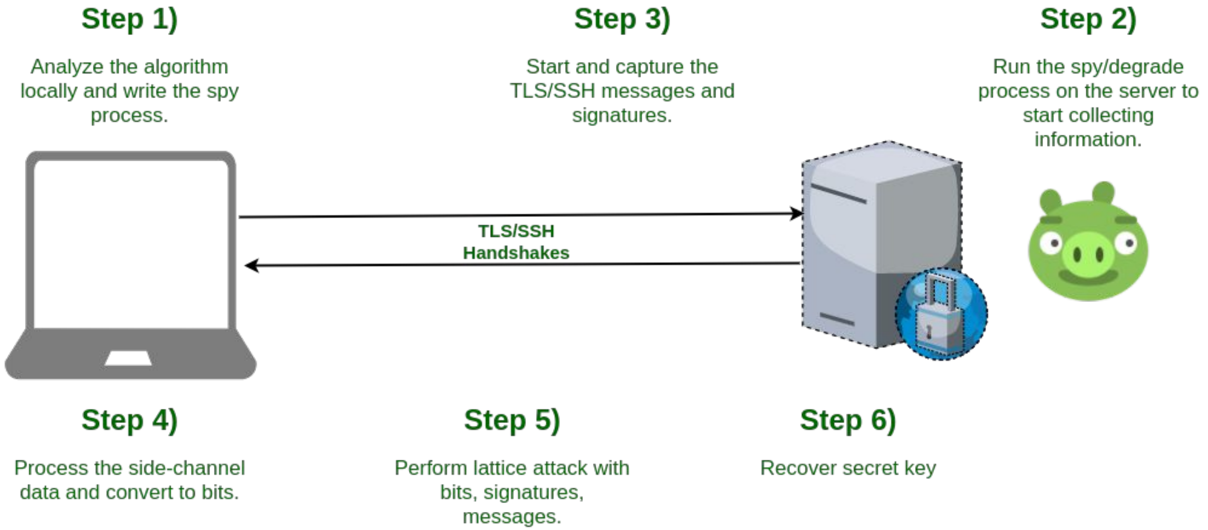
- Dimension $d + 2$
- Implemented in Sage
- BKZ reduction (block size 30)

| Source | Signa- tures | $d$ | $\ell$ | $j$ | $\mu_l$ | Success Rate (%) | CPU Minutes |
|---|---|---|---|---|---|---|---|
| Prev. [8] | 168 | 42 | 8 | — | 336.0 | 100.0 | 0.7 |
| Prev. [8] | 312 | 24 | 12 | — | 288.0 | 100.0 | 0.6 |
| This work | 50 | 50 | $\{4..7\}$ | 7 | 249.7 | 14.0 | 79.5 |
| This work | 55 | 55 | $\{4..7\}$ | 7 | 268.8 | 98.0 | 1.7 |
| This work | 60 | 60 | $\{4..7\}$ | 7 | 293.4 | 100.0 | 0.7 |
| This work | 70 | 70 | $\{3..5\}$ | 5 | 258.2 | 5.0 | 130.8 |
| This work | 80 | 80 | $\{3..5\}$ | 5 | 286.1 | 94.5 | 13.2 |
| This work | 90 | 90 | $\{3..5\}$ | 5 | 321.2 | 100.0 | 4.0 |

[8] Cabrera Aldaya et al. "SPA vulnerabilities of the binary extended Euclidean algorithm." *Journal of Cryptographic Engineering* (2016): 1-13.

# End-to-End Protocol Attack

# End-to-End Protocol Attack



# Cryptographic Libraries

- Crypto **libraries** are a prime **target** for CTA!
- We offered a patch to the libraries
- OpenSSL 1.0.1 development reached **EOL** starting **January 2017**.
- OpenSSL 1.0.1 shipped with **Ubuntu LTS 12.04** and **14.04**; **Debian 7.0** and **8.0**; and **SUSE**.
- **Upgrade** to OpenSSL **1.0.2 or higher.**
- Otherwise, apply the **patch**!

# Conclusions

- Constant-time implementations need to be **tested**.
- The **BEEA** modular inversion **enables** practical cache-timing attacks.
- The **performance degradation** technique **improves** trace quality.
- Different key bit recovery approaches **are possible**.
- Cache-Timing attacks are increasing in **popularity** and **complexity** every year.

# Thank you
# Questions?