
Peter Schwabe

January 18, 2015

ShmooCon 2015
Secure Crypto

Research over the past decades has produced several secure crypto algorithms:

- AES-256 block cipher
Secure Crypto

Research over the past decades has produced several secure crypto algorithms:

- AES-256 block cipher
- AES-CBC + HMAC-SHA256 authenticated encryption
Secure Crypto

Research over the past decades has produced several secure crypto algorithms:

- AES-256 block cipher
- AES-CBC + HMAC-SHA256 authenticated encryption
- RSA-2048 public-key encryption
Secure Crypto

Research over the past decades has produced several secure crypto algorithms:

- AES-256 block cipher
- AES-CBC + HMAC-SHA256 authenticated encryption
- RSA-2048 public-key encryption
- ECDSA signatures with the secp256k1 curve (used in Bitcoin)
Secure Crypto?

- Osvik, Shamir, Tromer, 2006: Recover AES-256 secret key of Linux’s dmcrypt in just 65 ms
Secure Crypto?

- Osvik, Shamir, Tromer, 2006: Recover AES-256 secret key of Linux’s dmcrypt in just 65 ms
- AlFardan, Paterson, 2013: “Lucky13” recovers plaintext of CBC-mode encryption in pretty much all TLS implementations
Secure Crypto?

- Osvik, Shamir, Tromer, 2006: Recover AES-256 secret key of Linux’s dm crypt in just 65 ms
- AlFardan, Paterson, 2013: “Lucky13” recovers plaintext of CBC-mode encryption in pretty much all TLS implementations
- Yarom, Falkner, 2014: Attack against RSA-2048 in GnuPG 1.4.13: “On average, the attack is able to recover 96.7% of the bits of the secret key by observing a single signature or decryption round.”
Secure Crypto?

- Osvik, Shamir, Tromer, 2006: Recover AES-256 secret key of Linux’s dmcrypt in just 65 ms
- AlFardan, Paterson, 2013: “Lucky13” recovers plaintext of CBC-mode encryption in pretty much all TLS implementations
- Yarom, Falkner, 2014: Attack against RSA-2048 in GnuPG 1.4.13: “On average, the attack is able to recover 96.7% of the bits of the secret key by observing a single signature or decryption round.”
- Benger, van de Pol, Smart, Yarom, 2014: “reasonable level of success in recovering the secret key” for OpenSSL ECDSA using secp256k1 “with as little as 200 signatures”
Secure Crypto?

- Osvik, Shamir, Tromer, 2006: Recover AES-256 secret key of Linux’s dmcrypt in just 65 ms
- AlFardan, Paterson, 2013: “Lucky13” recovers plaintext of CBC-mode encryption in pretty much all TLS implementations
- Yarom, Falkner, 2014: Attack against RSA-2048 in GnuPG 1.4.13: “On average, the attack is able to recover 96.7% of the bits of the secret key by observing a single signature or decryption round.”
- Benger, van de Pol, Smart, Yarom, 2014: “reasonable level of success in recovering the secret key” for OpenSSL ECDSA using secp256k1 “with as little as 200 signatures”

Those attacks all don’t break the math!
Timing Attacks

General idea of those attacks

- Secret data has influence on timing of software
- Attacker measures timing
- Attacker computes influence$^{-1}$ to obtain secret data
Timing Attacks

General idea of those attacks

- Secret data has influence on timing of software
- Attacker measures timing
- Attacker computes influence\(^{-1}\) to obtain secret data

Two kinds of remote...

- Timing attacks are a type of side-channel attacks
- Unlike other side-channel attacks, they work remotely:
  - Some need to run attack code in parallel to the target software
  - Attacker can log in remotely (ssh)
Timing Attacks

General idea of those attacks

- Secret data has influence on timing of software
- Attacker measures timing
- Attacker computes influence\(^{-1}\) to obtain secret data

Two kinds of remote...

- Timing attacks are a type of side-channel attacks
- Unlike other side-channel attacks, they work remotely:
  - Some need to run attack code in parallel to the target software
  - Attacker can log in remotely (ssh)
  - Some attacks work by measuring network delays
  - Attacker does not even need an account on the target machine
Timing Attacks

General idea of those attacks

- Secret data has influence on timing of software
- Attacker measures timing
- Attacker computes influence \(^{-1}\) to obtain secret data

Two kinds of remote... 

- Timing attacks are a type of side-channel attacks
- Unlike other side-channel attacks, they work remotely:
  - Some need to run attack code in parallel to the target software
  - Attacker can log in remotely (ssh)
  - Some attacks work by measuring network delays
  - Attacker does not even need an account on the target machine
- Can’t protect against timing attacks by locking a room
if (secret)
{
    do_A();
}
else
{
    do_B();
}
Exponentiation

- Core operation in RSA decryption: $a^d \mod n$ with secret key $d$
- Very similar operation involved in ElGamal, DSA, and ECC
Exponentiation

- Core operation in RSA decryption: $a^d \mod n$ with secret key $d$
- Very similar operation involved in ElGamal, DSA, and ECC

Example: exponent 105

- $105 = 64 + 32 + 8 + 1 = 2^6 + 2^5 + 2^3 + 2^0$
Exponentiation

- Core operation in RSA decryption: \(a^d \mod n\) with secret key \(d\)
- Very similar operation involved in ElGamal, DSA, and ECC

Example: exponent 105

- \(105 = 64 + 32 + 8 + 1 = 2^6 + 2^5 + 2^3 + 2^0\)
- \(105 = 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0\)
Exponentiation

- Core operation in RSA decryption: \( a^d \mod n \) with secret key \( d \)
- Very similar operation involved in ElGamal, DSA, and ECC

Example: exponent 105

- \( 105 = 64 + 32 + 8 + 1 = 2^6 + 2^5 + 2^3 + 2^0 \)
- \( 105 = 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \)
- \( 105 = (((((((((1 \cdot 2 + 1) \cdot 2) + 0) \cdot 2) + 1) \cdot 2) + 0) \cdot 2) + 0) \cdot 2) + 1 \) (Horner’s rule)
- \( a^{105} = (((((((((a^2 \cdot a)^2) \cdot 1)^2) \cdot a)^2) \cdot 1)^2) \cdot 1)^2) \cdot a \)
Exponentiation

- Core operation in RSA decryption: \( a^d \mod n \) with secret key \( d \)
- Very similar operation involved in ElGamal, DSA, and ECC

**Example: exponent 105**

- \( 105 = 64 + 32 + 8 + 1 = 2^6 + 2^5 + 2^3 + 2^0 \)
- \( 105 = 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \)
- \( 105 = ((((((1 \cdot 2 + 1) \cdot 2) + 0) \cdot 2) + 1) \cdot 2) + 0) \cdot 2) + 0) \cdot 2) + 1 \) (Horner’s rule)
- \( a^{105} = ((((((a^2 \cdot a^2) \cdot 1)^2) \cdot a)^2) \cdot 1)^2) \cdot 1)^2) \cdot a \)
- Cost: 6 squarings, 3 multiplications
- More generally: 1 squaring per bit, 1 multiplication per 1-bit
typedef unsigned long long uint64;
typedef uint32_t uint32;

/* This really wants to be done with long integers */
uint32 modexp(uint32 a, uint32 mod, unsigned char exp[4]) {
  int i,j;
  uint32 r = 1;
  for(i=3;i>=0;i--) {
    for(j=7;j>=0;j--) {
      r = ((uint64)r*r) % mod;
      if(exp[i] & (1<<j))
        r = ((uint64)a*r) % mod;
    }
  }
  return r;
}
/ * This really wants to be done with long integers */
uint32 modexp(uint32 a, uint32 mod, unsigned char exp[4]) {
    int i,j;
    uint32 r = 1,t;
    for(i=3;i>=0;i--) {
        for(j=7;j>=0;j--) {
            r = ((uint64)r*r) % mod;
            if(exp[i] & (1<<j))
                r = ((uint64)a*r) % mod;
            else
                t = ((uint64)a*r) % mod;
        }
    }
    return r;
}
/* This really wants to be done with long integers */
uint32 modexp(uint32 a, uint32 mod, unsigned char exp[4]) {
    int i,j;
    uint32 r = 1,t;
    for(i=3;i>=0;i--) {
        for(j=7;j>=0;j--) {
            r = ((uint64)r*r) % mod;
            if(exp[i] & (1<<j))
                r = ((uint64)a*r) % mod;
            else
                t = ((uint64)a*r) % mod;
        }
    }
    return r;
}

Compiler may optimize else clause away, but can avoid that
Square-and-multiply-always

/* This really wants to be done with long integers */
uint32 modexp(uint32 a, uint32 mod, unsigned char exp[4]) {
    int i,j;
    uint32 r = 1,t;
    for(i=3;i>=0;i--) {
        for(j=7;j>=0;j--) {
            r = ((uint64)r*r) % mod;
            if(exp[i] & (1<<j))
                r = ((uint64)a*r) % mod;
            else
                t = ((uint64)a*r) % mod;
        }
    }
    return r;
}

► Compiler may optimize else clause away, but can avoid that
► Still not constant time, reasons:
  ► Branch prediction
  ► Instruction cache
Eliminating branches

- So, what do we do with code like this?

```plaintext
define s
    if s then
        r ← A
    else
        r ← B
    end if
```

Eliminating branches

- So, what do we do with code like this?

```plaintext
if s then
    r ← A
else
    r ← B
end if
```

- Replace by

```plaintext
r ← sA + (1 - s)B
```
Eliminating branches

So, what do we do with code like this?

\[
\text{if } s \text{ then} \\
\quad r \leftarrow A \\
\text{else} \\
\quad r \leftarrow B \\
\text{end if}
\]

Replace by

\[
r \leftarrow sA + (1 - s)B
\]

Can expand \( s \) to all-one/all-zero mask and use XOR instead of addition, AND instead of multiplication.
Eliminating branches

- So, what do we do with code like this?
  ```plaintext
  if s then
      r ← A
  else
      r ← B
  end if
  ```
- Replace by
  ```plaintext
  r ← sA + (1 − s)B
  ```
- Can expand $s$ to all-one/all-zero mask and use XOR instead of addition, AND instead of multiplication
- For very fast $A$ and $B$ this can even be faster
Fixing Square-and-multiply-always

```c
uint32 modexp(uint32 a, uint32 mod, unsigned char exp[4]) {
    int i, j;
    uint32 r = 1, t;
    for(i=3; i>=0; i--) {
        for(j=7; j>=0; j--) {
            r = ((uint64)r*r) % mod;
            t = ((uint64)a*r) % mod;
            cmov(&r, &t, (exp[i] & (1<<j)) >> j);
        }
    }
    return r;
}
```
/* decision bit b has to be either 0 or 1 */
void cmov(uint32 *r, uint32 *a, uint32 b)
{
    uint32 t;

    b = -b; /* Now b is either 0 or 0xffffffff */
    t = (*r ^ *a) & b;
    *r ^= t;
}
Problem No. 2

table[secret]
The Advanced Encryption Standard (AES)

- Block cipher Rijndael proposed by Rijmen, Daemen in 1998
- Selected as AES by NIST in October 2000
The Advanced Encryption Standard (AES)

- Block cipher Rijndael proposed by Rijmen, Daemen in 1998
- Selected as AES by NIST in October 2000
- Block size: 128 bits (AES state: $4 \times 4$ matrix of 16 bytes)
- Key size 128/192/256 bits (resp. 10/12/14 rounds)
The Advanced Encryption Standard (AES)

- Block cipher Rijndael proposed by Rijmen, Daemen in 1998
- Selected as AES by NIST in October 2000
- Block size: 128 bits (AES state: $4 \times 4$ matrix of 16 bytes)
- Key size 128/192/256 bits (resp. 10/12/14 rounds)
- AES with $n$ rounds uses $n + 1$ 16-byte rounds keys $K_0, \ldots, K_n$
The Advanced Encryption Standard (AES)

- Block cipher Rijndael proposed by Rijmen, Daemen in 1998
- Selected as AES by NIST in October 2000
- Block size: 128 bits (AES state: $4 \times 4$ matrix of 16 bytes)
- Key size 128/192/256 bits (resp. 10/12/14 rounds)
- AES with $n$ rounds uses $n + 1$ 16-byte rounds keys $K_0, \ldots, K_n$
- Four operations per round: SubBytes, ShiftRows, MixColumns, and AddRoundKey
- Last round does not have MixColumns
Implementing AES on 32-bit machines

“The different steps of the round transformation can be combined in a single set of table lookups, allowing for very fast implementations on processors with word length 32 or above.”

Implementing AES on 32-bit machines

“The different steps of the round transformation can be combined in a single set of table lookups, allowing for very fast implementations on processors with word length 32 or above.”


The first round of AES in C

- **Input**: 32-bit integers $y_0$, $y_1$, $y_2$, $y_3$
- **Output**: 32-bit integers $z_0$, $z_1$, $z_2$, $z_3$
- **Round keys in 32-bit-integer array $rk[44]$**

\[
\begin{align*}
  z_0 &= T_0[(y_0 >> 24)] ^ T_1[(y_1 >> 16) & 0xff] ^ rk[4]; \\
  z_1 &= T_0[(y_1 >> 24)] ^ T_1[(y_2 >> 16) & 0xff] ^ rk[5]; \\
  z_2 &= T_0[(y_2 >> 24)] ^ T_1[(y_3 >> 16) & 0xff] ^ rk[6]; \\
  z_3 &= T_0[(y_3 >> 24)] ^ T_1[(y_0 >> 16) & 0xff] ^ rk[7];
\end{align*}
\]
Cache-timing attacks

- AES and the attackers program run on the same CPU
- Tables are in cache
Cache-timing attacks

- AES and the attackers program run on the same CPU
- Tables are in cache
- The attacker’s program replaces some cache lines
Cache-timing attacks

- AES and the attackers program run on the same CPU
- Tables are in cache
- The attacker’s program replaces some cache lines
- AES continues, loads from table again

<table>
<thead>
<tr>
<th>(T0[0]) (\ldots) (T0[15])</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(T0[16]) (\ldots) (T0[31])</td>
<td></td>
</tr>
<tr>
<td>???</td>
<td></td>
</tr>
<tr>
<td>???</td>
<td></td>
</tr>
<tr>
<td>(T0[64]) (\ldots) (T0[79])</td>
<td></td>
</tr>
<tr>
<td>(T0[80]) (\ldots) (T0[95])</td>
<td></td>
</tr>
<tr>
<td>???</td>
<td></td>
</tr>
<tr>
<td>???</td>
<td></td>
</tr>
<tr>
<td>???</td>
<td></td>
</tr>
<tr>
<td>???</td>
<td></td>
</tr>
<tr>
<td>(T0[160]) (\ldots) (T0[175])</td>
<td></td>
</tr>
<tr>
<td>(T0[176]) (\ldots) (T0[191])</td>
<td></td>
</tr>
<tr>
<td>(T0[192]) (\ldots) (T0[207])</td>
<td></td>
</tr>
<tr>
<td>(T0[208]) (\ldots) (T0[223])</td>
<td></td>
</tr>
<tr>
<td>???</td>
<td></td>
</tr>
<tr>
<td>???</td>
<td></td>
</tr>
</tbody>
</table>
Cache-timing attacks

- AES and the attackers program run on the same CPU
- Tables are in cache
- The attacker’s program replaces some cache lines
- AES continues, loads from table again
- Attacker loads his data:
Cache-timing attacks

- AES and the attackers program run on the same CPU
- Tables are in cache
- The attacker’s program replaces some cache lines
- AES continues, loads from table again
- Attacker loads his data:
  - Fast: cache hit (AES did not just load from this line)
Cache-timing attacks

- AES and the attackers program run on the same CPU
- Tables are in cache
- The attacker’s program replaces some cache lines
- AES continues, loads from table again
- Attacker loads his data:
  - Fast: cache hit (AES did not just load from this line)
  - Slow: cache miss (AES just loaded from this line)
The general case

Loads from and stores to addresses that depend on secret data leak secret data.
“Countermeasure”

- Observation: This simple \textit{cache-timing attack} does not reveal the secret address, only the cache line

- Idea: Lookups \textit{within one cache line} should be safe
“Countermeasure”

- Observation: This simple cache-timing attack does not reveal the secret address, only the cache line
- Idea: Lookups within one cache line should be safe... or are they?
“Countermeasure”

- Observation: This simple *cache-timing attack* does not reveal the secret address, only the cache line
- Idea: Lookups *within one cache line* should be safe... or are they?
- Bernstein, 2005: “*Does this guarantee constant-time S-box lookups? No!*”
Observation: This simple cache-timing attack does not reveal the secret address, only the cache line.

Idea: Lookups within one cache line should be safe... or are they?

Bernstein, 2005: “Does this guarantee constant-time S-box lookups? No!”

Osvik, Shamir, Tromer, 2006: “This is insufficient on processors which leak low address bits”
Observation: This simple cache-timing attack does not reveal the secret address, only the cache line

Idea: Lookups within one cache line should be safe... or are they?

Bernstein, 2005: “Does this guarantee constant-time S-box lookups? No!”

Osvik, Shamir, Tromer, 2006: “This is insufficient on processors which leak low address bits”

Reasons:
- Cache-bank conflicts
- Failed store-to-load forwarding
- ...
“Countermeasure”

- Observation: This simple cache-timing attack does not reveal the secret address, only the cache line.
- Idea: Lookups within one cache line should be safe... or are they?
- Bernstein, 2005: “Does this guarantee constant-time S-box lookups? No!”
- Osvik, Shamir, Tromer, 2006: “This is insufficient on processors which leak low address bits”
- Reasons:
  - Cache-bank conflicts
  - Failed store-to-load forwarding
  -...
- OpenSSL is using it in BN_mod_exp_mont_consttime
“Countermeasure”

- Observation: This simple *cache-timing attack* does not reveal the secret address, only the cache line
- Idea: Lookups *within one cache line* should be safe... or are they?
- Bernstein, 2005: “*Does this guarantee constant-time S-box lookups? No!*”
- Osvik, Shamir, Tromer, 2006: “*This is insufficient on processors which leak low address bits*”
- Reasons:
  - Cache-bank conflicts
  - Failed store-to-load forwarding
  - ...
- OpenSSL is using it in `BN_mod_exp_mont_constttime`
- Brickell (Intel), 2011: yeah, it’s fine as a countermeasure
“Countermeasure”

- Observation: This simple *cache-timing attack* does not reveal the secret address, only the cache line
- Idea: Lookups *within one cache line* should be safe... or are they?
- Bernstein, 2005: “*Does this guarantee constant-time S-box lookups? No!*”
- Osvik, Shamir, Tromer, 2006: “*This is insufficient on processors which leak low address bits*”
- Reasons:
  - Cache-bank conflicts
  - Failed store-to-load forwarding
  - ...
- OpenSSL is using it in `BN_mod_exp_mont_consttime`
- Brickell (Intel), 2011: yeah, it’s fine as a countermeasure
- Bernstein, Schwabe, 2013: Demonstrate timing variability for access within one cache line
“Countermeasure”

- Observation: This simple *cache-timing attack* does not reveal the secret address, only the cache line
- Idea: Lookups *within one cache line* should be safe... or are they?
- Bernstein, 2005: "*Does this guarantee constant-time S-box lookups? No!*"
- Osvik, Shamir, Tromer, 2006: "*This is insufficient on processors which leak low address bits*"
- Reasons:
  - Cache-bank conflicts
  - Failed store-to-load forwarding
  - ...  
- OpenSSL is using it in `BN_mod_exp_mont_consttime`
- Brickell (Intel), 2011: yeah, it’s fine as a countermeasure
- Bernstein, Schwabe, 2013: Demonstrate timing variability for access within one cache line
- TODO: Real attack against, e.g., OpenSSL
Countermeasure

```c
uint32 table[TABLE_LENGTH];

uint32 lookup(size_t pos)
{
    size_t i;
    int b;
    uint32 r = table[0];
    for(i=1;i<TABLE_LENGTH;i++)
    {
        b = (i == pos);
        cmov(&r, &table[i], b);
    }
    return r;
}
```
Countermeasure

```c
uint32 table[TABLE_LENGTH];

uint32 lookup(size_t pos)
{
    size_t i;
    int b;
    uint32 r = table[0];
    for(i=1;i<TABLE_LENGTH;i++)
    {
        b = (i == pos); /* DON’T! Compiler may do funny things! */
        cmov(&r, &table[i], b);
    }
    return r;
}
```
Countermeasure

```c
uint32 table[TABLE_LENGTH];

uint32 lookup(size_t pos)
{
    size_t i;
    int b;
    uint32 r = table[0];
    for(i=1;i<TABLE_LENGTH;i++)
    {
        b = isequal(i, pos);
        cmov(&r, &table[i], b);
    }
    return r;
}
```
int isequal(uint32 a, uint32 b)
{
    size_t i; uint32 r = 0;
    unsigned char *ta = (unsigned char *)&a;
    unsigned char *tb = (unsigned char *)&b;
    for(i=0;i<sizeof(uint32);i++)
    {
        r |= (ta[i] ^ tb[i]);
    }
    r = (-r) >> 31;
    return (int)(1-r);
}
Back to AES

How could AES be chosen?

“Table lookup: not vulnerable to timing attacks; relatively easy to effect a defense against power attacks by software balancing of the lookup address.”

How could AES be chosen?

“Table lookup: not vulnerable to timing attacks; relatively easy to effect a defense against power attacks by software balancing of the lookup address.” —Report on the Development of the Advanced Encryption Standard (AES), October 2000

What now?

- You can use generic constant-time lookups for AES tables
- It’s horribly inefficient
Back to AES

How could AES be chosen?
“Table lookup: not vulnerable to timing attacks; relatively easy to effect a defense against power attacks by software balancing of the lookup address.”

What now?
- You *can* use generic constant-time lookups for AES tables
- It’s horribly inefficient
- Intel’s answer: let’s do it in hardware (AES-NI, since Westmere)
Back to AES

How could AES be chosen?
“Table lookup: not vulnerable to timing attacks; relatively easy to effect a defense against power attacks by software balancing of the lookup address.”

What now?
- You can use generic constant-time lookups for AES tables
- It’s horribly inefficient
- Intel’s answer: let’s do it in hardware (AES-NI, since Westmere)
- ARM’s answer: let’s do it in hardware (crypto extension in ARMv8)
Back to AES

How could AES be chosen?
“Table lookup: not vulnerable to timing attacks; relatively easy to effect a defense against power attacks by software balancing of the lookup address.”

What now?
- You can use generic constant-time lookups for AES tables
- It’s horribly inefficient
- Intel’s answer: let’s do it in hardware (AES-NI, since Westmere)
- ARM’s answer: let’s do it in hardware (crypto extension in ARMv8)
- Solutions in software:
  - AES with vector-permute instructions (Hamburg, 2009)
  - Bitslicing (Biham, 1997, for DES)
Bitslicing

- Imagine registers that have only one bit
- Perform arithmetic on those registers using XOR, AND, OR
- Essentially the same as hardware implementations
Bitslicing

- Imagine registers that have only one bit
- Perform arithmetic on those registers using XOR, AND, OR
- Essentially the same as hardware implementations
- But wait, registers are longer!
- Think of them as vectors of bits
- Perform the simulated hardware implementations on many independent data streams
Imagine registers that have only one bit
Perform arithmetic on those registers using XOR, AND, OR
Essentially the same as hardware implementations
But wait, registers are longer!
Think of them as vectors of bits
Perform the simulated hardware implementations on many independent data streams
Bitslicing works for every algorithm
Bitslicing is inherently protected against timing attacks
Efficient bitslicing needs a huge amount of data-level parallelism
Bitslicing binary polynomials

4-coefficient binary polynomials

\[(a_3x^3 + a_2x^2 + a_1x + a_0), \text{ with } a_i \in \{0, 1\}\]

4-coefficient bitsliced binary polynomials

typedef unsigned char poly4; /* 4 coefficients in the low 4 bits */
typedef unsigned long long poly4x64[4];

void poly4_bitslice(poly4x64 r, const poly4 x[64])
{
    int i,j;
    for(i=0;i<4;i++)
    {
        r[i] = 0;
        for(j=0;j<64;j++)
            r[i] |= (unsigned long long)(1 & (x[j] >> i))<<j;
    }
}
typedef unsigned long long poly4x64[4];
typedef unsigned long long poly7x64[7];

void poly4x64_mul(poly7x64 r, const poly4x64 a, const poly4x64 b) {
    r[0] = a[0] & b[0];
    r[1] = (a[0] & b[1]) ^ (a[1] & b[0]);
    r[2] = (a[0] & b[2]) ^ (a[1] & b[1]) ^ (a[2] & b[0]);
}
Is that all?

Lesson so far

- Avoid all data flow from secrets to branch conditions and memory addresses
- This can always be done; cost highly depends on the algorithm
Is that all?

Lesson so far

- Avoid all data flow from secrets to branch conditions and memory addresses
- This can *always* be done; cost highly depends on the algorithm
- Test this with `valgrind` and *uninitialized secret data* (or use Langley’s `ctgrind`)

Is that all?

Lesson so far

- Avoid all data flow from secrets to branch conditions and memory addresses
- This can *always* be done; cost highly depends on the algorithm
- Test this with valgrind and *uninitialized secret data* (or use Langley’s ctgrind)

“In order for a function to be constant time, the branches taken and memory addresses accessed must be independent of any secret inputs. (That’s assuming that the fundamental processor instructions are constant time, but that’s true for all sane CPUs.)”

—Langley, Apr. 2010
Lesson so far

- Avoid all data flow from secrets to branch conditions and memory addresses
- This can always be done; cost highly depends on the algorithm
- Test this with valgrind and uninitialized secret data (or use Langley’s ctgrind)

“In order for a function to be constant time, the branches taken and memory addresses accessed must be independent of any secret inputs. (That’s assuming that the fundamental processor instructions are constant time, but that’s true for all sane CPUs.)”

—Langley, Apr. 2010

“So the argument to the DIV instruction was smaller and DIV, on Intel, takes a variable amount of time depending on its arguments!”

—Langley, Feb. 2013
Dangerous arithmetic (examples)

- DIV, IDIV, FDIV on pretty much all Intel/AMD CPUs
- Various math instructions on Intel/AMD CPUs (FSIN, FCOS...)
Dangerous arithmetic (examples)

- DIV, IDIV, FDIV on pretty much all Intel/AMD CPUs
- Various math instructions on Intel/AMD CPUs (FSIN, FCOS...)
- MUL, MULHW, MULHWU on many PowerPC CPUs
- UMULL, SMULL, UMLAL, and SMLAL on ARM Cortex-M3.
Dangerous arithmetic (examples)

- DIV, IDIV, FDIV on pretty much all Intel/AMD CPUs
- Various math instructions on Intel/AMD CPUs (FSIN, FCOS...)
- MUL, MULHW, MULHWU on many PowerPC CPUs
- UMULL, SMULL, UMLAL, and SMLAL on ARM Cortex-M3.

Solution

- Avoid these instructions
- Make sure that inputs to the instructions don’t leak timing information
References I

- Osvik, Shamir, Tromer, 2006: *Cache Attacks and Countermeasures: the Case of AES.*
  http://eprint.iacr.org/2005/271/

  http://www.isg.rhul.ac.uk/tls/Lucky13.html

  http://eprint.iacr.org/2013/448/

- Benger, van de Pol, Smart, Yarom, 2014: “Ooh Aah... Just a Little Bit”: A small amount of side channel can go a long way.
  http://eprint.iacr.org/2014/161/
References II

- Bernstein, 2005: *Cache-timing attacks on AES.*
  http://cr.yp.to/papers.html#cachetiming


- Bernstein, Schwabe, 2013: *A word of warning.*
  https://cryptojedi.org/peter/data/chesrump-20130822.pdf
  https://cryptojedi.org/peter/data/cacheline.tar.bz2


- Biham, 1997: “*A Fast New DES Implementation in Software.*”
Contact

http://cryptojedi.org