Side-Channel Attack against RSA Key Generation Algorithms

CHES 2014

Aurélie Bauer, Eliane Jaulmes, Victor Lomné,
Emmanuel Prouff and Thomas Roche
Agence Nationale de la Sécurité des Systèmes d'Information
(French Network and Information Security Agency)

Thursday, September 25th, 2014

Outline

Outline

SCA: Principle

- SCA consist in measuring a physical leakage of a device when it handles sensitive information
  - e.g. cryptographic keys
- Handled info. is correlated with the physical leakage
  - e.g. a register leaking as the Hamming Weight of its value
- The attacker can then apply statistical methods to extract the secret from the measurements
  - Simple Side-Channel Attacks (SSCA)
  - Differential Side-Channel Attacks (DSCA)
  - Template Attacks (TA)
  - Collision-based Side-Channel Attacks
  - ...

RSA (Rivest - Shamir - Adelman)

- RSA: the most used public-key cryptosystem

- Key Generation
  - Generate p, q two prime numbers of same size
  - Compute $n = p \cdot q$, and $\phi(n) = (p-1) \cdot (q-1)$
  - Choose an integer e such that e and $\phi(n)$ are coprime
  - Compute d, the multiplicative inverse of e modulo $\phi(n)$

  $\Rightarrow$ Public Key: (e, n) / Private Key: d

- Encryption-Decryption / Signature-Verification
  - Encryption / Verification: $c = m^e \pmod{n}$
  - Decryption / Signature: $m = c^d \pmod{n}$

SCA on RSA 1/2

- Attacking during the Key Generation
- Key Generation
  - Generate p, q two prime numbers of same size
  - Compute $n = p \cdot q$, and $\phi(n) = (p-1) \cdot (q-1)$
  - Choose an integer e such that e and $\phi(n)$ are coprime
  - Compute d, the multiplicative inverse of e modulo $\phi(n)$

  $\Rightarrow$ Public Key: (e, n) / Private Key: d
- Encryption-Decryption / Signature-Verification
  - Encryption / Verification: $c = m^e \ (mod\ n)$
  - Decryption / Signature: $m = c^d \ (mod\ n)$

SCA on RSA 2/2

- Attacking during the Decryption / Signature

- Key Generation
  - Generate p, q two prime numbers of same size
  - Compute $n = p \cdot q$, and $\phi(n) = (p-1) \cdot (q-1)$
  - Choose an integer e such that e and $\phi(n)$ are coprime
  - Compute d, the multiplicative inverse of e modulo $\phi(n)$

  $\Rightarrow$ Public Key: (e, n) / Private Key: d

- Encryption-Decryption / Signature-Verification
  - Encryption / Verification: $c = m^e \ (mod\ n)$
  - Decryption / Signature: $m = c^d \ (mod\ n)$

RSA Key Generation exposed ?

- Most of the works about Physical Cryptanalysis on RSA focus on attacking during Decryption / Signature
- Until recent years, RSA Key Generation was performed during device personalisation
- This is no longer the case, due to new security services (mobile payment, e-ticketing, OTP generations, ...)
- Some devices can perform RSA Key generation during their life cycle

This Work $\Rightarrow$ case 1/2

- Attacking during the Prime Number Generation

- Key Generation
  - Generate p, q two prime numbers of same size
  - Compute $n = p \cdot q$, and $\phi(n) = (p-1) \cdot (q-1)$
  - Choose an integer e such that e and $\phi(n)$ are coprime
  - Compute d, the multiplicative inverse of e modulo $\phi(n)$

  $\Rightarrow$ Public Key: (e, n) / Private Key: d

- Encryption-Decryption / Signature-Verification
  - Encryption / Verification: $c = m^e \ (mod\ n)$
  - Decryption / Signature: $m = c^d \ (mod\ n)$

Outline

How to generate a prime number ?

- Two methods to generate a prime number:
  - Provable prime generation algorithms
    1. pick up a random odd value
    2. perform a provable primality test
    3. if test fails, increment the random value and go to step 2
  - Probable prime generation algorithms
    1. pick up a random odd value
    2. perform a probable primality test
    3. if test fails, increment the random value and go to step 2
- Probable algorithms generally used for embedded systems due to timing constraints

Algorithm: Probable Prime Generation Algorithm v1

Input : A bit-length ℓ, the set $\mathcal{S} = \{s_0, \cdots, s_{52}\}$ of all odd primes lower than 256
Output: A probable prime p

/* Generate a seed */
1 Randomly generate an odd ℓ-bit integer v_0
/* Prime Sieve */
2 $v \leftarrow v_0$
3 $s \leftarrow s_0$
4 $i = 0$
5 **while** $(v \mod s \neq 0)$ and $(i < 53)$ **do**
6     $i = i + 1$
7     $s \leftarrow s_i$
8 **if** $(i \neq 53)$ **then**
9     $v = v + 2$
10   **goto** Step 3
/* Probabilistic primality tests */
11 **else**
12     $i = 0$
    /* Process t Miller-Rabin's tests (stop if one fails) */
13   **while** (Miller-Rabin(v) = ok) and $(i < t)$ **do**
14         $i = i + 1$
/* Process one Lucas' test */
15 **if** $(i = t)$ and (Lucas(v) = ok) **then**
16   **return** v
17 **else**
18     $v = v + 2$
19   **goto** Step 3

Algorithm: Probable Prime Generation Algorithm v1

Input : A bit-length ℓ, the set $\mathcal{S} = \{s_0, \cdots, s_{52}\}$ of all odd primes lower than 256
Output: A probable prime p

```
   /* Generate a seed                                              */
1  Randomly generate an odd ℓ-bit integer v_0
   /* Prime Sieve                                                  */
2  v ← v_0
3  s ← s_0
4  i = 0
5  while (v mod s ≠ 0) and (i < 53) do
6      i = i + 1
7      s ← s_i
8  if (i ≠ 53) then
9      v = v + 2
10     goto Step 3
   /* Probabilistic primality tests                                */
11 else
12     i = 0
       /* Process t Miller-Rabin's tests (stop if one fails)       */
13     while (Miller-Rabin(v) = ok) and (i < t) do
14         i = i + 1
   /* Process one Lucas' test                                      */
15 if (i = t) and (Lucas(v) = ok) then
16     return v
17 else
18     v = v + 2
19     goto Step 3
```

Algorithm: Probable Prime Generation Algorithm v1

Input : A bit-length ℓ, the set $\mathcal{S} = \{s_0, \cdots, s_{52}\}$ of all odd primes lower than 256
Output: A probable prime p

```
   /* Generate a seed                                                  */
1  Randomly generate an odd ℓ-bit integer v_0
   /* Prime Sieve                                                      */
2  v ← v_0
3  s ← s_0
4  i = 0
5  while (v mod s ≠ 0) and (i < 53) do
6  |   i = i + 1
7  |_  s ← s_i
8  if (i ≠ 53) then
9  |   v = v + 2
10 |_  goto Step 3
   /* Probabilistic primality tests                                    */
11 else
12 |   i = 0
   |   /* Process t Miller-Rabin's tests (stop if one fails)           */
13 |   while (Miller-Rabin(v) = ok) and (i < t) do
14 |   |_  i = i + 1
   |_
   /* Process one Lucas' test                                          */
15 if (i = t) and (Lucas(v) = ok) then
16 |_  return v
17 else
18 |   v = v + 2
19 |_  goto Step 3
```

Algorithm: Probable Prime Generation Algorithm v1

Input : A bit-length ℓ, the set $\mathcal{S} = \{s_0, \cdots, s_{52}\}$ of all odd primes lower than 256

Output: A probable prime p

```
/* Generate a seed                                                   */
1  Randomly generate an odd ℓ-bit integer v_0
/* Prime Sieve                                                       */
2  v ← v_0
3  s ← s_0
4  i = 0
5  while (v mod s ≠ 0) and (i < 53) do
6      i = i + 1
7      s ← s_i
8  if (i ≠ 53) then
9      v = v + 2
10     goto Step 3
/* Probabilistic primality tests                                     */
11 else
12     i = 0
       /* Process t Miller-Rabin's tests (stop if one fails)         */
13     while (Miller-Rabin(v) = ok) and (i < t) do
14         i = i + 1
/* Process one Lucas' test                                           */
15 if (i = t) and (Lucas(v) = ok) then
16     return v
17 else
18     v = v + 2
19     goto Step 3
```

Algorithm: Probable Prime Generation Algorithm v1

Input : A bit-length ℓ, the set $\mathcal{S} = \{s_0, \cdots, s_{52}\}$ of all odd primes lower than 256
Output: A probable prime p

```
    /* Generate a seed                                                  */
1   Randomly generate an odd ℓ-bit integer v0
    /* Prime Sieve                                                      */
2   v ← v0
3   s ← s0
4   i = 0
5   while (v mod s ≠ 0) and (i < 53) do
6       i = i + 1
7       s ← s_i
8   if (i ≠ 53) then
9       v = v + 2
10      goto Step 3
    /* Probabilistic primality tests                                    */
11  else
12      i = 0
        /* Process t Miller-Rabin's tests (stop if one fails)           */
13      while (Miller-Rabin(v) = ok) and (i < t) do
14          i = i + 1
    /* Process one Lucas' test                                          */
15  if (i = t) and (Lucas(v) = ok) then
16      return v
17  else
18      v = v + 2
19      goto Step 3
```

Attack on Probable Prime Generation Algorithm v1

- Attack of [Finke+09]:
  - Each prime sieve execution ends as soon as $v \bmod s_i = 0$
  - Each prime sieve execution leaks through SPA
  - Allows to construct equation system with v_0 as unknown:

$$\left.\begin{array}{ccc} v_0 & \bmod & s_{i_0} = 0 \\ v_0 + 2 & \bmod & s_{i_1} = 0 \\ \vdots & & \\ v_0 + k \times 2 & \bmod & s_{i_k} = 0 \end{array}\right\} \Longleftrightarrow v_0 = x \bmod s_{i_0} \times s_{i_1} \times \ldots \times s_{i_k} \quad (1)$$

  - Chinese Reminder Theorem allows to deduce equation (1)
    $\Rightarrow v_0 \bmod s_{i_0} \times s_{i_1} \times \ldots \times s_{i_k}$
    $\Rightarrow p \bmod s_{i_0} \times s_{i_1} \times \ldots \times s_{i_k}$
  - Coppersmith technique $\Rightarrow p$

Algorithm: Probable Prime Generation Algorithm v2

Input : A bit-length ℓ, the set $\mathcal{S} = \{s_0, \cdots, s_{52}\}$ of all odd primes lower than 256
Output: A probable prime p

/* Generate a seed */
1 Randomly generate an odd ℓ-bit integer v_0

/* Costly Prime Sieve for v_0 */
2 **for** $j = 0$ **to** 52 **do**
3     $R[j] \leftarrow v_0 \mod s_j$ /* costly modular reduction over ℓ-bit integers */
4

/* Efficient Prime Sieve for v_i with $i > 0$ */
5 $v \leftarrow v_0$
6 **while** (R contains a null remainder) **do**
7     $v = v + 2$
8   **for** $j = 0$ **to** 52 **do**
9         $R[j] \leftarrow R[j] + 2 \mod s_j$ /* efficient modular reduction over 8-bit integers */
10

/* Probabilistic primality tests */
11 $i = 0$
/* Process t Miller-Rabin's tests (stop if one fails) */
12 **while** (Miller-Rabin(v) = ok) and ($i < t$) **do**
13     $i = i + 1$

/* Process one Lucas' test */
14 **if** ($i = t$) and (Lucas(v) = ok) **then**
15   **return** v
16 **else**
17     $v = v + 2$
18   **goto** Step 6

Algorithm: Probable Prime Generation Algorithm v2

Input : A bit-length ℓ, the set $\mathcal{S} = \{s_0, \cdots, s_{52}\}$ of all odd primes lower than 256
Output: A probable prime p

/* Generate a seed */
1 Randomly generate an odd ℓ-bit integer v_0

/* Costly Prime Sieve for v_0 */
2 **for** $j = 0$ **to** 52 **do**
3     $R[j] \leftarrow v_0 \mod s_j$ /* costly modular reduction over ℓ-bit integers */
4

/* Efficient Prime Sieve for v_i with $i > 0$ */
5 $v \leftarrow v_0$
6 **while** (R contains a null remainder) **do**
7     $v = v + 2$
8   **for** $j = 0$ **to** 52 **do**
9         $R[j] \leftarrow R[j] + 2 \mod s_j$ /* efficient modular reduction over 8-bit integers */
10

/* Probabilistic primality tests */
11 $i = 0$
/* Process t Miller-Rabin's tests (stop if one fails) */
12 **while** (Miller-Rabin(v) = ok) and ($i < t$) **do**
13     $i = i + 1$

/* Process one Lucas' test */
14 **if** ($i = t$) and (Lucas(v) = ok) **then**
15   **return** v

16 **else**
17     $v = v + 2$
18   **goto** Step 6

Algorithm: Probable Prime Generation Algorithm v2

Input : A bit-length ℓ, the set $\mathcal{S} = \{s_0, \cdots, s_{52}\}$ of all odd primes lower than 256
Output: A probable prime p

/* Generate a seed */
1 Randomly generate an odd ℓ-bit integer v_0

/* Costly Prime Sieve for v_0 */
2 **for** $j = 0$ **to** 52 **do**
3     $R[j] \leftarrow v_0 \mod s_j$ /* costly modular reduction over ℓ-bit integers */
4

/* Efficient Prime Sieve for v_i with $i > 0$ */
5 $v \leftarrow v_0$
6 **while** (R contains a null remainder) **do**
7     $v = v + 2$
8   **for** $j = 0$ **to** 52 **do**
9         $R[j] \leftarrow R[j] + 2 \mod s_j$ /* efficient modular reduction over 8-bit integers */
10

/* Probabilistic primality tests */
11 $i = 0$
/* Process t Miller-Rabin's tests (stop if one fails) */
12 **while** (Miller-Rabin(v) = ok) and ($i < t$) **do**
13     $i = i + 1$

/* Process one Lucas' test */
14 **if** ($i = t$) and (Lucas(v) = ok) **then**
15   **return** v
16 **else**
17     $v = v + 2$
18   **goto** Step 6

Algorithm: Probable Prime Generation Algorithm v2

Input : A bit-length ℓ, the set $\mathcal{S} = \{s_0, \cdots, s_{52}\}$ of all odd primes lower than 256
Output: A probable prime p

```
   /* Generate a seed                                                        */
1  Randomly generate an odd ℓ-bit integer v_0

   /* Costly Prime Sieve for v_0                                             */
2  for j = 0 to 52 do
3  |   R[j] ← v_0  mod s_j        /* costly modular reduction over ℓ-bit integers */
4  |_

   /* Efficient Prime Sieve for v_i with i > 0                               */
5  v ← v_0
6  while (R contains a null remainder) do
7  |   v = v + 2
8  |   for j = 0 to 52 do
9  |   |   R[j] ← R[j] + 2  mod s_j   /* efficient modular reduction over 8-bit integers */
10 |   |_
   |_

   /* Probabilistic primality tests                                          */
11 i = 0
   /* Process t Miller-Rabin's tests (stop if one fails)                     */
12 while (Miller-Rabin(v) = ok) and (i < t) do
13 |_  i = i + 1

   /* Process one Lucas' test                                                */
14 if (i = t) and (Lucas(v) = ok) then
15 |_  return v

16 else
17 |   v = v + 2
18 |_  goto Step 6
```

Algorithm: Probable Prime Generation Algorithm v2

Input : A bit-length ℓ, the set $\mathcal{S} = \{s_0, \cdots, s_{52}\}$ of all odd primes lower than 256
Output: A probable prime p

/* Generate a seed */
1 Randomly generate an odd ℓ-bit integer v_0

/* Costly Prime Sieve for v_0 */
2 **for** $j = 0$ **to** 52 **do**
3     $R[j] \leftarrow v_0 \mod s_j$ /* costly modular reduction over ℓ-bit integers */
4

/* Efficient Prime Sieve for v_i with $i > 0$ */
5 $v \leftarrow v_0$
6 **while** (R contains a null remainder) **do**
7     $v = v + 2$
8   **for** $j = 0$ **to** 52 **do**
9         $R[j] \leftarrow R[j] + 2 \mod s_j$ /* efficient modular reduction over 8-bit integers */
10

/* Probabilistic primality tests */
11 $i = 0$
/* Process t Miller-Rabin's tests (stop if one fails) */
12 **while** (Miller-Rabin(v) = ok) and ($i < t$) **do**
13     $i = i + 1$

/* Process one Lucas' test */
14 **if** ($i = t$) and (Lucas(v) = ok) **then**
15   **return** v
16 **else**
17     $v = v + 2$
18   **goto** Step 6

Algorithm: Probable Prime Generation Algorithm v2

Input : A bit-length ℓ, the set $\mathcal{S} = \{s_0, \cdots, s_{52}\}$ of all odd primes lower than 256
Output: A probable prime p

/* Generate a seed */
1 Randomly generate an odd ℓ-bit integer v_0

/* Costly Prime Sieve for v_0 */
2 **for** $j = 0$ **to** 52 **do**
3     $R[j] \leftarrow v_0 \mod s_j$ /* costly modular reduction over ℓ-bit integers */
4

/* Efficient Prime Sieve for v_i with $i > 0$ */
5 $v \leftarrow v_0$
6 **while** (R contains a null remainder) **do**
7     $v = v + 2$
8   **for** $j = 0$ **to** 52 **do**
9         $R[j] \leftarrow R[j] + 2 \mod s_j$ /* efficient modular reduction over 8-bit integers */
10

/* Probabilistic primality tests */
11 $i = 0$
/* Process t Miller-Rabin's tests (stop if one fails) */
12 **while** (Miller-Rabin(v) = ok) and ($i < t$) **do**
13     $i = i + 1$

/* Process one Lucas' test */
14 **if** ($i = t$) and (Lucas(v) = ok) **then**
15   **return** v

16 **else**
17     $v = v + 2$
18   **goto** Step 6

Probable Prime Generation Algorithm v2

- Prime sieve of algorithm v2 is regular
- Attack of [Finke+09] becomes ineffective
- Algorithm v2 is more efficient than algorithm v1
- Algorithm v2 recommended in:
  - ANSI X9.31
  - FIPS 186-4

Outline

Attack on Probable Prime Generation Algorithm v2

- Attacker records side-channels of following computations:
  (each line corresponds to a prime sieve execution)

$$\begin{cases} r_{0,0} = v_0 \bmod 3 & r_{0,1} = v_0 \bmod 5 & \dots & r_{0,52} = v_0 \bmod 251 \\ r_{1,0} = v_1 \bmod 3 & r_{1,1} = v_1 \bmod 5 & \dots & r_{1,52} = v_1 \bmod 251 \\ \vdots & \vdots & & \vdots \\ r_{n,0} = v_n \bmod 3 & r_{n,1} = v_n \bmod 5 & \dots & r_{n,52} = v_n \bmod 251 \end{cases}$$

Attack on Probable Prime Generation Algorithm v2

- As $v_i = v_0 + i \times 2$, one gets:

$$\begin{cases} r_{0,0} = v_0 \bmod 3 & r_{0,1} = v_0 \bmod 5 & \dots & r_{0,52} = v_0 \bmod 251 \\ r_{1,0} = v_0 + 2 \bmod 3 & r_{1,1} = v_0 + 2 \bmod 5 & \dots & r_{1,52} = v_0 + 2 \bmod 251 \\ \vdots & \vdots & & \vdots \\ r_{n,0} = v_0 + n \times 2 \bmod 3 & r_{n,1} = v_0 + n \times 2 \bmod 5 & \dots & r_{n,52} = v_0 + n \times 2 \bmod 251 \end{cases}$$

Attack on Probable Prime Generation Algorithm v2

- As n can be guessed by SPA, the attacker can then perform partial DPA for each small prime number:

$$\begin{cases} r_{0,0} = v_0 \bmod 3 & r_{0,1} = v_0 \bmod 5 & \dots & r_{0,52} = v_0 \bmod 251 \\ r_{1,0} = v_0 + 2 \bmod 3 & r_{1,1} = v_0 + 2 \bmod 5 & \dots & r_{1,52} = v_0 + 2 \bmod 251 \\ \vdots & \vdots & & \vdots \\ r_{n,0} = v_0 + n \times 2 \bmod 3 & r_{n,1} = v_0 + n \times 2 \bmod 5 & \dots & r_{n,52} = v_0 + n \times 2 \bmod 251 \end{cases}$$

$\Rightarrow$ allows to get $v_0 \bmod 3$

Attack on Probable Prime Generation Algorithm v2

- As n can be guessed by SPA, the attacker can then perform partial DPA for each small prime number:

$$\left\{\begin{array}{cccc} r_{0,0} = v_0 \bmod 3 & r_{0,1} = v_0 \bmod 5 & \ldots & r_{0,52} = v_0 \bmod 251 \\ r_{1,0} = v_0 + 2 \bmod 3 & r_{1,1} = v_0 + 2 \bmod 5 & \ldots & r_{1,52} = v_0 + 2 \bmod 251 \\ \vdots & \vdots & & \vdots \\ r_{n,0} = v_0 + n \times 2 \bmod 3 & r_{n,1} = v_0 + n \times 2 \bmod 5 & \ldots & r_{n,52} = v_0 + n \times 2 \bmod 251 \end{array}\right.$$

$\Rightarrow$ allows to get $v_0 \bmod 5$

Attack on Probable Prime Generation Algorithm v2

- As n can be guessed by SPA, the attacker can then perform partial DPA for each small prime number:

$$
\begin{cases}
r_{0,0} = v_0 \bmod 3 & r_{0,1} = v_0 \bmod 5 & \dots & r_{0,52} = v_0 \bmod 251 \\
r_{1,0} = v_0 + 2 \bmod 3 & r_{1,1} = v_0 + 2 \bmod 5 & \dots & r_{1,52} = v_0 + 2 \bmod 251 \\
\vdots & \vdots & & \vdots \\
r_{n,0} = v_0 + n \times 2 \bmod 3 & r_{n,1} = v_0 + n \times 2 \bmod 5 & \dots & r_{n,52} = v_0 + n \times 2 \bmod 251
\end{cases}
$$

$\Rightarrow \dots$

Attack on Probable Prime Generation Algorithm v2

- As n can be guessed by SPA, the attacker can then perform partial DPA for each small prime number:

$$
\begin{cases}
r_{0,0} = v_0 \bmod 3 & r_{0,1} = v_0 \bmod 5 & \dots & r_{0,52} = v_0 \bmod 251 \\
r_{1,0} = v_0 + 2 \bmod 3 & r_{1,1} = v_0 + 2 \bmod 5 & \dots & r_{1,52} = v_0 + 2 \bmod 251 \\
\vdots & \vdots & & \vdots \\
r_{n,0} = v_0 + n \times 2 \bmod 3 & r_{n,1} = v_0 + n \times 2 \bmod 5 & \dots & r_{n,52} = v_0 + n \times 2 \bmod 251
\end{cases}
$$

$\Rightarrow$ allows to get $v_0 \bmod 251$

Attack on Probable Prime Generation Algorithm v2

- Similarly to [Finke+09], one constructs an equation system with v_0 as unknown:

$$\left.\begin{array}{c} v_0 \mod 3 \\ v_0 \mod 5 \\ \vdots \\ v_0 \mod 251 \end{array}\right\} \iff v_0 = x \bmod 3 \times 5 \times \ldots \times 251 \quad (2)$$

- Chinese Reminder Theorem allows to deduce equation (2)
$\Rightarrow v_0 \bmod 3 \times 5 \times \ldots \times 251$
$\Rightarrow p \bmod 3 \times 5 \times \ldots \times 251$

- Coppersmith technique $\Rightarrow p$

Attack Analysis

- Attack success depends on number n of prime sieve executions
- Unlike classical SCA, n cannot be chosen by attacker
- In the sequel, we focus on 512-bit case
- When all the 53 partial DPA succeed, one gets roughly 350 bits of p
- If at least 256 consecutive bits of p are retrived, Coppersmith technique can allow to get the others

Attack Analysis

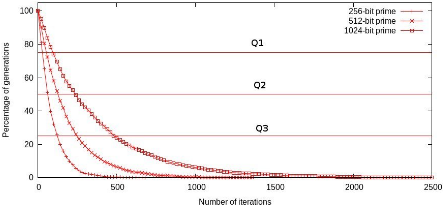

Figure : Cumulative distrib. fct. of n for diff. prime bit-lengths ℓ

- 512-bit prime number generation imply at least:
  (estimations over 2000 generations)
  - 53 prime sieve executions in 75% of the cases (Q_1)
  - 126 prime sieve executions in 50% of the cases (Q_2)
  - 246 prime sieve executions in 25% of the cases (Q_3)

Attack Analysis

σ	Q_1	Q_2	Q_3
0	1	1	1
1	1	1	1
2	0.46	1	1
3	0	0.99	1
4	0	0.08	1
5	0	0	0.7

Figure : Success rates for different noise levels to recover 256 bits of p depending on the number of prime sieve executions

Toy Implementation

- 8-bit ATMega128 micro-controller at 8MHz
- Implementation of 300 prime sieve executions from a random seed v_0
- EM measurements with sampling rate at 1GSa/s
- Partial DPA performed with Pearson correlation as distinguisher
- Experiment repeated 200 times

Attacking the Toy Implementation

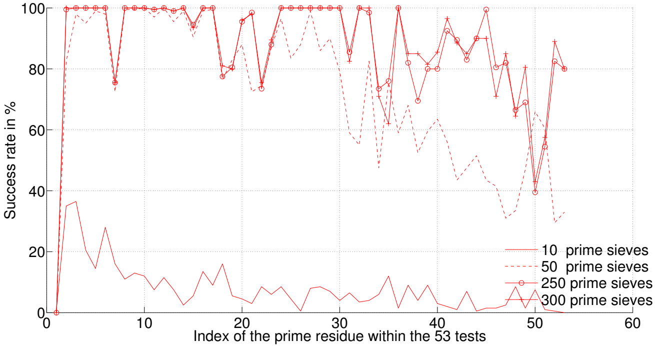

Figure : Success rates for each prime sieve elements

Attacking the Toy Implementation

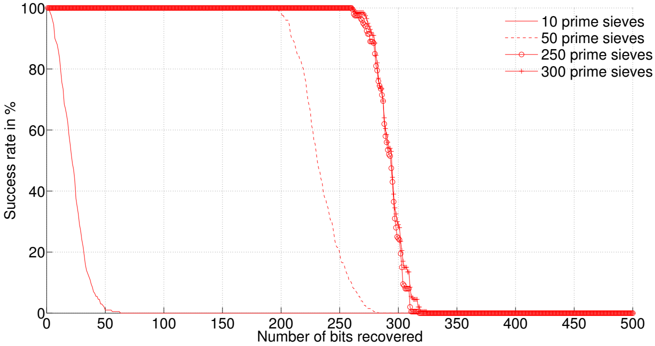

Figure : Success rates for recovering x bits of information on the generated prime

Improving the Attack Success

- Unsuccessful partial DSCA can be discarded thanks to Key Enumeration Algorithm
- The attacker can attack both p and q generations and use the RSA public modulus n to increase the success of the attack
- The initial costly prime sieve can also be used to get more information on p

Practical Issues

- Record long side-channel trace corr. to full prime generation
  - use high-end oscilloscope w. huge memory depth
  - use several cascaded oscilloscopes
- Find patterns corr. to n prime sieve executions
  - located between patterns corr. to Miller-Rabin tests
  - once one is found, use pattern matching techniques
- Find sub-patterns corr. to trial divisions
  - use classical peak extraction techniques used in SCA

Attack Flow in Practice

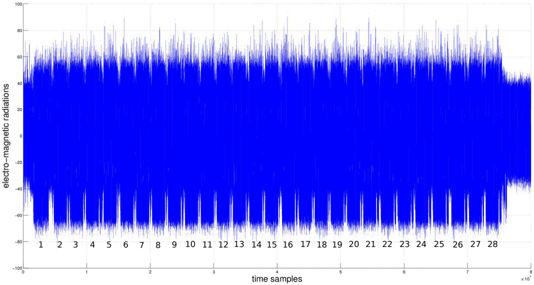

Figure : EM radiations measured during a prime number generation computation on a commercial smartcard

- Pattern 1 ⇒ initial costly prime sieve
- Patterns 2 to 28 ⇒ Miller-Rabin tests

Attack Flow in Practice

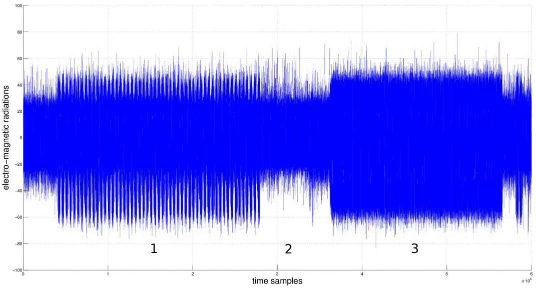

Figure : Zoom on the two first patterns of previous figure

- Pattern 1 ⇒ initial costly prime sieve
- Pattern 2 ⇒ efficient prime sieve executions
- Pattern 3 ⇒ first Miller-Rabin test

Outline

Possible Countermeasures

- Our attack exploits two features:
  - use of a prime sieve
  - deterministic candidate generation
- Approaches to thwart our attack:
  - Add randomly dummy trial divisions in each prime sieve computation
  - Perform prime sieve computation in pseudo-random order
  - Prime generation w. non-deterministic generation $\Rightarrow$ [Fouque+11]
  - Efficient provable prime generation algorithm $\Rightarrow$ [Clavier+12]

Thanks for your attention !

Questions ?