Practical realisation and elimination of an ECC-related software bug attack

B.B. Brumley, M. Barbosa, D. Page and F. Vercauteren

Department of Information and Computer Science,
Aalto University School of Science, P.O. Box 15400, FI-00076 Aalto, Finland.
billy.brumley@aalto.fi

HASLab/INESC TEC
Universidade do Minho, Braga, Portugal.
mbb@di.uminho.pt

Department of Computer Science, University of Bristol,
Merchant Venturers Building, Woodland Road, Bristol, BS8 1UB, UK.
page@cs.bris.ac.uk

Department of Electrical Engineering, Katholieke Universiteit Leuven,
Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium.
fvercaut@esat.kuleuven.ac.be

CT-RSA 29/02/12

Overview

- **Motivation**:

Quote

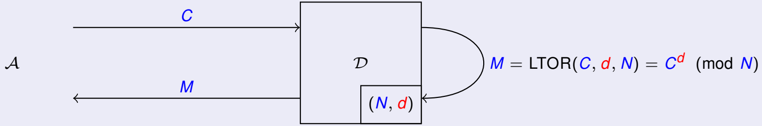
Decrypting ciphertexts on any computer which multiplies even one pair of numbers incorrectly can lead to full leakage of the secret key, sometimes with a single well-chosen ciphertext.

– Biham et. al. [2, Page 1]

- **Contribution**:
  1. an attack of this type on OpenSSL 0.9.8*g*, and
  2. an investigation of methods to detect and prevent such attacks.

Background: "bug attacks" (1)

Example: RSA bug attack



- **Rules**:
  - The attacker $\mathcal{A}$ wants to recover the private exponent d housed in a target device $\mathcal{D}$.
  - $\mathcal{D}$ uses a $(w \times w)$-bit integer multiplier whose operands are x and y.
  - Although generalisations are possible, assume that if
    1. $x \neq \alpha$ or $y \neq \beta$ their product is computed correctly, but
    2. $x = \alpha$ and $y = \beta$ their product is computed incorrectly.

Background: “bug attacks” (2)

Algorithm (LTOR)

Input: Integers x and y, and a modulus N.
Output: The result $x^y \pmod{N}$.

$t \leftarrow 1$
for $i = |y| - 1$ **downto** 1 **step** -1 **do**
1 $\quad t \leftarrow t^2 \pmod{N}$
2 $\quad$ **if** $y_i = 1$ **then**
3 $\quad\quad t \leftarrow t \cdot x \pmod{N}$
$\quad$ **end**
end
return t

Attack (Biham et. al. [2, Section 4.2])

At the j-th step, the attacker

- knows d', some more-significant portion of the binary expansion of d, and
- aims to recover the next less-significant unknown bit

so proceeds as follows:

1. Using d', select a C st. during decryption using LTOR, when $i = j$ at line #2
   - β occurs in the representation of x,
   - α occurs in the representation of t

   meaning that if
   - $y_i = 1$ then t is then multiplied by x and the bug is triggered,
   - $y_i = 0$ then t is then squared and the bug is not triggered.
2. Have the device decrypt C using d; if the result
   - is incorrect then the bug was triggered and hence $d_j = 1$,
   - is correct then the bug wasn’t triggered and hence $d_j = 0$.

Feature #1: NIST-P-{256, 384} implementation (1)

Quote

The function `BN_nist_mod_384` *(in* `crypto/bn/bn_nist.c`*) gives wrong results for some inputs.*

– Reimann [4], on the `openssl-dev` *mailing list*

Feature $\#1$: NIST-P-$\{256, 384\}$ implementation (2)

Algorithm (NIST-P-256-REDUCE, per Solinas [5, Example 3, Page 20])

Input: For $w = 32$-bit words, a 16-word integer product $z = x \cdot y$ and the modulus $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$.

Output: The result $z \pmod{p}$.

1. Form the nine, 8-word intermediate variables

$$
\begin{array}{lcllllllllll}
S_0 & = & \langle & z_0, & z_1, & z_2, & z_3, & z_4, & z_5, & z_6, & z_7 & \rangle \\
S_1 & = & \langle & 0, & 0, & 0, & z_{11}, & z_{12}, & z_{13}, & z_{14}, & z_{15} & \rangle \\
S_2 & = & \langle & 0, & 0, & 0, & z_{12}, & z_{13}, & z_{14}, & z_{15}, & 0 & \rangle \\
S_3 & = & \langle & z_8, & z_9, & z_{10}, & 0, & 0, & 0, & z_{14}, & z_{15} & \rangle \\
S_4 & = & \langle & z_9, & z_{10}, & z_{11}, & z_{13}, & z_{14}, & z_{15}, & z_{13}, & z_8 & \rangle \\
S_5 & = & \langle & z_{11}, & z_{12}, & z_{13}, & 0, & 0, & 0, & z_8, & z_{10} & \rangle \\
S_6 & = & \langle & z_{12}, & z_{13}, & z_{14}, & z_{15}, & 0, & 0, & z_9, & z_{11} & \rangle \\
S_7 & = & \langle & z_{13}, & z_{14}, & z_{15}, & z_8, & z_9, & z_{10}, & 0, & z_{12} & \rangle \\
S_8 & = & \langle & z_{14}, & z_{15}, & 0, & z_9, & z_{10}, & z_{11}, & 0, & z_{13} & \rangle
\end{array}
$$

2. Compute

$$r = S_0 + 2S_1 + 2S_2 + S_3 + S_4 - S_5 - S_6 - S_7 - S_8 \pmod{p}.$$

3. Return $0 \leq r < p$.

Feature #1: NIST-P-{256, 384} implementation (3)

Algorithm (NIST-P-256-REDUCE, per OpenSSL)

Input: For $w = 32$-bit words, a 16-word integer product $z = x \cdot y$ and the modulus $p = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$.

Output: The (potentially incorrect) result $z \pmod{p}$.

1. Form the nine, 8-word intermediate variables

$$
\begin{array}{lclcccccccc}
S_0 & = & \langle & z_0, & z_1, & z_2, & z_3, & z_4, & z_5, & z_6, & z_7 & \rangle \\
S_1 & = & \langle & 0, & 0, & 0, & z_{11}, & z_{12}, & z_{13}, & z_{14}, & z_{15} & \rangle \\
S_2 & = & \langle & 0, & 0, & 0, & z_{12}, & z_{13}, & z_{14}, & z_{15}, & 0 & \rangle \\
S_3 & = & \langle & z_8, & z_9, & z_{10}, & 0, & 0, & 0, & z_{14}, & z_{15} & \rangle \\
S_4 & = & \langle & z_9, & z_{10}, & z_{11}, & z_{13}, & z_{14}, & z_{15}, & z_{13}, & z_8 & \rangle \\
S_5 & = & \langle & z_{11}, & z_{12}, & z_{13}, & 0, & 0, & 0, & z_8, & z_{10} & \rangle \\
S_6 & = & \langle & z_{12}, & z_{13}, & z_{14}, & z_{15}, & 0, & 0, & z_9, & z_{11} & \rangle \\
S_7 & = & \langle & z_{13}, & z_{14}, & z_{15}, & z_8, & z_9, & z_{10}, & 0, & z_{12} & \rangle \\
S_8 & = & \langle & z_{14}, & z_{15}, & 0, & z_9, & z_{10}, & z_{11}, & 0, & z_{13} & \rangle
\end{array}
$$

2. Compute

$$
\begin{array}{lcl}
S & = & S_0 + 2S_1 + 2S_2 + S_3 + S_4 - S_5 - S_6 - S_7 - S_8 \\
 & = & t + c \cdot 2^{256}
\end{array}
$$

3. Compute

$$
\begin{array}{lcll}
r & = & t - c \cdot p & \pmod{2^{256}} \\
 & = & t - \mathrm{sign}(c) \cdot T[|c|] & \pmod{2^{256}}
\end{array}
$$

   for pre-computed $T[i] = i \cdot p$.

4. If $r \geq p$ (resp. $r < 0$) then update $r \leftarrow r - p$ (resp. $r \leftarrow r + p$), return r.

Feature #1: NIST-P-{256, 384} implementation (4)

- Some (limited) analysis: incorrect result (i.e., $\pm 2^{256}$)
  1. is triggered randomly with probability $\sim 10 \cdot 2^{-29}$,
  2. can be triggered deliberately with special-form operands, e.g.,

$$\begin{array}{lllllll} x & = & (2^{32}-1)\cdot 2^{224} & + & 3\cdot 2^{128} & + & x_0 \\ y & = & (2^{32}-1)\cdot 2^{224} & + & 1\cdot 2^{96} & + & y_0 \end{array}$$

for any random $0 \leq x_0, y_0 < 2^{32}$.

Feature #2: ECDHE implementation (1)

Algorithm (ephemeral ECDH between $\mathcal{A}$ and $\mathcal{D}$)

$\mathcal{A}$		$\mathcal{D}$
$k^i_{\mathcal{A}} \xleftarrow{\$} \{1, 2, \ldots, n-1\}$		$k^i_{\mathcal{D}} \xleftarrow{\$} \{1, 2, \ldots, n-1\}$
$Q^i_{\mathcal{A}} \leftarrow [k^i_{\mathcal{A}}]G$		$Q^i_{\mathcal{D}} \leftarrow [k^i_{\mathcal{D}}]G$
	$\xrightarrow{\quad Q^i_{\mathcal{A}} \quad}$	
	$\xleftarrow{\quad Q^i_{\mathcal{D}} \quad}$	
$R^i_{\mathcal{A}} \leftarrow [k^i_{\mathcal{A}}]Q^i_{\mathcal{D}} = [k^i_{\mathcal{A}} \cdot k^i_{\mathcal{D}}]G$		$R^i_{\mathcal{D}} \leftarrow [k^i_{\mathcal{D}}]Q^i_{\mathcal{A}} = [k^i_{\mathcal{D}} \cdot k^i_{\mathcal{A}}]G$

Feature #2: ECDHE implementation (1)

Algorithm (ephemeral-static ECDH between $\mathcal{A}$ and $\mathcal{D}$)

$\mathcal{A}$		$\mathcal{D}$
$k^i_{\mathcal{A}} \xleftarrow{\$} \{1, 2, \ldots, n-1\}$		$k_{\mathcal{D}} \xleftarrow{\$} \{1, 2, \ldots, n-1\}$
$Q^i_{\mathcal{A}} \leftarrow [k^i_{\mathcal{A}}]G$		$Q_{\mathcal{D}} \leftarrow [k_{\mathcal{D}}]G$
	$\xrightarrow{Q^i_{\mathcal{A}}}$	
	$\xleftarrow{Q_{\mathcal{D}}}$	
$R^i_{\mathcal{A}} \leftarrow [k^i_{\mathcal{A}}]Q_{\mathcal{D}} = [k^i_{\mathcal{A}} \cdot k_{\mathcal{D}}]G$		$R^i_{\mathcal{D}} \leftarrow [k_{\mathcal{D}}]Q^i_{\mathcal{A}} = [k_{\mathcal{D}} \cdot k^i_{\mathcal{A}}]G$

Feature #2: ECDHE implementation (2)

- OpenSSL implements this as follows

ssl/s3_lib.c

```
                if (!(s->options & SSL_OP_SINGLE_ECDH_USE))
                        {
                        if (!EC_KEY_generate_key(ecdh))
                                {
                                EC_KEY_free(ecdh);
                                SSLerr(SSL_F_SSL3_CTRL,ERR_R_ECDH_LIB);
                                return(ret);
                                }
                        }
```

meaning ECDHE

- uses a per-invocation (of the library) rather than a per-session key, unless
- one explicitly uses `SSL_CTX_set_options` to set `SSL_OP_SINGLE_ECDH_USE`.

Attack (1)

Feature	Biham et. al. [2, Section 4.2]	Brumley et. al. [3, Section 3]
Target	Fixed d	Fixed $k_{\mathcal{D}}$ (ECDH or ephemeral-static ECDHE)
Leakage	Re-encrypt M using e, check against C	Handshake success/failure
Input	Arbitrary poisoned integer $C \in \mathbb{Z}_N^*$	Controlled distinguisher point $Q_{\mathcal{A}}^i = [k_{\mathcal{A}}^i]G \in E(\mathbb{F}_p)$
Computation	Left-to-right binary exponentiation	Left-to-right (modified) wNAF scalar multiplication

Attack (2)

Attack (Brumley et. al. [3, Section 3])

At the j-th step, the attacker

- knows a, some more-significant portion of the wNAF expansion of $k_{\mathcal{D}}$, and
- aims to recover the next less-significant unknown non-zero digit $b \in S$ for some digit set S

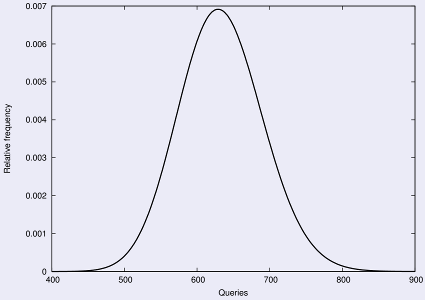
so proceeds as follows:

1. Select a distinguisher point
$$D_{a,b} = [l]G$$
for known l, st. for (enough) random paddings d
$$[a \parallel b \parallel d]D_{a,b} \notin E(\mathbb{F}_p)$$
for all $b \in S$, and
$$[a \parallel c \parallel d]D_{a,b} \in E(\mathbb{F}_p)$$
for all $c \in S \setminus \{0, b\}$.
2. Use each distinguisher point as an input to $\mathcal{D}$: if the handshake fails, that guess for b was correct.
3. Apply wNAF rules to cope with any subsequent zero digits.

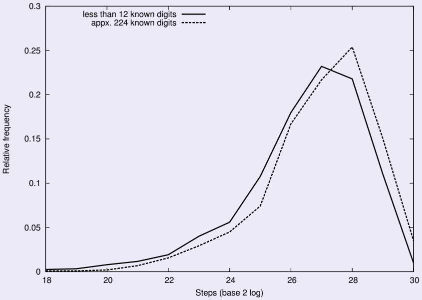
Attack (3)

- Cost: for a prototype $\mathcal{D}$ based on `s_server` ...

Queries to $\mathcal{D}$ by $\mathcal{A}$



Effort by $\mathcal{A}$ to find $D_{a,b}$



- ... when NIST-P-256 is used, $\mathcal{A}$
  - can recover the fixed $k_{\mathcal{D}}$ using $\sim$ 633 queries to $\mathcal{D}$, where
  - each query implies a $\sim 2^{27}$ step brute-force distinguisher point search (assuming no pre-computation).

Conclusions (1)

- Reactive countermeasures:
  1. The bug in NIST-P-256-REDUCE is *already* patched in OpenSSL 0.9.8*h* and higher.
  2. Restarting the library to refresh $k_{\mathcal{D}}$ limits impact ...
  3. ... but you may as well just opt-out of ephemeral-static ECDHE instead!
  4. Point or scalar blinding, or a randomised scalar multiplication algorithm prevent selection of suitable distinguisher points.
- Proactive countermeasures (or, "second half of paper"): given
  1. testing doesn't seem robust enough, and
  2. there seems to be a connection between performance-enhancing optimisations and security

  how can we make formal verification (e.g., of OpenSSL) technically and economically viable?

Conclusions (2)

Questions?

References and Further Reading

[1] I. Biehl, B. Meyer, and V. Müller.
Differential fault attacks on elliptic curve cryptosystems.
In *Advances in Cryptology (CRYPTO)*, volume 1880 of *LNCS*, pages 131–146. Springer-Verlag, 2000.

[2] E. Biham, Y. Carmeli, and A. Shamir.
Bug attacks.
In *Advances in Cryptology (CRYPTO)*, volume 5157 of *LNCS*, pages 221–240. Springer-Verlag, 2008.

[3] B. Brumley, M. Barbosa, D. Page, and F. Vercauteren.
Practical realisation and elimination of an ECC-related software bug attack.
In *Topics in Cryptology (CT-RSA)*, 2012.

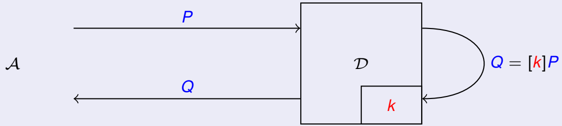
[4] H. Reimann.
`BN_nist_mod_384` gives wrong answers.
`openssl-dev` mailing list #1593, 2007.
Available from `http://marc.info/?t=119271238800004`.

References and Further Reading (cont.)

[5] J.A. Solinas.
Generalized mersenne numbers.
Technical Report CORR 99-39, Centre for Applied Cryptographic Research (CACR), University of Waterloo, 1999.

Extra – Invalid Curve Attack (1)

Example: ECC invalid curve attack



Attack (Biehl et. al. [1, Section 4.1])

1. Given a curve E' of order $|E'| = \prod r_i$, for each i:
   1.1 Select a point $P_i \in E'$ with order r_i.
   1.2 Send $P_i \in E'$ to $\mathcal{D}$ and have it compute $Q_i = [k]P_i \in E'$.
   1.3 Solve ECDLP in subgroup to get $k \pmod{r_i}$.
2. Use CRT to recover k given all $k \pmod{r_i}$.

Extra – Invalid Curve Attack (2)

- Observation: if $\mathcal{D}$ uses OpenSSL, it will validate each input $P = (x_P, y_P)$ by comparing the LHS and RHS of

$$y_P^2 = x_P^3 + a_4 x_P + a_6$$

  and hence prevent an invalid curve attack.
- Idea: select point $P = (x_P, y_P)$ as follows,
  1. Select x_P such that during the computation of $t = (x_P^2 + a_4) \cdot x_P + a_6 \pmod{p}$:
     - The step $t_0 = x_P^2 \pmod{p}$ *does not* trigger the bug.
     - The step $t_1 = (t_0 + a_4) \cdot x_P \pmod{p}$ *does* trigger the bug, i.e., the correct result would be $t_1 \pm 2^{256} \pmod{p}$.
     - The incorrect result t is a quadratic residue modulo p.
  2. Compute $y_P = \sqrt{t} \pmod{p}$.

  meaning P now passes the OpenSSL point validation, but is actually on some curve E' rather than E.

Extra – Invalid Curve Attack (3)

- (Open) problem:
  - The characteristics of the bug mean it produces results that are incorrect by $\pm 2^{256}$.
  - This limits the invalid curves to

$$
\begin{array}{rcl}
E'_{+256} & : & y^2 = x^3 + a_4 x + (a_6 + 2^{256}) \\
E'_{-256} & : & y^2 = x^3 + a_4 x + (a_6 - 2^{256})
\end{array}
$$

$$
\begin{array}{rcl}
|E'_{+256}| & = & FFFFFFFF00000000FFFFFFFFFFFFFFFF\backslash \\
 & & DA0A4439003A5730FA6F898036B17E90_{(16)} \\
 & \approx & 2^4 \cdot 2^{11} \cdot 2^{31} \cdot 2^{209}
\end{array}
$$

$$
\begin{array}{rcl}
|E'_{-256}| & = & FFFFFFFF00000001000000000000000\backslash \\
 & & 304C2CB870EB2102DEB81758D8933A44_{(16)} \\
 & \approx & 2^2 \cdot 2^{11} \cdot 2^{14} \cdot 2^{16} \cdot 2^{57} \cdot 2^{154}
\end{array}
$$

  and hence also the P_i.
  - Even so, the 128-bit security level of NIST-P-256 is reduced to that of E'_{-256}.

A First-Order Leak-Free Masking Countermeasure

Houssem MAGHREBI, Emmanuel PROUFF,
Sylvain GUILLEY, Jean-Luc DANGER
< houssem.maghrebi@TELECOM-ParisTech.fr >

Institut TELECOM / TELECOM-ParisTech
CNRS – LTCI (UMR 5141)

RSA CONFERENCE'12, San Francisco
Session Track: Cryptography **Session Code**: CRYP-204
Scheduled Date: 02/29/2012 **Session Title**: Secure Implementation Methods **Session Classification**: Advanced

Presentation Outline

Masking: principle

- Aims at making power consumption random
- The sensitive variable Z is randomly split into two shares:

$$(M_1 \ , M_0 \ = \ Z \ \theta \ M_1 \ )$$

  M_0 is the masked variable and θ is an inversible operation
- Boolean masking is based on exclusive-or (xor) operations:

$$M_0 \ = \ Z \ \oplus \ M_1$$

- The application of a transformation S on a variable Z split in two shares leads the processing of two new shares M_0' and M_1' such that:

$$S(Z) \ = \ M_0' \ \oplus \ M_1'$$

- The critical point is to deduce M_0' from M_0, M_1 and M_1'

Linear Function

- $S(Z) = S(M_0 \oplus M_1) = S(M_0) \oplus S(M_1)$
- $M'_0 = S(M_0) \oplus S(M_1) \oplus M'_1$

Non-Linear Function (NL)

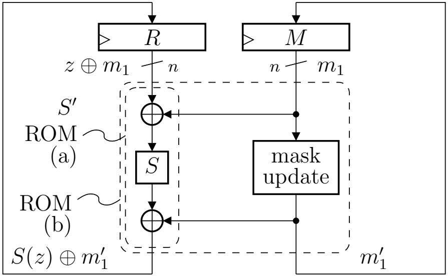
- Achieving first-order security is much more difficult
- Commonly, there are three strategies:
  - (a) *Global Look-up Table*: a precomputed ROM is associated to the function $S' : (X, Y, Y') \mapsto S(X \oplus Y)$. M'_0 is computed by performing a single operation: $S'[Z \oplus M_1, M_1, M'_1]$
  - (b) *The re-computation method*: M_1 and M'_1 are generated and a table representing the function $S' : Y \mapsto S(Y \oplus M_1) \oplus M'_1$ is computed from S and stored in RAM
  - (c) *The sbox secure calculation*: the sbox outputs are computed *on-the-fly* by using a mathematical representation of the sbox
- The GLUT method seems to be the most appropriate method

Generic Structure

The ROM lookup-table represents a $(3n, n)$-function S' such that:
$S'(Z \oplus M_1, M_1, M_1') = S(Z) \oplus M_1'$

Security Evaluation

It manipulates the masked data $Z \oplus M_1$ and the mask M_1 at the same time (*i.e.* potentially exploitable)



Assumption: Only the updating of the registers leak information

- The masked data register leakage is: $L_R = A(Z \oplus M_1, Z' \oplus M_1') + N_R$
- The mask register leakage is: $L_M = A(M_1, M_1') + N_M$

Property #1: For any pair (X, Y), we have $A(X, Y) = \mathcal{A}(X \oplus Y)$

- The power consumption L related to the simultaneous updating of the registers equals $L_R + L_M$: $L = \mathcal{A}(\Delta(Z) \oplus \Delta(M)) + \mathcal{A}(\Delta(M)) + N_R + N_M$, where $\Delta(Z)$ and $\Delta(M)$ respectively denote $Z \oplus Z'$ and $M_1 \oplus M_1'$
- The distribution of L (and in particular its variance) depends on the sensitive variable $\Delta(Z)$

How to break the dependency between L and $\Delta(Z)$?

- A simple solution is to choose a function $@$ such that:

$$Z \; @ \; M_1 = Z \; \oplus \; F(M_1)$$

- M_1 and Z do no longer need to have the same dimension n, so F is a (p, n)-function
- The deterministic part of the leakage can be rewritten:

$$\begin{aligned} & A(Z@M_1, Z'@M_1') + A(M_1, M_1') \\ \doteq \;& \mathcal{A}(Z \oplus Z' \oplus F(M_1) \oplus F(M_1')) + \mathcal{A}(M_1 \oplus M_1') \\ = \;& \mathcal{A}(\Delta(Z) \oplus F(M_1) \oplus F(M_1')) + \mathcal{A}(\Delta(M_1)) \end{aligned}$$

Necessary Conditions to be Satisfied

L is independent of $\Delta(Z)$ if:

1. **[Constant Masks Difference]**: $M_1 \oplus M_1'$ is constant and
2. **[Difference Uniformity]**: $F(M_1) \oplus F(M_1')$ is uniform

Presentation Outline

One Simple Solution

- Fix the condition $M_1' = M_1 \oplus \alpha$ for some nonzero constant α
- Design F *s.t.* $Y \mapsto F(Y) \oplus F(Y \oplus \alpha)$ is uniform for this α

First Construction Proposal

- Choose $p = n + 1$ and split $\mathbb{F}_2^{n+1}$ into $E \oplus (E \oplus \alpha)$
- Choose a bijective function G from E into $\mathbb{F}_2^n$
- Define F such that for every $Y \in \mathbb{F}_2^{n+1}$, we have $F(Y) = G(Y)$ if $Y \in E$ and $F(Y) = 0$ otherwise

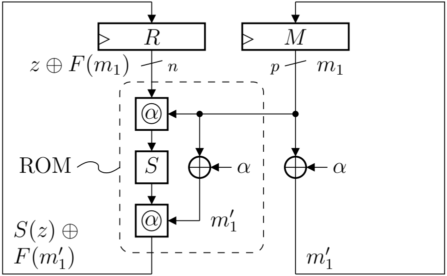
Example for $n = 3$: $E = \{0\} \times \mathbb{F}_2^n \subset \mathbb{F}_2^{n+1}$ and the constant α is equal to 1000 in binary, and $F(x_3x_2x_1x_0) = 0$ if $x_3 = 1$ or $x_2x_1x_0$ otherwise.

Second Construction Proposal

- Choose $p = n + n'$ with $n' < n$ and select one injective function G from $\mathbb{F}_2^{n'}$ into $\mathbb{F}_2^n - \{0\}$
- For every $(X, Y) \in \mathbb{F}_{2^{n'}} \times \mathbb{F}_{2^n} = \mathbb{F}_{2^p}$ $F(X, Y) = G(X) \cdot Y$
- The outputs of the (p, n)-function F are uniformly distributed over $\mathbb{F}_2^n$

- The two constructions of F satisfy the *difference uniformity* condition
- The mask dimension p for the first construction is only slightly greater than the dimension n of the data to be masked

Hardware Implementation



- The registers contain $Z \oplus F(M_1)$ and M_1
- The mask update operation is constrained to be a $\oplus$ with α
- Every computation is protected with the single pair of masks $(M_1,\ M'_1 = M_1 \oplus \alpha)$
- $S(Z) \oplus F(M'_1)$ is got by accessing the ROM table

Evaluation Methodology

- *The target implementation*: the proposed countermeasure
- *The target secret*: the sensitive variable $\Delta(Z)$
- *The Adversary model*: the non-adaptive known plaintext model, the attacker is not able to perform HO-SCA
- *The Leakage model*: the Hamming distance model

Mutual Information Analysis

$$\mathrm{I}[\mathcal{A}(\Delta(Z) \oplus F(M_1) \oplus F(M_1'))+\mathcal{A}(\Delta(M)); \Delta(Z)] = 0$$

$$(\textit{perfect masking of register } R \Longrightarrow \mathrm{I}[L_R; \Delta(Z)] = 0)$$

- $\Delta(M)$ is constant and $F(M_1) \oplus F(M_1')$ is uniformly distributed over $\mathbb{F}_2^n$ and independent of $\Delta(Z)$
- Our proposal is *leak-free* and immune against first-order attacks

Evaluation Methodology

- *The target implementation*: the proposed countermeasure
- *The target secret*: the sensitive variable $\Delta(Z)$
- *The Adversary model*: the non-adaptive known plaintext model, the attacker is not able to perform HO-SCA
- *The Leakage model*: the Hamming distance model

Mutual Information Analysis

$$\mathrm{I}[\mathcal{A}(\Delta(Z) \oplus F(M_1) \oplus F(M_1')) + \mathcal{A}(\Delta(M)); \Delta(Z)] = 0$$
$$(\textit{hiding of register } M \Longrightarrow \mathrm{I}[L_M; \Delta(Z)] = 0)$$

- $\Delta(M)$ is constant and $F(M_1) \oplus F(M_1')$ is uniformly distributed over $\mathbb{F}_2^n$ and independent of $\Delta(Z)$
- Our proposal is *leak-free* and immune against first-order attacks

Evaluation Methodology

- *The target implementation*: the proposed countermeasure
- *The target secret*: the sensitive variable $\Delta(Z)$
- *The Adversary model*: the non-adaptive known plaintext model, the attacker is not able to perform HO-SCA
- *The Leakage model*: the Hamming distance model

Mutual Information Analysis

$$\mathrm{I}[\mathcal{A}(\Delta(Z) \oplus F(M_1) \oplus F(M_1')) + \mathcal{A}(\Delta(M)); \Delta(Z)] = 0$$
$$(\textit{first-order resistance} \Longrightarrow \mathrm{I}[L_R + L_M; \Delta(Z)] = 0)$$

- $\Delta(M)$ is constant and $F(M_1) \oplus F(M_1')$ is uniformly distributed over $\mathbb{F}_2^n$ and independent of $\Delta(Z)$
- Our proposal is *leak-free* and immune against first-order attacks

Evaluation Methodology

- *The target implementation*: the proposed countermeasure
- *The target secret*: the sensitive variable $\Delta(Z)$
- *The Adversary model*: the non-adaptive known plaintext model, the attacker is not able to perform HO-SCA
- *The Leakage model*: the Hamming distance model

Mutual Information Analysis

$$\mathrm{I}[\mathcal{A}(\Delta(Z) \oplus F(M_1) \oplus F(M_1')),\ \mathcal{A}(\Delta(M)); \Delta(Z)] = 0$$
$$(\textit{second-order resistance} \Longrightarrow \mathrm{I}[L_R, L_M; \Delta(Z)] = 0)$$

- $\Delta(M)$ is constant and $F(M_1) \oplus F(M_1')$ is uniformly distributed over $\mathbb{F}_2^n$ and independent of $\Delta(Z)$
- Our proposal is *leak-free* and immune against first-order attacks *and certain second-order attacks!*

Context: Memory Access in Von-Neumann Architecture

```
mov dptr, #tab
mov acc, y
movc acc, @acc+dptr
```

- dptr: the data memory pointer
- #tab: the address of a table stored in data
- y: the index of the value that must be read in table tab
- The accumulator register acc contains the value tab[y]

Analogy

- #tab and y refer respectively to the ROM and $(Z@M_1, M'_1)$
- The most significant bits of acc is associated to the register R and its least significant bits to the register M
- Taking advantage from our proposal, the memory access is made completely secure

Presentation Outline

- In reality $A(X, Y)$ is a polynomial $P(X_1, \cdots, X_n, Y_1, \ldots, Y_n)$
- We study $\mathrm{I}[L_R + L_M; Z \oplus Z']$ when P is of degree $\leqslant d$

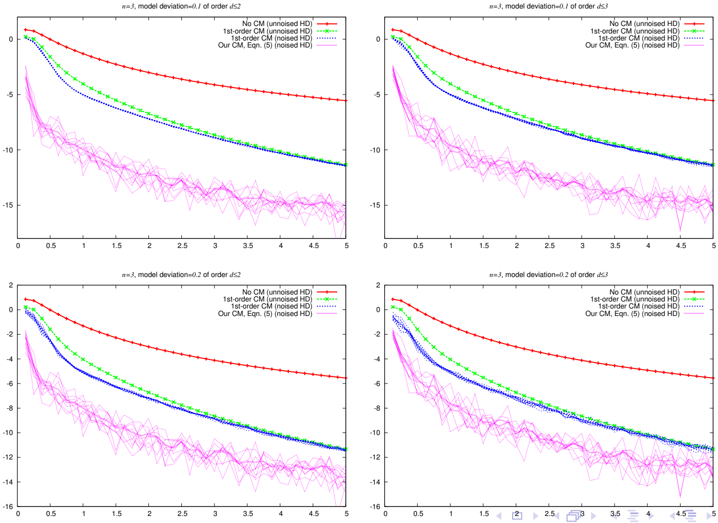
Methodology

- The leakage function is:
  $P(X_1, \cdots, X_n, Y_1, \cdots, Y_n) = \sum_{\substack{(u,v)\in\mathbb{F}_2^n\times\mathbb{F}_2^n, \\ \mathrm{HW}(u)+\mathrm{HW}(v)\leq d}} a_{(u,v)} X_1^{u_1} \cdots X_n^{u_n} Y_1^{v_1} \cdots Y_n^{v_n}$
- The coefficients $a_{(u,v)}$ are drawn at random from this law:

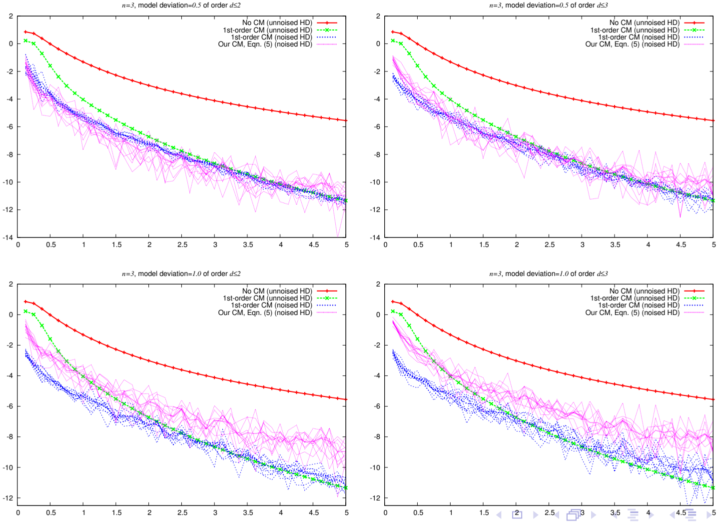
$$a_{(u,v)} \sim a^{\mathrm{HD}}_{(u,v)} + \mathcal{U}\left(\left[-\frac{\text{deviation}}{2}, +\frac{\text{deviation}}{2}\right]\right)$$
$$a_{(u,v)} = 0 \quad \text{if} \quad \mathrm{HW}(u,v) > d \ .$$

- The deviation is $\{0.1, 0.2, 0.5, 1.0\}$, *i.e.* 10%, 20%, 50% or 100%
- The computed mutual information is $\mathrm{I}[L; Z, Z']$, where $L = P(Z \oplus F(M), Z' \oplus F(M \oplus \alpha)) + N_R + P(M, M \oplus \alpha) + N_M$

Simulation Results for low deviation



Simulation Results for high deviation



Presentation Outline

Conclusions

- A new masking scheme for hardware sbox implementations is presented
- The countermeasure proposed is a leak-free countermeasure under some realistic assumptions about the device architecture
- The solution has been evaluated within an information-theoretic study, proving its security against 1O-SCA under the Hamming distance assumption
- When the leakage function deviates slightly from this assumption, our solution still achieves excellent results

Perspective

- Adapt the countermeasure to reach 2nd-order security

Thanks
For Your
Attention.

An up-to-date version of the paper (with some corrections in the construction of the F functions (in §4.1)) is on the eprint: [1].

References

[1] Houssem Maghrebi, Emmanuel Prouff, Sylvain Guilley, and Jean-Luc Danger.
A First-Order Leak-Free Masking Countermeasure.
Cryptology ePrint Archive, Report 2012/028, 2012.
http://eprint.iacr.org/2012/028.

A First-Order Leak-Free Masking Countermeasure

Houssem MAGHREBI, Emmanuel PROUFF,
Sylvain GUILLEY, Jean-Luc DANGER
<houssem.maghrebi@TELECOM-ParisTech.fr>

Institut TELECOM / TELECOM-ParisTech
CNRS – LTCI (UMR 5141)

RSA CONFERENCE'12, San Francisco
Session Track: Cryptography **Session Code**: CRYP-204
Scheduled Date: 02/29/2012 **Session Title**: Secure Implementation Methods **Session Classification**: Advanced