Formal Software Methods for Cryptosystems' Implementation Security

Pablo Rauzy
`rauzy@enst.fr`

`pablo.rauzy.name`

Telecom ParisTech
LTCI / COMELEC / SEN

December 4, 2013

Implementation Security

- Security of the physical implementations of cryptosystems.

There are two main categories of physical attacks:

- side-channel attacks, which are passive,
- fault injection attacks, which are active.

Side-Channel Attacks

A *side-channel attack* is any attack based on information gained from the physical implementation of a cryptosystem, rather than brute force or theoretical weaknesses in the algorithms.

Examples of side-channel information:

- timing,
- power consumption,
- electromagnetic leaks.

Fault Injection Attacks

A *fault injection attack* consists in modifying parameters or intermediate values of a cryptosystem's computation to make the final result of the computation leak sensitive information about the system, often by comparing the compromised result with a correct one (*differential fault attack*).

There are many form of fault injections:

- invasive / non-invasive,
- destructive / non-destructive,
- global / precise.

Formal Methods

- Security of implementation is a relatively new topic (about 15 years old).
- Formal study of the physical attacks and their countermeasures is still confidential.

Are Seldom Used…

- Big participation of the industry to the field of implementation security:
  - more engineering than research;
  - development of security by trial-and-error.
- Concrete, physical implementations appear to be too complex to formally study:
  - discrepancy between model and implementation;
  - existing formal analysis tools work with functional properties, not physical ones.

⇒ Thus, formal methods are seldom used in our field.

. . . But are a Necessity

- Cryptosystems' software should be bug-free and rely as little as possible on hand-written code for critical parts.
- Moreover, being able to prove the security enable (often much needed) speed-oriented and security-oriented optimizations.

⇒ We need tools to formally assess the security of implementations, and where possible automatically generate or insert countermeasures against physical attacks.

Implementation Security
  - Side-Channel Attacks
  - Fault Injection Attacks

Formal Methods
  - Are Seldom Used...
  - ... But are a Necessity

Formally Proved Security of Assembly Code Against Power Analysis
  - Power Analysis
  - Power Analysis Countermeasures
  - Dual-Rail with Precharge Logic (DPL)
  - Formally Proven DPL Countermeasure
  - Automatic Insertion of the DPL Countermeasure
  - Formally Proving the Absence of Leakage
  - Results and Contributions

Formal Proofs of CRT-RSA Countermeasures Against BellCoRe Attacks
  - RSA
  - CRT-RSA
  - The BellCoRe Attack
  - Countermeasures
  - Shortcomings
  - Formal Analysis
  - Results and Contributions

Perspectives

Formally Proved Security of Assembly Code Against Power Analysis

Power Analysis
Power Analysis Countermeasures
Dual-Rail with Precharge Logic (DPL)
Formally Proven DPL Countermeasure
Automatic Insertion of the DPL Countermeasure
    Generic Assembly Language
    Sensitive Instructions
    Code Transformation
    Correctness Proof of the Transformation
Formally Proving the Absence of Leakage
    The Attacker
    The Security Invariant
    Computed Proof of Constant Activity
    Hardware Characterization
Results and Contributions

Power Analysis

- A form of side-channel attack in which the attacker measures the *power consumption* of a cryptographic device.
- *Simple Power Analysis* (SPA).
- *Differential Power Analysis* [KJJ99] (DPA).
- Power consumption is often modeled by Hamming weight of values or Hamming distance of values' updates as it is very correlated with actual measures.

Power Analysis Countermeasures

- Thwarting side-channel analysis is complicated since an unprotected implementation leaks at every step.
- Serious power analysis countermeasures can be classified in two categories:
  - Those that use randomness to make the leakage statistically independent from sensitive data (*masking*).
  - Those that make the leakage indistinguishable (*balancing*).
- Automated masking has already been explored but most efforts have yet to be done for balancing.
- Randomness is a strong requirement and is hard to capture formally, thus we chose to work with a balancing countermeasure, namely *dual-rail with precharge logic* (DPL).

Dual-Rail with Precharge Logic (DPL)

- The DPL countermeasure consists in computing on a redundant representation: each bit b is implemented as a pair $(y_{\text{False}}, y_{\text{True}})$.
- The bit pair is then used in a protocol made up of two phases:
  1. a *precharge* phase, during which all the bit pairs are zeroized $(y_{\text{False}}, y_{\text{True}}) = (0, 0)$, such that the computation starts from a known reference state;
  2. an *evaluation* phase, during which the pair $(y_{\text{False}}, y_{\text{True}})$ is equal to $(1, 0)$ if it carries the logical value 0, or $(0, 1)$ if it carries the logical value 1.

⇒ Two physical resources which have the same leakage properties have to exist.

Formally Proven DPL Countermeasure

- The semantics of the code must not be altered by the transformation that adds the countermeasure (correctness).
- The countermeasure must be efficient (security).

⇒ We need formal models of the possible side-channel leakages, and then use them to prove that the obtained code is protected against those leakages.

Automatic Insertion of the DPL Countermeasure

- We want to be able to transform any assembly code to make it respect the DPL protocol.
- We want to prove that the transformation is correct.

Generic Assembly Language

- We need to be able to manipulate any assembly code. For that we designed a generalist assembly that our tools manipulate.
- It is generalist enough for us to be able to easily map instructions from most assembly one-to-one and back.
- Instructions follow this pattern:

```
opcode destination operand1 operand2
```

Sensitive Instructions

Sensitive value

A value is said *sensitive* if it depends on sensitive data. A sensitive data depends on both the secret key and the cleartext (as usually admitted in the "*only computation leaks*" paradigm; see for instance [RP10, §4.1]).

Sensitive instruction

A *sensitive instruction* is an instruction which may modify the Hamming weight of a sensitive value.

Code Transformation

- Bitslice code (in practice, use a bitsliced implementation).
- Expand sensitive instructions to DPL macro.
- Transform all sensitive data into their DPL encoded counterparts.

$$\begin{array}{lcl}
r_1 & \leftarrow & r_0 \\
r_1 & \leftarrow & a \\
r_1 & \leftarrow & r_1 \wedge 3 \\
r_1 & \leftarrow & r_1 \ll 1 \\
r_1 & \leftarrow & r_1 \ll 1 \\
r_2 & \leftarrow & r_0 \\
r_2 & \leftarrow & b \\
r_2 & \leftarrow & r_2 \wedge 3 \\
r_1 & \leftarrow & r_1 \vee r_2 \\
r_3 & \leftarrow & r_0 \\
r_3 & \leftarrow & op[r_1] \\
d & \leftarrow & r_0 \\
d & \leftarrow & r_3
\end{array}$$

DPL macro for $d = a \mathrm{\ op\ } b$.

Correctness Proof of the Transformation

Correct DPL transformation

Let S be a valid state of the system (values in registers and memory). Let c be a sequence of instructions of the system. Let $\widehat{S}$ be the state of the system after the execution of c with state S, we denote that by $S \xrightarrow{c} \widehat{S}$.
We write $dpl(S)$ for the DPL state (with DPL encoded values of the 1s and 0s in memory and registers) equivalent to the state S.
We say that c' is a *correct DPL transformation* of the code c if $S \xrightarrow{c} \widehat{S} \implies dpl(S) \xrightarrow{c'} dpl(\widehat{S})$.

- Proof for each instruction by exhaustive case enumeration that its macro expansion is a correct DPL transformation;
- Proof by induction that any sequence of correct DPL transformations is a correct DPL transformation.

Formally Proving the Absence of Leakage

- We want to prove a security property on the code resulting from the transformation.
- We need to show that the formal proof on the software can be relevant on a concrete physical implementation.

The Attacker

The attacker can measure the power consumption of parts of the cryptosystem.

Leakage model

We model power consumption by the Hamming distance of values updates, *i.e.*, the number of bit flips. It is a commonly accepted model for power analysis, for instance with DPA [KJJ99] or CPA [BCO04]. We write $H(a, b)$ the Hamming distance between the values a and b.

The Security Invariant

The activity of a cryptosystem is said to be constant if its power consumption does not depend on the sensitive data and is thus always the same.

Constant activity

Formally, let $P(s)$ be a program which has s as parameter (*e.g.*, the key and the cleartext). According to our leakage model, a program $P(s)$ is of *constant activity* if:

- for every values s_1 and s_2 of the parameter s, for each cycle i, for every sensitive value v, v is updated at cycle i in the run of $P(s_1)$ if and only if it is in the run of $P(s_2)$;
- whenever an instruction modifies some sensitive value from v to v', then the value of $H(v, v')$ does not depend on s.

Computed Proof of Constant Activity

We want to statically determine if the code is correctly balanced.

- We use *symbolic execution*, to run the program independently of the sensitive data.
- We compute on sets of values instead of values directly, so we do not have to make hypothesis on the initial values of sensible data.
- Avoid combinatorial explosion thanks to bitslicing, as a value can initially be only 1 or 0 or both (or their DPL encoded counterparts).

- We implemented an interpreter for our generic assembly language.
- Our interpreter is equipped to measure all the possible Hamming distances of each value update.
- If for one of these value updates there are different possible Hamming distances, then we consider that there is a leak of information.
- Otherwise, the code is proven well-balanced.

Computed Proof of Constant Activity

We want to statically determine if the code is correctly balanced.

- We use *symbolic execution*, to run the program independently of the sensitive data.
- We compute on sets of values instead of values directly, so we do not have to make hypothesis on the initial values of sensible data.
- Avoid combinatorial explosion thanks to bitslicing, as a value can initially be only 1 or 0 or both (or their DPL encoded counterparts).

- We implemented an interpreter for our generic assembly language.
- Our interpreter is equipped to measure all the possible Hamming distances of each value update.
- If for one of these value updates there are different possible Hamming distances, then we consider that there is a leak of information.
- Otherwise, the code is proven well-balanced.

Hardware Characterization

- The DPL countermeasure relies on the fact that the pair of bits used to store the DPL encoded values leak the same way.
- This property is generally not true in non-specialized hardware.
- However, using the two closest bits (in term of leakage) for the DPL protocol still helps reaching a better immunity to power analysis attacks.
- Using stochastic profiling [SLP05], or monobit CPA attack to measure the Pearson correlation coefficient between the actual power consumption of the targeted bit and the logical Hamming distance of its updates, it is possible to find a pair of bits that have close leakage properties and that are at suitable positions for the DPL protocol.

Results and Contributions

- Design method to generate code provably protected against power analysis, including a tool to automatically insert the DPL countermeasure against power analysis, and a way to profile the hardware on which it will be run for customization of the countermeasure.
- A case study with a PRESENT encryption algorithm running on an AVR smartcard.
- A paper that will be submitted to CHES 2014.

Formal Proofs of CRT-RSA Countermeasures Against BellCoRe Attacks

- RSA
- CRT-RSA
- The BellCoRe Attack
  - How it works?
- Countermeasures
  - Shamir's Countermeasure
  - Shamir's Countermeasure / Algorithm
  - Aumüller *et al.*'s Countermeasure
  - Aumüller *et al.*'s Countermeasure / Algorithm
  - Vigilant's Countermeasure
  - Vigilant's Countermeasure / Algorithm
- Shortcomings
- Formal Analysis
  - CRT-RSA Computation
  - Fault Injection
  - Algorithm Description
  - finja
  - How finja Works?
  - Mathematical Framework
  - Testing Attacks
- Results and Contributions

RSA

RSA (*Rivest, Shamir, Adleman*)

RSA [RSA78] is an algorithm for public key cryptography. It can be used as both an encryption and a signature algorithm.

It works as follows (for simplicity we omit the padding operations):

- Let m be the message, (N, e) the public key, and (N, d) the private key such that $d \cdot e \equiv 1 \mod \varphi(N)$.
- The signature S is computed by $S \equiv m^d \mod N$.
- The signature can be verified by checking that $m \equiv S^e \mod N$.

RSA

RSA (*Rivest, Shamir, Adleman*)

RSA [RSA78] is an algorithm for public key cryptography. It can be used as both an encryption and a signature algorithm.

It works as follows (for simplicity we omit the padding operations):

- Let m be the message, (N, e) the public key, and (N, d) the private key such that $d \cdot e \equiv 1 \mod \varphi(N)$.
- The signature S is computed by $S \equiv m^d \mod N$.
- The signature can be verified by checking that $m \equiv S^e \mod N$.

CRT-RSA

CRT (*Chinese Remainder Theorem*)

CRT-RSA [Koç94] is an optimization of the RSA computation which allows a fourfold speedup.

It works as follows:

- Let p and q be the primes from the key generation ($N = p \cdot q$).
- These values are pre-computed (considered part of the private key):
  - $d_p \doteq d \mod (p-1)$
  - $d_q \doteq d \mod (q-1)$
  - $i_q \doteq q^{-1} \mod p$
- S is then computed as follows:
  - $S_p = m^{d_p} \mod p$
  - $S_q = m^{d_q} \mod q$
  - $S = S_q + q \cdot (i_q \cdot (S_p - S_q) \mod p)$

CRT (*Chinese Remainder Theorem*)

CRT-RSA [Koç94] is an optimization of the RSA computation which allows a fourfold speedup.

It works as follows:

- Let p and q be the primes from the key generation ($N = p \cdot q$).
- These values are pre-computed (considered part of the private key):
  - $d_p \doteq d \mod (p-1)$
  - $d_q \doteq d \mod (q-1)$
  - $i_q \doteq q^{-1} \mod p$
- S is then computed as follows:
  - $S_p = m^{d_p} \mod p$
  - $S_q = m^{d_q} \mod q$
  - $S = S_q + q \cdot (i_q \cdot (S_p - S_q) \mod p)$

The BellCoRe Attack

BellCoRe (*Bell Communications Research*)

The BellCoRe attack [BDL97] consists in revealing the secret primes p and q by faulting the computation. It is very powerful as it works even with very random faulting.

It works as follows:

- The intermediate variable S_p (resp. S_q) is faulted as $\widehat{S_p}$ (resp. $\widehat{S_q}$).
- The attacker thus gets an erroneous signature $\widehat{S}$.
- The attacker can recover p (resp. q) as $\gcd(N, S - \widehat{S})$.

The BellCoRe Attack

BellCoRe (*Bell Communications Research*)

The BellCoRe attack [BDL97] consists in revealing the secret primes p and q by faulting the computation. It is very powerful as it works even with very random faulting.

It works as follows:

- The intermediate variable S_p (resp. S_q) is faulted as $\widehat{S_p}$ (resp. $\widehat{S_q}$).
- The attacker thus gets an erroneous signature $\widehat{S}$.
- The attacker can recover p (resp. q) as $\gcd(N, S - \widehat{S})$.

How it works?

For all integer x, $\gcd(N, x)$ can only take 4 values:

- 1, if N and x are co-prime,
- p, if x is a multiple of p,
- q, if x is a multiple of q,
- N, if x is a multiple of both p and q, *i.e.*, of N.

How it works?

If S_p is faulted (*i.e.*, replaced by $\widehat{S_p} \neq S_p$):

- $S - \widehat{S} = q \cdot \left((i_q \cdot (S_p - S_q) \mod p) - (i_q \cdot (\widehat{S_p} - S_q) \mod p) \right)$

$\Rightarrow$ $\gcd(N, S - \widehat{S}) = q$

How it works?

If S_q is faulted (*i.e.*, replaced by $\widehat{S_q} \neq S_q$):

- $S - \widehat{S} \equiv (S_q - \widehat{S_q}) - (q \mod p) \cdot i_q \cdot (S_q - \widehat{S_q}) \equiv 0 \mod p$ (because $(q \mod p) \cdot i_q \equiv 1 \mod p$)

$\Rightarrow$ $\gcd(N, S - \widehat{S}) = p$

How it works?

If S_q is faulted (*i.e.*, replaced by $\widehat{S_q} \neq S_q$):

- $S - \widehat{S} \equiv (S_q - \widehat{S_q}) - (q \mod p) \cdot i_q \cdot (S_q - \widehat{S_q}) \equiv 0 \mod p$
  (because $(q \mod p) \cdot i_q \equiv 1 \mod p$)

$\Rightarrow$ $\gcd(N, S - \widehat{S}) = p$

Countermeasures

Several protections against the BellCoRe attacks have been proposed.

Some of them are given below:

- Obvious countermeasures: no CRT, or with signature verification;
- Shamir [Sha99];
- Aumüller *et al.* [ABF$^+$02];
- Vigilant, original [Vig08] and with some corrections by Coron *et al.* [CGM$^+$10];

Shamir's Countermeasure

- Introduces a small random number r, co-prime with p and q.
- Carries out computations modulo $p' = p \cdot r$ and $q' = q \cdot r$.

⇒ Allows retrieval of the results by reduction modulo p and modulo q.

⇒ Enables verification by reduction modulo r.

Shamir's Countermeasure / Algorithm

```
Input  : Message $m$, key $(p, q, d, i_q)$, 32-bit random prime $r$
Output: Signature $m^d \mod N$, or error if some fault injection is detected.
$p' = p \cdot r$
$d_p = d \mod (p-1) \cdot (r-1)$
$S'_p = m^{d_p} \mod p'$
$q' = q \cdot r$
$d_q = d \mod (q-1) \cdot (r-1)$
$S'_q = m^{d_q} \mod q'$
$S_p = S'_p \mod p$
$S_q = S'_q \mod q$
$S = S_q + q \cdot (i_q \cdot (S_p - S_q) \mod p)$
if $S'_p \not\equiv S'_q \mod r$ then
    return error
else
    return $S$
end
```

Aumüller *et al.*'s Countermeasure

- Variation of Shamir's countermeasure primarily intended to fix two shortcomings:
  - removes the need for d during the computation;
  - checks the CRT recombination step.
- Uses *asymmetrical* verification (computations modulo p' and q' operate on two different objects).
- Also adds some verifications of the intermediate computations.

Aumüller *et al.*'s Countermeasure / Algorithm

Input : Message m, key (p, q, d_p, d_q, i_q), 32-bit random prime t

Output : Signature $m^d \mod N$, or error if some fault injection is detected.

```
$p' = p \cdot t$
$d'_p = d_p + \mathrm{random}_1 \cdot (p - 1)$
$S'_p = m^{d'_p} \mod p'$
if $(p' \mod p \neq 0)$ or $(d'_p \not\equiv d_p \mod (p - 1))$ then
    return error
end
$q' = q \cdot t$
$d'_q = d_q + \mathrm{random}_2 \cdot (q - 1)$
$S'_q = m^{d'_q} \mod q'$
if $(q' \mod q \neq 0)$ or $(d'_q \not\equiv d_q \mod (q - 1))$ then
    return error
end
$S_p = S'_p \mod p$
$S_q = S'_q \mod q$
$S = S_q + q \cdot (i_q \cdot (S_p - S_q) \mod p)$
if $(S - S'_p \not\equiv 0 \mod p)$ or $(S - S'_q \not\equiv 0 \mod q)$ then
    return error
end
$S_{pt} = S'_p \mod t$
$S_{qt} = S'_q \mod t$
$d_{pt} = d'_p \mod (t - 1)$
$d_{qt} = d'_q \mod (t - 1)$
if $S_{pt}^{d_{qt}} \not\equiv S_{qt}^{d_{pt}} \mod t$
then
    return error
else
    return $S$
end
```

Vigilant's Countermeasure

- Different approach than Aumüller *et al.*'s one.
- All the CRT computation (even the recombination) is carried out in an overring of $\mathbb{Z}_{Nr^2}$ of $\mathbb{Z}_N$.
- The $\mathbb{Z}_{r^2}$ subring is used to make an additional check that uses the Binomial theorem.

- *"Formal proof of the FA-resistance of Vigilant's scheme including our countermeasures is still an open (and challenging) issue."*

Vigilant's Countermeasure / Algorithm

```
Input  : Message $M$, key $(p, q, d_p, d_q, i_q)$.
Output: Signature $M^d \mod N$.
Choose random numbers $r$, $R_1$, $R_2$, $R_3$, and $R_4$.
$p' = pr^2$
$M_p = M \mod p'$
$i_{pr} = p^{-1} \mod r^2$
$B_p = p \cdot i_{pr}$
$A_p = 1 - B_p \mod p'$
$M'_p = A_p M_p + B_p \cdot (1 + r) \mod p'$
if $M'_p \not\equiv M \mod p$ then
    return error
end
$d'_p = d_p + R_1 \cdot (p - 1)$
$S_{pr} = {M'_p}^{d'_p} \mod p'$
if $d'_p \not\equiv d_p \mod p - 1$ then
    return error
end
if $B_p S_{pr} \not\equiv B_p \cdot (1 + d'_p r) \mod p'$ then
    return error
end
$S'_p = S_{pr} - B_p \cdot (1 + d'_p r - R_3)$
$q' = qr^2$
$M_q = M \mod q'$
$i_{qr} = q^{-1} \mod r^2$
$B_q = q \cdot i_{qr}$
$A_q = 1 - B_q \mod q'$
$M'_q = A_q M_q + B_q \cdot (1 + r) \mod q'$
if $M'_q \not\equiv M \mod q$ then
    return error
end
if $M_p \not\equiv M_q \mod r^2$ then
    return error
end
$d'_q = dq + R_2 \cdot (q - 1)$
$S_{qr} = {M'_q}^{d'_q} \mod q'$
if $d'_q \not\equiv dq \mod q - 1$ then
    return error
end
if $B_q S_{qr} \not\equiv B_q \cdot (1 + d'_q r) \mod q'$ then
    return error
end
$S'_q = S_{qr} - B_q \cdot (1 + d'_q r - R_4)$
$S = S'_q + q \cdot (i_q \cdot (S'_p - S'_q) \mod p')$
$N = pq$
if $N \cdot (S - R_4 - q \cdot i_q \cdot (R_3 - R_4)) \not\equiv 0 \mod Nr^2$ then
    return error
end
if $q \cdot i_q \not\equiv 1 \mod p$ then
    return error
end
return $S \mod N$
```

Shortcomings

- All these countermeasures are hand crafted iteratively, by trial-and-error.
- No proof of their efficiency is given.

Formal Analysis

- The goal is to make sure countermeasures are trustable.
- We want to cover a very general attacker model.
- We want our proof to apply to any implementation that is a refinement of the abstract algorithm.

CRT-RSA Computation

- A CRT-RSA computation takes as input a message m, assumed known by the attacker, and a secret key (p, q, d_p, d_q, i_q).
- The implementation is free to instantiate any variable, but must return a result equal to: $S = S_q + q \cdot (i_q \cdot (S_p - S_q) \mod p)$, where:
  - $S_p = m^{d_p} \mod p$, and
  - $S_q = m^{d_q} \mod q$.

Fault Injection

- An attacker can request a CRT-RSA computation.
- During the computation, the attacker can fault any intermediate value.
- A faulted value can be zero or random.
- The attacker can read the final result of the computation.

- Faulting can occur in the global memory (*permanent fault*) or in a local register or bus (*transient fault*).
- The control flow graph is untouched.

Fault Injection

- An attacker can request a CRT-RSA computation.
- During the computation, the attacker can fault any intermediate value.
- A faulted value can be zero or random.
- The attacker can read the final result of the computation.

- Faulting can occur in the global memory (*permanent fault*) or in a local register or bus (*transient fault*).
- The control flow graph is untouched.

Algorithm Description

- Low level enough for the attack to work if protections are not implemented.
- Intermediate variable that would appear during refinement could be the target of an attack, but such a fault would propagate to an intermediate variable of the high level description.

finja

- Input:
  - A high level description of the computation, and
  - an attack success condition.
- Output:
  - Either the list of possible attacks, or
  - a proof that the computation is resistant to fault injections.

- Source code (including examples) is available at `http://pablo.rauzy.name/sensi/finja.html`.

How finja Works?

- The description of the computation is transformed into a *term*.
- The term is a tree which encodes:
    - dependencies between the intermediate values, and
    - properties of the intermediate values (such as being null, being null modulo another term, or being a multiple of another term).
- Each intermediate value (subterms of the tree) can be faulted, in such case its properties become:
    - nothing, in the case of a randomizing fault, or
    - being null, in the case of a zeroing fault.

How finja Works?

- The description of the computation is transformed into a *term*.
- The term is a tree which encodes:
    - dependencies between the intermediate values, and
    - properties of the intermediate values (such as being null, being null modulo another term, or being a multiple of another term).
- Each intermediate value (subterms of the tree) can be faulted, in such case its properties become:
    - nothing, in the case of a randomizing fault, or
    - being null, in the case of a zeroing fault.

Mathematical Framework

finja uses symbolic computation to simplify the term.

It uses the computed properties of the intermediate values and rules from:

- arithmetic in the $\mathbb{Z}$ ring;
- modular arithmetic in the $\mathbb{Z}/n\mathbb{Z}$ rings;
- plus a few theorems:
  - little Fermat's theorem;
  - its generalization, *i.e.*, Euler's theorem;
  - Chinese remainder theorem;
  - a special case of the Binomial theorem.

Testing Attacks

- Simplified faulted terms are then fed into the attack success condition.
- The attack success condition is then simplified to either true or false.

Results and Contributions

- finja
- We have a formal proof of the resistance of Aumüller *et al.*'s and Vigilant's countermeasures against the BellCoRe attack by fault injection on CRT-RSA.
- We also have simplified Vigilant's countermeasures.
- Three publications: PROOFS 2013 [RG13], JCEN, PPREW 2014.

Perspectives

Power analysis:

- Clean/rewrite and release tools.
- Use the same methods for other algorithms and other hardware.
- Automated bitslicing.
- Cache behavior model for timing attack.

Fault injection:

- Fault injections in the instructions [HMER13].
- Using EasyCrypt [BGZB09] and program synthesis to find countermeasures.
- High-order fault injections countermeasures.

References I

Christian Aumüller, Peter Bier, Wieland Fischer, Peter Hofreiter, and Jean-Pierre Seifert.
Fault Attacks on RSA with CRT: Concrete Results and Practical Countermeasures.
In Burton S. Kaliski, Jr., Çetin Kaya Koç, and Christof Paar, editors, *CHES*, volume 2523 of *Lecture Notes in Computer Science*, pages 260–275. Springer, 2002.

Éric Brier, Christophe Clavier, and Francis Olivier.
Correlation Power Analysis with a Leakage Model.
In *CHES*, volume 3156 of *LNCS*, pages 16–29. Springer, August 11–13 2004.
Cambridge, MA, USA.

Dan Boneh, Richard A. DeMillo, and Richard J. Lipton.
On the Importance of Checking Cryptographic Protocols for Faults.
In *Proceedings of Eurocrypt'97*, volume 1233 of *LNCS*, pages 37–51. Springer, May 11-15 1997.
Konstanz, Germany. DOI: 10.1007/3-540-69053-0_4.

Gilles Barthe, Benjamin Grégoire, and Santiago Zanella-Béguelin.
Formal certification of code-based cryptographic proofs.
In *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*, pages 90–101.
ACM, 2009.

Jean-Sébastien Coron, Christophe Giraud, Nicolas Morin, Gilles Piret, and David Vigilant.
Fault Attacks and Countermeasures on Vigilant's RSA-CRT Algorithm.
In Luca Breveglieri, Marc Joye, Israel Koren, David Naccache, and Ingrid Verbauwhede, editors, *FDTC*, pages 89–96.
IEEE Computer Society, 2010.

Karine Heydemann, Nicolas Moro, Emmanuelle Encrenaz, and Bruno Robisson.
Formal Verification of a Software Countermeasure Against Instruction Skip Attacks.
Cryptology ePrint Archive, Report 2013/679, 2013.
`http://eprint.iacr.org/`.

References II

Paul C. Kocher, Joshua Jaffe, and Benjamin Jun.
Differential Power Analysis.
In *Proceedings of CRYPTO'99*, volume 1666 of *LNCS*, pages 388–397. Springer-Verlag, 1999.

Sung-Kyoung Kim, Tae Hyun Kim, Dong-Guk Han, and Seokhie Hong.
An efficient CRT-RSA algorithm secure against power and fault attacks.
J. Syst. Softw., 84:1660–1669, October 2011.

Çetin Kaya Koç.
High-Speed RSA Implementation, November 1994.
Version 2, `ftp://ftp.rsasecurity.com/pub/pdfs/tr201.pdf`.

Pablo Rauzy and Sylvain Guilley.
A Formal Proof of Countermeasures Against Fault Injection Attacks on CRT-RSA.
Cryptology ePrint Archive, Report 2013/506, 2013.
`http://eprint.iacr.org/`.

Matthieu Rivain and Emmanuel Prouff.
Provably Secure Higher-Order Masking of AES.
In Stefan Mangard and François-Xavier Standaert, editors, *CHES*, volume 6225 of *LNCS*, pages 413–427. Springer, 2010.

Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman.
A Method for Obtaining Digital Signatures and Public-Key Cryptosystems.
Commun. ACM, 21(2):120–126, 1978.

Adi Shamir.
Method and apparatus for protecting public key schemes from timing and fault attacks, November 1999.
US Patent Number 5,991,415; also presented at the rump session of EUROCRYPT '97.

References III

Werner Schindler, Kerstin Lemke, and Christof Paar.
A Stochastic Model for Differential Side Channel Cryptanalysis.
In LNCS, editor, *CHES*, volume 3659 of *LNCS*, pages 30–46. Springer, Sept 2005.
Edinburgh, Scotland, UK.

David Vigilant.
RSA with CRT: A New Cost-Effective Solution to Thwart Fault Attacks.
In Elisabeth Oswald and Pankaj Rohatgi, editors, *CHES*, volume 5154 of *Lecture Notes in Computer Science*, pages 130–145. Springer, 2008.

That's it. Questions?

- Implementation Security
  - Side-Channel Attacks
  - Fault Injection Attacks
- Formal Methods
  - Are Seldom Used...
  - ... But are a Necessity
- Formally Proved Security of Assembly Code Against Power Analysis
  - Power Analysis
  - Power Analysis Countermeasures
  - Dual-Rail with Precharge Logic (DPL)
  - Formally Proven DPL Countermeasure
  - Automatic Insertion of the DPL Countermeasure
  - Formally Proving the Absence of Leakage
  - Results and Contributions
- Formal Proofs of CRT-RSA Countermeasures Against BellCoRe Attacks
  - RSA
  - CRT-RSA
  - The BellCoRe Attack
  - Countermeasures
  - Shortcomings
  - Formal Analysis
  - Results and Contributions
- Perspectives

`rauzy@enst.fr`