A Formal Proof of Countermeasures against Fault Injection Attacks on CRT-RSA

Pablo Rauzy
rauzy@enst.fr
pablo.rauzy.name

Sylvain Guilley
sylvain.guilley@enst.fr
perso.enst.fr/~guilley

Telecom ParisTech
LTCI / COMELEC / SEN

August 24, 2013 — 9h45–10h15
PROOFS 2013 @ Santa Barbara

IACR ePrint 2013/506

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 (*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 (*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:
  - $S_p = m^{d_p} \mod p$
  - $S_q = m^{d_q} \mod q$
  - $S = S_q + q \cdot (i_q \cdot (S_p - S_q) \mod p)$

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 $\gcd(N, S - \widehat{S})$.

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 $\gcd(N, S - \widehat{S})$.

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 $\widehat{S_p} \neq S_p$):

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

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

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 $(q \mod p) \cdot i_q \equiv 1 \mod p$)

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

BellCoRe attack refinement

This attack has been improved [JLQ99] so it only needs the faulty signature to recover p or q, by computing $\gcd(N, m - \widehat{S}^e)$.

- If S_p if faulted, then *most likely* $\gcd(N, S - \widehat{S}) = q$,
- which means that we have $S \not\equiv \widehat{S} \mod p$
  thus, $S^e \not\equiv \widehat{S}^e \mod p$;
- and that we also have $S \equiv \widehat{S} \mod q$
  thus, $S^e \equiv \widehat{S}^e \mod q$.

$\Rightarrow$ As $S^e \equiv m \mod N$, this proves the result.

A symmetrical reasoning can be done if the fault occurs during the computation of S_q.

BellCoRe attack refinement

This attack has been improved [JLQ99] so it only needs the faulty signature to recover p or q, by computing $\gcd(N, m - \widehat{S}^e)$.

- If S_p if faulted, then *most likely* $\gcd(N, S - \widehat{S}) = q$,
- which means that we have $S \not\equiv \widehat{S} \mod p$
  thus, $S^e \not\equiv \widehat{S}^e \mod p$;
- and that we also have $S \equiv \widehat{S} \mod q$
  thus, $S^e \equiv \widehat{S}^e \mod q$.

$\Rightarrow$ As $S^e \equiv m \mod N$, this proves the result.

A symmetrical reasoning can be done if the fault occurs during the computation of S_q.

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];
- Kim *et al.* [KKHH11].

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];
- Kim *et al.* [KKHH11].

Shamir Countermeasure

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

$\Rightarrow$ Allows retrieval of the results by reduction modulo p and modulo q.

$\Rightarrow$ Enables verification by reduction modulo r.

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

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.* Countermeasure

- Variation of Shamir 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.

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.

$p' = p \cdot t$

$d'_p = d_p + \text{random}_1 \cdot (p-1)$

$S'_p = m^{d'_p} \mod p'$

if $(p' \mod p \neq 0)$ **or** $(d'_p \not\equiv d_p \mod (p-1))$ **then**

  return error

end

$q' = q \cdot t$

$d'_q = d_q + \text{random}_2 \cdot (q-1)$

$S'_q = m^{d'_q} \mod q'$

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

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

- The goal is making 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.

- 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 injection.

How it 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 it 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

The simplification is done by a recursive traversal of the term tree.

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 such as little Fermat's theorem and its generalization, *i.e.*, Euler's theorem.

Testing Attacks

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

Already released!

- Source code (including examples) is already available at `http://pablo.rauzy.name/sensi/finja.html`.
- I still need to write a user manual (I will do that Real Soon Now$^{\text{TM}}$).

Minimal Example of Usage

- Computation: $t = a + b \times c$.
- Let's say the "attack" works if $t \not\equiv a \mod b$.
- Demo.

- Demo.

- Demo.

- Demo.

- We have a formal proof of the resistance of the Aumüller *et al.* countermeasure against the BellCoRe attack by fault injection on CRT-RSA.

⇒ We have shown the **importance** of **formal analysis** in the field of **implementation security**.

- We have a formal proof of the resistance of the Aumüller *et al.* countermeasure against the BellCoRe attack by fault injection on CRT-RSA.

$\Rightarrow$ We have shown the **importance** of **formal analysis** in the field of **implementation security**.

- We would like to handle the repaired countermeasure of Vigilant [CGM+10].
- We would like to handle the countermeasure of Kim *et al.* [KKHH11].
- We also want to extend the capabilities of our tool to take into account fault injection in the control flow.

Regarding the CRT-RSA algorithm from Vigilant, the difficulty our verification framework in OCaml shall overcome is to decide how to inject the remarkable identity $(1+r)^{d_p} \equiv 1 + d_p \cdot r \mod r^2$.

The conclusion of their own article states:

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

- We would like to handle the repaired countermeasure of Vigilant [CGM+10].
- We would like to handle the countermeasure of Kim *et al.* [KKHH11].
- We also want to extend the capabilities of our tool to take into account fault injection in the control flow.

Regarding the CRT-RSA algorithm from Kim *et al.*, the computation is very detailed (it goes down to the multiplication level), and involves Boolean operations (and, xor, *etc.*), so more expertise about both arithmetic and logic must be added to our software.

- We would like to handle the repaired countermeasure of Vigilant [CGM+10].
- We would like to handle the countermeasure of Kim *et al.* [KKHH11].
- We also want to extend the capabilities of our tool to take into account fault injection in the control flow.

References

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.

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.

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.

Marc Joye, Arjen K. Lenstra, and Jean-Jacques Quisquater.
Chinese Remaindering Based Cryptosystems in the Presence of Faults.
J. Cryptology, 12(4):241–245, 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`.

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.

References

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.

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?

`rauzy@enst.fr`

- The attacker only has access to the output value, not the intermediate ones.
- Thus, implementation details are not important: the result of a computation is either faulted or it is not.

$\Rightarrow$ Our high-level model does capture that.

- It's free software.
- I think research software should be:
  - Free software (open access!);
  - Publicly demonstrated and discussed (presented at workshops / conferences);
  - **peer reviewed**.

$\Rightarrow$ Releasing research software should be like publishing an article (and should count as such, by the way).