Toward Trustable Homomorphic Computation

Pablo Rauzy

pablo.rauzy@univ-paris8.fr

pablo.rauzy.name

UNIVERSITÉ PARIS 8 VINCENNES-SAINT-DENIS

séminaire “Protection de l’information”

2017-05-04

- Protecting CRT-RSA against fault injection attacks.
- Generalizing modular extension.
- Trustable homomorphic cryptography?

- RSA, CRT-RSA, and the BellCoRe attack.
- Modular extension.

RSA (*Rivest, Shamir, Adleman*) — **Definition**

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

- 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*) — Definition

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

- 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):
  - $d_p \doteq d \bmod (p-1)$
  - $d_q \doteq d \bmod (q-1)$
  - $i_q \doteq q^{-1} \bmod p$
- S is then computed as follows:
  - $S_p = M^{d_p} \bmod p$
  - $S_q = M^{d_q} \bmod q$
  - $S = S_q + q \cdot (i_q \cdot (S_p - S_q) \bmod p)$
    (Garner recombination).

BellCoRe (*Bell Communications Research*) — Definition

The BellCoRe attack 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.

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

Why does it Work?

- 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.

- If S_p is faulted (i.e., replaced by $\widehat{S_p} \neq S_p$):
  - $S - \widehat{S} = q \cdot \Big((i_q \cdot (S_p - S_q) \mod p) - (i_q \cdot (\widehat{S_p} - S_q) \mod p) \Big)$

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

- If S_q is faulted (i.e., replaced by $\widehat{S_q} \neq S_q$):
  - $S - \widehat{S} \equiv (S_q - \widehat{S_q}) - (q \mod p) \cdot i_q \cdot (S_q - \widehat{S_q}) \mod p$

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

Countermeasures

- Many countermeasures have been proposed:
  - ∼20 papers,
  - from 1999 to now,
  - both from academia and industry.
- Including:
  - Shamir (1999),
  - Aumüller et al. (2002),
  - Vigilant (2008) + Coron et al. (2010).

finja

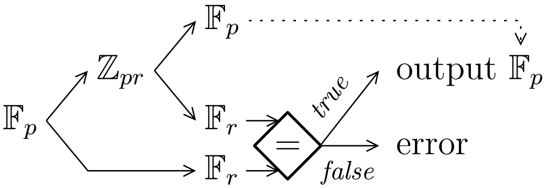
- Inputs:
  - a high-level description of the algorithm,
  - an attack success condition,
  - a fault model.
- Output:
  - the list of working attacks, or
  - a proof that the computation is resistant to fault injections.
- `https://pablo.rauzy.name/sensi/finja.html`

Classification

Countermeasure	Family	Verification method/count	Order intended	Order actual	Small subrings usage
Shamir (1999)	Shamir	test / 1	1	0	$r_1 = r_2$, consistency of intermediate signatures
Joye et al. (2001)	Shamir	test / 2	1	0	checksums of the intermediate CRT signatures
Aumüller et al. (2002)	Shamir	test / 5	1	1	$r_1 = r_2$, consistency of the checksums of both intermediate signatures
Blömer et al. (2003)	Shamir	infection / 2	1	1	direct verification of the intermediate CRT signatures, CRT recombination happens in overring
Ciet & Joye (2005)	Shamir	infection / 2	2	1	checksums of the intermediate CRT signatures, CRT recombination happens in overring
Giraud (2006)	Giraud	test / 1	1	1	NA
Boscher et al. (2007)	Giraud	test / 1	1	1	NA
Vigilant (2008)	Shamir	test / 7	1	1	$r_1 = r_2$, embedded control values, CRT recombination happens in overring
Rivain (2009)	Giraud	test / 2	1	1	NA
Kim et al. (2011)	Giraud	infection / 6	1	1	NA

Modular Extension

- Many of these countermeasures are patented.
- Most of them are doing the exact same thing: *modular extension*.
- The idea is to use the isomorphism between $\mathbb{F}_p \times \mathbb{F}_r$ and $\mathbb{Z}_{pr}$.



Notation: $\mathbb{Z}_n$ is a shorthand for $\mathbb{Z}/n\mathbb{Z}$.

- Modular extension is not tied to RSA.
- Automation and application to elliptic curve cryptography.

Generic Integrity Verification

- The working factors of modular extension based countermeasures:
  - are not tied to the BellCoRe attack,
  - nor to the CRT-RSA algorithm.
- Cost-effective integrity verification of any arithmetic computation.

Inversion in Direct Products

Divisions optimization

Proposition

To get the inverse of z in $\mathbb{F}_p$ while computing in $\mathbb{Z}_{pr}$, one has:

- $z = 0 \bmod r \implies (z^{p-2} \bmod pr) \equiv z^{-1} \mod p$,
- otherwise $(z^{-1} \bmod pr) \equiv z^{-1} \mod p$.

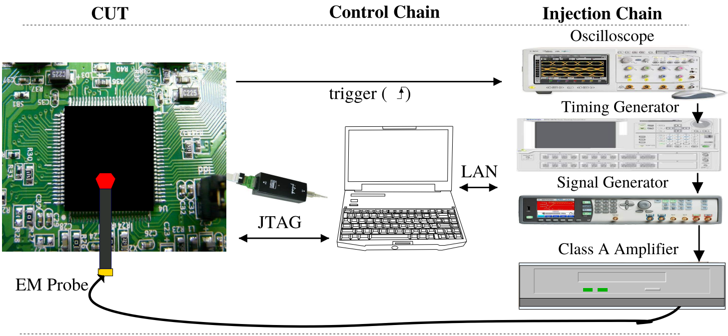
proof sketch:

- If $z = 0 \mod r$, then z is not invertible in $\mathbb{Z}_{pr}$:
  - but z^{p-2} exists in $\mathbb{Z}_{pr}$,
  - and $(z^{p-2} \bmod pr) \bmod p = z^{p-2} \bmod p = z^{-1} \bmod p$.

enredo

- Inputs:
  - an asymmetric cryptography algorithm to be protected,
  - a desired redundancy level.
- Output:
  - the (proved to be the) same algorithm
  - provably protected against fault injection attacks.
- `https://pablo.rauzy.name/sensi/enredo.html`

Practical Case Study with ECSM on 32-bit ARMv7



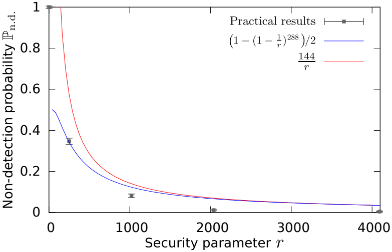
Field characteristic	p = `0xfffffffffffffffffffffffffffffffeffffffffffffffff`
Curve equation coefficients	a = `0xfffffffffffffffffffffffffffffffefffffffffffffffc` b = `0x64210519e59c80e70fa7e9ab72243049feb8deecc146b9b1`
Point coordinates	Gx = `0x188da80eb03090f67cbf20eb43a18800f4ff0afd82ff1012` Gy = `0x07192b95ffc8da78631011ed6b24cdd573f977a11e794811`
Point order	ord = `0xffffffffffffffffffffffff99def836146bc9b1b4d22831`

Parameters of our ECSM implementation (namely NIST P-192)

Security Results

r value	r size (bit)	Positives (%)		Negatives (%)	
		true	false	true	false
1	1	0.00	0.00	2.74	97.26
251	8	63.65	0.00	2.56	33.79
1021	10	89.09	0.00	2.96	7.95
2039	11	98.82	0.00	0.00	1.18
4093	12	97.61	0.00	1.91	0.48
65521	16	97.79	0.00	2.21	0.00
4294967291	32	97.19	0.00	2.81	0.00
18446744073709551557	64	99.79	0.00	0.21	0.00

$\approx$ 1000 tests for each value of r



Performance Results

r value	r size (bit)	time (ms) $\mathbb{Z}_{pr}$	$\mathbb{F}_r$	test	overhead
1	1	683	24	$\ll 1$	$\times 1.04$
251	8	883	91	$\ll 1$	$\times 1.43$
1021	10	899	100	$\ll 1$	$\times 1.46$
2039	11	902	197	$\ll 1$	$\times 1.61$
4093	12	903	197	$\ll 1$	$\times 1.61$
65521	16	883	189	$\ll 1$	$\times 1.56$
4294967291	32	832	172	$\ll 1$	$\times 1.47$
18446744073709551557	64	996	246	$\ll 1$	$\times 1.82$

Signature verification overhead $\approx \times 4.5$.

Code C + `mini-gmp` compiled with `gcc -O0` (no optimization).

Modular Extension is Parallel

- The computation in $\mathbb{F}_r$ and in $\mathbb{Z}_{pr}$ are completely independent.
- They can be executed in any order and in particular on *different machines*.

- Homomorphic cryptography, privacy concerns.
- Related works.
- Leveraging modular extension.

Homomorphic encryption — **Definition**

Homomorphic encryption allows computations to be carried out on ciphertext, generating an encrypted result which when decrypted matches the result of operations performed on the plaintext.

- Allows to delegate computation on privacy-sensitive data...
- ... provided that we trust the third-party with the computation.

Trivial Example

- An insurance company offers multiple plans for weight-related diseases.
- It also provides a free service for computing one's body mass index.

$$BMI = \frac{\text{mass}}{\text{height}^2}$$

Privacy Concerns

- People do not want to send their personal information in clear over the network.
- They also do not want to reveal them to the insurance company.
- Thus the BMI computation service uses homomorphic cryptography:
  - the user do not send $m = \text{mass}$ and $h = \text{height}$,
  - instead, they send $\mathcal{E}(m)$ and $\mathcal{E}(h)$, homomorphically encrypted values.

Trust

- The user must now trust the insurance company with computing

$$\mathcal{E}(BMI) = \frac{\mathcal{E}(m)}{\mathcal{E}(h) \times_{\mathcal{E}} \mathcal{E}(h)}\varepsilon.$$

- While the insurance company would benefit from computing instead

$$\mathcal{E}(BMI') = \frac{\mathcal{E}(m) +_{\mathcal{E}} \mathcal{E}(20)}{\mathcal{E}(h) \times_{\mathcal{E}} \mathcal{E}(h)}\varepsilon.$$

Related Works

- *How to Delegate and Verify in Public: Verifiable Computation from Attribute-based Encryption*
  Parno et al. 2011 (IACR ePrint 2011/597).
- *Pinocchio: Nearly Practical Verifiable Computation*
  Parno et al. 2013 (IACR ePrint 2013/279).
- *Efficiently Verifiable Computation on Encrypted Data*
  Fiore et al. 2014 (IACR ePrint 2014/202).
- *Verifiable Computation on Outsourced Encrypted Data*
  Lai et al. 2014.

Problems

- Introduce new complicated cryptographic constructions (e.g., “homomorphic encrypted authenticator” in Lai et al.).
- Stay at a theoretical level: only implemented in $\LaTeX$.
- No actual implementation and novelty means no real tests:
  - Security flaws?
  - Complexity?
  - Practical feasibility of implementation?

Leveraging Modular Extension

- Modular extension is practical, proved, and formally studied.
- Let’s try to apply it to homomorphic cryptography!

Project

- Getting to know better the field of homomorphic cryptography.
- Finding (or inventing?) a (somewhat) fully homomorphic cryptosystem which would support modular extension.
  - No tests depending on the modular values.
- Finding (or writing?) an hackable implementation of it.
  - YASHE from Bos et al. 2013 (IACR ePrint 2013 2013/075)?
- Prototyping the verification of delegated computation using modular extension.

Appendix

Formal Study of Countermeasures

- The goal is making sure countermeasures are trustworthy:
  - by proving the algorithm at high-level (the proof should apply to any refinement),
  - by covering a very general attacker model.

$$M \longrightarrow \boxed{\text{RSA}} \longrightarrow S$$

vs

```
.file "mp_mul_256.c"
.text
.p2align 4,,15
.globl mp_mul_256
.type mp_mul_256, @function
mp_mul_256:
.LFB0:
.cfi_startproc
# uint64_t mp_mul_256(uint64_t c[8], uint64_t a[4], uint64_t b[4])
movdqa (%rsi), %xmm4
movdqa 16(%rsi), %xmm5
movdqa (%rdx), %xmm6
movdqa 16(%rdx), %xmm7
movq %r12, %xmm14
movq %r14, %xmm15
pinsrq $1, %r13, %xmm14
pinsrq $1, %r15, %xmm15
pextrq $1, %xmm4, %r12
pextrq $1, %xmm5, %r13
pextrq $1, %xmm6, %r14
pextrq $1, %xmm7, %r15
xorq %r9, %r9
xorq %r10, %r10
movq %xmm6, %r11
movq %xmm4, %rax
mulq %r11
movq %rax, %xmm0
movq %rdx, %r8
movq %xmm4, %rax
mulq %r14
addq %rax, %r8
adcq %rdx, %r9
movq %xmm6, %rax
mulq %r12
addq %rax, %r8
adcq %rdx, %r9
adcq $0, %r10
pinsrq $1, %r8, %xmm0
xorq %r8, %r8
movq %xmm7, %r11
movq %xmm4, %rax
mulq %r11
addq %rax, %r9
adcq %rdx, %r10
adcq $0, %r8cde
# ...
```

Threat Model

Attacker model — Definition

The attacker can request CRT-RSA computations, inject fault(s) during the computation, and read the final result of the computation.

- Data fault (on intermediate values):
  - *zeroing* or *randomizing*,
  - *permanent* or *transient*.
- Code fault:
  - *skipping* any number of consecutive instructions.
- Attack *order*:
  - number of fault injections during the computation (an attack is said *high-order* if its order is > 1).

Data-Code Faulting Equivalence Lemma

Equivalence between faults on the code and on the data — **Lemma**

The effect of a skipping fault (i.e., fault on the code) can be captured by considering only randomizing and zeroing faults (i.e., fault on the data).

proof sketch:

- If the skipped instructions are part of an arithmetic operation:
  - either the computation has not been done at all: its result becomes zero (if initialized) or random (if not),
  - or the computation has partly been done: its result is thus considered random at our modeling level.
- If the skipped instruction is a branching instruction, it is equivalent to fault the result of the branching condition:
  - at zero (i.e., `false`), to avoid branching,
  - at random (i.e., `true`), to force branching.

finja

- Inputs:
  - a high-level description of the algorithm,
  - an attack success condition,
  - a fault model.
- Output:
  - the list of working attacks, or
  - a proof that the computation is resistant to fault injections.
- `https://pablo.rauzy.name/sensi/finja.html`

Overview

1. The algorithm is parsed into an internal representation (an AST):
   - that **finja** can execute symbolically (simplified),
   - that encodes properties of the intermediate variables.

2. **finja** makes a copy of the original tree and simplifies it.

3. For each possible fault(s) injection(s) in the fault model, **finja**:
   - produces a copy of the original tree,
   - injects the fault in the copy,
   - simplifies the faulted tree,
   - checks attack success condition holds,
     if yes, the working attack is reported,
     if not, the countermeasure is considered secure against this attack.

4. **finja** outputs an HTML report.

Rewriting System

- Most of the $\mathbb{Z}$ ring axioms,
- $\mathbb{Z}_N$ subrings,
- And a few theorems.

Rewriting System

- Most of the $\mathbb{Z}$ ring axioms:
  - neutral elements (0 for sums, 1 for products);
  - absorbing element (0, for products);
  - inverses and opposites;
  - associativity and commutativity;
  - but no distributivity (not confluent).
- $\mathbb{Z}_N$ subrings,
- And a few theorems.

Rewriting System

- Most of the $\mathbb{Z}$ ring axioms,
- $\mathbb{Z}_N$ subrings:
  - identity:
    - $(a \mod N) \mod N = a \mod N$,
    - $N^k \mod N = 0$;
  - inverse:
    - $(a \mod N) \times (a^{-1} \mod N) \mod N = 1$,
    - $(a \mod N) + (-a \mod N) \mod N = 0$;
  - associativity and commutativity:
    - $(b \mod N) + (a \mod N) \mod N = a + b \mod N$,
    - $(a \mod N) \times (b \mod N) \mod N = a \times b \mod N$;
  - subrings: $(a \mod N \times m) \mod N = a \mod N$.
- And a few theorems.

Rewriting System

- Most of the $\mathbb{Z}$ ring axioms,
- $\mathbb{Z}_N$ subrings,
- And a few theorems:
  - Fermat's little theorem;
  - its generalization, Euler's theorem;
  - Chinese remainder theorem;
  - Binomial theorem in $\mathbb{Z}_{r^2}$ rings
    $(1 + r)^d \equiv 1 + dr \mod r^2$.

Minimal Example of Usage

minimal-example.fia

```
noprop a, b, c ;
t := a + b * c ;
return t ;

%%

@ !=[b] a
```

- Computation: $t = a + b \times c$.
- Attack success condition: $t \not\equiv a \mod b$.
- `finja -r minimal-example.fia`
- `finja -z minimal-example.fia`

Minimal Example of Simplification

randomizing fault on c

```
noprop a, b, c ;
t := a + b * Random ;
return t ;

%%

@ !=[b] a
```

- ```
     @ !=[b] a
  => a + b * Random !=[b] a
  => a != a
  => false
  ```
- Harmless fault injection.

zeroing fault on a

```
noprop a, b, c ;
t := Zero + b * c ;
return t ;

%%

@ !=[b] a
```

- ```
     @ !=[b] a
  => Zero + b * c !=[b] a
  => b * c !=[b] a
  => 0 != a
  => true
  ```
- Attack successful.

Proven Countermeasure

- Using finja, I have proved the security of:
  - Aumüller et al. (2002) at PROOFS 2013 and
  - Vigilant (2008) + Coron et al. (2010) at PPREW 2014.

- I have optimized:
  - Aumüller: from 7 to 5 verifications,
  - Vigilant: from 9 to 3 verifications, from 5 to 1 random number (plus removed unnecessary computations).

High-Order Countermeasures?

- High-order attacks?
- High-order countermeasures?
- Proven high-order countermeasures?

High-Order Attacks

- High-order attacks have been studied and shown practical:
  - *Fault Attacks for CRT Based RSA:*
    New Attacks, New Results, and New Countermeasures,
    by C. H. Kim and J.-J. Quisquater at WISTP'07.
  - *Multi Fault Laser Attacks on Protected CRT-RSA*,
    by E. Trichina and R. Korkikyan at FDTC'10.

Existing High-Order Countermeasures?

- A few countermeasures claim to be second-order:
  - *Practical fault countermeasures for chinese remaindering based RSA*, by M. Ciet and M. Joye at FDTC'05.
  - *On Second-Order Fault Analysis Resistance for CRT-RSA Implementations*, by E. Dottax, C. Giraud, M. Rivain, and Y. Sierra at WISTP'09.
- But they do not work in our more general fault model:
  - `finja -t -n 2 -z -z -s crt-rsa_ciet-joye.fia`
  - `finja -t -n 2 -r -z -s crt-rsa_dottax-etal.fia`
- We found no countermeasure claiming to resist > 2 faults.

Towards a Proven High-Order Countermeasure

- If we want a high-order countermeasure, we have to create it:
  - What is a countermeasure?
  - What makes a countermeasure work? What makes it fail?
  - How do the existing first-order countermeasures work?

Countermeasures Classification

- What are the methods used by the existing countermeasures?
- We used 4 main parameters to classify countermeasures:
  1. Shamir's or Giraud's family,
  2. test-based or infective,
  3. intended order,
  4. usage of the small subrings.

Classification Recap

Countermeasure	Family	Verification method / count	Order intended	Order actual	Small subrings usage
Shamir (1999)	Shamir	test / 1	1	0	$r_1 = r_2$, consistency of intermediate signatures
Joye et al. (2001)	Shamir	test / 2	1	0	checksums of the intermediate CRT signatures
Aumüller et al. (2002)	Shamir	test / 5	1	1	$r_1 = r_2$, consistency of the checksums of both intermediate signatures
Blömer et al. (2003)	Shamir	infection / 2	1	1	direct verification of the intermediate CRT signatures, CRT recombination happens in overring
Ciet & Joye (2005)	Shamir	infection / 2	2	1	checksums of the intermediate CRT signatures, CRT recombination happens in overring
Giraud (2006)	Giraud	test / 1	1	1	NA
Boscher et al. (2007)	Giraud	test / 1	1	1	NA
Vigilant (2008)	Shamir	test / 7	1	1	$r_1 = r_2$, embedded control values, CRT recombination happens in overring
Rivain (2009)	Giraud	test / 2	1	1	NA
Kim et al. (2011)	Giraud	infection / 6	1	1	NA

Building Better Countermeasures

- Formal study and classification of countermeasures:
  - provided a better understanding of their working factors,
  - allow to fix broken countermeasures, and build better ones.

Correcting Shamir's Countermeasure

Algorithm: CRT-RSA with Shamir's countermeasure

Input: Message M, key (p, q, d, i_q) **Output:** Signature $M^d \mod N$, or error

Choose a small random integer r.

$p' = p \cdot r$

$q' = q \cdot r$

$S'_p = M^{d \mod \varphi(p')} \mod p'$ // Intermediate signature in $\mathbb{Z}_{pr}$

$S'_q = M^{d \mod \varphi(q')} \mod q'$ // Intermediate signature in $\mathbb{Z}_{qr}$

if $S'_p \not\equiv S'_q \mod r$ **then return** error

$S_p = S'_p \mod p$ // Retrieve intermediate signature in $\mathbb{Z}_p$

$S_q = S'_q \mod q$ // Retrieve intermediate signature in $\mathbb{Z}_q$

$S = S_q + q \cdot (i_q \cdot (S_p - S_q) \mod p)$ // Recombination in $\mathbb{Z}_N$

return S

Correcting Shamir's Countermeasure

Algorithm: CRT-RSA with Shamir's countermeasure

Input: Message M, key (p, q, d, i_q) **Output:** Signature $M^d \mod N$, or error

```
Choose a small random integer r.
p' = p · r
q' = q · r
if p' ≢ 0 mod p or q' ≢ 0 mod q then return error
S'_p = M^(d mod φ(p')) mod p'          // Intermediate signature in Z_pr
S'_q = M^(d mod φ(q')) mod q'          // Intermediate signature in Z_qr
if S'_p ≢ S'_q mod r then return error
S_p = S'_p mod p                       // Retrieve intermediate signature in Z_p
S_q = S'_q mod q                       // Retrieve intermediate signature in Z_q
S = S_q + q · (i_q · (S_p − S_q) mod p) // Recombination in Z_N
if S ≢ S'_p mod p or S ≢ S'_q mod q then return error
return S
```

Choose a small random integer r.

$p' = p \cdot r$

$q' = q \cdot r$

if $p' \not\equiv 0 \mod p$ **or** $q' \not\equiv 0 \mod q$ **then return** error

$S'_p = M^{d \mod \varphi(p')} \mod p'$ // Intermediate signature in $\mathbb{Z}_{pr}$

$S'_q = M^{d \mod \varphi(q')} \mod q'$ // Intermediate signature in $\mathbb{Z}_{qr}$

if $S'_p \not\equiv S'_q \mod r$ **then return** error

$S_p = S'_p \mod p$ // Retrieve intermediate signature in $\mathbb{Z}_p$

$S_q = S'_q \mod q$ // Retrieve intermediate signature in $\mathbb{Z}_q$

$S = S_q + q \cdot (i_q \cdot (S_p - S_q) \mod p)$ // Recombination in $\mathbb{Z}_N$

if $S \not\equiv S'_p \mod p$ **or** $S \not\equiv S'_q \mod q$ **then return** error

return S

Simplifying Vigilant's Countermeasure

- Simplification of Vigilant's countermeasure in 4 steps:
  - our first simplifications + those of Coron et al.'s corrections,
  - remove additional computation with random numbers,
  - verify the 3 necessary invariants in a single step,
  - bonus: transform the countermeasure into an infective variant.

Algorithm: CRT-RSA with Vigilant's countermeasure

Input: Message M, key (p, q, d_p, d_q, i_q) **Output:** Signature $M^d \mod N$, or error

Choose a small random integer r, R_1, R_2, R_3, R_4. $N = p \cdot q$
$p' = p \cdot r^2$
$i_{pr} = p^{-1} \mod r^2$
$M_p = M \mod p'$
$B_p = p \cdot i_{pr}$; $A_p = 1 - B_p \mod p'$
$M'_p = A_p \cdot M_p + B_p \cdot (1 + r) \mod p'$ // CRT insertion of verification value in M'_p
$d'_p = d_p + R_3 \cdot (p - 1)$
$S'_p = {M'_p}^{d'_p \mod \varphi(p')} \mod p'$ // Intermediate signature in $\mathbb{Z}_{pr^2}$
if $M'_p \not\equiv M \mod p$ **or** $d'_p \not\equiv d_p \mod p - 1$ **or** $B_p \cdot S'_p \not\equiv B_p \cdot (1 + d'_p \cdot r) \mod p'$ **then return** error
$S_{pr} = S'_p - B_p \cdot (1 + d'_p \cdot r - R_1)$ // Verification value of S'_p swapped with R_1
$q' = q \cdot r^2$
$i_{qr} = q^{-1} \mod r^2$
$M_q = M \mod q'$
$B_q = q \cdot i_{qr}$; $A_q = 1 - B_q \mod q'$
$M'_q = A_q \cdot M_q + B_q \cdot (1 + r) \mod q'$ // CRT insertion of verification value in M'_q
$d'_q = d_q + R_4 \cdot (q - 1)$
$S'_q = {M'_q}^{d'_q \mod \varphi(q')} \mod q'$ // Intermediate signature in $\mathbb{Z}_{qr^2}$
if $M'_q \not\equiv M \mod q$ **or** $d'_q \not\equiv d_q \mod q - 1$ **or** $B_q \cdot S'_q \not\equiv B_q \cdot (1 + d'_q \cdot r) \mod q'$ **then return** error
$S_{qr} = S'_q - B_q \cdot (1 + d'_q \cdot r - R_2)$ // Verification value of S'_q swapped with R_2
if $M_p \not\equiv M_q \mod r^2$ **then return** error
$S_r = S_{qr} + q \cdot (i_q \cdot (S_{pr} - S_{qr}) \mod p')$ // Recombination checksum in $\mathbb{Z}_{Nr^2}$

if $N \cdot (S_r - R_2 - q \cdot i_q \cdot (R_1 - R_2)) \not\equiv 0 \mod Nr^2$ **then return** error
if $q \cdot i_q \not\equiv 1 \mod p$ **then return** error
return $S = S_r \mod N$ // Retrieve result in $\mathbb{Z}_N$

Algorithm: CRT-RSA with Vigilant's countermeasure

Input: Message M, key (p, q, d_p, d_q, i_q) **Output:** Signature $M^d \mod N$, or error

Choose a small random integer r, R_1, R_2, R_3, R_4. $N = p \cdot q$

$p' = p \cdot r^2$

$i_{pr} = p^{-1} \mod r^2$

$M_p = M \mod p'$

$B_p = p \cdot i_{pr}$; $A_p = 1 - B_p \mod p'$

$M'_p = A_p \cdot M_p + B_p \cdot (1 + r) \mod p'$ // CRT insertion of verification value in M'_p

$d'_p = d_p + R_3 \cdot (p - 1)$

$S'_p = {M'_p}^{d'_p \mod \varphi(p')} \mod p'$ // Intermediate signature in $\mathbb{Z}_{pr^2}$

if $M'_p \not\equiv M \mod p$ **or** $d'_p \not\equiv d_p \mod p - 1$ **or** $B_p \cdot S'_p \not\equiv B_p \cdot (1 + d'_p \cdot r) \mod p'$ **then return** error

$S_{pr} = S'_p - B_p \cdot (1 + d'_p \cdot r - R_1)$ // Verification value of S'_p swapped with R_1

$q' = q \cdot r^2$

$i_{qr} = q^{-1} \mod r^2$

$M_q = M \mod q'$

$B_q = q \cdot i_{qr}$; $A_q = 1 - B_q \mod q'$

$M'_q = A_q \cdot M_q + B_q \cdot (1 + r) \mod q'$ // CRT insertion of verification value in M'_q

$d'_q = d_q + R_4 \cdot (q - 1)$

$S'_q = {M'_q}^{d'_q \mod \varphi(q')} \mod q'$ // Intermediate signature in $\mathbb{Z}_{qr^2}$

if $M'_q \not\equiv M \mod q$ **or** $d'_q \not\equiv d_q \mod q - 1$ **or** $B_q \cdot S'_q \not\equiv B_q \cdot (1 + d'_q \cdot r) \mod q'$ **then return** error

$S_{qr} = S'_q - B_q \cdot (1 + d'_q \cdot r - R_2)$ // Verification value of S'_q swapped with R_2

$S_r = S_{qr} + q \cdot (i_q \cdot (S_{pr} - S_{qr}) \mod p')$ // Recombination checksum in $\mathbb{Z}_{Nr^2}$

if $pq \cdot (S_r - R_2 - q \cdot i_q \cdot (R_1 - R_2)) \not\equiv 0 \mod Nr^2$ **then return** error

return $S = S_r \mod N$ // Retrieve result in $\mathbb{Z}_N$

Algorithm: CRT-RSA with Vigilant's countermeasure

Input: Message M, key (p, q, d_p, d_q, i_q) **Output:** Signature $M^d \mod N$, or error

Choose a small random integer r, R_1, R_2. $N = p \cdot q$

$p' = p \cdot r^2$

$i_{pr} = p^{-1} \mod r^2$

$M_p = M \mod p'$

$B_p = p \cdot i_{pr}$; $A_p = 1 - B_p \mod p'$

$M'_p = A_p \cdot M_p + B_p \cdot (1 + r) \mod p'$ // CRT insertion of verification value in M'_p

$S'_p = {M'_p}^{d_p \mod \varphi(p')} \mod p'$ // Intermediate signature in $\mathbb{Z}_{pr^2}$

if $M'_p \not\equiv M \mod p$ **or** $B_p \cdot S'_p \not\equiv B_p \cdot (1 + d_p \cdot r) \mod p'$ **then return** error

$S_{pr} = S'_p - B_p \cdot (1 + d_p \cdot r - R_1)$ // Verification value of S'_p swapped with R_1

$q' = q \cdot r^2$

$i_{qr} = q^{-1} \mod r^2$

$M_q = M \mod q'$

$B_q = q \cdot i_{qr}$; $A_q = 1 - B_q \mod q'$

$M'_q = A_q \cdot M_q + B_q \cdot (1 + r) \mod q'$ // CRT insertion of verification value in M'_q

$S'_q = {M'_q}^{d_q \mod \varphi(q')} \mod q'$ // Intermediate signature in $\mathbb{Z}_{qr^2}$

if $M'_q \not\equiv M \mod q$ **or** $B_q \cdot S'_q \not\equiv B_q \cdot (1 + d_q \cdot r) \mod q'$ **then return** error

$S_{qr} = S'_q - B_q \cdot (1 + d_q \cdot r - R_2)$ // Verification value of S'_q swapped with R_2

$S_r = S_{qr} + q \cdot (i_q \cdot (S_{pr} - S_{qr}) \mod p')$ // Recombination checksum in $\mathbb{Z}_{Nr^2}$

if $pq \cdot (S_r - R_2 - q \cdot i_q \cdot (R_1 - R_2)) \not\equiv 0 \mod Nr^2$ **then return** error

return $S = S_r \mod N$ // Retrieve result in $\mathbb{Z}_N$

Algorithm: CRT-RSA with Vigilant's countermeasure

Input: Message M, key (p, q, d_p, d_q, i_q) **Output:** Signature $M^d \mod N$, or error

Choose a small random integer r. $N = p \cdot q$

$p' = p \cdot r^2$

$i_{pr} = p^{-1} \mod r^2$

$M_p = M \mod p'$

$B_p = p \cdot i_{pr}$; $A_p = 1 - B_p \mod p'$

$M'_p = A_p \cdot M_p + B_p \cdot (1 + r) \mod p'$ // CRT insertion of verification value in M'_p

$S'_p = {M'_p}^{d_p \mod \varphi(p')} \mod p'$ // Intermediate signature in $\mathbb{Z}_{pr^2}$

if $M'_p + N \not\equiv M \mod p$ **then return** error

$S_{pr} = 1 + d_p \cdot r$ // Checksum in $\mathbb{Z}_{r^2}$ for S'_p

$q' = q \cdot r^2$

$i_{qr} = q^{-1} \mod r^2$

$M_q = M \mod q'$

$B_q = q \cdot i_{qr}$; $A_q = 1 - B_q \mod q'$

$M'_q = A_q \cdot M_q + B_q \cdot (1 + r) \mod q'$ // CRT insertion of verification value in M'_q

$S'_q = {M'_q}^{d_q \mod \varphi(q')} \mod q'$ // Intermediate signature in $\mathbb{Z}_{qr^2}$

if $M'_q + N \not\equiv M \mod q$ **then return** error

$S_{qr} = 1 + d_q \cdot r$ // Checksum in $\mathbb{Z}_{r^2}$ for S'_q

$S_r = S_{qr} + q \cdot (i_q \cdot (S_{pr} - S_{qr}) \mod p')$ // Recombination checksum in $\mathbb{Z}_{r^2}$

$S' = S'_q + q \cdot (i_q \cdot (S'_p - S'_q) \mod p')$ // Recombination in $\mathbb{Z}_{Nr^2}$

if $S' \not\equiv S_r \mod r^2$ **then return** error

return $S = S' \mod N$ // Retrieve result in $\mathbb{Z}_N$

Algorithm: CRT-RSA with Vigilant's countermeasure

Input: Message M, key (p, q, d_p, d_q, i_q) **Output:** Signature $M^d \mod N$, or a random value in $\mathbb{Z}_N$

Choose a small random integer r. $N = p \cdot q$

$p' = p \cdot r^2$

$i_{pr} = p^{-1} \mod r^2$

$M_p = M \mod p'$

$B_p = p \cdot i_{pr}$; $A_p = 1 - B_p \mod p'$

$M'_p = A_p \cdot M_p + B_p \cdot (1 + r) \mod p'$ // CRT insertion of verification value in M'_p

$S'_p = {M'_p}^{d_p \mod \varphi(p')} \mod p'$ // Intermediate signature in $\mathbb{Z}_{pr^2}$

$c_p = M'_p + N - M + 1 \mod p$

$S_{pr} = 1 + d_p \cdot r$ // Checksum in $\mathbb{Z}_{r^2}$ for S'_p

$q' = q \cdot r^2$

$i_{qr} = q^{-1} \mod r^2$

$M_q = M \mod q'$

$B_q = q \cdot i_{qr}$; $A_q = 1 - B_q \mod q'$

$M'_q = A_q \cdot M_q + B_q \cdot (1 + r) \mod q'$ // CRT insertion of verification value in M'_q

$S'_q = {M'_q}^{d_q \mod \varphi(q')} \mod q'$ // Intermediate signature in $\mathbb{Z}_{qr^2}$

$c_q = M'_q + N - M + 1 \mod q$

$S_{qr} = 1 + d_q \cdot r$ // Checksum in $\mathbb{Z}_{r^2}$ for S'_q

$S_r = S_{qr} + q \cdot (i_q \cdot (S_{pr} - S_{qr}) \mod p')$ // Recombination checksum in $\mathbb{Z}_{r^2}$

$S' = S'_q + q \cdot (i_q \cdot (S'_p - S'_q) \mod p')$ // Recombination in $\mathbb{Z}_{Nr^2}$

$c_S = S' - S_r + 1 \mod r^2$

return $S = {S'}^{c_p c_q c_S} \mod N$ // Retrieve result in $\mathbb{Z}_N$

High-Order Countermeasures

High-Order Countermeasures — **Proposition**

Against randomizing faults, all first-order correct countermeasures are high-order.

However, there are no generic high-order countermeasures if the three types of faults in our attack model are taken into account, but it is possible to build Dth-order countermeasures for any D.

proof sketch:

- Random faults cannot induce a verification skip (whether test-based or infective).
- Repeating verifications D times can force to inject $D+1$ faults.

Generating High-Order Countermeasures

Algorithm: Generation of CRT-RSA with Vigilant's countermeasure at order D

Input: order D **Output:** CRT-RSA algorithm protected against fault injection attack of order D

print Choose a small random integer r.
print $N = p \cdot q$
print $p' = p \cdot r^2$; $i_{pr} = p^{-1} \mod r^2$; $M_p = M \mod p'$; $B_p = p \cdot i_{pr}$; $A_p = 1 - B_p \mod p'$
print $M'_p = A_p \cdot M_p + B_p \cdot (1 + r) \mod p'$
print $q' = q \cdot r^2$; $i_{qr} = q^{-1} \mod r^2$; $M_q = M \mod q'$; $B_q = q \cdot i_{qr}$; $A_q = 1 - B_q \mod q'$
print $M'_q = A_q \cdot M_q + B_q \cdot (1 + r) \mod q'$
print $S'_p = {M'_p}^{d_p \mod \varphi(p')} \mod p'$
print $S'_q = {M'_q}^{d_q \mod \varphi(q')} \mod q'$
print $S_{pr} = 1 + d_p \cdot r$
print $S_{qr} = 1 + d_q \cdot r$
print $S_r = S_{qr} + q \cdot (i_q \cdot (S_{pr} - S_{qr}) \mod p')$
print $S' = S'_q + q \cdot (i_q \cdot (S'_p - S'_q) \mod p')$
print $S_0 = S' \mod N$
for $i \leftarrow 1$ ***to*** D **do**
  print if $M'_p + N \not\equiv M \mod p$ **then return** error
  print S'; **print** $_i$ **print** $= S$; **print** $_{i-1}$
  print if $M'_q + N \not\equiv M \mod q$ **then return** error
  print S''; **print** $_i$ **print** $= S'$; **print** $_i$
  print if $S \not\equiv S_r \mod r^2$ **then return** error
  print S; **print** $_i$ **print** $= S''$; **print** $_i$
end
print return S; **print** $_D$

Example of countermeasure of order 3

Algorithm: CRT-RSA with Vigilant's countermeasure at order 3

Input: Message M, key (p, q, d_p, d_q, i_q) **Output:** Signature $M^d \mod N$, or error

Choose a small random integer r.
$N = p \cdot q$
$p' = p \cdot r^2$; $i_{pr} = p^{-1} \mod r^2$; $M_p = M \mod p'$; $B_p = p \cdot i_{pr}$; $A_p = 1 - B_p \mod p'$
$M'_p = A_p \cdot M_p + B_p \cdot (1 + r) \mod p'$
$q' = q \cdot r^2$; $i_{qr} = q^{-1} \mod r^2$; $M_q = M \mod q'$; $B_q = q \cdot i_{qr}$; $A_q = 1 - B_q \mod q'$
$M'_q = A_q \cdot M_q + B_q \cdot (1 + r) \mod q'$
$S'_p = {M'_p}^{d_p \mod \varphi(p')} \mod p'$; $S_{pr} = 1 + d_p \cdot r$
$S'_q = {M'_q}^{d_q \mod \varphi(q')} \mod q'$; $S_{qr} = 1 + d_q \cdot r$
$S_r = S_{qr} + q \cdot (i_q \cdot (S_{pr} - S_{qr}) \mod p')$
$S' = S'_q + q \cdot (i_q \cdot (S'_p - S'_q) \mod p')$
$S_0 = S' \mod N$
if $M'_p + N \not\equiv M \mod p$ **then return** error **else** $S'_1 = S_0$
if $M'_q + N \not\equiv M \mod q$ **then return** error **else** $S''_1 = S'_1$
if $S \not\equiv S_r \mod r^2$ **then return** error **else** $S_1 = S''_1$
if $M'_p + N \not\equiv M \mod p$ **then return** error **else** $S'_2 = S_1$
if $M'_q + N \not\equiv M \mod q$ **then return** error **else** $S''_2 = S'_2$
if $S \not\equiv S_r \mod r^2$ **then return** error **else** $S_2 = S''_2$
if $M'_p + N \not\equiv M \mod p$ **then return** error **else** $S'_3 = S_2$
if $M'_q + N \not\equiv M \mod q$ **then return** error **else** $S''_3 = S'_3$
if $S \not\equiv S_r \mod r^2$ **then return** error **else** $S_3 = S''_3$
return S_3

enredo

- Inputs:
  - an asymmetric cryptography algorithm to be protected,
  - a desired redundancy level.
- Output:
  - the (proved to be the) same algorithm
  - provably protected against fault injection attacks.
- `https://pablo.rauzy.name/sensi/enredo.html`

Overview

1. The algorithm is parsed and type-checked:
   - type-checker gather necessary information for the transformation.

2. **enredo** applies the *modular extension* transformation:
   - the transformation has been formally defined,
   - and it is proved correct (semantic preserving).

3. **enredo** outputs the protected algorithm.

Correctness of the Transformation

Correctness (Proposition)

The transformation is correct if at all time during the execution the invariant defining the transformation of the memory holds, and when a value is returned, it is the same as in the original program.
The **enredo** transformation is correct.

proof sketch:

$$\begin{tikzcd} m \arrow[r, "{[\![\mathrm{s}]\!]_\Gamma}"] \arrow[d, "\langle\cdot\rangle_r"'] & m' \arrow[d, "\langle\cdot\rangle_r"] \\ \langle m\rangle_r \arrow[r, dashed, "{[\![\langle \mathrm{s}\rangle_{r,\Gamma}]\!]_{\langle\Gamma\rangle_r}}"'] & \langle m'\rangle_r \end{tikzcd}$$

during the execution, or

$$\begin{tikzcd} m \arrow[r, "{[\![\mathrm{s}]\!]_\Gamma}"] \arrow[d, "\langle\cdot\rangle_r"'] & v \arrow[d, equal] \\ \langle m\rangle_r \arrow[r, dashed, "{[\![\langle \mathrm{s}\rangle_{r,\Gamma}]\!]_{\langle\Gamma\rangle_r}}"'] & v' \end{tikzcd}$$

when the algorithm terminates.

Security — Proposition

The algorithm is secure if when it returns a value it is either the right one or an error constant. It is not secure only with a probability asymptotically inversely proportional to the security parameter r.

proof sketch:

- Faulted results are polynomials of corrupted intermediate values:
  - the result can be expressed as a polynomial of the inputs and the faults,
  - a fault on x is not detected if: $P(\widehat{x}) = P(x) \bmod r$ and $P(\widehat{x}) \neq P(x) \bmod p$,
  - i.e., when $\widehat{x_1}$ is a root of $\Delta P(\widehat{x_1}) = P(\widehat{x_1}) - P(x_1)$ in $\mathbb{Z}_r$.
- Non-detection probability $\mathbb{P}_{\text{n.d.}}$ is inversely proportional to r:
  - $\mathbb{P}_{\text{n.d.}} \approx \#roots(\Delta P)/r$ in $\mathbb{Z}_r$,
  - If ΔP is uniformly distributed, when $r \to \infty$, $\#roots(\Delta P)$ tends to 1,
  - in practice $\mathbb{P}_{\text{n.d.}} \gtrsim \frac{1}{r}$, i.e., $\mathbb{P}_{\text{n.d.}} \geq \frac{1}{r}$ but is close to $\frac{1}{r}$.