Formal Software Methods for Cryptosystems Implementation Security

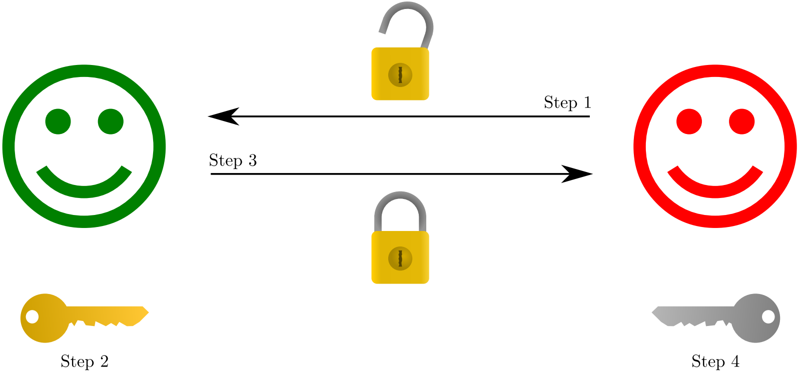
Pablo Rauzy
rauzy@enst.fr
pablo.rauzy.name

Advisor: Sylvain Guilley

PhD Defense

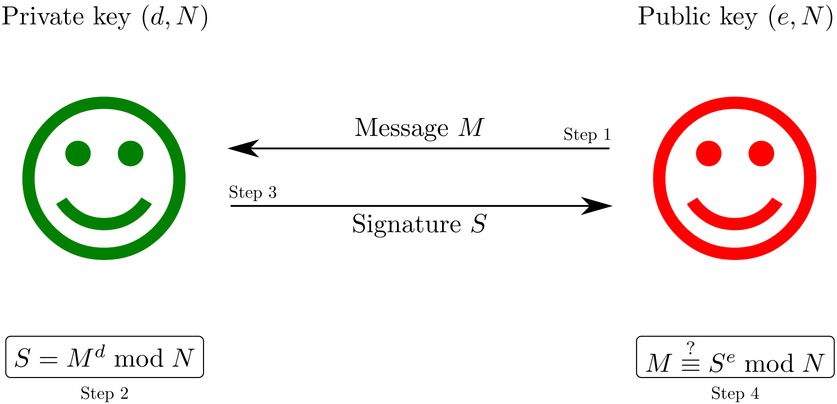
July 13, 2015 @ ENS Ulm, Paris.

Asymmetric Cryptography



The RSA Cryptosystem

$$N = p \cdot q$$
$$d \cdot e \equiv 1 \bmod \varphi(N)$$



Private key (d, N)

Public key (e, N)

Message M

Step 1

Step 3

Signature S

$$\boxed{S = M^d \bmod N}$$

Step 2

$$\boxed{M \overset{?}{\equiv} S^e \bmod N}$$

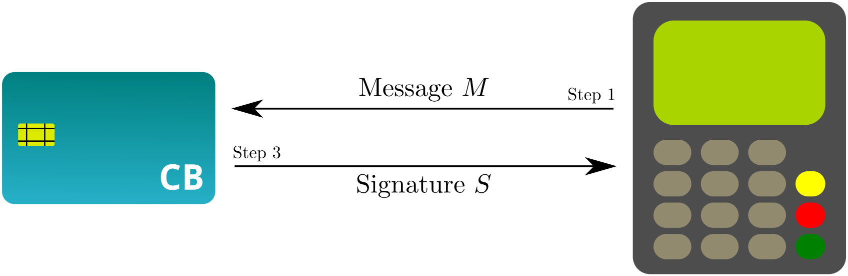
Step 4

Implementations

$$N = p \cdot q$$
$$d \cdot e \equiv 1 \bmod \varphi(N)$$

Private key (d, N)

Public key (e, N)



$\boxed{S = M^d \bmod N}$

Step 2

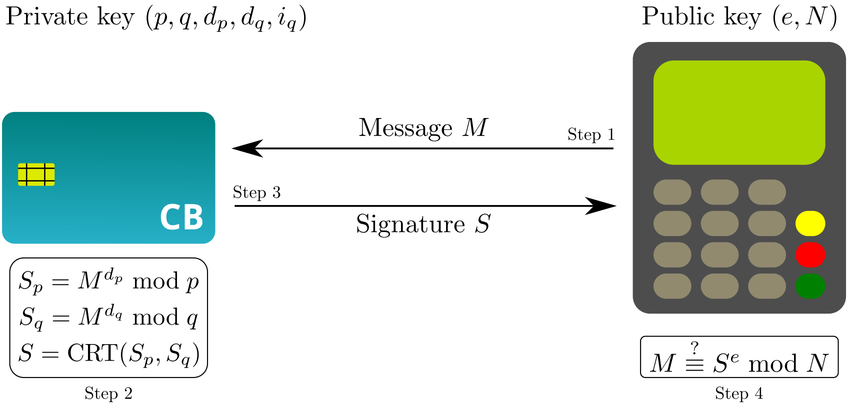
$\boxed{M \overset{?}{\equiv} S^e \bmod N}$

Step 4

CRT-RSA

$$d_p \doteq d \bmod \varphi(p)$$
$$d_q \doteq d \bmod \varphi(q)$$
$$i_q \doteq q^{-1} \bmod p$$

$$N = p \cdot q$$
$$d \cdot e \equiv 1 \bmod \varphi(N)$$



Private key (p, q, d_p, d_q, i_q)

Public key (e, N)

Message M

Step 1

Step 3

Signature S

$$\boxed{\begin{aligned} S_p &= M^{d_p} \bmod p \\ S_q &= M^{d_q} \bmod q \\ S &= \mathrm{CRT}(S_p, S_q) \end{aligned}}$$

Step 2

$$\boxed{M \overset{?}{\equiv} S^e \bmod N}$$

Step 4

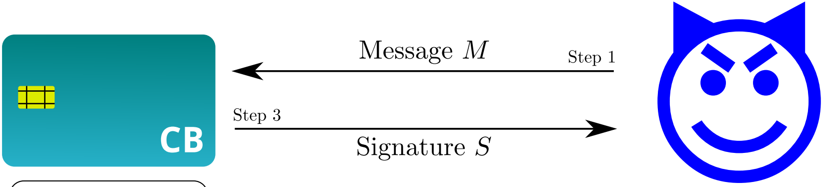
The BellCoRe Attack

$$\begin{aligned} d_p &\doteq d \bmod \varphi(p) \\ d_q &\doteq d \bmod \varphi(q) \\ i_q &\doteq q^{-1} \bmod p \end{aligned}$$

$$\begin{aligned} N &= p \cdot q \\ d \cdot e &\equiv 1 \bmod \varphi(N) \end{aligned}$$

Private key (p, q, d_p, d_q, i_q)

Public key (e, N)



Message M

Step 1

Step 3

Signature S

$$\boxed{\begin{aligned} S_p &= M^{d_p} \bmod p \\ S_q &= M^{d_q} \bmod q \\ S &= \mathrm{CRT}(S_p, S_q) \end{aligned}}$$

Step 2

$$\boxed{M \overset{?}{\equiv} S^e \bmod N}$$

Step 4

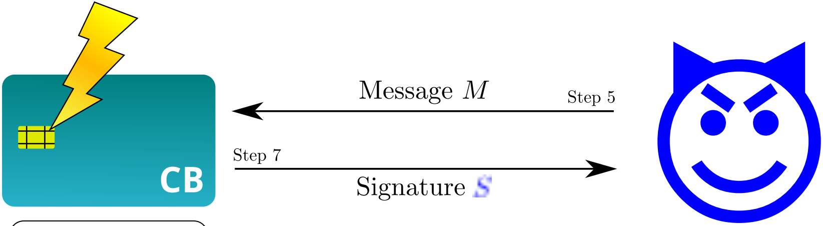
The BellCoRe Attack

$$d_p \doteq d \bmod \varphi(p)$$
$$d_q \doteq d \bmod \varphi(q)$$
$$i_q \doteq q^{-1} \bmod p$$

$$N = p \cdot q$$
$$d \cdot e \equiv 1 \bmod \varphi(N)$$

Private key (p, q, d_p, d_q, i_q)

Public key (e, N)



$$\boxed{\begin{array}{l} S_p = M^{d_p} \bmod p \\ \widehat{S_q} = M^{d_q} \bmod q \\ \widehat{S} = \mathrm{CRT}(S_p, \widehat{S_q}) \end{array}}$$

Step 6

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

Step 8

Research Questions

- Protection against this kind of attacks?

- Proof of the protection?

- Automation of the protection? Of the proof?

- Fault injection attacks:
  - introduced in 1996 by Boneh, DeMillo, and Lipton.
- Side-channel attacks:
  - introduced in 1996 by Kocher.

Countermeasures

- Since then, many countermeasures have been developed:
  - at hardware- and software-level,
  - mostly in the industry, but also in academia.
- However, virtually all of them are a matter of artisanal work:
  - there is no automation,
  - parameter choices are scarcely motivated,
  - there are seldom rigorous security proofs.

- Use tools from mathematics and theoretical computer science:
  - prove that systems respect some functional and security properties,
  - automatize the proof.

- Enable optimization:
  - security is costly,
  - mechanized proofs can be used as non-regression tests.

Implementation Security

- Formal methods are uncommon in the cryptologic community:
  - cryptology is a fast-moving field,
  - animated by many engineers and hackers.
- And even more unusual for security against physical attacks:
  - even more engineering (e.g., trial-and-error development),
  - physical behavior seemingly difficult to model.

Previous Works

- However, a few attempts have been made before I started my PhD.
- Proofs:
  - Rivain and Prouff (2010): provable masking.
  - Christofi et al. (2012): proved BellCoRe countermeasure.
- Automation:
  - Bayrak et al. (2011): automatic random precharging.
  - Moss et al. (2012): assisted masking.

Contributions

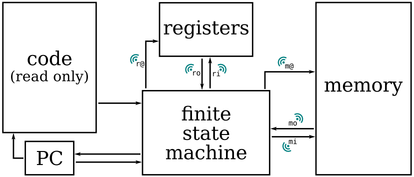
1. Proof and automation of a generic side-channel countermeasure.

2. Proof and simplifications of fault injection countermeasures.

3. Automation of a proven generic fault injection countermeasure.

1. paioli

- Proof and automation of a *balancing* countermeasure:
  - null side-channel signal-to-noise ratio,
  - i.e., constant leakage,
  - using *Dual-rail with Precharge Logic* in software.
- Automation of the (proven correct) code transformation.
- Automation of the security proof:
  - symbolic execution ensures the constant-leakage property holds.



- Formal study of CRT-RSA countermeasures against fault attacks:
    - security proofs,
    - optimization (speed, number of tests, randomness requirement),
    - symbolic evaluation by term rewriting in an arithmetic framework.
- Classification of a family of existing countermeasures:
    - extract protection principles from the employed techniques,
    - design method to resist an arbitrary (but fixed) number of faults.

3. enredo

- Extraction of the principles of CRT-RSA countermeasures:
  - integrity verification,
  - independent from the fault model,
  - independent from the algorithm,
  - applicable to all asymmetric cryptography.
- Security proof of the generic countermeasure.
- Automation of the (proved correct) code transformation:
  - protection of previously unprotected (but attacked) algorithms.

- Introduction
- Towards Generic Countermeasures Against Fault Injection Attacks
  - RSA, CRT-RSA, the BellCoRe attack, countermeasures.
  - Formal study, finja, proved countermeasures.
  - Classification of countermeasures, building better countermeasures.
  - Integrity verification, enredo, generic countermeasures.
- Conclusions and Perspectives

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).

The BellCoRe Attack

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

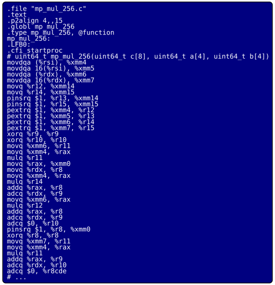
- Many countermeasures have been proposed:
  - $\sim$20 papers,
  - from 1999 to now,
  - both from academia and industry.
- Including:
  - Shamir (1999),
  - Aumüller et al. (2002),
  - Vigilant (2008) + Coron et al. (2010).

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.

$$M \to \boxed{\mathrm{RSA}} \to S$$

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.
- `http://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

contribution

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

contribution

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 $r$.
$p' = p \cdot r$
$q' = q \cdot r$
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 $M'_p$
$d'_p = d_p + R_3 \cdot (p - 1)$
$S'_p = {M'_p}^{d'_p \bmod \varphi(p')} \mod p'$                 // 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 $M'_q$
$d'_q = d_q + R_4 \cdot (q - 1)$
$S'_q = {M'_q}^{d'_q \bmod \varphi(q')} \mod q'$                 // 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 $p' = p \cdot r^2$ ; $i_{pr} = p^{-1} \bmod r^2$ ; $M_p = M \bmod p'$ ; $B_p = p \cdot i_{pr}$ ; $A_p = 1 - B_p \bmod p'$
print $M'_p = A_p \cdot M_p + B_p \cdot (1 + r) \bmod p'$
print $q' = q \cdot r^2$ ; $i_{qr} = q^{-1} \bmod r^2$ ; $M_q = M \bmod q'$ ; $B_q = q \cdot i_{qr}$ ; $A_q = 1 - B_q \bmod q'$
print $M'_q = A_q \cdot M_q + B_q \cdot (1 + r) \bmod q'$
print $S'_p = {M'_p}^{d_p \bmod \varphi(p')} \bmod p'$
print $S'_q = {M'_q}^{d_q \bmod \varphi(q')} \bmod q'$
print $S_{pr} = 1 + d_p \cdot r$
print $S_{qr} = 1 + d_q \cdot r$
print $S_r = S_{qr} + q \cdot (i_q \cdot (S_{pr} - S_{qr}) \bmod p')$
print $S' = S'_q + q \cdot (i_q \cdot (S'_p - S'_q) \bmod p')$
print $S_0 = S' \bmod N$
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 $S_3$
```

Generalization

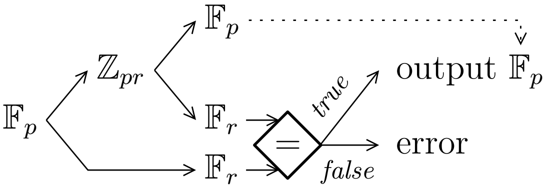
- The working factors of countermeasures:
  - are not tied to the BellCoRe attack,
  - nor to the CRT-RSA algorithm.
- Cost-effective integrity verification of any arithmetic computation.

Integrity Verification

- Obvious idea: repeat the computation and compare the results:
  - may be easy to inject the same fault twice,
  - costs too much.
- Signature verification is an RSA-specific possibility:
  - but it requires to have both parts of the key,
  - may cost too much.
- Existing countermeasures are optimizations of the redundancy idea.

Entanglement

- Use the isomorphism between $\mathbb{F}_p \times \mathbb{F}_r$ and $\mathbb{Z}_{pr}$.



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

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.
- `http://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 *entanglement* 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:

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

during the execution, or

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

when the algorithm terminates.

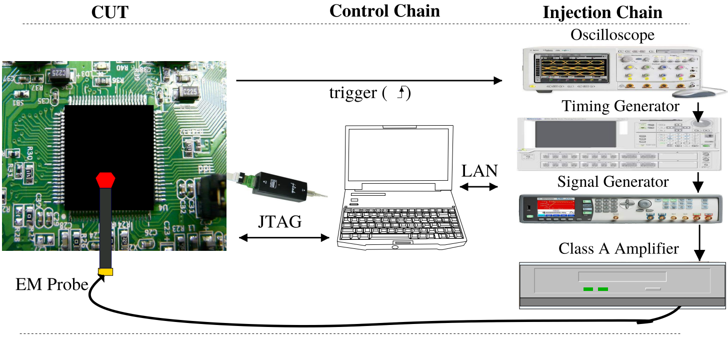
Security **Proposition**

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 $\frac{1}{r}$.

Practical Case Study with ECSM on 32-bit ARMv7



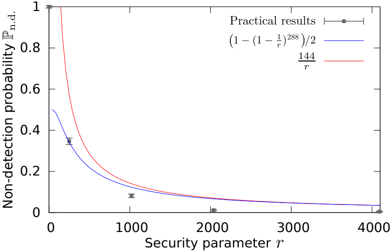
Field characteristic	$p =$ 0xfffffffffffffffffffffffffffffffeffffffffffffffff
Curve equation coefficients	$a =$ 0xfffffffffffffffffffffffffffffffefffffffffffffffc $b =$ 0x64210519e59c80e70fa7e9ab72243049feb8deecc146b9b1
Point coordinates	$Gx =$ 0x188da80eb03090f67cbf20eb43a18800f4ff0afd82ff1012 $Gy =$ 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	time (ms)			overhead
	(bit)	$\mathbb{Z}_{pr}$	$\mathbb{F}_r$	test	
1	1	683	24	$\ll 1$	×1.04
251	8	883	91	$\ll 1$	×1.43
1021	10	899	100	$\ll 1$	×1.46
2039	11	902	197	$\ll 1$	×1.61
4093	12	903	197	$\ll 1$	×1.61
65521	16	883	189	$\ll 1$	×1.56
4294967291	32	832	172	$\ll 1$	×1.47
18446744073709551557	64	996	246	$\ll 1$	×1.82

Signature verification overhead $\approx \times 4.5$.

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

Conclusions

- CRT-RSA implementations can provably resist the BellCoRe attack.
- CRT-RSA implementations can provably resist multiple faults.
- CRT-RSA implementations can provably resist new fault attacks.
- Any asymmetric cryptography implementations can be protected.

Conclusions

Aumuller: OK

- CRT-RSA implementations can provably resist the BellCoRe attack.
- CRT-RSA implementations can provably resist multiple faults.
- CRT-RSA implementations can provably resist new fault attacks.
- Any asymmetric cryptography implementations can be protected.

Conclusions

Aumuller: OK

Vigilant: OK

- CRT-RSA implementations can provably resist the BellCoRe attack.
- CRT-RSA implementations can provably resist multiple faults.
- CRT-RSA implementations can provably resist new fault attacks.
- Any asymmetric cryptography implementations can be protected.

Conclusions

Aumuller: OK

Vigilant: OK

- CRT-RSA implementations can provably resist the BellCoRe attack.

VALIDATED

- CRT-RSA implementations can provably resist multiple faults.
- CRT-RSA implementations can provably resist new fault attacks.
- Any asymmetric cryptography implementations can be protected.

Conclusions

Aumuller: OK

Shamir: FIXED

Vigilant: OK

VALIDATED

- CRT-RSA implementations can provably resist the BellCoRe attack.
- CRT-RSA implementations can provably resist multiple faults.
- CRT-RSA implementations can provably resist new fault attacks.
- Any asymmetric cryptography implementations can be protected.

Aumuller: OK

Shamir: FIXED

Vigilant: OK

- CRT-RSA implementations can provably resist the BellCoRe attack.

VALIDATED

- CRT-RSA implementations can provably resist multiple faults.
- CRT-RSA implementations can provably resist new fault attacks.
- Any asymmetric cryptography implementations can be protected.

HIGH-ORDER: BRING IT ON!

Conclusions

Aumuller: OK

Shamir: FIXED

Vigilant: OK

- CRT-RSA implementations can provably resist the BellCoRe attack.

VALIDATED

- CRT-RSA implementations can provably resist multiple faults.

VALIDATED

- CRT-RSA implementations can provably resist new fault attacks.
- Any asymmetric cryptography implementations can be protected.

HIGH-ORDER: BRING IT ON!

Conclusions

Aumuller: OK

Shamir: FIXED

BellCoRe: OUT

Vigilant: OK

- CRT-RSA implementations can provably resist the BellCoRe attack.

VALIDATED

- CRT-RSA implementations can provably resist multiple faults.

VALIDATED

- CRT-RSA implementations can provably resist new fault attacks.
- Any asymmetric cryptography implementations can be protected.

HIGH-ORDER: BRING IT ON!

Conclusions

Aumuller: OK

Shamir: FIXED

BellCoRe: OUT

Vigilant: OK

- CRT-RSA implementations can provably resist the BellCoRe attack.

VALIDATED

- CRT-RSA implementations can provably resist multiple faults.

VALIDATED

- CRT-RSA implementations can provably resist new fault attacks.
- Any asymmetric cryptography implementations can be protected.

NEW ATTACKS: WHO'S NEXT?

HIGH-ORDER: BRING IT ON!

Aumuller: OK

Shamir: FIXED

BellCoRe: OUT

Vigilant: OK

- CRT-RSA implementations can provably resist the BellCoRe attack.
- CRT-RSA implementations can provably resist multiple faults.
- CRT-RSA implementations can provably resist new fault attacks.
- Any asymmetric cryptography implementations can be protected.

VALIDATED

VALIDATED

VALIDATED

NEW ATTACKS: WHO'S NEXT?

HIGH-ORDER: BRING IT ON!

Conclusions

Aumuller: OK

Shamir: FIXED

BellCoRe: OUT

Vigilant: OK

- CRT-RSA implementations can provably resist the BellCoRe attack.

VALIDATED

- CRT-RSA implementations can provably resist multiple faults.

VALIDATED

- CRT-RSA implementations can provably resist new fault attacks.

VALIDATED

- Any asymmetric cryptography implementations can be protected.

ECSM: PROTECTED

NEW ATTACKS: WHO'S NEXT?

HIGH-ORDER: BRING IT ON!

Conclusions

Aumuller: OK

Shamir: FIXED

BellCoRe: OUT

Vigilant: OK

- CRT-RSA implementations can provably resist the BellCoRe attack.

VALIDATED

- CRT-RSA implementations can provably resist multiple faults.

VALIDATED

- CRT-RSA implementations can provably resist new fault attacks.

VALIDATED

- Any asymmetric cryptography implementations can be protected.

ECSM: PROTECTED

PAIRING: SOON

NEW ATTACKS: WHO'S NEXT?

HIGH-ORDER: BRING IT ON!

Conclusions

AUMULLER: OK

SHAMIR: FIXED

BELLCORE: OUT

VIGILANT: OK

- CRT-RSA implementations can provably resist the BellCoRe attack.

VALIDATED

- CRT-RSA implementations can provably resist multiple faults.

VALIDATED

- CRT-RSA implementations can provably resist new fault attacks.

VALIDATED

- Any asymmetric cryptography implementations can be protected.

VALIDATED

ECSM: PROTECTED

PAIRING: SOON

NEW ATTACKS: WHO'S NEXT?

HIGH-ORDER: BRING IT ON!

- Articles in conferences with proceedings:
  - PPREW 2014, FDTC 2014, HOST 2015.
- Articles in journals
  - Journal of Cryptographic Engineering ($\times 2$).
- Communications in workshops without proceedings:
  - COSADE 2013 (short paper + talk),
  - PROOFS 2013 and 2014 (full papers + talks),
  - CHES 2013 and 2015 (posters),
  - TRUDEVICE 2015 (short paper + talk),
  - talks: DigiCosme working group, GDR SoC-SiP Security Day, Formal Methods and Security seminar Inria/DGA, Crypto'Spain Itinerant Seminar, SAS deptartment seminar @ EMSE, ...
- Two articles under submission and one book chapter in progress.

Perspectives

- Direct follow-up.
- Formalize security beyond cryptology.

The End

Questions?

Classification of CRT-RSA Countermeasures in Shamir's Family

1. Shamir's or Giraud's Family of Countermeasures

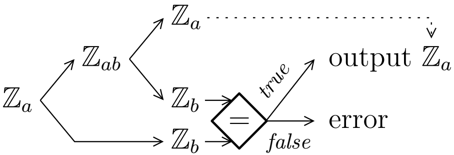
- Two main families of countermeasures:
  - descendants of Giraud's countermeasure (2006),
  - descendants of Shamir's countermeasure (1999).

1.1. Giraud's Family

- Use particular exponentiation algorithms that:
  - keep track of variables involved in intermediate steps,
  - check an invariant that is supposed to be spread till the last steps.
- Examples of countermeasures in this family include:
  - Boscher et al. (2007),
  - Rivain (2009) (and its recently improved version from 2014),
  - Kim et al. (2011).
- Study of Giraud's family countermeasures was left as future work:
  - Ágnes Kiss did it with Juliane Krämer at TU Berlin,
  - we co-authored a paper (to be published).

1.2. Shamir's Family

- Use a "checksum" in a smaller mathematical structure:
    - RSA computes in rings $\mathbb{Z}_a$ (typically, $a = p$, $a = q$, or $a = pq$),
    - any small number b is coprime with a,
    - there is an isomorphism between the *direct product* $\mathbb{Z}_{ab}$ and $\mathbb{Z}_a \times \mathbb{Z}_b$,
    - original computation and checksum can be conducted together in $\mathbb{Z}_{ab}$,
    - redundant checksum can be computed in parallel in $\mathbb{Z}_b$.
- Verify that invariants on the computations and the checksums hold.



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

2. Test-Based or Infective Countermeasures

- Invariant verification can be done in two ways:
  - step-wise internal checks during the CRT computation,
  - computationally, using an infective computation strategy.

Test-based countermeasure

Definition

A countermeasure is said to be *test-based* if it attempts to detect fault injections by verifying that some arithmetic invariants are respected, and branch to return an error instead of the numerical result of the algorithm in case of invariant violation.

- Examples of test-based countermeasures:
  - Shamir (1999),
  - Joye et al. (2001),
  - Aumüller et al. (2002),
  - Vigilant (2008).

Infective countermeasure

Definition

A countermeasure is said to be *infective* if rather than testing arithmetic invariants it uses them to compute a neutral element of some arithmetic operation in a way that would not result in this neutral element if the invariant is violated.
It then uses the results of these computations to infect the result of the algorithm before returning it to make it unusable by the attacker (thus, it does not need branching instructions).

- Examples of infective countermeasures:
  - Blömer et al. (2003),
  - Ciet & Joye (2005),
  - Kim et al. (2011).

Equivalence between test-based and infective verification **Proposition**

Each test-based (resp. infective) countermeasure has a direct equivalent infective (resp. test-based) countermeasure.

proof sketch:

- Invariants verified by countermeasures are modular equality:
  - $a \overset{?}{\equiv} b \mod m$.
  - Test-based: `if a != b [mod m] then return error`.
  - Infective: `c := a - b + 1 mod m; ... return S`$^{\texttt{c}}$.

3. Intended Order

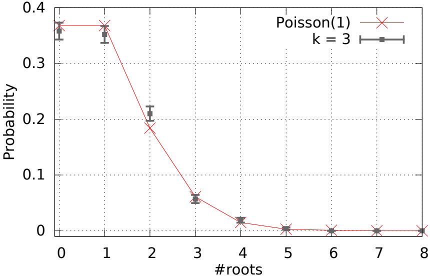
- In our fault model, all existing countermeasures:
  - resist any number of randomizing faults,
  - can be broken by a second-order attack:
    a well targeted fault injection, and
    a verification skip (e.g., using a skipping or a zeroing fault).
- The concept of integrity verification does not depend on the order.

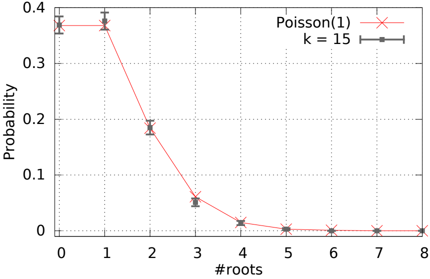
4. Usage of the Small Subrings

- In most countermeasures:
  - computations in $\mathbb{Z}_p$ and $\mathbb{Z}_q$ are lifted in $\mathbb{Z}_{pr_1}$ and $\mathbb{Z}_{qr_2}$,
  - allowing the retrieval of the results modulo p and q, and
  - the verification of the signature modulo r_1 and r_2.
- The following questions were raised:
  - Are the smaller rings used to verify the intermediate signatures?
  - Or are they used directly to compute checksums that are verified?
  - Does CRT recombination takes place in an overring?
  - When $r_1 = r_2$, what is permitted by the resulting symmetry?

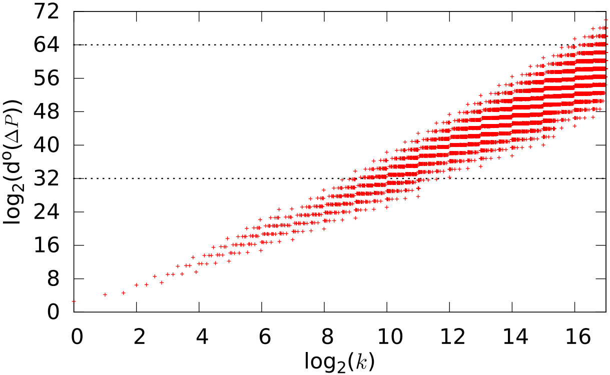
Polynomials of Faults

#roots probability for ECSM $[k]G$.





Degree of the polynomial ΔP against the value of k (in log-log scale)



Formally Proved Security of Assembly Code Against Power Analysis

Dual-rail with Precharge Logic

- The DPL countermeasure consists in computing on a redundant representation: each bit y is implemented as a pair $(y_{\mathsf{False}}, y_{\mathsf{True}})$.

- 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 $(y_{\mathsf{False}}, y_{\mathsf{True}}) = (0, 0)$, such that the computation starts from a known reference state;
  2. an *evaluation* phase, during which the $(y_{\mathsf{False}}, y_{\mathsf{True}})$ pair is equal to $(1, 0)$ if it carries the logical value 0, or $(0, 1)$ if it carries the logical value 1.

DPL Macro

- Each sensitive instruction should replaced by a *DPL macro*.
- The DPL macro assumes that the system is in a valid DPL state.
- And leaves it in a valid DPL state to make the macros chainable.

- The basic idea is to concatenate two DPL encoded values.
- Then use the result as an index in a look-up table.

Example Using the Two Least Significant Bit

- In this example we use the two LSB.
- Logical value 1 is 1 (`01`).
- Logical value 0 is 2 (`10`).
- Precharge phases (activity: 1 if sensitive)
- Evaluation phases (activity: 1)
- Masks (activity: normally 0)
- Shifts (activity: 2)
- Concatenation (activity: 1)
- Look-up (activity: $1 + 2$)

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

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

Example Using the Two Least Significant Bit

- In this example we use the two LSB.
- Logical value 1 is 1 ($\texttt{01}$).
- Logical value 0 is 2 ($\texttt{10}$).

- Precharge phases (activity: 1 if sensitive)
- Evaluation phases (activity: 1)
- Masks (activity: normally 0)
- Shifts (activity: 2)
- Concatenation (activity: 1)
- Look-up (activity: 1 + 2)

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

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

Example Using the Two Least Significant Bit

- In this example we use the two LSB.
- Logical value 1 is 1 (`01`).
- Logical value 0 is 2 (`10`).

- Precharge phases (activity: 1 if sensitive)
- Evaluation phases (activity: 1)
- Masks (activity: normally 0)
- Shifts (activity: 2)
- Concatenation (activity: 1)
- Look-up (activity: 1 + 2)

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

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

Example Using the Two Least Significant Bit

- In this example we use the two LSB.
- Logical value 1 is 1 (`01`).
- Logical value 0 is 2 (`10`).

- Precharge phases (activity: 1 if sensitive)
- Evaluation phases (activity: 1)
- Masks (activity: normally 0)
- Shifts (activity: 2)
- Concatenation (activity: 1)
- Look-up (activity: $1 + 2$)

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

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

Example Using the Two Least Significant Bit

- In this example we use the two LSB.
- Logical value 1 is 1 (`01`).
- Logical value 0 is 2 (`10`).

- Precharge phases (activity: 1 if sensitive)
- Evaluation phases (activity: 1)
- Masks (activity: normally 0)
- Shifts (activity: 2)
- Concatenation (activity: 1)
- Look-up (activity: 1 + 2)

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

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

Example Using the Two Least Significant Bit

- In this example we use the two LSB.
- Logical value 1 is 1 ($\texttt{01}$).
- Logical value 0 is 2 ($\texttt{10}$).
- Precharge phases (activity: 1 if sensitive)
- Evaluation phases (activity: 1)
- Masks (activity: normally 0)
- Shifts (activity: 2)
- Concatenation (activity: 1)
- Look-up (activity: 1 + 2)

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

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

Example Using the Two Least Significant Bit

- In this example we use the two LSB.
- Logical value 1 is 1 (`01`).
- Logical value 0 is 2 (`10`).

- Precharge phases (activity: 1 if sensitive)
- Evaluation phases (activity: 1)
- Masks (activity: normally 0)
- Shifts (activity: 2)
- Concatenation (activity: 1)
- Look-up (activity: 1 + 2)

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

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

Code Transformation

1. Bitslice code.
2. DPL macros expansion.
3. Look-up tables.

1. Bitslicing Code

- Always possible (by Turing machines equivalence theorem)
- But, hard to do automatically in practice.
- However, there are a lot of already (manually) bitsliced implementations, since it is a common optimization technique.

→ We take already bitsliced code as input.

2.1. Sensitive Instructions

Sensitive value **Definition**

A *value* is said *sensitive* if it depends on sensitive data. A sensitive data depends on the secret key or the plaintext.

Sensitive instruction **Definition**

An *instruction* is said *sensitive* if it may modify the Hamming weight of a sensitive value.

- All the sensitive instructions must be expanded to a DPL macro.
- Thus, all the sensitive data must be transformed too.

2.2. Which Instructions are Sensitive?

- Bitsliced code means that only the logical (bit level) operators, except shifts, are used in sensitive instructions.
- DPL protocol implies that `not` instructions are replaced by `xor`.

→ Only `and`, `or`, and `xor` instructions need to be expanded to DPL macros.

3. Look-Up Tables

- Addresses of the look-up tables are sensitive too: their indices are sensitive values.
- Thus, the addresses bits corresponding to the accessed cell must be 0.
- In our example, the look-up table addresses must be multiple of 16.

index	0000	0001	0010	0011	0100	0101	0110	0111
and	00	00	00	00	00	01	10	00
or	00	00	00	00	00	01	01	00
xor	00	00	00	00	00	10	01	00

index	1000	1001	1010	1011	1100	1101	1110	1111
and	00	10	10	00	00	00	00	00
or	00	01	10	00	00	00	00	00
xor	00	01	10	00	00	00	00	00

3. Look-Up Tables

- Addresses of the look-up tables are sensitive too: their indices are sensitive values.
- Thus, the addresses bits corresponding to the accessed cell must be 0.
- In our example, the look-up table addresses must be multiple of 16.

index	0000	0001	0010	0011	0100	0101	0110	0111
and	00	00	00	00	00	01	10	00
or	00	00	00	00	00	01	01	00
xor	00	00	00	00	00	10	01	00

index	1000	1001	1010	1011	1100	1101	1110	1111
and	00	10	10	00	00	00	00	00
or	00	01	10	00	00	00	00	00
xor	00	01	10	00	00	00	00	00

Correctness Proof of the Transformation

Correct DPL transformation

Definition

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 equivalent to the state S.

We say that c' is a *correct DPL transformation* of the code c if $S \xrightarrow{c} \widehat{S} \implies dpl(S) \xrightarrow{c'} dpl(\widehat{S})$.

Correctness of our code transformation

Proposition

The expansion of the sensitive instructions into DPL macros is a correct DPL transformation.

- Proof in the paper.

- Example execution for `and`.

a, b	10, 10				Sensitive activity
	d	r1	r2	r3	
mov r1 r0	?	0	?	?	0
mov r1 a	?	10	?	?	1
and r1 r1 #3	?	10	?	?	0
shl r1 r1 #1	?	100	?	?	2
shl r1 r1 #1	?	1000	?	?	2
mov r2 r0	?	1000	0	?	0
mov r2 b	?	1000	10	?	1
and r2 r2 #3	?	1000	10	?	0
orr r1 r1 r2	?	1010	10	?	1
mov r3 r0	?	1010	10	0	0
mov r3 !r1, *and*	?	1010	10	10	3
mov d r0	0	1010	10	10	0
mov d r3	10	1010	10	10	1

- Example execution for `and`.

a, b		10, 01			Sensitive activity
	d	`r1`	`r2`	`r3`	
`mov r1 r0`	?	0	?	?	0
`mov r1` a	?	10	?	?	1
`and r1 r1 #3`	?	10	?	?	0
`shl r1 r1 #1`	?	100	?	?	2
`shl r1 r1 #1`	?	1000	?	?	2
`mov r2 r0`	?	1000	0	?	0
`mov r2` b	?	1000	01	?	1
`and r2 r2 #3`	?	1000	01	?	0
`orr r1 r1 r2`	?	1001	01	?	1
`mov r3 r0`	?	1001	01	0	0
`mov r3 !r1,`*and*	?	1001	01	10	3
`mov` d `r0`	0	1001	01	10	0
`mov` d `r3`	10	1001	01	10	1

- Example execution for `and`.

a, b	01, 10				Sensitive activity
	d	r1	r2	r3	
mov r1 r0	?	0	?	?	0
mov r1 a	?	01	?	?	1
and r1 r1 #3	?	01	?	?	0
shl r1 r1 #1	?	010	?	?	2
shl r1 r1 #1	?	0100	?	?	2
mov r2 r0	?	0100	0	?	0
mov r2 b	?	0100	10	?	1
and r2 r2 #3	?	0100	10	?	0
orr r1 r1 r2	?	0110	10	?	1
mov r3 r0	?	0110	10	0	0
mov r3 !r1, *and*	?	0110	10	10	3
mov d r0	0	0110	10	10	0
mov d r3	10	0110	10	10	1

- Example execution for `and`.

a, b	01, 01				Sensitive activity
	d	r1	r2	r3	
`mov r1 r0`	?	0	?	?	0
`mov r1` a	?	01	?	?	1
`and r1 r1 #3`	?	01	?	?	0
`shl r1 r1 #1`	?	010	?	?	2
`shl r1 r1 #1`	?	0100	?	?	2
`mov r2 r0`	?	0100	0	?	0
`mov r2` b	?	0100	01	?	1
`and r2 r2 #3`	?	0100	01	?	0
`orr r1 r1 r2`	?	0101	01	?	1
`mov r3 r0`	?	0101	01	0	0
`mov r3 !r1,`*and*	?	0101	01	01	3
`mov` d `r0`	0	0101	01	01	0
`mov` d `r3`	01	0101	01	01	1

Computed Proof of Constant Activity

- Our tool does this verification automatically for the whole program.
- It uses *symbolic computations* to keep track of possible leakages.

- The strategy is to *simulate* a CPU and memory in software, and *compute with sets* of values.
- Initially, all sensitive data values can be either 0 or 1.
- At each cycle and for each possible combination of actual values:
  - it looks at the Hamming weight of values and Hamming distance of updates in registers, memory, and addresses; and
  - if one can have different values, it reports a leak.

- This verification is independent from the code transformation.

Cost of the Countermeasure

	bitslice	DPL	cost
code (B)	1620	3056	$\times 1.88$
RAM (B)	288	352	$+64$
#cycles	$78,403$	$235,427$	$\times 3$

DPL cost.

Attacks

- We attacked three implementations:
  - a bitsliced but unprotected one,
  - a DPL protected one using the two less significant bits,
  - a DPL protected one taking the hardware characterization into account.
- We took $100,000$ execution traces.
- We computed the success rate of using $monobit\ CPA$ of the output of the S-Box as a model.

Results

- The unprotected implementation breaks using about 400 traces.
- The poorly balanced one is still not broken using $100,000$ traces.

→ But we want to show that the hardware characterization is beneficial!

- Let's make the attacker "cheat".
- We used our knowledge of the key to select a narrow part of the traces where we knew that the attack would work.
- We used the NICV to select the point where the signal-to-noise ratio of the CPA attack is the highest.

Results

- The unprotected implementation breaks using about 400 traces.
- The poorly balanced one is still not broken using $100,000$ traces.

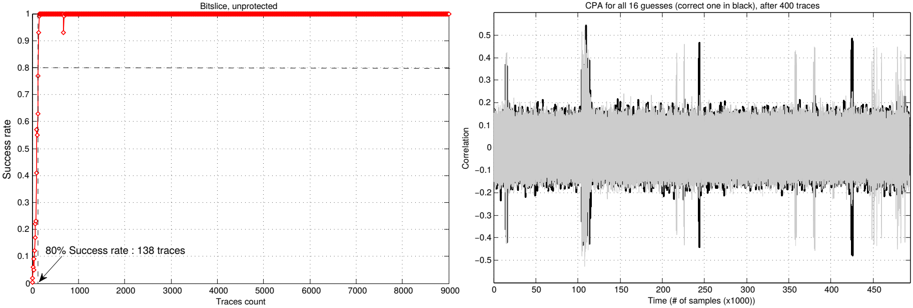
$\rightarrow$ But we want to show that the hardware characterization is beneficial!

- Let's make the attacker "cheat".
- We used our knowledge of the key to select a narrow part of the traces where we knew that the attack would work.
- We used the NICV to select the point where the signal-to-noise ratio of the CPA attack is the highest.

Results for the “Cheating Attacker”

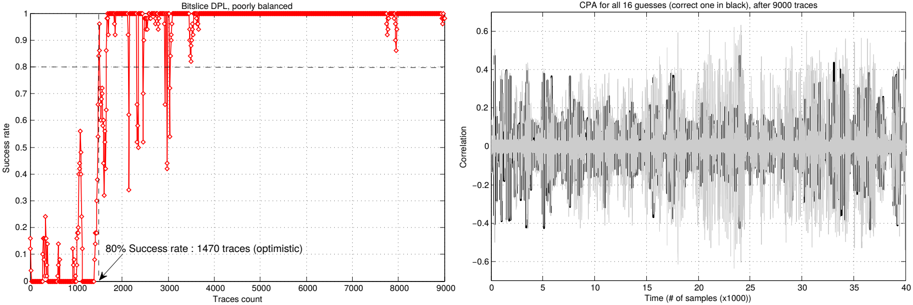
- The unprotected implementation breaks using 138 traces.
- The poorly balanced one breaks using $1,470$ traces.
- The better balanced one breaks using $4,810$ traces.

Results for the “Cheating Attacker”: unprotected



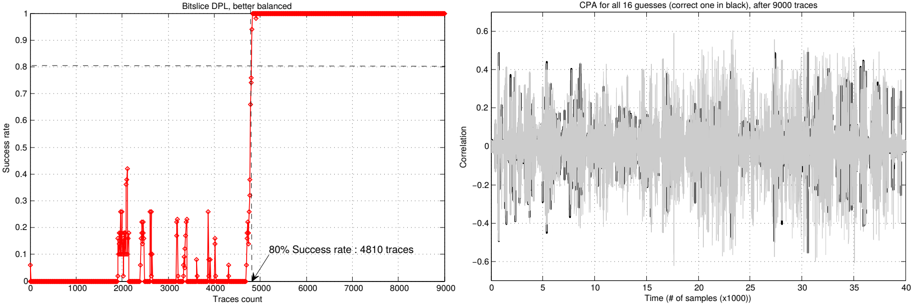
Monobit CPA attack on unprotected bitslice implementation.

Results for the “Cheating Attacker”: poorly balanced



Monobit CPA attack on poorly balanced DPL implementation (bits 0 and 1).

Results for the “Cheating Attacker”: better balanced



Monobit CPA attack on better balanced DPL implementation (bits 1 and 2).