Using Linear Codes as a Fault Countermeasure for Non-Linear Operations : Application to AES and Formal Verification

work co-funded by the PRINCE project

Sabine Azzi, Bruno Barras,
Maria Christofi, David Vigilant

PROOFS workshop 2015, September 17th

Introduction

Motivation

- Is it possible to design a countermeasure for embedded devices based on linear codes to protect linear (obviously yes) and non-linear parts of a block cipher ?
- Can we formally verify it ?

Our Contribution

- Not yet studied for the non-linear operations (such as substitution step)
  $\longrightarrow$ This is discussed in this paper !
- Willingness to get a formal verification of AES based on linear code as a fault countermeasure.

1 Introduction

- Existing techniques
- Recent interest in using linear codes to protect block ciphers
- Linear codes well suited for software implementation

Existing Techniques to Protect Block Ciphers Against Fault Attacks

- **Time redundancy :** The algorithm itself is not modified. The whole algorithm or some parts of it are executed several times sequentially, and it is verified that the replayed computations lead to the same results.
- **Hybrid redundancy :** The consistency is verified in its context. For example, verifying the encryption result can be done by deciphering the result and verify that the original plaintext is recovered after decryption.
- **Information redundancy :** Add some duplication of the information which allows to detect any modification of any part of the data, with a consistency check with its duplication part.

Existing Techniques to Protect Block Ciphers Against Fault Attacks

- **Time redundancy :** The algorithm itself is not modified. The whole algorithm or some parts of it are executed several times sequentially, and it is verified that the replayed computations lead to the same results.
- **Hybrid redundancy :** The consistency is verified in its context. For example, verifying the encryption result can be done by deciphering the result and verify that the original plaintext is recovered after decryption.
- **Information redundancy : Add some duplication of the information which allows to detect any modification of any part of the data, with a consistency check with its duplication part.**

1 Introduction

- Existing techniques
- Recent interest in using linear codes to protect block ciphers
- Linear codes well suited for software implementation

Recent Interest in Using Linear Codes to Protect Block Ciphers

- **Side-Channel and specific for AES :** ”A New Masking Scheme for Side-Channel Protection of AES” (Bringer et al. - 2012)
- **Bloc cipher generic : Fault Attacks and Side-Channel :** ”Orthogonal Direct Sum Masking” (Bringer et al. - 2014)
- **Bloc cipher generic : Side-Channel Resistance study :** ”Complementary Dual Codes for Counter-measures to Side-Channel Attacks (Carlet et al. - 2015)

1 Introduction

- Existing techniques
- Recent interest in using linear codes to protect block ciphers
- Linear codes well suited for software implementation

Some Systematic Linear Codes Operations can be Implemented Efficiently in Software

- For systematic codes, data x is represented by $x||Gx$
- e.g. $C[16, 8, 5]$
  - Decoded data representation : 1 byte
  - Encoded data representation : 2 byte
  - Redundancy part generation is a lookup table (256 bytes)
  - Verification that $x||Gx$ is in C is a lookup table (256 bytes)

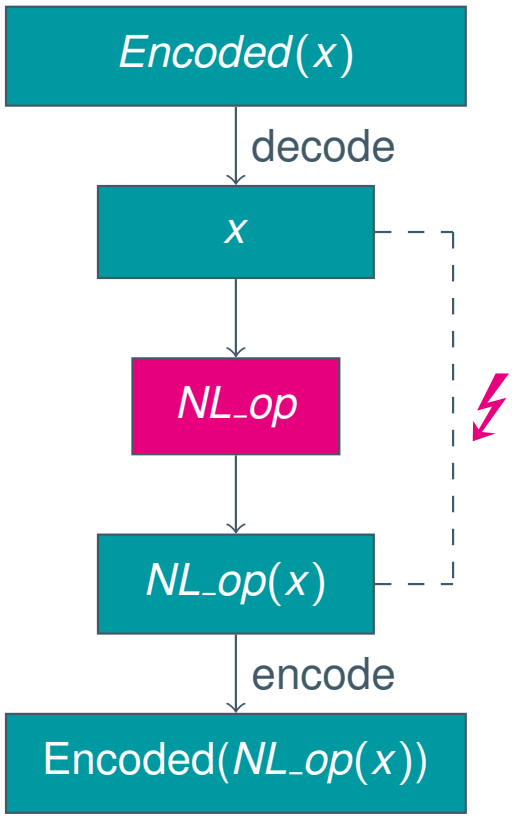
2 Contribution

- Study of the usage of systematic linear codes for non-linear operations of block ciphers
- Formal Verification Methodology
- AES case study implementation and formal verification

Linear Codes and Non Linear Operations

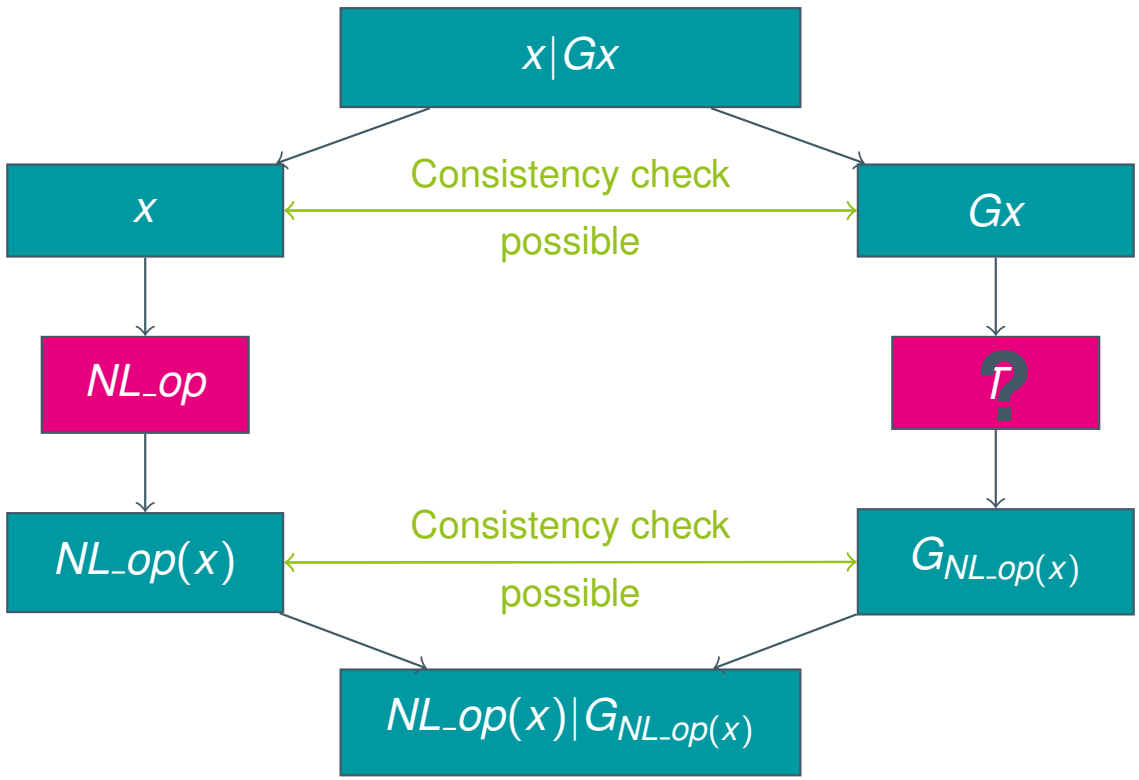
- Open question in cited papers.
- Common technique : Decode data before the non linear step and re-encoded it after it.
- decoded data for the non linear step $\Rightarrow$ the fault resistance is significantly decreased
- This paper
  - studies the fault resistance especially during non linear operations
  - proposes a formal verification of a block cipher implementation with a countermeasure based on linear codes

State of the art on constrained devices (not enough room for a lookup table with encoded entries).



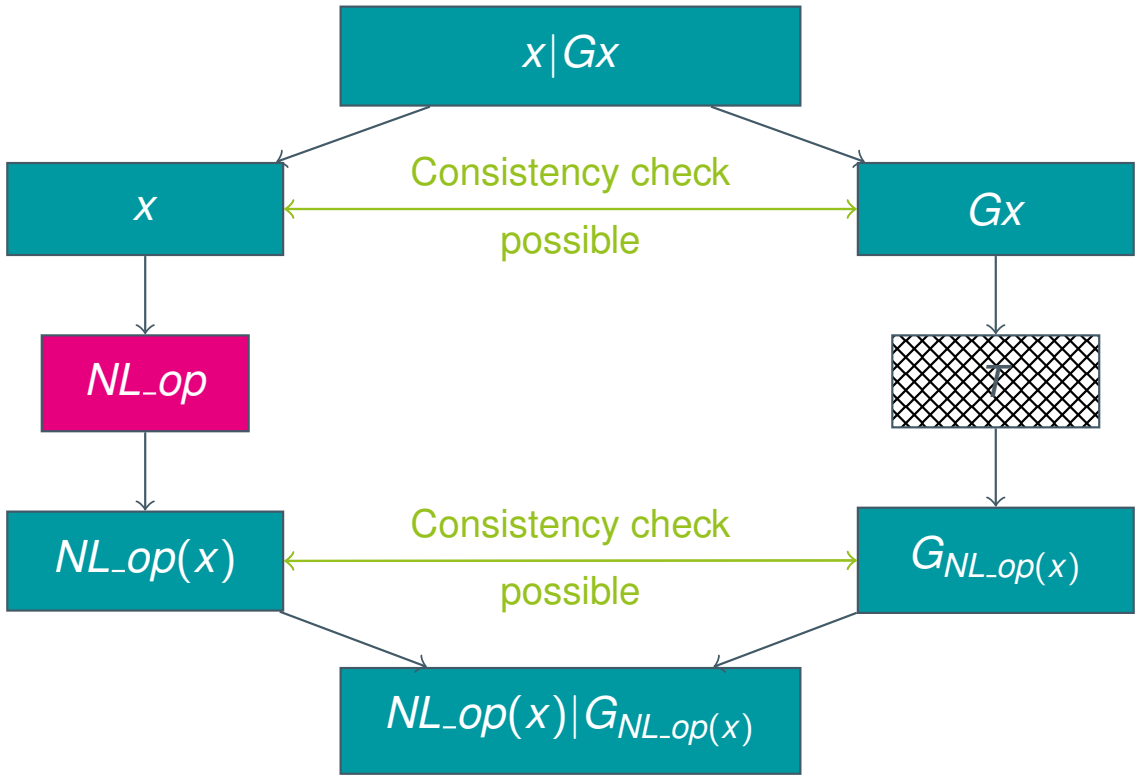
Systematic Linear Codes and Non Linear Operations

Systematic codes may be interesting :
2 lookup tables



But Gx may not determine uniquely x.

Example : $C[16, 8, 5]$ and Non Linear Operations

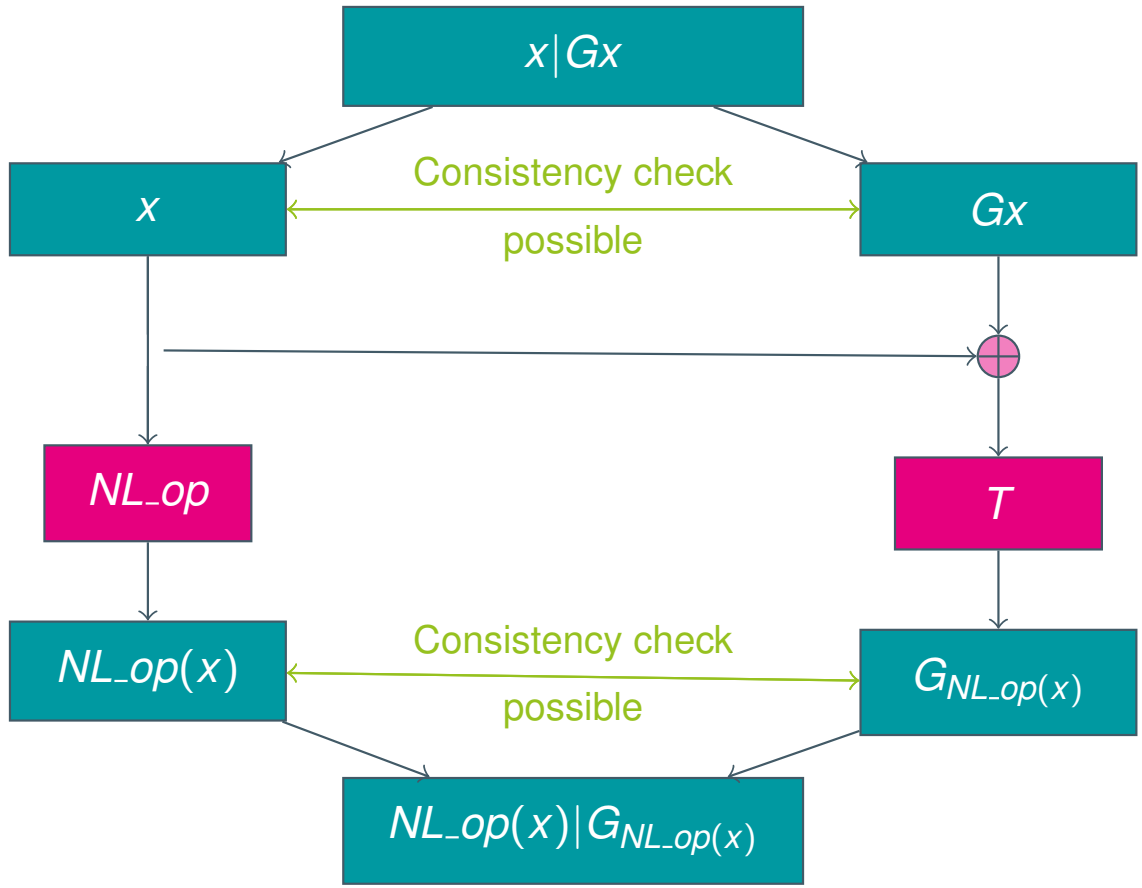


Gx does not determine uniquely x.

Example : $C[16, 8, 5]$ and Non Linear Operations

$C[16, 8, 5]$:

- has an orthogonal/complementary code
- Gx does not determine uniquely x
- BUT... $Gx \oplus x$ determines uniquely x



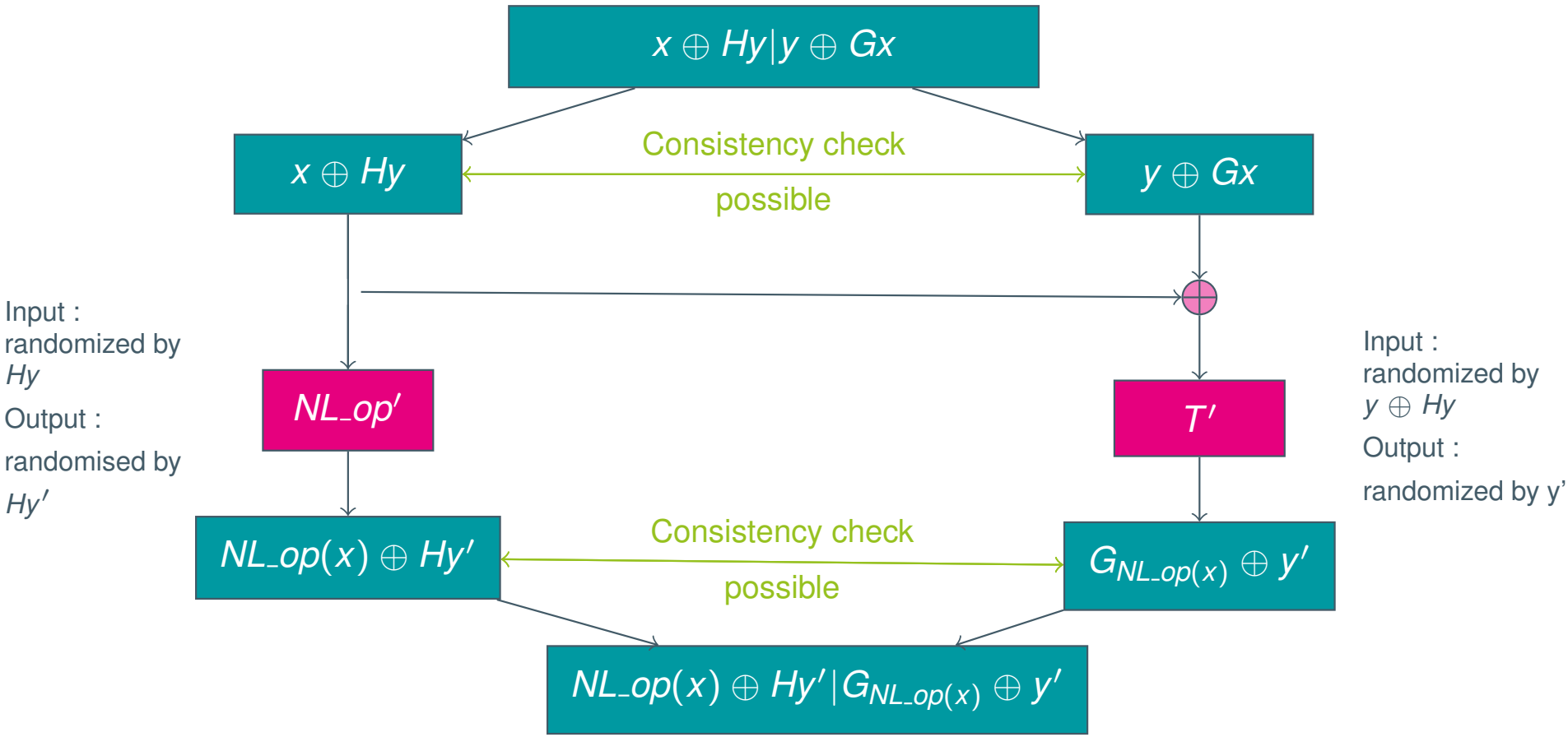
Method Applications/Generalisations

The paper discusses how this approach can be :

- Generalized for all systematic linear codes with a square generator matrix (or concatenation of square matrices)
- Applied whatever the non linear operation (for all block ciphers)
- Combined with masking methods to prevent side channel attacks
- Applied to the orthogonal sum code technique

Example : C[16,8,5], Complementary and Non Linear Operations

C[16,8,5] has an orthogonal/complementary code C2 with a generator matrix H



2 Contribution

- Study of the usage of systematic linear codes for non-linear operations of block ciphers
- Formal Verification Methodology
- AES case study implementation and formal verification

Motivation

Many software countermeasures presented to thwart attacks ...

Motivation

Many software countermeasures presented to thwart attacks ...

... which are ”quickly” broken.

Motivation

Many software countermeasures presented to thwart attacks ...

... which are ”quickly” broken.

Their security has to be verified ...

Motivation

Many software countermeasures presented to thwart attacks ...

... which are ”quickly” broken.

Their security has to be verified ...

... but it is costly.

Motivation

Many software countermeasures presented to thwart attacks ...

... which are ”quickly” broken.

Their security has to be verified ...

... but it is costly.

Use tools from mathematics and theoretical computer science...

Motivation

Many software countermeasures presented to thwart attacks ...

... which are ”quickly” broken.

Their security has to be verified ...

... but it is costly.

Use tools from mathematics and theoretical computer science...

...provide mechanized proofs that can be used as non-regression tests.

Motivation

Many software countermeasures presented to thwart attacks ...

... which are ”quickly” broken.

Their security has to be verified ...

... but it is costly.

Use tools from mathematics and theoretical computer science...

...provide mechanized proofs that can be used as non-regression tests.

So ... formal methods should be used to prove that systems respect some functional and security properties.

Motivation

Many software countermeasures presented to thwart attacks ...

... which are ”quickly” broken.

Their security has to be verified ...

... but it is costly.

Use tools from mathematics and theoretical computer science...

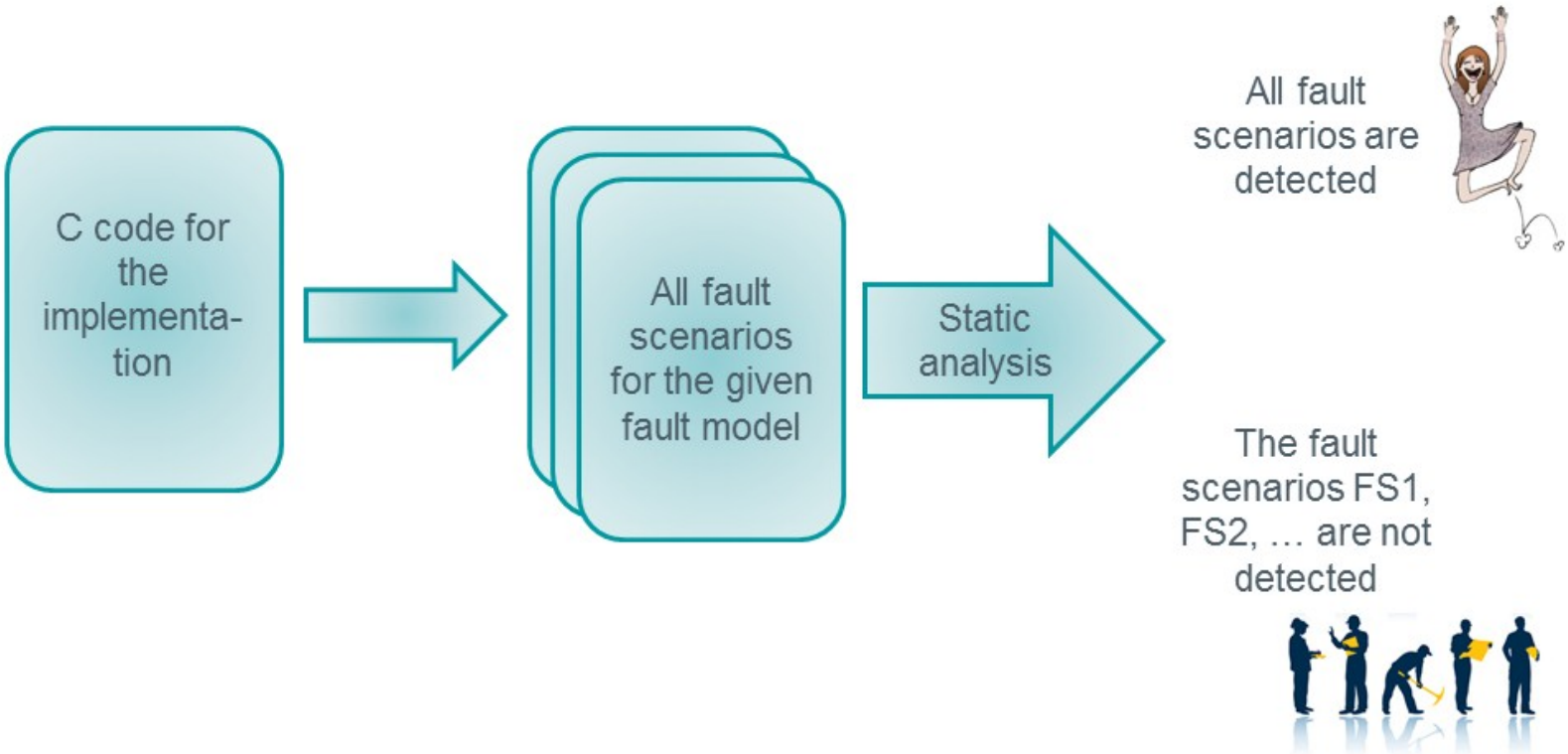
...provide mechanized proofs that can be used as non-regression tests.

So ... formal methods should be used to prove that systems respect some functional and security properties.

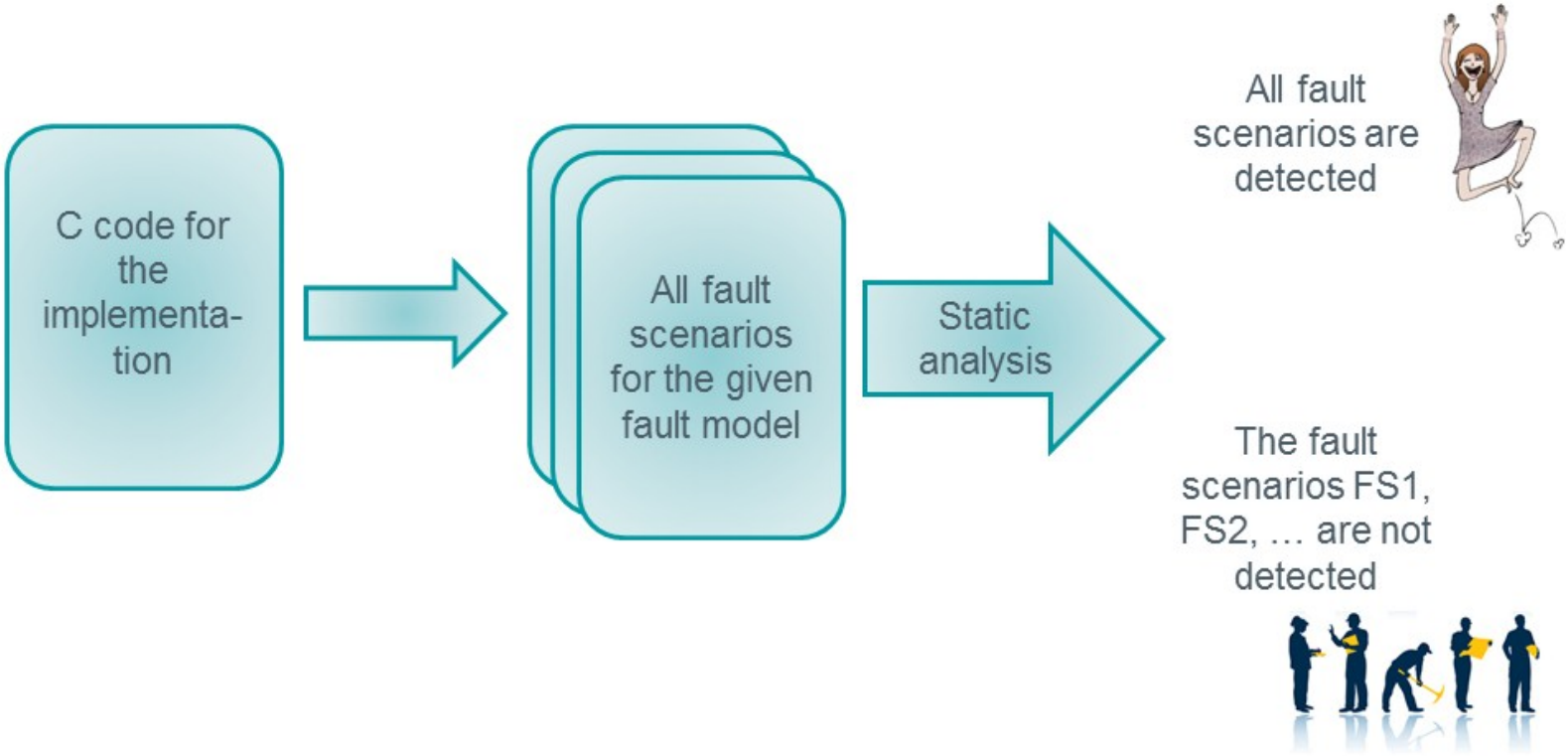
Objective

Given an implementation of a cryptographic algorithm, with a set of countermeasures, formally verify its functionality and its resistance to a set of attacks pre-defined.

The goal !

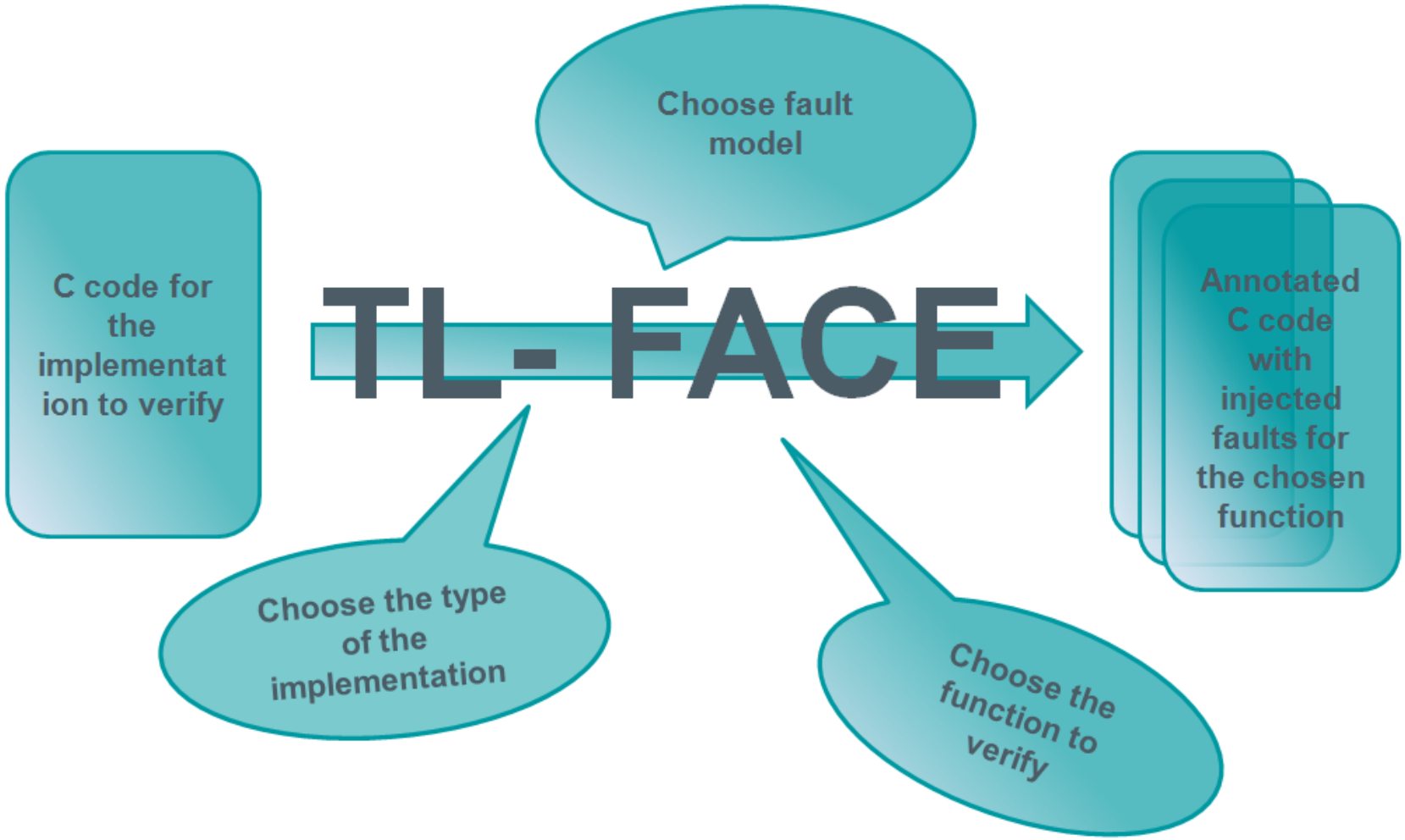

C code for the implementa-tion
All fault scenarios for the given fault model
Static analysis
All fault scenarios are detected
The fault scenarios FS1, FS2, … are not detected

The goal !



But how do we generate all fault scenarios ?

TL-FACE - User's interaction



TL-FACE implements the method described in *"Formal verification of an implementation of CRT-RSA Vigilant's algorithm", 2012, Christofi, Chetali, Goubin, Vigilant*

Define the Fault Model - Parameters I

- Number of faults authorized per execution
  - single-fault attack
  - double-fault attack
  - multi-fault attack

Define the Fault Model - Parameters I

- Number of faults authorized per execution
  - single-fault attack
  - double-fault attack
  - multi-fault attack
- Fault location (attacked variable)
  - no control : the target is one or more unknown bits
  - limited control : the target is a chosen variable
  - full control : the target is a set of chosen bits

Define the Fault Model - Parameters I

- Number of faults authorized per execution
  - single-fault attack
  - double-fault attack
  - multi-fault attack
- Fault location (attacked variable)
  - no control : the target is one or more unknown bits
  - limited control : the target is a chosen variable
  - full control : the target is a set of chosen bits
- Appearance time (attacked instruction)
  - no control
  - limited control : the fault is injected during the execution of a bloc of some identified operations
  - full control : the attacker chooses the ”moment” of the injection

Define the Fault Model - Parameters I

- Number of faults authorized per execution
  - single-fault attack
  - double-fault attack
  - multi-fault attack
- Fault location (attacked variable)
  - no control : the target is one or more unknown bits
  - limited control : the target is a chosen variable
  - full control : the target is a set of chosen bits
- Appearance time (attacked instruction)
  - no control
  - limited control : the fault is injected during the execution of a bloc of some identified operations
  - full control : the attacker chooses the ”moment” of the injection
- Fault persistence
  - transient fault : fault effect disappears after a period of time
  - permanent fault : fault effect persists until the attacked variable be refreshed
  - destructive fault : physical structure of the chip will be destroyed

Define the Fault Model - Parameters II

- Number of affected bits
  - only one bit
  - some bits (up to one byte)
  - a random number of bits (bound by the size of the affected variables or the component's bus)

Define the Fault Model - Parameters II

- Number of affected bits
  - only one bit
  - some bits (up to one byte)
  - a random number of bits (bound by the size of the affected variables or the component's bus)
- Fault type
  - bit flip : set bit value to its complementary value
  - stuck-at : set bit value to its initial value
  - random fault : set bit value to a random value
  - bit set or bit reset : set bit value to a known value

Define the Fault Model - Parameters II

- Number of affected bits
  - only one bit
  - some bits (up to one byte)
  - a random number of bits (bound by the size of the affected variables or the component's bus)
- Fault type
  - bit flip : set bit value to its complementary value
  - stuck-at : set bit value to its initial value
  - random fault : set bit value to a random value
  - bit set or bit reset : set bit value to a known value
- Attacker success probability

TL-FACE : ”Fault Attack Checking Engine” How it works

- Generate C code to simulate the faults

  Automatic generation

- Annotate C code to express the correctness of the crypto implementation and the fact that these faults are detected

  Annotations using the ACSL language

- Use static analysis tools/provers to check that all annotations are matched (i.e. no fault may modify the result without detection)

  Integration using frama-c [1]

- And if the proof does not finish ?

  Report the potential point of attack (line code + attacked variable)

1. frama-c is a source code analysis tool of C software, developed by CEA-LIST and INRIA-Saclay

2 Contribution

- Study of the usage of systematic linear codes for non-linear operations of block ciphers
- Formal Verification Methodology
- AES case study implementation and formal verification

Use Case : Formal Verification of an AES Implementation

- An AES software ANSI-C implementation based on systematic linear code C[16,8,5] has been developed.
- A formal verification for a basic fault model has been performed.
- Chosen Fault Model : An attacker can :
  - inject one fault per execution
  - inject permanent faults
  - modify data (not the code execution)
  - flip one bit of the AES state
- Implementation is believed resistant under a stronger attacker model (2, 3, 4) but has to be (formally) verified

Details About AES Implementation

- Each byte of the AES state x is replaced by a 16-bit codeword of C[16,8,5]

$$X = x||Gx = x * G$$

  (G being the generator matrix)
- Design is straightforward for linear operations of AES.
- No matrix operation.
- Only lookup tables are used.
- Technique described above for non linear step is applied.

Results - Found Locations

Once the previous method is applied, the number of identified locations are :

Functions	Possible locations
AddRoundKey	13
SubBytes	13
ShiftRow	16
MixColumns	101
Encrypt	15

We focused on

- one linear operation : AddRoundKey
- one non linear operation : SubBytes

Results

AddRoundKey operation

- 3 possible locations for the considered fault model
- All fault injections are detected by the countermeasure

SubBytes operation

- 1 possible location for the considered fault model
- It seems to be detected by the countermeasure

Security of the whole system

Further (manual) study shows that other operations are secure too

Conclusion

- We discuss a novel method for using systematic linear codes in order to protect non linear operations of block ciphers against fault attacks.
  - Can be applied to all block ciphers.
  - Can be efficiently implemented in software.
- The presented method is to be combined with other methods (such as tour counter, byte index -for the ShiftRow operation- etc) as some attack paths are not covered by codes
- We develop an AES software implementation with this method and verify it formally under a basic fault model.

Future Work

- Formal verification of the actual AES implementation considering stronger fault models.
- Provide precise bench (code/RAM/performance/Fault detection) of the method compared to other existing ones.
- Develop an AES implementation implementing orthogonal sum technique and combined with this method, and then verify it formally.

Thank you for your attention !

maria.christofi@trusted-labs.com

Overhead Compared to Unsecure Implementation

AES software implementation using C[16,8,5]

- Performance (assuming at least 16-bit CPU is used)
  - All linear operations : almost free
  - Nonlinear operations : 2 LUT + 1 XOR instead of 1 LUT
- RAM
  - Each internal variable byte represented on 2 bytes : x2
  - Nonlinear operations : 1 LUT in code, 2 randomized LUTs in RAM (512 bytes)