SYSTEMATIC
PARIS REGION SYSTEMS & ICT CLUSTER

VeriSiCC
Deliverable L4.1-4.2
Test Vectors

CALL: FUI25

NAME OF THIS PROJECT: VERISICC

Leader of this deliverable

- Partner: IDEMIA
- Contact name: Aurélien Greuet
- Contact information: aurelien.greuet@idemia.com

Leader of the project

- Company: CryptoExperts
- Contact name: Sonia Belaïd
- Contact information: sonia.belaid@cryptoexperts.com, 06 68 75 30 66

Partners

- SMEs: CryptoExperts and NinjaLab
- Big Business: IDEMIA
- Public Institutions: INRIA, ANSSI, and Université du Luxembourg

Version : 1.0

Table of contents

1. Introduction

1.1. Purpose

This deliverable describe the test vectors developed and used for tasks 4.1 and 4.2. Since these tasks overlap on some points, we decide to write one deliverable for both.

Test vectors were developed to estimate the practical security provided by the tools from SP3. These tests are implementations of cryptographic primitives, designed for embedded systems like smartcards. These implementations are verified or generated by automatic tools. Thus, they are expected to be secure in a specified model, e.g. probing model. However, because of leakages coming from the device, on which we have no control, the practical security must be assessed too.

To this end, we propose several implementations of the same S-Box, at several sharing orders. These implementations rely on different theoretical models.

In addition, we develop wrappers to harmonize the random management and data representation between implementations. As a result, the functions to call the different implementations have the same signature, and exact same inputs/outputs can be used for any implementation. Switching between implementations is then very convenient.

Moreover, the masking order is set as a parameter instead of being hardcoded. Hence, test vectors and S-Box routines can be loaded to the smartcard once for all to test all supported orders.

1.2. Notations

In the sequel, we will consider some bitslice implementations of PRESENT S-Boxes. Their inputs and outputs have to be in a specific format. Let x be a 64-bit PRESENT state, viewed as 16 4-bit nibbles, where the i-th nibble is $b_0^i\, b_1^i\, b_2^i\, b_3^i$:

$$x = b_0^0\, b_1^0\, b_2^0\, b_3^0 \;\; b_0^1\, b_1^1\, b_2^1\, b_3^1 \;\; \cdots \;\; b_0^{14}\, b_1^{14}\, b_2^{14}\, b_3^{14} \;\; b_0^{15}\, b_1^{15}\, b_2^{15}\, b_3^{15}\,.$$

Let $S(x)$ be the output of PRESENT S-Box for input value x and $\widetilde{b_0^i}\, \widetilde{b_1^i}\, \widetilde{b_2^i}\, \widetilde{b_3^i}$ be its i-th nibble:

$$S(x) = \widetilde{b_0^0}\, \widetilde{b_1^0}\, \widetilde{b_2^0}\, \widetilde{b_3^0} \;\; \widetilde{b_0^1}\, \widetilde{b_1^1}\, \widetilde{b_2^1}\, \widetilde{b_3^1} \;\; \cdots \;\; \widetilde{b_0^{14}}\, \widetilde{b_1^{14}}\, \widetilde{b_2^{14}}\, \widetilde{b_3^{14}} \;\; \widetilde{b_0^{15}}\, \widetilde{b_1^{15}}\, \widetilde{b_2^{15}}\, \widetilde{b_3^{15}}\,.$$

A bitslice implementation takes as input the following 4 16-bit values, referred in the sequel as the bitslice format:

- $r_0 = b_0^0\, b_0^1\, b_0^2 \cdots b_0^{14}\, b_0^{15}$ (bits at position 0 in each input nibble)
- $r_1 = b_1^0\, b_1^1\, b_1^2 \cdots b_1^{14}\, b_1^{15}$ (bits at position 1 in each input nibble)
- $r_2 = b_2^0\, b_2^1\, b_2^2 \cdots b_2^{14}\, b_2^{15}$ (bits at position 2 in each input nibble)
- $r_3 = b_3^0\, b_3^1\, b_3^2 \cdots b_3^{14}\, b_3^{15}$ (bits at position 3 in each input nibble)

and outputs:

- $\widetilde{r_0} = \widetilde{b_0^0}\, \widetilde{b_0^1}\, \widetilde{b_0^2} \cdots \widetilde{b_0^{14}}\, \widetilde{b_0^{15}}$ (bits at position 0 in each output nibble)
- $\widetilde{r_1} = \widetilde{b_1^0}\, \widetilde{b_1^1}\, \widetilde{b_1^2} \cdots \widetilde{b_1^{14}}\, \widetilde{b_1^{15}}$ (bits at position 1 in each output nibble)
- $\widetilde{r_2} = \widetilde{b_2^0}\, \widetilde{b_2^1}\, \widetilde{b_2^2} \cdots \widetilde{b_2^{14}}\, \widetilde{b_2^{15}}$ (bits at position 2 in each output nibble)
- $\widetilde{r_3} = \widetilde{b_3^0}\, \widetilde{b_3^1}\, \widetilde{b_3^2} \cdots \widetilde{b_3^{14}}\, \widetilde{b_3^{15}}$ (bits at position 3 in each output nibble)

2. Implementation choices

2.1. Algorithm

We chose to implement the PRESENT S-Box. This is a 4-bit to 4-bit S-Box, applied 16 times in parallel to each 4-bit nibbles of the 64-bit state.

Since it is lightweight and has a simple algebraic structure, formal methods for the verification or generation of countermeasures are expected to be runnable with PRESENT. On the contrary, it could require too much memory and/or could be too time-consuming to deal with more complex cryptographic primitives.

2.2. Target

The tests are developed for ARM Cortex-M3. This CPU family is used in most of current smartcards and secure elements. In particular, ARM Cortex-M3 based components are used in products like bank cards, passports, ID cards, SIM cards, eUICC, connected devices (watch, automotive), etc.

2.3. Security Order

We decided, as a first step, to restrict the test vectors to implementations secure at first and second order only.

This restriction makes the tests easier to develop and deploy. Moreover, it allows to quickly get feedbacks from the practical evaluation.

Higher orders can still be developed in a second step, taking into accounts feedbacks from the first and second order evaluation.

3. PRESENT S-Box Implementations

3.1. Unprotected LUT Implementation

A naive implementation, without countermeasures against side-channel attacks, is provided. It is used to verify the correctness of more complex implementations. It can also be used to mount a 1st order attack, proving the need to protect such implementations.

This algorithm relies on the following look-up table, describing the outputs of the PRESENT S-Box:

```
SBox[16] = {0xC, 0x5, 0x6, 0xB, 0x9, 0x0, 0xA, 0xD,
            0x3, 0xE, 0xF, 0x8, 0x4, 0x7, 0x1, 0x2};
```

Then, a 64-bit input is viewed as the concatenation of 16 4-bit nibbles $\mathtt{n}_0\mathtt{n}_1 \dots \mathtt{n}_{15}$. The table is used to compute the output $\mathtt{r}_0\mathtt{r}_1 \dots \mathtt{r}_{15}$, where each $\mathtt{r}_i = \mathtt{SBox}[\mathtt{n}_i]$.

3.2. Inria Unprotected Bitsliced Implementation

This implementation of the PRESENT S-Box is provided by Inria. This is a bitsliced implementation without masking. It is used as a reference implementation to compare with masked implementation. The code is given in C language.

3.2.1. Data Representation

The function computing the S-Box takes as parameters:

- two `[4]` arrays of 32-bit numbers. The first one is used to store the result and the second one contains the input. Inputs and outputs are both bitslice format.

3.3. Tornado Implementation

3.3.1. Tool Description

Tornado is a compiler producing masked bitsliced implementations proven secure in the bit/register probing model. It was introduced in [BDM+20].

C code was generated by Tornado from an Usuba PRESENT S-Box implementation. The Usuba source code comes from the file `Tornado/src/usuba/samples/usuba/present.ua`, available at https://github.com/CryptoExperts/Tornado.

The C code was generated in the register probing model, n-sliced (using `-V` option).

Tornado can't generate ARM Cortex-M3 assembly code. However, the assembly code generated by the ARM C compiler from the C code output by Tornado is very close to "handwritten" assembly code. Hence, the C code can be used without further specific optimization or additional countermeasure.

3.3.2. Masking Order

The code coming from Tornado is fully generic and can work at any order. The order is specified with a C preprocessor `#define` macro.

3.3.3. Data Representation

The function computing the S-Box takes 8 pointers as parameters:

- `X3`, `X2`, `X1`, `X0`: arrays of size `MASKING_ORDER+1` containing the inputs in bitslice format: each `Xi` contains the `MASKING_ORDER+1` shares representing the i-th bits of input.
- `Y3`, `Y2`, `Y1`, `Y0`: arrays of size `MASKING_ORDER+1` where the outputs are stored in bitslice format like the input above.

3.3.4. Random Generation

The fresh random needed in ISW multiplications and share refreshes is obtained by calling the `get_random` function. This routine returns a 32-bit random number. By default, it relies on the C library `rand()` function, but it can be modified to use other sources of random, like hardware RNG.

3.4. Inria Implementation

3.4.1. Tool Description

This implementation of PRESENT comes from [BGG+21]. It has been formally verified by the `scVerif` tool, an adaption of `maskVerif` [BBC+19] able to take into account device-specific leakage effects.

The code is given in ARM Cortex-M0+ assembly language, that is a subset of ARM Cortex-M3. Hence, such a code can be used without modification on our target.

3.4.2. Masking Order

The code is not generic, each order of masking is given as a specific function. For the tests, we consider 2 functions: one to compute PRESENT S-Box secure at order 1, and the other for order 2.

3.4.3. Data Representation

Order 1. The function computing the S-Box protected against first order attacks takes as parameters:

- a pointer to random data that will be used to pick fresh random numbers,
- two `[4][2]` arrays of 32-bit numbers. The first one is used to store the result and the second one contains the input. Inputs and outputs are both represented with 2 shares, both in bitslice format.

Order 2. The function computing the 2nd order S-Box takes as parameters:

- a pointer to random data that will be used to pick fresh random numbers,
- two `[4][3]` arrays of 32-bit numbers. The first one is used to store the result and the second one contains the input. Inputs and outputs are both represented with 3 shares, all three in bitslice format.

3.4.4. Random Generation

The fresh random used in ISW multiplications, refreshes and scrubbing is accessed with the pointer given as first parameter. It must be filled with enough random before entering the S-Box function. For order 1, 8 random numbers of 32 bits are needed. For order 2, 27 random numbers of 32 bits are used.

4. Test Vectors Description

4.1. Harmonization

In the sequel, we explain the choices made for the tests. In particular, we harmonize the random management and data representation, so that the same set of inputs/outputs can be used, regardless of the underlying S-Box implementation.

As a result, we provide two test functions, `test_present_2shares`, for first order implementations, and `test_present_3shares`, for second order implementations. These functions take as input a function pointer `presentOrder1` (resp. `presentOrder2`), expected to be a bitslice implementation of the PRESENT S-Box at the given order.

Both test functions perform the following:

- Test the correctness of the S-Box computation with the fixed input `0x0123456789ABCDEF`:
  - Compute the expected output using the unprotected LUT implementation
  - Split the input into 2 shares (resp. 3 shares). To this end, 1 (resp. 2) 64-bit random is generated as the first share (resp. the first and second shares), the second (resp. the third) being the xor of the input and the first one (resp. the xor of the input and the xor of the first two shares).

  - Transform the shares to bitslice format
  - Call the input function `presentOrder1` (resp. `presentOrder2`) with the shares in bitslice format
  - Transform back the output shares from bitslice to standard format
  - Recombine the output shares and compare the result with the expected output.

- Test the correctness of the S-Box computation with random inputs. The same steps as with the fixed input are performed. This is done 255 times.

4.2. Masking Order

We chose to first implement tests for first and second order PRESENT S-Box.

Inria implementations are used without modification.

Since the masking order with Tornado code is specified with a C preprocessor `#define` macro, changing the masking order can't be done dynamically: the `#define` has to be changed, the code re-compiled and the binary re-loaded to the smartcard target.

To keep a generic code for multiple orders, we replaced the preprocessor macros with parameters. Then, the same function can be called to compute the S-Box with data split in either 2 shares or 3 shares, without re-compilation. Hence, the code has to be loaded only once into our target, but can be used to test both order 1 and order 2.

4.3. Data Representation

We chose to use the same data representation as the Inria implementation:

Order 1. The function computing the S-Box protected against 1st order attacks takes as parameters:

- a pointer to random data that will be used to pick fresh random numbers,
- two `[4][2]` arrays of 32-bit numbers. The first one is used to store the result and the second one contains the input. Inputs and outputs are both represented with 2 shares, in bitslice format.

Order 2. The function computing the 2nd order S-Box takes as parameters:

- a pointer to random data that will be used to pick fresh random numbers,
- two `[4][3]` arrays of 32-bit numbers. The first one is used to store the result and the second one contains the input. Inputs and outputs are both represented with 3 shares, in bitslice format.

Inria code doesn't need modification, but Tornado implementation has to be adapted to handle this representation. To do so while keeping the exact same S-Box algorithm, a wrapper is used. Since the wrappers for order 1 and 2 are similar, we only give a description of the order 1 wrapper . Let `sbox__V1` be the Tornado S-Box. It takes as input 8 pointers `X3`, `X2`, `X1`, `X0`, `Y3`, `Y2`, `Y1`, `Y0`. The `Yi`'s will contain the result while the `Xi`'s contain the input values.

Hence, given two arrays `input[4][2]` and `output[4][2]` following Inria data format, our wrapper calls `sbox__V1` with

- `X3 = input[0]`
- `X2 = input[1]`
- `X1 = input[2]`
- `X0 = input[3]`
- `Y3 = output[0]`
- `Y2 = output[1]`
- `Y1 = output[2]`
- `Y0 = output[3]`

The modification to handle the random data is described in the next section.

4.4. Random Generation

We chose to get the fresh random through a pre-filled buffer, whose pointer is given as input parameter, instead of calling a function.

This choice provides a more flexible management of random numbers. In particular, the following scenarios, that are relevant for a characterization or evaluation process, can very easily be achieved:

- replay the exact same sequence of random numbers from one execution to another,
- replay the exact same sequence of random numbers to different implementations,
- give to the algorithm a sequence of "bad" random numbers, e.g. with some bias.

References

[BBC+19] G. Barthe, S. Belaïd, Gaëtan Cassiers, Pierre-Alain Fouque, B. Grégoire, and François-Xavier Standaert. maskVerif: Automated Verification of Higher-Order Masking in Presence of Physical Defaults. In *ESORICS*, 2019.

[BDM+20] Sonia Belaïd, Pierre-Évariste Dagand, Darius Mercadier, Matthieu Rivain, and Raphaël Wintersdorff. Tornado: Automatic Generation of Probing-Secure Masked Bitsliced Implementations. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020*, pages 311–341, Cham, 2020. Springer International Publishing.

[BGG+21] Gilles Barthe, Marc Gourjon, Benjamin Grégoire, Maximilian Orlt, Clara Paglialonga, and Lars Porth. Masking in Fine-Grained Leakage Models: Construction, Implementation and Verification. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(2):189–228, Feb. 2021.