# Read Me

Artifacts for TCHES'2022, Issue 3: Submission #35

This document is a brief introduction of the artifacts for the accepted paper "On Efficient and Secure Code-based Masking: A Pragmatic Evaluation" (Submission #35) for TCHES'2022, Issue 3.

## 1 About Artifacts

The artifact consists of five implementations for AES-128 masked by code-based masking.

- **2-share_BM:** a specific implementation tailored for the generator matrix $\mathbf{A}$ shown in Table 1. Since the generator matrix is fixed, no input is required.
- **2-share_IPM:** a specific implementation tailored for the generator matrix $\mathbf{A}$ (with $L_1 = 91$) shown in Table 1. Since the generator matrix is fixed, no input is required.
- **3-share_BM:** a specific implementation tailored for the generator matrix $\mathbf{A}$ shown in Table 1. Since the generator matrix is fixed, no input is required.
- **3-share_IPM:** a specific implementation tailored for the generator matrix $\mathbf{A}$ (with $L_1 = 91$ and $L_2 = 239$) shown in Table 1. Since the generator matrix is fixed, no input is required.
- **General case:** the generic implementation that can be fed with any legal generator matrix $\mathbf{A}$. Recall that $\mathbf{A}$ is an $(k+m) \times n$ matrix over $\mathbb{F}_{2^8}^{(k+m)\times n}$, for $n \geq k+m$. Since it is generic, some inputs (e.g. the generator matrix $\mathbf{A}$) should be prepared and some parameters should accordingly be tuned (e.g., $k$, $m$, and $n$). All the relevant inputs and parameters are defined in the source file "aes.c" and detailed in the annotations. However in the current version, this implementation has been well-tuned for the same generator matrix $\mathbf{A}$ as in "2-share_IPM", hence it can run normally with no modification.

It is worth noting that specific implementations (including "2-share_BM", "2-share_IPM", "3-share_BM", and "3-share_IPM") are fully-optimized for speed, and they are used to conduct a direct comparison regarding computational overhead (introduced in Section 3.5 of the paper submission) with the implementations in [BFG+17]. Hence we actually follow the same setting of parameters (the $L_1$ and $L_2$ in Table 1) with [BFG+17], which differ from the parameters utilized for security evaluations in Section 4 of the paper submission. However, according to our implementation strategies, the computational overhead is irrelevant with the parameters (the $L_1$ and $L_2$ in Table 1) of the generator matrix $\mathbf{A}$ as long as the parameters are greater than 1. Hence from a perspective of performance (in clock cycles), the "2-share_IPM" and "3-share_IPM" implementations should be constant-time regardless various choices of $L_1$ and $L_2$ of $\mathbf{A}$ for $L_1, L_2 \geq 1$. Whist the "general" implementation is designed for trace acquisition for the security evaluation detailed in Section 4 of the paper submission.

| | 2-share_BM | 2-share_IPM | 3-share_BM | 2-share_IPM |
|---|---|---|---|---|
| **A** | $\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$ | $\begin{pmatrix} 1 & 0 \\ L_1 & 1 \end{pmatrix}$ | $\begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$ | $\begin{pmatrix} 1 & 0 & 0 \\ L_1 & 1 & 0 \\ L_2 & 0 & 1 \end{pmatrix}$ |

Table 1: various choices of generator matrix $\mathbf{A}$ over $\mathbb{F}_{2^8}^{(k+m)\times n}$ for different implementations.

## 2 Running the Codes

### 2.1 Compiling

Our implementations are tailored for LEGACY STM32F407 (specifically STM32F407VG) whose micro-controller is ARM Cortex-M4. To compile the codes, `arm-none-eabi-gcc` tool-chain and a couple more libraries for the STM32F407 (STM32F4xx_StdPeriph_Driver and CMSIS) are required. The required libraries are included in the artifact and stored in the directory called "STLib".

In order to get everything up and running, here are the steps to follow (we provide the steps using a Linux shell syntax). Note that the steps are consistent among the 5 implementations to build the artifacts, here we set 2-share_BM as the example.

1. Install the `arm-none-eabi-gcc` tool-chain first and add the address of tool-chain to the PATH (system environment variable), or you can change the value of TOOLCHAIN_PATH in Makefile to the address of the tool-chain where you installed it (shown below). Note that the tested gcc version is 10.3.1 20210621 (release) (GNU Arm Embedded Toolchain 10.3-2021.07).

```
TOOLCHAIN_PATH=/yourAddress/arm-gcc/gcc-arm-none-eabi-10.3-2021.07/bin/
```

2. Compile the code step by step as below. After compiling, a HEX file called "main.hex" will be obtained as the firmware for the STM32F407 board.

```
$ cd */2-share_BM
$ make clean
$ make
```

## 2.2 Programming the Codes

To flash the firmware onto the board, you can use *stlink* tool or other researchers and designers. To use *stlink* tool for programming, please go for https://github.com/stlink-org/stlink to see the tutorial and install this tool. After installing, you can use the command lines below to flash the firmware.

```
    $ cd addressOfStlink/stlink/build-mingw/bin
    $ ./st-flash --format ihex write addressOfImplementations/2-share_BM/main.hex
```

## 2.3 Communicating with the board

All implementations in this artifact are designed for the same communication way. To communicate with the board by PC, you may need a USB to UART serial communication module and FT232 is recommended. On the board, the pin PA2 is utilized to send messages, while the pin PA3 is set to receive messages. Below we will show how to use the shell command to communicate with the board. Also, other serial communication assistants on PC can work well with the board for communication.

First, baud rate should be set to 9600.

```
    $ stty -F /dev/ttyUSB0 raw speed 9600 -echo
```

And you can use the command line to check if the baud rate is correct.

```
    $ stty -F /dev/ttyUSB0
```

There are four types of messages for communication. The length of each type of message is two bytes. The first byte is commonly set to "80"(hexadecimal) as an identifier, while the second byte represents various instructions. Hence after receiving a message, the board will firstly check whether the identifier byte is correct or not. If correct, the board will reply with the second byte of the received message right after. If not, the board will discard and ignore this message until the identifier byte emerges. The four instructions represented by the second byte are shown below. Note that the command strings of related examples shown below are all hexadecimal.

- **Assign the key.** The byte code is "12"(hexadecimal). This instruction is to set the key value (the key already has the initial value) stored in the RAM of the board. After receiving the reply from the board, the key value (for 16 bytes in total) should be issued continuously byte by byte. If the board receives the 16 bytes successfully, it will change the key value. Then it will reply with a fixed code "91"(hexadecimal). Usually, this instruction will be issued first.

```
PC: 80 12 // instruct the board to assign the key
board: 12 // reply
// recive the correct reply and send the 16 bytes key
PC: 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
board: 91 // have set the key
```

  To communicate with board by shell, two terminals may be required. one (T1) is for transmitting data to the serial port, and the other one (T2) is for printing the data from the board.

```
T2: $ hexdump -C -n 1 /dev/ttyUSB0
T1: $ echo -e -n "\x80\x12" > /dev/ttyUSB0
// "12" shall be print on T2
T2: $ hexdump -C -n 1 /dev/ttyUSB0
T1: $ echo -e -n "\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\
    x0d\x0e\x0f" > /dev/ttyUSB0
// "91" shall be printed on T2
```

- **Instruct to encrypt.** The byte code is "04"(hexadecimal). This instruction is to instruct the board to encrypt with the issued plaintexts (the plaintexts already have the initial value stored in the RAM of the board). After receiving the reply from the board, the plaintexts (for 16 bytes in total) for encryption should be issued continuously byte by byte. If the board receives the 16 bytes successfully, it will reply with a fixed code "92"(hexadecimal) first and then starts the AES-128 encryption (masked by code-based masking) with the received plaintexts. After the encryption, the board will send a fixed byte code "93"(hexadecimal) for the status reminder. Right after it will send 4 bytes to transfer the clock cycle results counted for such an encryption process. Note that this is a Little Endian transmission for 32-bit data. Usually, this instruction is utilized for trace acquisition.

```
PC: 80 04 // instruct the board to encrypt with the issued plaintexts
board: 04 // reply
// recive the correct reply and send the 16 bytes plaintexts
PC: 00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff
board: 92 // have received the plaintexts and start the encryption
board: 93 // remind that the encryption is completed
board: b6 5d 02 00 // the true clock cycle is 0x00025db6
```

To communicate with board by shell, two terminals may be required. one (T1) is for transmitting data to the serial port, and the other one (T2) is for printing the data from the board.

```
T2: $ hexdump -C -n 1 /dev/ttyUSB0
T1: $ echo -e -n "\x80\x04" > /dev/ttyUSB0
// "04" shall be print on T2
T2: $ hexdump -C -n 6 /dev/ttyUSB0
T1: $ echo -e -n "\x00\x11\x22\x33\x44\x55\x66\x77\x88\x99\xaa\xbb\xcc\
    xdd\xee\xff" > /dev/ttyUSB0
// "92 93 b6 5d 02 00" shall be printed on T2
```

- **Encryption test.** The byte code is “81”(hexadecimal). This instruction is to instruct the board to encrypt with the key and plaintexts stored in the RAM of the board. After receiving the corresponding message and replying, the board will then directly perform the key expansion and the encryption (of AES-128 masked by code-based masking) using the stored key value and plaintexts. After the encryption, the board will send a fixed byte code “95”(hexadecimal) for the status reminder. Right after it will send 4 bytes to transfer the clock cycle results counted for such an encryption process. Note that this is a Little Endian transmission for 32-bit data. Usually, this instruction is used to count the clock cycles for an encryption process.

```
PC: 80 81 // instruct the board to encrypt with the stored key and
    plaintexts
board: 81 // reply and start the encryption
board: 95 // remind that the encryption is completed
board: b6 5d 02 00 // the true clock cycle is 0x00025db6
```

To communicate with board by shell, two terminals may be required. one (T1) is for transmitting data to the serial port, and the other one (T2) is for printing the data from the board.

```
T2: $ hexdump -C -n 6 /dev/ttyUSB0
T1: $ echo -e -n "\x80\x81" > /dev/ttyUSB0
// "81 95 b6 5d 02 00" shall be printed on T2
```

- **Return the ciphertexts.** The byte code is “10”(hexadecimal). This instruction is to instruct the board to transfer the ciphertexts (initialized to all 0) stored in RAM.

After receiving the corresponding message and replying, the board will then transfer the ciphertexts (for 16 bytes in total) byte by byte. To remind that such transmission is completed, the board will finally send a fixed code "94"(hexadecimal).

```
PC: 80 10 // instruct the board to transfer the 16 bytes cyphertexts
board: 10 // reply and start the encryption
// send the ciphertexts
board: 69 c4 e0 d8 6a 7b 04 30 d8 cd b7 80 70 b4 c5 5a
board: 94 // remind that the issue is over
```

To communicate with board by shell, two terminals may be required. one (T1) is for transmitting data to the serial port, and the other one (T2) is for printing the data from the board.

```
T2: $ hexdump -C -n 18 /dev/ttyUSB0
T1: $ echo -e -n "\x80\x10" > /dev/ttyUSB0
// "10 69 c4 e0 d8 6a 7b 04 30 d8 cd b7 80 70 b4 c5 5a 94" shall be
    printed on T2
```

The above-mentioned "key", "plaintexts" and "ciphertexts" are declared and initialized in the source file "main.c" for all implementations. And they are defined as "key_aes", "pt_aes" and "ct_aes", respectively. The initial value of key is 0x000102030405060708090a0b0c0d0e0f and the initial value of plaintexts is 0x00112233445566778899aabbccddeeff. Hence if the encryption process is correct, the ciphertexts should be 0x69c4e0d86a7b0430d8cdb78070b4c55a. Therefore, you can simply use the "encryption test" and "return the ciphertexts" instructions to check the correctness of the encryption.

## 2.4 Clock Cycle Counting

It is recommended to directly use the instruction "Encryption test" to count the clock cycles of once encryption process. It is worth noting that the clock cycles counted by the instruction "Instruct to encrypt" and "Encryption test" sometimes are different. They are usually times of 16 clock cycles apart. However, this difference is very small and originates from different instructions. The records of clock cycles in the paper depend on the clock cycles counted by "Encryption test".

## 2.5 Acquisition

In this artifact, only the "general" implementation is designed for trace acquisition. However, the other implementations can be exploited for acquisition if with some light modifications (e.g., adding triggers). In our codes, the pin PA7 of the board is leveraged for trigger.

# 3 About Source Code

## 3.1 File Organization

For the five implementations in the artifact, there are some files with the same functions that are listed below.

- **"comm.h" & "comm.c".** These files declare and implement functions for serial communication of the STM32F407VG board.
- **"DWT_clock.h" & "DWT_clock.c".** These files declare and implement functions for the clock cycle counter of the STM32F407VG board.
- **"random.h" & "random.c".** These files declare and implement functions for activating TRNG (True Random Numbers Generator) of the STM32F407VG board.
- **"common.h".** Some global variables are declared and macros are defined in this file.
- **"aes.h" & "aes.c".** These files declare and implement functions for the key expansion and the masked encryption of AES-128. Note that the specific implementations have an extra assembly source file called "aes.s" that implements the whole masked AES-128 algorithm (excluding the key expansion) in assembly code. While the "general" implementation uses the assembly code to implement masked gadgets only.
- **"main.c".** This file implements the communication (introduced above) with the STM32F407VG board.

For each implementation, there exist some assembly source files to implement masked gadgets (and masked AES-128 only for specific implementations) by code-based masking. For the detailed functions, please see the annotations covered in the files. In addition, there are some common system files and "Makefile" to compile the code for the five implementations. For the "general" implementation, it also includes "trigger.h", "trigger.c" and "trigger.s" for activating trigger in STM32F407VG board.

## 3.2 Input Parameters

Concerning the specific implementations, since the generator matrix **A** is fixed, no inputs are required and there is no need to tune any parameters. While regarding the "general" implementation, the generator matrix **A** and the associated matrices should be required as inputs and some parameters should accordingly be tuned. Those inputs and parameters are all covered in the source file "aes.c" and detailed in the annotations. However, in the current version of "general" implementation, the parameters are well-tuned for the input generator matrix

$$\mathbf{A} = \begin{pmatrix} 1 & 0 \\ 91 & 1 \end{pmatrix},$$

hence it can run normally with no modification if required.

### 3.3 About Random Numbers

With respect to specific implementations, all the random numbers for a whole masked AES-128 algorithm should be generated first before the encryption. The generated random numbers are stored in an array variable "randomSeed", which is defined in the source file "main.c". The function "getRand()" for generating random numbers is implemented in the source file "aes.c".

Concerning the "general" implementation, random numbers will be generated when required then during the encryption. And the functions involving generating random numbers include "getRandom1()", "getRandom2()" and "AES_Encrypt_GCB()", which are all implemented in the source file "aes.c".

## References

[BFG+17] Josep Balasch, Sebastian Faust, Benedikt Gierlichs, Clara Paglialonga, and François-Xavier Standaert. Consolidating inner product masking. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 724–754. Springer, 2017.