Software Camouflage

Sylvain Guilley*[1,2], Damien Marion[3], Zakaria Najm[1],
Youssef Souissi[2], and Antoine Wurcker[3]

[1] TELECOM-ParisTech, COMELEC dpt — UMR CNRS 5141,
39 rue Dareau, 75 014 Paris, FRANCE.
[2] Secure-IC S.A.S., 80 avenue des Buttes de Coësmes, 35 700 Rennes, FRANCE.
[3] XLIM — UMR CNRS 7252,
123, avenue Albert Thomas, 87 060 Limoges Cedex, FRANCE.

Abstract. Obfuscation is a software technique aimed at protecting high-value programs against reverse-engineering. In embedded devices, it is harder for an attacker to gain access to the program machine code; of course, the program can still be very valuable, as for instance when it consists in a secret algorithm. In this paper, we investigate how obscurity techniques can be used to protect a secret customization of substitution boxes in symmetric ciphers, when the sole information available by the attacker is a side-channel. The approach relies on a combination of a universal evaluation algorithm for vectorial Boolean functions with indistinguishable opcodes that are randomly shuffled. The promoted solution is based on the noting that different logic opcodes, such as AND/OR or AND/XOR, happen to be very close one from each other from a side-channel leakage point of view. Moreover, our solution is very amenable to masking owing to the fact the substitution boxes are computed (combinationally).

Keywords: Side-channel analysis, reverse-engineering, cryptography, substitution boxes (sboxes), CNF, ANF, masking, camouflage, obscurity.

1 Introduction

Side-channel analysis is a well-known technique to extract keys from secure devices. It can be adapted to recover secret algorithms. In this case, it takes on the name of side-channel analysis for reverse-engineering (SCARE). Several scholar papers describe use-cases how SCARE can work in various contexts. For example, Novak shows in [1,2,3] how to attack GSM algorithms. Such attacks are later improved by Clavier [4]. Some other papers analyse general methods to attack software implementations (references are [5,6,7,8,9]).

To combat SCARE, protections shall be implemented. Usually proposed solutions to prevent SCARE attacks consist in applying standard side-channel countermeasures (as described for instance in the "Differential Power Analysis" book [10]). Now, robust side-channel countermeasures are costly. For instance, a provable first-order masking scheme for the AES [11] makes its cost increase from 3 kcycles to 129 kcycles.

* Corresponding author.

It is noteworthy that lowest-level solutions exist to prevent embedded information stealth. For instance, at silicon-level, they consist in shaping standard cells in such a way that different functions appear alike. This is illustrated on the example displayed in Fig. 1. Thus, an attacker that wishes to recover the circuit's functionality by its destratification will eventually be deceived when attempting to recognize gates. This is why this technique is termed *hardware camouflage*[4]. However, this technological option is low-level, namely circuit layout level, and thus might not be applicable to all business models (especially *fabless* models).

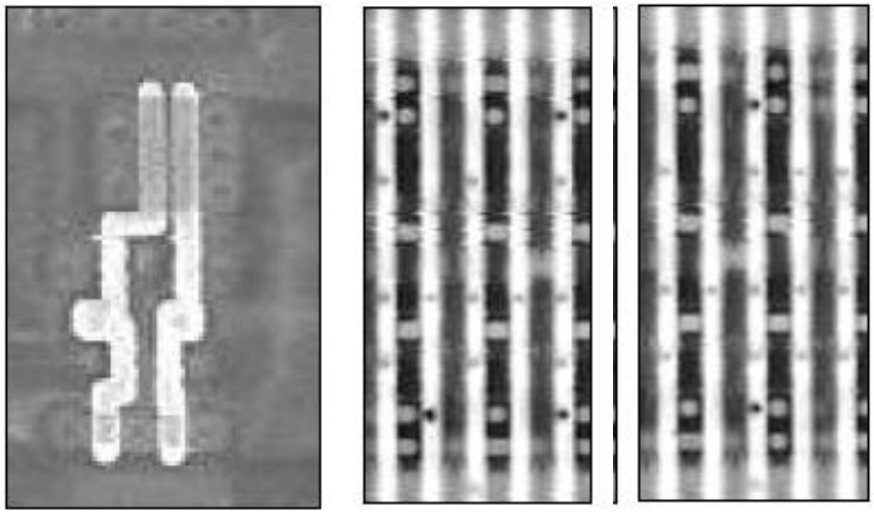

Fig. 1. Hardware-level camouflage of gates. Left: an unprotected gate, whose function is easy to identify. Center, right: *almost* indistinguishable AND/OR camouflaged gates. *[courtesy of SMI / SypherMedia Library]*

In this article, we place ourselves in a context where the secret application (or data) to be protected against SCARE is a code executed by a processor. For this purpose, we explore a software solution based on *camouflaged instructions*[5]. The idea is to take advantage of indistinguishable instructions, from a side-

[4] This is colloquially known as *hardware "camo"*; there are many such examples of technologies, such as this patent [12] by IBM and the tens of patents cited by this patent.

[5] We notice that the paper [13] also tackles a similar issue, but requires to process simultaneously a *decoil* value, like in *dual-rail with precharge logics* [14]. In our *software camouflage* technique, the opcodes are balanced natively *per se*, without any *deus ex machina* support.

channel point of view, to deceive an attacker eager to uncover the secret running code.

The rest of the paper is organized as follows. In Sec. 2 we show that some classes of instructions are delicate to recognize using side-channel leakage (which will be all the more true as the data processed by the instruction is unknown). These conclusions are derived from real-world side-channel studies on representative embedded devices. In Sec. 3 we provide a software camouflage based on the use of AND and OR, or AND and XOR (shown as hardly distinguishable), and compare the cost of our countermeasure to alternative protections against SCA or SCARE. This section considers the case-study of the secret customization of substitution boxes (sboxes) in some block cipher. The remarkable compatibility of our software camouflage method with masking is also explained. The Sec. 4 discusses further considerations, such as the possibility to use cryptographically strong sboxes that are parametrized in a more lightweight way than the generic sboxes computations, or the resistance of software camouflage to fault injection attacks. Eventually, the conclusions are given in Sec. 5. Source codes for the most important algorithms are given in Appendix A. The source code for the truth table to ANF conversion algorithm is given in Appendix A.

2 Investigation on the Indistinguishability of Opcodes AND / OR / XOR by Side-Channel Analysis

In this section, we intend to show that it is possible to choose a set of opcodes that are difficult to distinguish one from each other by side-channel analysis. We have conducted three distinct studies on two distinct platforms. The first one is an ASIC that contains a *6502 microcontroller*; the second one is an AT91 with an ARM7-TDMI processor executing a *virtual machine* JAVA simple RTJ (32 bits); finally, the third one is the same platform, but executing *native code* this time.

2.1 Investigation on 6502 Microcontroller

Our work on the 6502 CISC microcontroller is realized with power consumption traces of the execution of a known code sample of an AES SubBytes function (16 loops for the entire state). The studied circuit has been synthesized and founded in STMicroelectronics CMOS 130 nm technology, from a behavioral description in VHDL of the 6502 processor [15]. Thanks to MODELSIM[6] simulations of the 6502 VHDL code, we know the values of several internals (registers, flags, signals, ...) at each clock cycle of the execution. The best linear model of the consumption by a pair of internals using the *least-squares* method (also known as "stochastic method" [16]) reveals that more than 98% of the consumption is explained by the values of the `Rd` (Read data) and `We` (Write enable)

[6] MODELSIM is a commercial tool, sold by Mentor Graphics, capable of simulating a behavioral event-based HDL codes (*e.g.* VHDL or Verilog codes).

signals [17]. These signals are generated by the 6502 CPU to control the RAM memory; they can take three values:

1. `Rd/We = 0/0`, noted `O`, when the memory is not accessed,
2. `Rd/We = 1/0`, noted `R`, when the memory is read, and
3. `Rd/We = 0/1`, noted `W`, when the memory is written to.

Of course, the last case (`Rd/We = 1/1`) never happens in practice. Using MODELSIM we observed that the sequence of values of the `Rd/We` signals during the execution of a given opcode is always the same.

Calling *signature* of an opcode its sequence of `Rd/We` signals, we interestingly noticed that, for any given addressing mode, some sets of distinct opcodes share the same signature. This is illustrated in Table 1 for the following set of opcodes: AND, AOR, EOR (*logic AND, OR, XOR operations*), and ADC and SBC (*arithmetic addition and subtraction*). Given the tight relation between the power consumption and the `Rd` and `We` signals, such set of opcodes which share the same signature are virtually indistinguishable by side-channel analysis. The interpretation of this interesting remark is that signals that are driving the RAM are heavily loaded and thus have a strong leakage, whereas which operation is executed by the ALU (arithmetic and logic unit) is selected by a local signal that leaks little. This motivates the assumptions made in the sequel that, in particular, AND, OR and XOR, cannot be distinguished.

	AND (AND)	**OR** (AOR)	**XOR** (EOR)	**ADD** (ADC)	**SUB** (SBC)
IMM	ORR	ORR	ORR	ORR	ORR
Z-PAGE	ORR	ORR	ORR	ORR	ORR
Z-PAGE, X	ORR	ORR	ORR	ORR	ORR
ABS	RORRR	RORRR	RORRR	RORRR	RORRR
ABS, X	RORRR	RORRR	RORRR	RORRR	RORRR
ABS, Y	RORRR	RORRR	RORRR	RORRR	RORRR
(IND, X)	ORRORRR	ORRORRR	ORRORRR	ORRORRR	ORRORRR
(IND), Y	ORRORRR	ORRORRR	ORRORRR	ORRORRR	ORRORRR

Table 1. `Rd/We` signature for a set of opcodes and for all addressing modes of the 6502

2.2 Investigation on ARM7 Java

The goal of our work done with the ARM7 and the virtual machine is the same as in the previous section with the 6502: find whether there are sets of bytecodes that are hard to distinguish by side-channel analysis. For this purpose, we choose a set of bytecodes (ADDV1, DUP, ICONST0, ...) executed on our platform and record their electromagnetic emission (average of 50 traces for each bytecode). Next we compute the cross-correlation between each pair of

traces. Fig. 2 shows the results we have obtained. The fact that we get a high level of cross-correlation (always beyond 0.86) is due to the common part of each trace which is related to the instruction fetch at the Java level. We can identify groups of bytecodes that have really similar electromagnetic leakages: *e.g.* (MULV1, MULV2) or (IXOR, IAND, IOR, SUBV1, SUBV2, ADDV1, ADDV2).

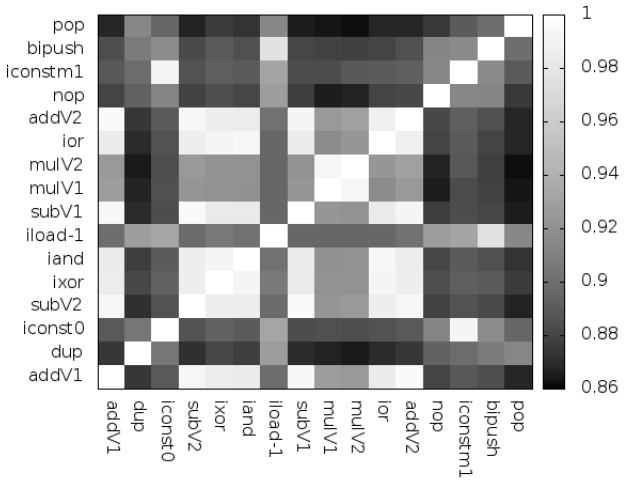

Fig. 2. Representation of the cross-correlation between each opcodes' traces

2.3 Investigation on ARM7 Native Machine Code

We have practiced another experience on our ARM7 platform: this time, it is profiled at the assembly level. This practice follows the same protocol that we have used for our study at the Java level. The traces of the electromagnetic leakage during the execution of an opcode are captured and analyzed. The result of this work is plotted on the three figures 3, 4 and 5. The first one, namely Fig. 3, shows all the traces obtained after the execution of all the chosen opcodes (SUB, ADD, OR, AND, XOR and MUL). The second one, namely Fig. 4 shows the electromagnetic leakage obtained with the execution of a ADD and a SUB; this graphic reveals that it is possible to distinguish these two opcodes. The last one, namely Fig. 5, concerns the execution of the opcodes AND, OR and XOR; these traces show that it is difficult to distinguish between this three opcodes using side-channel analysis. The same arguments as given for the 6502 apply to account for this noting: the only difference while executing a logic bitwise operation is a selection in the ALU block within the CPU core.

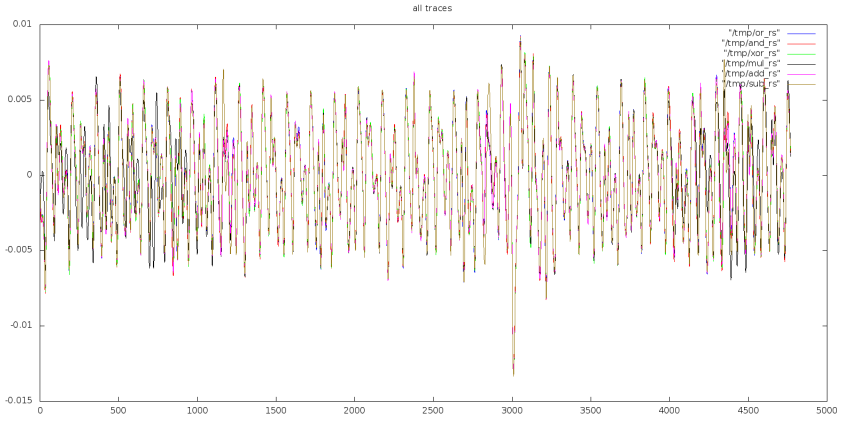

Fig. 3. Traces of the electromagnetic leakage of all the opcodes

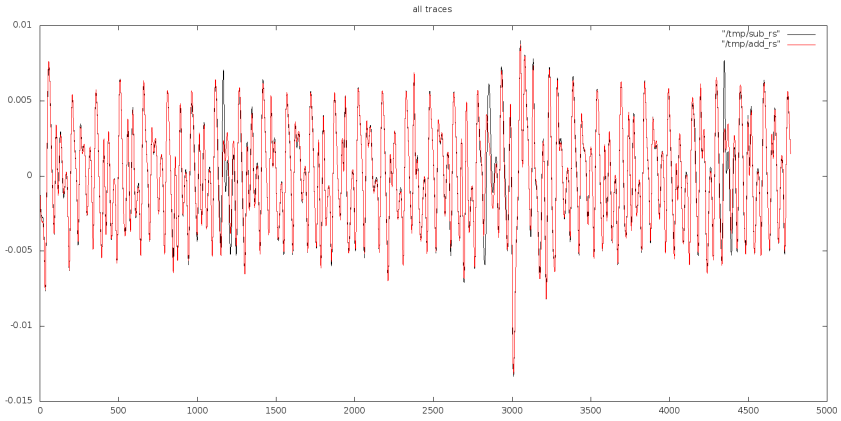

Fig. 4. Traces of the electromagnetic leakage of the ADD and SUB

3 Universal Computation Schemes for Substitution Boxes based on AND, OR & XOR

This section shows how the indistinguishability of AND, OR & XOR opcodes can be taken advantage of to dissimulate the functionality of an sbox from a side-channel attacker. We assume the context in which a block cipher, such as AES, is customized, so as to make its attack still more complex. This secret cryptography practice is common in some market verticals, such as the conditional access (Pay-TV smartcards) or the telecom protocols (encryption algorithms between the terminal and the base station). Instead of redesigning a completely new block cipher, which is error-prone, it is often observed that a standardized algorithm, trustworthy since well analyzed by crytanalysts, is slightly cus-

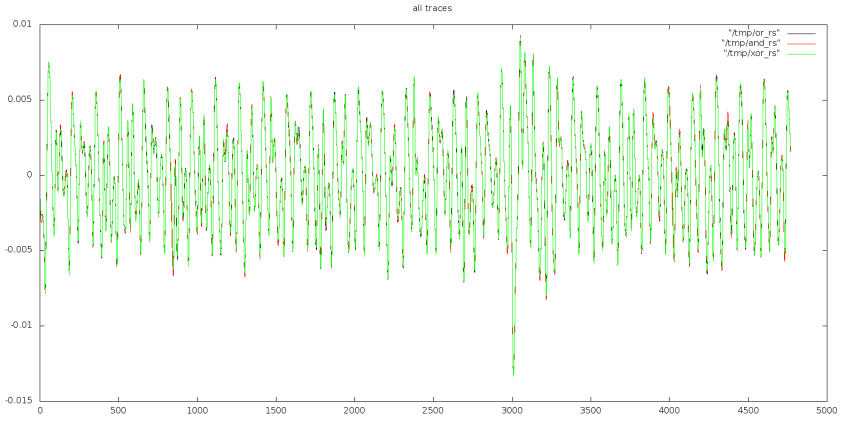

Fig. 5. Traces of the electromagnetic leakage of the AND, XOR and OR

tomized. One classical customization is the replacement of the sboxes by others that have similar properties in terms of linear and differential characteristics. We thus assume in this section that the goal of the designer is to compute a secret sbox in such a way it cannot be uncovered even in the presence of side-channel leakage.

3.1 Tabulated *versus* Computed Substitution Boxes

The sboxes implemented in memories (look-up-tables) have been shown to be attackable thanks to a divide-and-conquer approach in [18]. The reason is that every coordinate of the sbox can be guessed independently. Thus, a leakage model can be determined by selecting some sub-functions, for example $2 \rightarrow 1$ functions (there are 16 of them) and attempting to correlate with them. Therefore, it is advised to use standard countermeasures against side-channel attacks to prevent those exploits. They are usually classified in two categories:

1. **hiding** [10, Chap. 7], that consists in balancing the leakage *statically* (*e.g.* thanks to dual-rail with precharge logic), and
2. **masking** [10, Chap. 9], that consists in balancing the leakage *statistically* (*e.g.* thanks to a secret sharing representation of the intermediate variables).

Direct application of masked logic or dual-rail logic is awkward, due to the huge overhead in terms of area it requires. Indeed, the memory size, initially $n \rightarrow m$, becomes in both cases $2n \rightarrow 2m$ (with the notation: number of input bits $\rightarrow$ number of output bits). Let us call $f : \mathbb{F}_2^n \rightarrow \mathbb{F}_2^m$ the unprotected sbox, and $\tilde{f} : \mathbb{F}_2^{2n} \rightarrow \mathbb{F}_2^{2m}$ the protected sbox.

Regarding first-order masking, the equation of the masked table can for instance write as:

$$Y = \tilde{f}(X) = (f(X_{\text{mask}}), f(X_{\text{mask}} \oplus X_{\text{masked\_data}}) \oplus f(X_{\text{mask}})) \ , \qquad (1)$$

where $X = (X_{\text{mask}}, X_{\text{masked\_data}}) \in \mathbb{F}_2^{2n}$ and $Y = (Y_{\text{mask}}, Y_{\text{masked\_data}}) \in \mathbb{F}_2^{2m}$. The two items in X and Y are called the *shares*. The Eqn. (1) satisfies the masking property that $X_{\text{mask}} \oplus X_{\text{masked\_data}}$ is actually the unmasked sensitive variable; the same holds at the output of the sbox for $Y_{\text{mask}} \oplus Y_{\text{masked\_data}}$, because $Y_{\text{mask}} \oplus Y_{\text{masked\_data}} = f(X_{\text{mask}} \oplus X_{\text{masked\_data}})$. The masking is first-order only because it makes use of only one mask per sensitive variable to protect, and because the mask used at the input of the sbox is reused to mask its output.

For dual-rail logic, the new sbox has this equation:

$$Y = \tilde{f}(X) = \begin{cases} (\neg f(\neg X_{\text{true}}), f(X_{\text{true}})) & \text{if } \neg X_{\text{false}} = X_{\text{true}}, \\ (0,0) & \text{otherwise, } \textit{i.e.}\ X_{\text{false}} = X_{\text{true}} = 0, \end{cases}$$

where $X = (X_{\text{false}}, X_{\text{true}}) \in \mathbb{F}_2^{2n}$ and $Y = (Y_{\text{false}}, Y_{\text{true}}) \in \mathbb{F}_2^{2m}$. The property satisfied by this redundant representation of the sbox is that the number of transitions of variables in $\mathbb{F}_2^{2n}$ (or $\mathbb{F}_2^{2m}$) is constant if they are operated through the precharge-evaluation protocol: X is equal to $(0,0)$ in precharge phase, and then takes the valid value $X = (\neg X_{\text{true}}, X_{\text{true}})$ at evaluation phase.

So, to summarize with a typical example, when $n = m = 8$, the number of memory points in the memory is raised from $2\,048$ (8×2^8) for f to $1,048,576$ (16×2^{16}) for $\tilde{f}$, *i.e.* an about $500\times$ increase in size. This is considered unaffordable for many applications.

Therefore, other protections shall be envisioned. Concerning masking, the alternative of the *global look-up table* (Eqn. (1)) is the *table recomputation*. This terminology stems from this paper [19] by Prouff *et al*.

The recomputation in "masking" is actually also prohibitive. For instance, the execution time of an AES secured at first order is multiplied by 43 while at the same time its implementation size is multiplied by 3 [11]. However, this masking protects the data but not the operations. Thus, the cost to mask both the data and the operations is expected to still be greater (*cf*. Sec. 4.1 for a quantitative analysis).

We therefore explore the possibility to use the "hiding" paradigm applied to sbox recomputation. This has been first mentioned in asymmetric cryptography, to protect against simple power attacks, *i.e.* against attacks that attempt to break the key with one sole trace. The idea is to make two operations (in asymmetric cryptography, it is the squaring and the multiplication) indistinguishable. To that end, each time the algorithm involves a key dependent choice, the two branches are coded in such a way they look alike when analyzed from a side-channel perspective. This approach is called the *atomicity countermeasure* [20]. We leverage on this idea to protect not only two portions of codes, but any of the 2^n executions of an $n \to m$ sbox.

Our solution against the SCARE attack of the sbox relies on two features:

1. The opcodes used for the sbox computation leak, irrespective of the data they process. It is thus important to already provide a regular sbox evaluation that does not leak the computation in simple power analysis. This aspect requires universal regular sbox evaluation algorithms, that are able to

compute any sbox in an identical way (from a side-channel analysis standpoint). This is covered in section 3.2.

2. Second, even if a *horizontal* analysis of the leakage is not possible owing to the indistinguishability countermeasure, the implementation shall all the same resist *vertical* attacks that aim at recovering the computed function based on the possibility (or not) to correlate with a guessed leakage. The solutions described in section 3.2 are amendable to *shuffling* [21], *i.e.* a countermeasure that consists in executing the code in a random order. This aspect is the subject of section 3.3.

3.2 Universal and Regular Sbox Evaluation

In this section, we present several ways to compute a secret substitution box $\mathbb{F}_2^n \to \mathbb{F}_2^m$ using only AND & XOR instructions.

Conjunctive Normal Form (CNF) Each component $j \in [\![1, m]\!]$ of the sbox is written as:

$$f_j(x) = \bigvee_{y \in \mathbb{F}_2^n} f_j(y) \wedge \bigwedge_{i=1}^{n} (1 \oplus x_i \oplus y_i) \ . \tag{2}$$

As each term $\bigwedge_{i=1}^{n}(1 \oplus x_i \oplus y_i)$ (also known as *minterm*, that is equal to 1 if and only if x is equal to y) differs for different x, the CNF can be written alternatively only with AND & XOR operations:

$$f_j(x) = \bigoplus_{y \in \mathbb{F}_2^n} f_j(y) \wedge \bigwedge_{i=1}^{n} (1 \oplus x_i \oplus y_i) \ . \tag{3}$$

The overall cost is $m \times (2^n - 1) \times (2n + 1)$ bitwise operations. If the bitwidths are $n = m = 8$ and if the CPU registers are at least bytes, then the m coordinates can be computed in parallel, resulting in $(2^n - 1) \times (2n + 1)$ byte operations.

Algebraic Normal Form (ANF) Each component can also be written in a unique way [22, Sec. 2.1.] as:

$$f_j(x) = \bigoplus_{y \in \mathbb{F}_2^n} a_y \wedge \bigwedge_{i=1}^{n} {x_i}^{y_i} \ , \tag{4}$$

where for any $x_i, y_i \in \mathbb{F}_2, {x_i}^{y_i}$ is equal to x_i if $y_i = 1$, or 1 otherwise. The constant a_y is equal to $\bigoplus_{z \in \mathbb{F}_2^n / z \preceq y} f_j(z)$, where $(z \preceq y) = 1 \iff (z \wedge y) = z$, *i.e.* z has 1s at coordinates where y also has. We also say that z is *covered* by y. In particular, the a_y of Eqn. (4) are not connected to the $f_j(y)$ of Eqn. (3). A fast algorithm to compute all the a_y is given in the code listing 1.1 in Appendix A. The monomial $\bigwedge_{i=1}^{n} {x_i}^{y_i}$ is also noted x^y. It needs not be computed in a *bitslice* manner; indeed, all the $1 \leq j \leq m$ computations in Eqn. (4) can be handled *in parallel*, as illustrated in the code listing 1.2 in Appendix A. The overall cost is

$m \times (2^n - 1) \times (n+1)$ bitwise operations. Indeed, in average on y, the computation of x^y requires only $n/2$ AND. Therefore, ANF is computed about *twice faster* than CNF.

As noted for the CNF, an 8-bit CPU can evaluate the $m = 8$ coordinates at once, and thus a $8 \to 8$ function can be computed in $(2^n - 1) \times (n + 1)$ byte operations.

Finally, it must be underlined that ANF and CNF can execute truly in *constant time*; indeed, the sboxes are fully computed is a thorough method to prevent timing attacks (refer for instance to the notice by Bernstein *et al.* [23]). Even proven masking schemes (such as [11,24]) are prone to timing attacks if the Galois field multiplication is not implemented as a computation but as a look-up table.

3.3 Secure Sbox Evaluation

Additional caution must be taken to prevent an attacker from correlating with intermediate data that appears while Eqn. (4) is computed. This computation can be executed whichever the order of the y value. Therefore, there are $(2^n)!$ different permutations possible to shuffle the monomials x^y. In particular, the first operation realized is (almost) never the same, and thus a correlation power analysis is doomed to failure. The standard method to extract information from shuffled or disaligned instructions consists in averaging the side-channel trace over a window large enough to certainly contain the sensitive variable leakage. In the scientific literature, it is referred to as an *integrated DPA attack* [25]. Numerically, the expected correlation coefficient in an integrated DPA attack is divided by the square root of $(2^n)!$ [26, §3.2], *i.e.* divided by 2.9×10^{253}; this justifies why a correlation power analysis is considered impossible. Notice that this is an *intra-sbox* shuffling; it can also be combined with an *inter-sbox* shuffling, *i.e.* a shuffling between the 16 sboxes used by AES during each round.

A new attack on secret executed code consists in uncovering a hidden Markov chain, as used for instance in the removal of a random delay countermeasure [27]. The similarity of the AND/XOR guarantees that such an attack is infeasible in practice.

The AES block cipher calls the sbox, called `SubBytes`, 16 times per round, and those are all identical. But that could be made different in the customized version of the algorithm. One advantage of the *computation* over the *tabular* of the sbox is that the 16 sboxes can be made unique at the cost of a limited overhead. The reason is that the code for the sbox evaluation can be shared, and that only the constants (that is, either $f(y)$ in CNF or the a_y in ANF) change.

3.4 Software Camouflage Masking

It must be noted that despite the random shuffling of the sbox inner computations with ANF, at the end of evaluation, the result shall be stored; such operation might leak unless care is taken. This side effect can be devastating if the sbox computation is stored in memory: the leak is expected to be of similar

amplitude to that of the direct sbox evaluation as a table in RAM! This should not happen in practice, when the sbox ANF formula is computed in registers. But let us assume that the code is written in a high-level language such as C (instead of an assembly language). It thus happens that a leakage exists in the likely case the compiled program ends with a memory transfer. To point this out, two actual implementations of the sbox using a regular look-up-table and the ANF formula (obtained from a C program) have been evaluated in practice, using a setup identical to that of the DPA contest version 4 [28]. The C code is given in source listing 1.2; it has been compiled without optimizations (`-O0` flag in `avg-gcc`). The C code has been compiled without optimizations (`-O0` flag in `avg-gcc`). The code is loaded into an ATMega163 8-bit smartcard, and evaluated on a SASEBO-W platform [29], which is well-designed for side-channel attacks experiments. The measurements are done with a LeCroy wave-runner 6100A oscilloscope by mean of a Langer EMV 3 GHz electromagnetic probe. The smartcard is clocked at 3.57 MHz. Exemplary traces are shown for both ways of evaluating the sbox in Fig. 6. The result of a correlation power analysis (CPA [30]) on 200 traces is represented in Fig. 7. The window corresponds to the sbox evaluation; the horizontal units are samples, knowing that the sampling rate is 500 Msample/s. Clearly, when addressing a look-up-table, the leakage is *evident*. When the implementation is an ANF computation, the CPA profile is *flat*: there is no leakage during the computation, as expected. However, after the computation, when the result is saved in RAM, it leaks[7].

In order to prevent such problem, an option consists in masking the sbox. This operation is trivial with the ANF formula, as it costs only one operation. In the Eqn. (4), if the constant $a_{(0\cdots0)_2}$ is toggled, then the whole sbox is turned into the complementary sbox. Such computation makes it possible to defeat *first degree* attacks (refer for instance to [31]), at a virtually free overhead. This means that the peaks occurring at the end of Fig. 7(b) disappear.

4 Discussion

4.1 Comparison with an Approach Based on Masking the Data and the Code

As already mentioned, masking is costly when it comes to compute nonlinear operations. To solve our problem, namely computing a secret sbox with masked data, the cost of the masking will still be higher. In this section, we estimate the cost of masking at order one the data and the code of an sbox written in a generic way.

As argued by Rivain and Prouff in their seminal paper about hi-order masking on CPUs [11], the masking of nonlinear functions are done more efficiently at the word level rather than at the bit level[8].

[7] Notice that the storage of the sbox result is *one option* when computed in ANF, whereas it is *inherent* (i.e. unavoidable) to the computation with a Look-up-Table.

[8] The work by Kim *et al.* [24] has shown that for some specific problems, *e.g.* when the sbox has a given structure (which is the case of the AES), minor improvements can

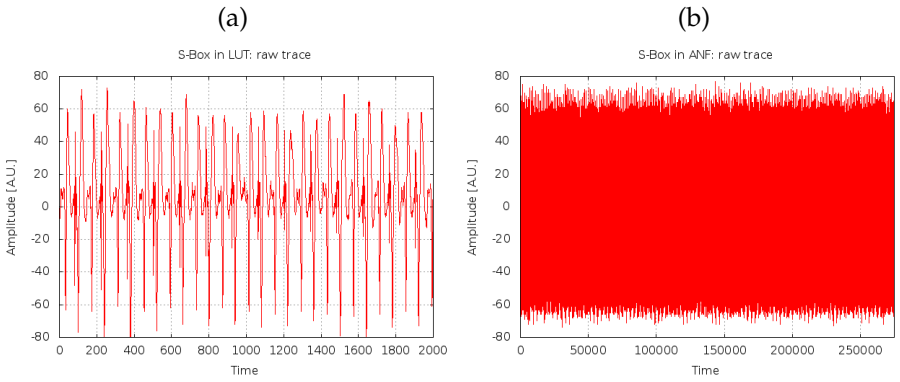

Fig. 6. Example of traces for the (a) *regular* LUT and the (b) *new* ANF implementation of `SubBytes`

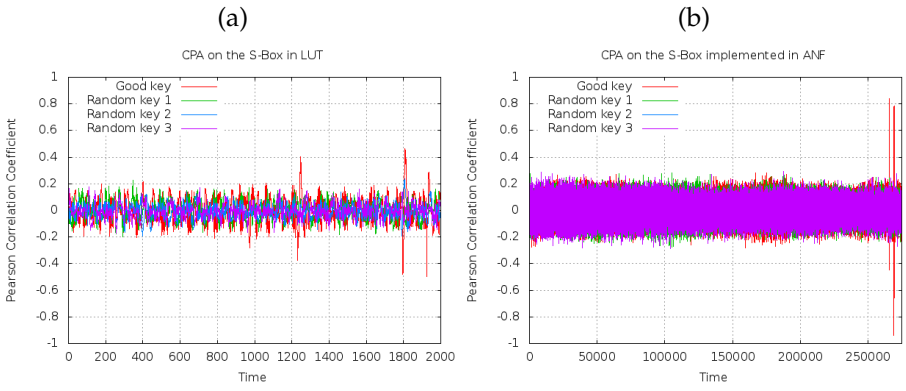

Fig. 7. Correlation Power Analysis of the Sbox (a) in a table, i.e., an array stored in a RAM, and (b) computed using the ANF formula

Therefore, the masking schemes are the most efficient when operating on the Galois field $(\mathbb{F}_{2^n}, \oplus, \odot)$. In this notation, $\oplus$ is the additive law (the XOR), and $\odot$ the multiplicative law. Using the Lagrange interpolation polynomial, any function $f : \mathbb{F}_{2^n} \to \mathbb{F}_{2^m}$ can write:

$$f(x) = \bigoplus_{i=0}^{2^n-2} b_i \odot x^i \ , \tag{5}$$

where, here, x^i is the ith power of $x \in \mathbb{F}_{2^n}$, and where $b_i \in \mathbb{F}_{2^m} \sim \mathbb{F}_2^m$ are constants.

The cost is $2^n - 2$ multiplications and $2^n - 1$ additions. When applying dth-order masking, the overhead is given below [11]:

- Any nonlinear operation requires: $(d+1)^2$ field products, $2d(d+1)$ field additions:
- Any linear operation requires: d field additions.

Thus, Eqn. (5) masked at order $d = 1$ requires:

- $(2^n - 2) \times 4$ field products, and
- $(2^n - 2) \times 4 + (2^n - 1)$ field additions.

A field product can be computed thanks to log-alog tables [32], which represents 6 clock cycles:

- two look-ups in the log table,
- one (modular) addition — which lasts three cycles, one for the addition, one to test the carry flag, and a last conditional reduction (always done to prevent from timing attacks),
- and one look-up in the alog table.

So, if $n = m = 8$, the cost of the computing Eqn. (5) masked at first order is: $254 \times 4 \times 6 + 254 \times 4 + 255 = 7,367$ cycles.

This is more than 3 times more than the shuffled ANF, which requires only $(2^n - 1) \times (n + 1) = 2,295$ byte operations, *i.e.* $2,295$ cycles.

However, it is expected that the proposed scheme is more efficient than masking, because it exploits a property of the hardware (namely the indistinguishability of the AND / OR or AND / XOR opcodes).

4.2 Strong Sboxes with a Compact Encoding

There are $2^{m \times 2^n}$ different sbox $\mathbb{F}_2^n \to \mathbb{F}_2^m$, hence $m \times 2^n$ bits of data is required to parametrize them. This number applies to general block ciphers, such as Feistel networks, that do not require the sboxes to be invertible. Some other generic

be got by computing on half-words, *e.g.* on nibble instead of bytes. But this result does not negate the noting by Rivain and Prouff that computing masking schemes on larger bitwidths is faster than computing at the bit level.

constructions of block ciphers, such as the substitution-permutation networks (SPN), need the sboxes to be invertible. In this case, $n = m$ and the number of sboxes is equal to $(2^n)! < 2^{n \times 2^n}$. Still, this number is very large.

It is certainly possible to use fewer parameters (*e.g.* 128 bits only, so as to make the complexity of the SCARE as hard as the key recovery) while still describing a cryptographically strong sbox. As this work is a trade-off between security and efficiency, it is left as an open issue for the interested reader.

4.3 Vulnerability of Secret Sboxes against Fault Injection Attacks

Fault injection attacks consist in perturbing a device in such a way to get incorrect results, that nonetheless contain some information on the secrets. This information is typically recovered by comparing the outputs of a correct encryption and of a faulted one. Naturally, fault injection attacks can be tuned to serve as the recognition of secret sboxes. In this case, the attack is termed fault injection for reverse-engineering (FIRE). Clearly, unless actions are taken, our countermeasure is vulnerable to FIRE attacks.

However, there exist simple ways to make those attacks inefficient. First of all, the randomization of operations within an sbox computation (as promoted in Sec. 3.3), as well as the possible randomization of sboxes evaluation order in an AES round, significantly complexifies fault injection attacks, as the position and the reproducibility of faults cannot be ascertained. Another option is to check the computation based on the assertion of an invariant. For example, the computation can be carried out nominally, which turns a plaintext into a ciphertext. Then, before outputting this ciphertext, the latter is injected in the decryption cipher. If the encryption and the decryption were successful, then the decrypted cipher shall match with the plaintext. Now, it is very unlikely that a fault during decryption manages to cancel the effect of a fault in encryption mode. The reason is that fault injection is usually modelled as a probabilistic effect, and that encryption is different from decryption. A third option consists to deny the possibility for the attacker to submit twice the same plaintext. This can be done easily simply by enforcing a mode of operation that makes use of a random initialization vector.

The existence of these simple practices justifies that FIRE is not considered a plausible threat.

5 Conclusions

It is discovered in this paper that logical operations of a CPU are almost indistinguishable using side-channel analysis. Therefore, generic constructions of functions computations can be devised. We jointly refer to them as *software camouflage*. We show that secret (customized) substitution boxes can be computed without simple side-channel leakage (since all opcodes used in the computation are alike), and without vulnerability against differential side-channel leakage (by a shuffling of the operations). In addition, the computation result

of this software camouflage can be masked at virtual no cost, which secures the leakage that can happen when moving this value (e.g., by a transfer from a register to a RAM). The resulting construction is shown to be more than three times faster than the state-of-the-art solution, namely the masking of the sbox computation on masked data. This result shows that the *a priori* knowledge of a device leakage can be constructively exploited to reduce the overhead of protections against side-channel analyses.

Acknowledgments

Parts of this work have been funded by the MARSHAL+ (*Mechanisms Against Reverse-engineering for Secure Hardware and Algorithms*) FUI #12 project, co-labellized by competitivity clusters System@tic and SCS.

We also thank the audience from PHISIC '13 for a positive feedback on this research.

References

1. Novak, R.: Side-Channel Attack on Substitution Blocks. In: ACNS. Volume 2846 of LNCS., Springer (2003) 307–318 Kunming, China.
2. Novak, R.: Sign-Based Differential Power Analysis. In: WISA. Volume 2908 of LNCS., Springer (2003) 203–216 Jeju Island, Korea.
3. Novak, R.: Side-Channel Based Reverse Engineering of Secret Algorithms. In Zajc, B., ed.: Proceedings of the Twelfth International Electrotechnical and Computer Science Conference (ERK 2003), Ljubljana, Slovenia, Slovenska sekcija IEEE (2003) 445–448
4. Clavier, C.: An Improved SCARE Cryptanalysis Against a Secret A3/A8 GSM Algorithm. In: ICISS. Volume 4812 of LNCS., Springer (2007) 143–155 Delhi, India. DOI: 10.1007/978-3-540-77086-2_11.
5. Daudigny, R., Ledig, H., Muller, F., Valette, F.: SCARE of the DES. In: ACNS. Volume 3531 of LNCS., Springer (2005) 393–406 New York, NY, USA.
6. Fournigault, M., Liardet, P.Y., Teglia, Y., Trémeau, A., Robert-Inacio, F.: Reverse Engineering of Embedded Software Using Syntactic Pattern Recognition. In: On the Move to Meaningful Internet Systems: OTM 2006 Workshops. Volume 4277 of LNCS., Springer (2006) 527–536 Montpellier, France, DOI: 10.1007/11915034.
7. Vermoen, D., Witteman, M.F., Gaydadjiev, G.: Reverse Engineering Java Card Applets Using Power Analysis. In Sauveron, D., Markantonakis, C., Bilas, A., Quisquater, J.J., eds.: WISTP. Volume 4462 of Lecture Notes in Computer Science., Springer (2007) 138–149 Heraklion, Greece.
8. Amiel, F., Feix, B., Villegas, K.: Power analysis for secret recovering and reverse engineering of public key algorithms. In: Selected Areas in Cryptography. (2007) 110–125 Ottawa, Ontario, Canada.
9. Réal, D., Dubois, V., Guilloux, A.M., Valette, F., Drissi, M.: SCARE of an Unknown Hardware Feistel Implementation. In: CARDIS. Volume 5189 of LNCS., Springer (2008) 218–227 London, UK.
10. Mangard, S., Oswald, E., Popp, T.: Power Analysis Attacks: Revealing the Secrets of Smart Cards. Springer (2006) ISBN 0-387-30857-1, http://www.dpabook.org/.

11. Rivain, M., Prouff, E.: Provably Secure Higher-Order Masking of AES. In Mangard, S., Standaert, F.X., eds.: CHES. Volume 6225 of LNCS., Springer (2010) 413–427
12. Hsu, L.L., Joshi, R.V., Kruger, D.W.: Techniques for impeding reverse engineering (2011) IBM. Patent US 7994042 B2.
13. Brier, E., Fortier, Q., Korkikian, R., Magld, K.W., Naccache, D., de Almeida, G.O., Pommellet, A., Ragab, A.H., Vuillemin, J.: Defensive Leakage Camouflage. [33] 277–295
14. Guilley, S., Sauvage, L., Flament, F., Hoogvorst, P., Pacalet, R.: Evaluation of Power-Constant Dual-Rail Logics Counter-Measures against DPA with Design-Time Security Metrics. IEEE Transactions on Computers **9** (2010) 1250–1263 DOI: 10.1109/TC.2010.104.
15. Kessner, D.: Free VHDL 6502 core (2000) `http://www.free-ip.com/` is no longer available, but `http://web.archive.org/web/20040603222048/http://www.free-ip.com/6502/index.html` is.
16. Schindler, W., Lemke, K., Paar, C.: A Stochastic Model for Differential Side Channel Cryptanalysis. In LNCS, ed.: CHES. Volume 3659 of LNCS., Springer (2005) 30–46 Edinburgh, Scotland, UK.
17. Marion, D., Wurcker, A.: Read/Write Signals Reconstruction Using Side Channel Analysis for Reverse Engineering (2013) COSADE, 2013; Short talk. TELECOM-ParisTech, Paris, France.
18. Guilley, S., Sauvage, L., Micolod, J., Réal, D., Valette, F.: Defeating Any Secret Cryptography with SCARE Attacks. In: LatinCrypt. Volume 6212 of LNCS., Springer (2010) 273–293 Puebla, México, DOI: 10.1007/978-3-642-14712-8_17.
19. Prouff, E., Rivain, M.: A Generic Method for Secure SBox Implementation. In Kim, S., Yung, M., Lee, H.W., eds.: WISA. Volume 4867 of Lecture Notes in Computer Science., Springer (2007) 227–244
20. Chevallier-Mames, B., Ciet, M., Joye, M.: Low-Cost Solutions for Preventing Simple Side-Channel Analysis: Side-Channel Atomicity. IEEE Trans. Computers **53** (2004) 760–768
21. Veyrat-Charvillon, N., Medwed, M., Kerckhof, S., Standaert, F.X.: Shuffling against Side-Channel Attacks: A Comprehensive Study with Cautionary Note. In Wang, X., Sako, K., eds.: ASIACRYPT. Volume 7658 of Lecture Notes in Computer Science., Springer (2012) 740–757
22. Carlet, C.: Boolean Functions for Cryptography and Error Correcting Codes: Chapter of the monography Boolean Models and Methods in Mathematics, Computer Science, and Engineering. Cambridge University Press, Y. Crama and P. Hammer eds (2010) On-line version available at `http://www.math.univ-paris13.fr/~carlet/chap-fcts-Bool-corr.pdf`.
23. Bernstein, D.J., Chou, T., Schwabe, P.: McBits: Fast Constant-Time Code-Based Cryptography. In Bertoni, G., Coron, J.S., eds.: CHES. Volume 8086 of Lecture Notes in Computer Science., Springer (2013) 250–272
24. Kim, H., Hong, S., Lim, J.: A Fast and Provably Secure Higher-Order Masking of AES S-Box. In Preneel, B., Takagi, T., eds.: CHES. Volume 6917 of LNCS., Springer (2011) 95–107
25. Clavier, C., Coron, J.S., Dabbous, N.: Differential Power Analysis in the Presence of Hardware Countermeasures. In Koç, c.K., Paar, C., eds.: CHES. Volume 1965 of Lecture Notes in Computer Science., Springer (2000) 252–263
26. Rivain, M., Prouff, E., Doget, J.: Higher-Order Masking and Shuffling for Software Implementations of Block Ciphers. In: CHES. Volume 5747 of Lecture Notes in Computer Science., Springer (2009) 171–188 Lausanne, Switzerland.

27. Durvaux, F., Renauld, M., Standaert, F.X., van Oldeneel tot Oldenzeel, L., Veyrat-Charvillon, N.: Efficient Removal of Random Delays from Embedded Software Implementations Using Hidden Markov Models. [33] 123–140
28. TELECOM ParisTech SEN research group: DPA Contest (4th edition) (2013–2014) http://www.DPAcontest.org/v4/.
29. RCIS-AIST, J.: SASEBO (Side-channel Attack Standard Evaluation Board, Akashi Satoh) development board: http://www.risec.aist.go.jp/project/sasebo/ (2013)
30. Brier, É., Clavier, C., Olivier, F.: Correlation Power Analysis with a Leakage Model. In: CHES. Volume 3156 of LNCS., Springer (2004) 16–29 Cambridge, MA, USA.
31. Bhasin, S., Danger, J.L., Guilley, S., Najm, Z.: A Low-Entropy First-Degree Secure Provable Masking Scheme for Resource-Constrained Devices. In: Proceedings of the Workshop on Embedded Systems Security. WESS '13, New York, NY, USA, ACM (2013) Montreal, Canada.
32. Daemen, J., Rijmen, V.: The Design of Rijndael: AES - The Advanced Encryption Standard. Springer (2002)
33. Mangard, S., ed.: Smart Card Research and Advanced Applications - 11th International Conference, CARDIS 2012, Graz, Austria, November 28-30, 2012, Revised Selected Papers. In Mangard, S., ed.: CARDIS. Volume 7771 of Lecture Notes in Computer Science., Springer (2013)

A Algorithms Source Code

It is shown by Carlet in [22, page 11] that there exists a simple divide-and-conquer butterfly algorithm to compute the ANF from the truth-table (or vice-versa). It is called the "Fast Möbius Transform". An implementation in `python` is given in code listing 1.1, for $n \to 1$ Boolean functions. As already underlined in Sec. 3.2, the very same code also works for $n \to n$ vectorial Boolean functions.

Code Listing 1.1. Truth table to ANF table transformation

```
from operator import xor

n = 8 # Problem's size (we handle bytes, i.e., 8-bit words)

def tt2anf( S ):
  """Truth table to ANF"""
  # S is an array of 256 bits or bytes
  # (depending the function is Boolean or vectorial Boolean)
  h = {}
  h[0] = {}
  for a in range( 1<<n ):
    h[0][a] = [S[a]]
  for k in range( n ):
    h[k+1] = {}
    for b in range( 1<<(n-k-1) ):
      #                 the "+" expresses the concatenation
      h[k+1][b] = h[k][2*b] + map( xor, h[k][2*b], h[k][2*b+1] )
  return h[n][0]
```

The application of the code listing 1.1 to $f = \texttt{SubBytes}$ (array noted `S_TT`) is given as `S_AND` in the code listing 1.2. The values in the array `S_TT` are $\{f(y), y \in \mathbb{F}_2^8\}$, in this order, whereas the values in the array `S_ANF` are $\{a_y, y \in \mathbb{F}_2^8\}$ (recall Eqn. (4)). In the same code listing, the function `anti_scare_eval` applies `SubBytes` on a byte x, with the formula of Eqn. (4). Furthermore, in this code, the y's are shuffled (See Sec. 3.3) by a simple XOR with a random byte r.

Code Listing 1.2. Shuffled evaluation of `SubBytes` with the ANF computation

```
/** Portable C types. We use uint8_t */
#include <stdint.h>

/** Original SubBytes [provided for self-containedness] */
static uint8_t S_TT[] = {
  0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,
  0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,
  0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,
  0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,
  0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,
  0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,
  0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,
  0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,
  0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,
  0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,
  0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,
  0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,
  0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,
  0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,
  0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,
  0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16 };

/** SubBytes under ANF form [obtained by the python script tt2anf] */
static uint8_t S_ANF[] = {
  0x63, 0x1f, 0x14, 0x13, 0x91, 0x86, 0x89, 0x20, 0x53, 0x2e, 0x43, 0x6e, 0x5f, 0x9e, 0x8b, 0xa9,
  0xa9, 0x57, 0x17, 0xef, 0xa1, 0x6d, 0x37, 0xc8, 0x34, 0x1f, 0x4f, 0xe6, 0x5e, 0x34, 0xd4, 0xbf,
  0xd4, 0x55, 0x30, 0xec, 0x10, 0xc5, 0x6c, 0xed, 0xd0, 0xf5, 0xb6, 0x14, 0x9b, 0xe5, 0xff, 0x6c,
  0x1a, 0xde, 0x14, 0x33, 0x3c, 0x63, 0xe8, 0x37, 0xb4, 0x12, 0x1a, 0xc8, 0x6a, 0xdb, 0x44, 0x34,
  0x6a, 0x95, 0x31, 0xaf, 0x83, 0x79, 0xed, 0x13, 0x08, 0xcd, 0xe2, 0xde, 0x36, 0xc2, 0x6d, 0xf7,
  0xf3, 0x5f, 0x61, 0x3c, 0xc0, 0xcc, 0x91, 0xa2, 0x56, 0xdf, 0x69, 0x1f, 0x64, 0x91, 0x36, 0x0f,
  0x0d, 0xe0, 0x6f, 0x3e, 0x91, 0x0b, 0x02, 0x08, 0x1e, 0x95, 0x2a, 0x0b, 0x74, 0x58, 0x9b, 0x7e,
  0xc1, 0x1b, 0x09, 0xb3, 0x0d, 0x0e, 0xff, 0x74, 0xae, 0xa9, 0x76, 0x52, 0xb9, 0x87, 0x1a, 0x08,
  0xae, 0xde, 0xca, 0x2d, 0x03, 0x8f, 0x4c, 0x85, 0x5a, 0x8c, 0x27, 0x70, 0x6d, 0xcd, 0x89, 0x7f,
  0x04, 0x77, 0xe6, 0xa3, 0x71, 0x8d, 0x6f, 0x0f, 0x1b, 0xf4, 0xfa, 0x8e, 0xb6, 0xa6, 0x60, 0x5f,
  0xf9, 0x46, 0x34, 0x30, 0x2b, 0x51, 0x1e, 0x9d, 0xfb, 0x94, 0x66, 0x37, 0x53, 0x3e, 0x51, 0x29,
  0xb0, 0x03, 0xef, 0xe8, 0x2f, 0x69, 0x14, 0xef, 0x32, 0x2f, 0x53, 0xcc, 0x1b, 0x93, 0x1c, 0x10,
  0x1d, 0x96, 0x70, 0x58, 0xb7, 0x08, 0x1f, 0xb7, 0x53, 0x98, 0x85, 0x57, 0x01, 0x2a, 0x04, 0x89,
  0x94, 0xf3, 0xca, 0x24, 0x8e, 0x51, 0x85, 0x4a, 0x3a, 0xd9, 0x2c, 0xc7, 0x56, 0xae, 0x3f, 0x17,
  0x7b, 0x28, 0x8d, 0xbb, 0x84, 0x4e, 0xd9, 0x43, 0x1d, 0x9f, 0x9c, 0xc4, 0x64, 0x8f, 0x3a, 0x2e,
  0xcc, 0x7e, 0xd4, 0x05, 0x3b, 0xa4, 0x29, 0x63, 0xdc, 0x10, 0xc3, 0xce, 0x0d, 0x9d, 0x04, 0x00 };

/**
 * Application of S_TT on input "x" via the ANF table (S_ANF),
 * with a random ordering equal to "yr" (yr = y XOR r),
 * where "r" is a uniformly distributed byte
 */
uint8_t anti_scare_eval( uint8_t* S_ANF, uint8_t x, uint8_t r )
{
  uint8_t result = 0x00u;
  uint8_t y = 0x00u, yr;

  do
  {
    yr = y^r;
    result ^= S_ANF[yr] & ((( x | ( 0xffu^yr )) == 0xffu ) ? 0xffu : 0x00u );
  }
  while( y++ != 0xffu );

  /* At this stage, the variable "result" is equal to S_TT[x], no matter the value of the shuffling random
       parameter "r" */
  return result;
}
```