Combining loop shuffling and code polymorphism for enhanced AES side-channel security

Nicolas Belleville, Loïc Masure

▶ **To cite this version:**

Nicolas Belleville, Loïc Masure. Combining loop shuffling and code polymorphism for enhanced AES side-channel security. COSADE 2024 - 15th International Workshop on Constructive Side-Channel Analysis and Secure Design, Apr 2024, Gardanne, France. pp.260-280, 10.1007/978-3-031-57543-3_14 . cea-04539481

HAL Id: cea-04539481
https://cea.hal.science/cea-04539481

Submitted on 9 Apr 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Combining Loop Shuffling and Code Polymorphism for Enhanced AES Side-Channel Security

Nicolas Belleville[1], Loïc Masure[2]

[1] Univ. Grenoble Alpes, CEA, List, F-38000 Grenoble, France
nicolas.belleville@cea.fr
[2] LIRMM, Univ. Montpellier, CNRS
loic.masure@lirmm.fr

Abstract. Combining countermeasures against side-channel attacks represents a promising approach to defend against powerful attackers. Existing works on this topic show that the hope for a significant increase of security is sometimes fulfilled, although not always. In this paper, we consider the combination of two hiding countermeasures, namely loop shuffling and code polymorphism. We study the combination on a custom implementation of AES, tailored to ease shuffling while providing a balance between performance and RAM usage. Our experimental study exploits real-world traces and simulated noiseless traces. On real-world traces, we show that code polymorphism effectively mitigates leakage stemming from the permutation variable employed for loop shuffling, and that both countermeasures resist surprisingly well to a deep learning attack that showed great success against code polymorphism in a former work. On simulated traces, we show that combining the countermeasures complicates both a simple CPA and a deep learning attack. As is, the combination of these countermeasures seems beneficial and should be particularly relevant in any context where loop shuffling benefits vanish due to the leakage of its permutation variables.

Keywords: side-channel attacks, countermeasures, hiding, loop shuffling, code polymorphism, aes, deep learning

1 Introduction

Side-channel attacks threaten the security of embedded systems and IoT devices. In particular, they are famous for how easily they break unprotected cryptography implementations. As the confidentiality of cryptographic keys is critical for the whole security of such systems, there is a strong need for countermeasures.

Two classes of countermeasures have emerged from this need: masking countermeasures and hiding countermeasures [19]. Masking countermeasures rely on the idea of secret-sharing, applied to all sensitive variables manipulated by the implementation. Hiding countermeasures leverage various techniques in order to lower the signal-to-noise ratio, but usually do not involve a modification of

variables manipulated by the implementation. Despite the large number of countermeasures developed along the years, side-channel analysis remains an active area of research.

While many hiding countermeasures have been explored and attacked, they have been studied mostly in isolation: the interest of their combination in general is still an open question. We notice in particular that code polymorphism and loop shuffling seem complementary from a theoretical point of view, as they operate on different scales: loop shuffling shuffles large sequences of instructions without modifying them, while code polymorphism acts by inserting, replacing, and shuffling instructions without being able to shuffle very large sequences. Moreover, attacks against loop shuffling frequently exploit leakage of the permutation used, and we believe that code polymorphism could help hiding this leakage. In this paper, we ask whether these remarks translate into a security gain in practice, i.e., whether code polymorphism and loop shuffling actually benefit from each other. As such, we propose to combine both countermeasures to protect an AES implementation.

Contributions

- We study this combination on a custom implementation of AES that features an execution-time faster than by-the-book 8-bit implementations while having a lower memory usage than T-table implementations.
- We pinpoint the interest of having dynamic code transformations in order to make loop iterations length less consistent, and propose the use of dynamic variants in code polymorphism as a response.
- We show that code polymorphism effectively hides the leakage of loop shuffling parameters.
- We consider both real-world traces as well as simulated noiseless traces to study the countermeasures' response to CPA and to a deep learning attack, and notice (1) how both countermeasures resist surprisingly well to the deep learning attack in the real-world setting, either alone or combined, (2) how combining the countermeasures increases the difficulty of both the CPA and the deep learning attack in the simulation setting.
- We release the dataset used for deep learning along with our paper, to allow future work to continue comparing the security of the different implementations considered.

2 Background & Related Work

2.1 Loop shuffling

Loop shuffling consists in executing independent loop iterations in a random order. Different strategies can be used for this purpose: (1) using a random start index [28], (2) performing a random walk [23], (3) generating a random permutation [28].

In the random start index strategy, one random number is drawn and the loop iterations are executed starting from this number. The ith iteration index is given by $i + s \mod b$, where s is the random start index and b is the loop bound. This strategy has a low randomness cost, at the price of a low number of achievable permutations. For a loop iterating 2^k times, this strategy only gives 2^k variants.

The random walk strategy extends the random start index strategy by using a more complex formula for iterating, while still using few random numbers. For instance, a random increment can be drawn as long as it is co-prime with the number of loop iterations. The ith iteration index is given by $i \cdot c + s \mod b$, where c is the random increment, s is the random start index and b is the loop bound. This strategy keeps a low randomness cost, while increasing notably the number of achievable permutations. For a loop iterating 2^k times, this strategy gives $2^k \cdot 2^{(k-1)}$ variants, as all odd numbers can be used as increments.

Finally, the random permutation strategy consists in drawing a random permutation among all possible ones, e.g. using Fisher-Yates algorithm. While this strategy is costly in terms of randomness in general, it is the only one that reaches the maximum number of variants. For a loop iterating 2^k times, this strategy gives $2^k!$ variants.

2.2 Code polymorphism

Code polymorphism consists in varying the sensitive code from one execution to another. Its effectiveness relies on several principles: (1) introduce temporal desynchronisation between the attacker's traces, (2) modify the leakage profile, (3) increase noise.

The countermeasure was first proposed by Amarilli et al. [5], using a program graph that guides a dynamic rewriting process to shuffle independent operations. In the same vein, Luo et al. [17] proposed later on a method to detect the independence between statements and to automatically shuffle them, this time without any rewriting. Malagón et al. [18] proposed to generate several versions of a same function using slightly varying compiler optimisation options, and to randomly choose between them at runtime. Meanwhile, Agosta et al. [3] proposed the use of dynamic code morphing engine to make the code vary using various code transformations. Couroussé et al. [14,9] later on followed a similar idea, using dynamic code generators. Agosta et al. [4] also proposed the use of switch statements generated statically in the code and controlled dynamically by random variables in order to avoid any dynamic code modification. Finally, Antognazza et al. [6] proposed a hardware approach to morph the code without any software change and with a smaller execution time overhead.

In this paper, we follow and extend the method proposed by Belleville et al. [9], based on dynamic code generation, that we present in more details below. Figure 1 presents the application flow of the countermeasure when using this approach. The countermeasure is automatically applied at compilation time on any function labelled as sensitive by the developer. The source-to-source compiler `Odo` is in charge of transforming the code for this purpose. `Odo` replaces any

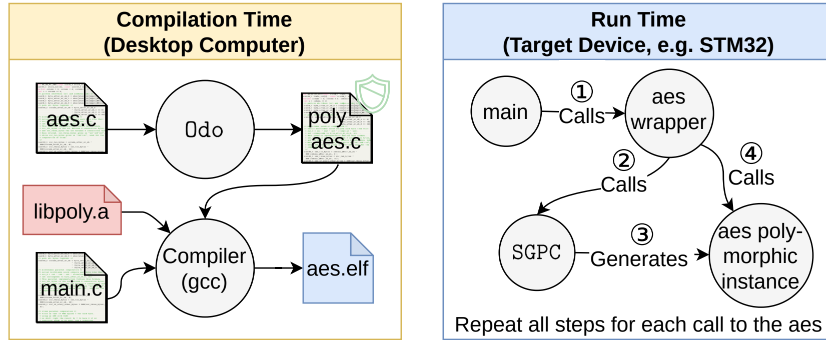


Fig. 1: Compilation flow and runtime code execution for a polymorphic AES, similar to [9]. `Odo` is a compiler that automatically applies code polymorphism. `SGPC` is a runtime generator of polymorphic code.

labelled function by a wrapper and a specialized generator of polymorphic code (`SGPC`), whose behaviour will be explained later on. At run time, the wrapper intercepts any call to the labelled function. The wrapper calls the `SGPC` to generate the new code in memory, and then calls the generated code (a.k.a. the `polymorphic instance`). At every call, the `polymorphic instance` is different from one another as the `SGPC` generates the code applying various code transformations guided by randomness. The `SGPC` has inherent knowledge of a reference assembly code, and generates variants of that code. It supports the following code transformations:

- *Random register permutation*: the `SGPC` draws a random permutation among the general purpose registers (except registers used for argument passing). For instance, the `SGPC` could emit an instruction using `R11` instead of `R5`. The change is global for all instructions to preserve the program semantics.
- *Instruction shuffling*: the `SGPC` randomly chooses the order of instructions that are independent of each other.
- *Semantic variants*: for some instructions, the `SGPC` knows several semantically equivalent sequences of instructions. The `SGPC` randomly chooses one of them. As an example, a `xor` can be replaced by a sequence of an `and`, an `or`, and a `xor`: $a \oplus b = (a \wedge b) \oplus (a \vee b)$.
- *Noise instructions*: the `SGPC` inserts a random number of useless instructions in between useful instructions. The noise instructions are randomly chosen among various frequently-used instructions.
- *Dynamic noise*: the `SGPC` sometimes inserts a sequence of contiguous noise instructions preceded by a branch instruction that will randomly jump inside the sequence during the execution of the polymorphic instance. A register is reserved to hold random data that is used at runtime to determine the branch offsets. This register value is updated throughout the execution by noise instructions, and is saved and restored between calls to make code execution different from one execution to the next. Dynamic noise's purpose is to partly

decorrelate what happens during code generation and code execution, and to maintain code variability even in between calls to the `SGPC` when the code generation is done less frequently for performance reasons.

The granularity of these transformations differs from the one of loop shuffling: they cannot permute large code sequences as a whole such as loop iterations, but they can largely impact the assembly instructions executed.

The code polymorphism countermeasure can be configured to use any subset of these code transformations, and some code transformations have an unlimited number of possible configurations: for noise instructions, the probability distribution used to decide how many noise instructions to insert can be tuned. For the rest of this paper, we will consider all transformations are enabled and code is regenerated before every execution. Also, we will use this probability law to determine how many noise instructions are inserted in between useful instructions:

$$P[X=0] = \frac{7}{8} + \frac{1}{64}$$
$$\forall i \in [1,7], P[X=i] = \frac{1}{64}$$

2.3 Attacks against loop shuffling and code polymorphism

To the best of our knowledge, most attacks against shuffling rely on template attacks [28,13,27], although some recent work uses neural networks [22]. The proposed attacks either sequentially exploit leakages from the permutation and the secret-related intermediate variables, or jointly exploit their leakages.

Attacks against code-polymorphism attempt to limit the effects of desynchronization through various ways: (1) pattern extraction using correlation to extract SBox related code [2], (2) re-alignment techniques combined with CPA or with template attacks [21], and (3) use of shift-invariant deep neural networks like convolutional neural networks [21]. The latter deep neural network approach stands out for its efficiency: once trained, about 10-20 traces were enough to find all key bytes on the considered AES implementation on a low-noise platform, while several hundreds of thousands of traces were needed with re-aligned CPA or template attacks. From this work, we additionally note that the trace amplitude changes when executing code in flash (before polymorphic instance) and in RAM (the polymorphic instances). It is probable that clear patterns visible during polymorphic instance execution are due to data flash access to the SBox [20, Fig. 7.10, p. 121]. They are the only flash accesses performed as instruction fetch loads the polymorphic instance code from RAM. Such flaw could have been implicitly leveraged by the trained model.

3 Countermeasures combination

Approaches combining countermeasures share a common goal, and so does our paper: find whether combining the countermeasures improves significantly security. Rivain et al. first showed that masking and shuffling could be advantageously

combined [25]. This idea was extended by Azouaoui et al. that compared the various possibilities of shuffling enabled by higher-order masking, and showed as a result that the security gain from shuffling could be amplified [7]. With affine masking, Fumaroli et al. combined multiplicative masking and Boolean masking and suggested this combination leads to much improved resistance to higher-order attacks [15]. Affine masking was latter on combined with shuffling [11,10]. Finally, Udvarhelyi et al. examined the combination of rekeying with masking and shuffling for ISAP cipher and showed that such combination hardly leads to any significant security improvement on low noise platforms [27].

4 Approach

Our main idea is to combine loop shuffling with code polymorphism in order to strengthen security. This approach builds onto the difference between both countermeasures. On the one hand, code polymorphism introduces desynchronization and alters leakage at instruction level. On the other hand, it is not able to reorder large code blocks such as loop iterations, while loop shuffling enables this possibility. As a result, the desynchronization obtained from the combination of both countermeasures is expected be much stronger. In addition, the fine-grain effect provided by code-polymorphism should prevent any easy recovering of loop shuffling parameters. For this study, we choose AES as a benchmark, and develop a custom implementation for this purpose.

4.1 AES implementation

We propose in this paper an AES implementation with less memory usage than T-tables implementations, while being faster than an 8bits by-the-book implementation. Our AES implementation exploits the following principles:

- Round functions are combined to avoid useless (and leaky) reads and writes of the state bytes in memory.
- The AES state is duplicated: during each round, we have different arrays for input state and output state, which enables the combination of all round functions, `shiftRows` included.
- We leverage parallelism as much as possible by doing computations in 32-bit words for `mixColumns` and `addRoundKey`.
- The key schedule is precomputed.
- SBox is stored in RAM: (1) RAM access are faster than flash accesses, (2) flash memory accesses induce clearly visible leakage when executing polymorphic instances as code is stored in RAM, (3) memory usage of our AES implementation is limited and having the SBox in RAM is not prohibitive.

The AES function first calls `addRoundKey`, that is performed word by word, then calls the combined round function `SR_SB_MC_ARK`, presented in Listing 1, once for each round, except for the last round where a similar function is called, without the `mixColumns` steps. The combined round function first performs the `shiftRows` and `subBytes` on state bytes, then packs all bytes from a column into a 32-bit word, and performs `mixColumns` and `addRoundKey` on 32-bit words.

```c
#define ROR(in, a) (((in) >> (a)) ^ ((in) << (32 - (a))))

// gives shiftRows input index from an output index.
// shiftRows would map state[input_SR(i)] to state[i]
uint32_t input_SR(uint32_t dest) {
  return (dest + ((dest & 3) << 2)) & 15;
}

void SR_SB_MC_ARK(const uint8_t state_in[16],
  uint32_t state_out[4], const uint32_t key[4]) {
  for(int column = 0; column < 4; column++) {
    int i = column << 2;
    // perform shiftRows (sr) and subBytes (sb)
    uint8_t byte_sr_sb_0 = sbox[state_in[input_SR(i)]];
    uint8_t byte_sr_sb_1 = sbox[state_in[input_SR(i+1)]];
    uint8_t byte_sr_sb_2 = sbox[state_in[input_SR(i+2)]];
    uint8_t byte_sr_sb_3 = sbox[state_in[input_SR(i+3)]];
    // pack all bytes together
    uint32_t column = byte_sr_sb_0 ^ byte_sr_sb_1<<8
                    ^ byte_sr_sb_2<<16 ^ byte_sr_sb_3<<24;

    //------- mixColumns parallel computation --------
    // serial mixColumns would compute each byte with:
    // out_0 = in1 ^ in2 ^ in3 ^ xtime(in0 ^ in1);
    // in0 corresponds to byte_sr_sb_0, etc...
    // This parallel implementation follows the same formula.
    // We call xor3 the result of in1 ^ in2 ^ in3,
    // xor2 the result of in0 ^ in1, and xt_xor2 the
    // result of xtime(in0 ^ in1).
    uint32_t xor2 = column ^ ROR(column, 8);
    uint32_t xor3 = ROR(xor2 ^ ROR(column, 16), 8);
    //------- xtime parallel computation --------
    // first we keep only the MSB of each byte
    uint32_t msbs = xor_two_bytes & 0x80808080;
    // MSB>>7 == 0 if no reduction is needed, 1 otherwise.
    uint32_t reduc_bool = msbs >> 7;
    // adding 0x1F gives 0x1F if no reduction is needed,
    // 0x20 otherwise. Then the NOT gives 0b11011111 if a
    // reduction is needed, and 0b11100000 otherwise.
    uint32_t reduc_mask = ~(reduc_bool + 0x1F1F1F1F);
    // AND with 0x1B to get 0x1B if a reduction is needed
    uint32_t reducer = reduc_mask & 0x1B1B1B1B;
    // shift left least significant bits and reduce
    uint32_t xt_xor2 = ((xor2 & 0x7F7F7F7F) << 1) ^ reducer;

    uint32_t mixcolumn_result = xor3 ^ xt_xor2;
    // perform addRoundKey on 32bits and store back result
    state_out[column] = mixcolumn_result ^ key[column];
  }
}
```

Listing 1: Implementation of AES round function that performs `subBytes`, `shiftRows`, `mixColumns` and `addRoundKey` in one loop.

```
uint8_t perm[24] = {0x1b, 0x1e, 0x27, 0x2d, 0x36, 0x39,
  0x4b, 0x4e, 0x63, 0x6c, 0x72, 0x78, 0x87, 0x8d, 0x93,
  0x9c, 0xb1, 0xb4, 0xc6, 0xc9, 0xd2, 0xd8, 0xe1, 0xe4};
```

Listing 2: Table containing byte encodings of all possible permutations between numbers strictly lower than 4. Each permutation of 4 2-bits values is encoded in one byte consisting of the concatenation of the 4 values.

4.2 Loop shuffling

The loop shuffling countermeasure consists in executing loop iterations in a random order. As explained in Section 2.1, loop shuffling can be implemented through the use of random start index, random walks, or random permutations. The random permutations strategy gives the largest amount of possibilities, at the cost of being more costly to implement in general.

As we iterate on columns, the opportunities of loop shuffling are reduced compared to a loop iterating on all bytes of the state. In order to maximise as much as possible the countermeasure impact, despite the low number of iterations, we decided to use a random permutation strategy, and to randomly shuffle as well the memory accesses and computations done for each of the 4 bytes used within the loop. This shuffles the use of the 4 bytes within the `shiftRows` and `subBytes` steps, while the `mixColumns` and `addRoundKey` still operates on the 4 bytes at once, packed in a 32bits word.

Such strategy gives $4! \times (4!)^4 = 331,776$ different possibilities when the whole loop is considered, as we make a first permutation choice for the loop iterations, and we then choose again how to shuffle the bytes 4 times (once per iteration).

Random permutation generation. Random permutation generation is costly in general but the restricted number of iterations and bytes to shuffle gives an opportunity to have a much faster permutation generation. We propose to tabulate all $4! = 24$ possible permutations. Such a table makes permutation generation very simple: a random number in $[0, 24[$ is drawn, and the corresponding permutation in the table is chosen.

We generate the random number in $[0, 24[$ as follows: we draw a random number on 32bits, discard the 2 most significant bits, and consider the remaining bits as a fixed point decimal value in the range $[0, 8[$, with 3 bits encoding the integer part and 27 bits encoding the decimal part. Multiplying this value by 3, which can be done in one instruction by adding the value with a shifted version of itself in ARM-Thumb2, gives us a fixed point decimal value in the range $[0, 24[$, with 5 bits encoding the integer part and 27 bits encoding the decimal part. Keeping 5 most significant bits then gives us an integer in $[0, 24[$. Such technique allows a good trade-off between speed and statistical uniformity, as all possible resulting integers probabilities can differ only by approximately $2^{-27} \approx 10^{-8}$. It also avoids the use of a costly modulo.

The table storing permutations encodes permutations as bytes. Each byte contains 4 2-bits integers, that give the list of 4 permuted values. More formally, for any permutation $p =$`{a,b,c,d}` of 2-bits integers, the encoding used is: `(a<<6)^(b<<4)^(c<<2)^d`. Listing 2 shows the obtained permutation table. Once one permutation byte is drawn from the table, the index of each iteration is obtained by extracting successively the 4 2-bits values composing the encoding.

Impact on AES implementation. The AES implementation with shuffling closely follows the implementation described in Section 4.1. In particular, all the `mixColumns` parallel computation remains unchanged. Compared to Listing 1, the following changes are done to enable shuffling:

- The loop on columns (line 11) iterates following a randomly drawn index permutation.
- Input state indexes (`i`, `i+1` etc, lines 14 to 17) are shuffled using a randomly drawn permutation. A new permutation is drawn at each loop iteration.
- The packed column representation (`column`, line 11) is computed by shifting each byte according to its line index: a byte corresponding to a line index j is shifted left by $8 * j$.

4.3 Code polymorphism

The code polymorphism countermeasure is implemented as proposed in [9]. This approach is presented in Section 2.2.

We propose to adapt the countermeasure in order to better hide loop iterations. Indeed, dynamic noise set apart, the code executed remains the same during consecutive loop iterations. Such constant behaviour could be used to construct attacks that work across loop iterations, as all samples from different iterations would be vertically aligned. Such attacks on loops have been demonstrated to recover masks during masked table precomputation [26], and could maybe translate to column permutation index recovery.

In order to have different behaviour during loop iterations, random branches are needed. Dynamic noise can fulfil this goal. While it was proposed to decorrelate what happens during generation and execution and to lower code generation frequency, the random branches it creates do decorrelate loop iterations. Dynamic noise works thanks to a reserved register that holds a random data at runtime, which is updated by some noise instructions, as explained in Section 2.2.

We extend the use of random branches to semantic variants, and propose the use of dynamic variants in combination of dynamic noise. As explained in Section 2.2, the use of semantic variants consists in randomly choosing during code generation a variant among several code sequences that all are semantically equivalent. When using dynamic variants, the code generator no longer chooses a variant, but instead generates a switch-case containing all variants. Such approach is inspired by the MEET approach, that leverages such mechanism to implement code polymorphism without code generation [4].

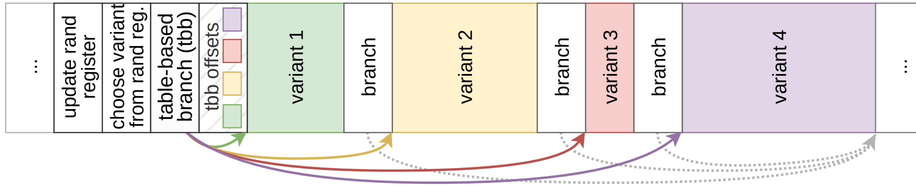


Fig. 2: Code structure of a dynamic variant. First, the random register is value is updated. Then, the variant to execute is randomly chosen. Then, the `tbb` instruction branches to the selected variant. Finally, after the variant is executed, the execution continues with the code right after the dynamic variant sequence.

Both dynamic noise and dynamic variants are implemented using the table-based branch ARM-Thumb2 instruction (`tbb`). This instruction consists in a PC-relative branch whose offset is stored in a table. Having different offsets in the table and accessing the table at random indexes results in an efficient random switch-case implementation. Concerning dynamic noise, the use of this instruction lowers the overhead compared to [9].

Figure 2 shows how a dynamic variant is structured. The dynamic variant is implemented by having a preamble responsible from selecting randomly a variant, then a `tbb` instruction (”table-based branch” in Figure 2) that jumps to different offsets (”tbb offsets” in Figure 2) depending on the random variable value, then the supported variants (”variant 1-4” in Figure 2). Branch instructions are inserted between variants to continue execution after one variant is executed.

As in dynamic noise, the switch-case used for dynamic variants is controlled by the random register, allowing the random choice of a variant during execution. The random register value may not change sufficiently along the execution depending on the noise instructions inserted. In particular, if several dynamic variants are inserted consecutively, the probability of absence of noise instruction updating the random register in between can be high. To solve this issue, we propose to update the register value before any dynamic variant (”update rand register” in Figure 2). We use the following formula: `r=r+ROR(r,7)`. While it is not a pseudo random number generator (PRNG) with good statistical properties, one instruction is sufficient to implement it in ARM-Thumb2 assembly, hence its choice. The only fixed point is 0, and the rotate right allows making most-significant bits impact the least-significant bits of the next random numbers. This choice is motivated by use of the least-significant bits in dynamic variants.

Dynamic variants require developing several instruction sequences semantically equivalent, which can be difficult for some instructions. As such, not all instructions are converted to a dynamic variant. In order to assess the coverage of our supported variants, we measure how many instructions of the AES implementation presented in Section 4.1 are supported: for the reference imple-

mentation, 75 out of 117 instructions are supported, while for the implementation with loop shuffling, 106 out of 202 instructions are supported.

5 Experimental results

This section aims at evaluating how the combination of countermeasures behaves compared to implementations featuring only one countermeasure.

5.1 Experimental setup

We use a STM32 Nucleo-144 development board featuring a STM32F756ZG with an ARM Cortex-M7 core. We capture electromagnetic emission using a Langer RF-B 3-2 electromagnetic probe, a Langer preamplifier PA 303 and a picoscope 5244B. The sampling resolution is set to 8bits, and the sampling frequency is set to 500Msamples/s. The STM32F7 CPU frequency is set to 166.666MHz to capture exactly 3 samples per clock cycle to ease traces analysis. A trigger is placed right before the call to the AES polymorphic instance to ease synchronisation (right before step 4 of Figure 1).

The electromagnetic probe is manually placed at a position where instructions boundaries and shapes are clearly visible on the oscilloscope measurements. This setup is validated by performing a non-specific t-test on the unprotected AES implementation, with 50k traces in each class. We obtain a maximal tvalue of 223.90, indicating strong leakage, hence a good probe placement.

Unprotected AES C code is compiled with `Odo` with the following options: `-O2 -target thumb-none-eabi -mcpu=cortex-m7`. C code is compiled using `arm-none-eabi-gcc`, with the following options: `-O2 -Wno-unused-function -mcpu=cortex-m7 -mthumb -static`. Memory usage is measured using `arm-none-eabi-size` on a minimally working elf file.

5.2 Performance, table size and code size

Our evaluation starts by validating the interest of our custom AES implementation compared to an 8-bits and a T-table implementation. Then, the impact of the countermeasures is studied.

Comparison of our AES implementation w.r.t. 8bits and T-table. We compare the execution time, table size, and code size for unprotected AES implementations. We use: (1) an 8-bits by-the-book implementation, (2) a T-table implementation from Mbed TLS library [1], (3) our implementation (Section 4.1). All implementations have constant tables in RAM to improve performance, and have the key schedule precomputed. Table 1 presents the results, with execution time averaged over 1000 executions. Our implementation is 1.88× slower than T-table implementation, while being 3.64× faster than 8-bits implementation. Meanwhile, its tables' size is only 16 bytes larger than 8-bits implementation,

Table 1: Comparison of execution time, table size, and code size for 3 unprotected AES implementations. We measure `.text` section to get code size.

Implementations (unprotected)	Execution time in clock cycles	Tables size in bytes SBox/T-tables	State(s)	Code size in bytes
8-bits (by the book)	13834	256	16	564
T-table (Mbed TLS)	2017	4096	32	960
Ours (Section 4.1)	3800	256	32	536

Table 2: Execution time and size of our AES implementation presented in Section 4.1 with each countermeasure configuration considered.

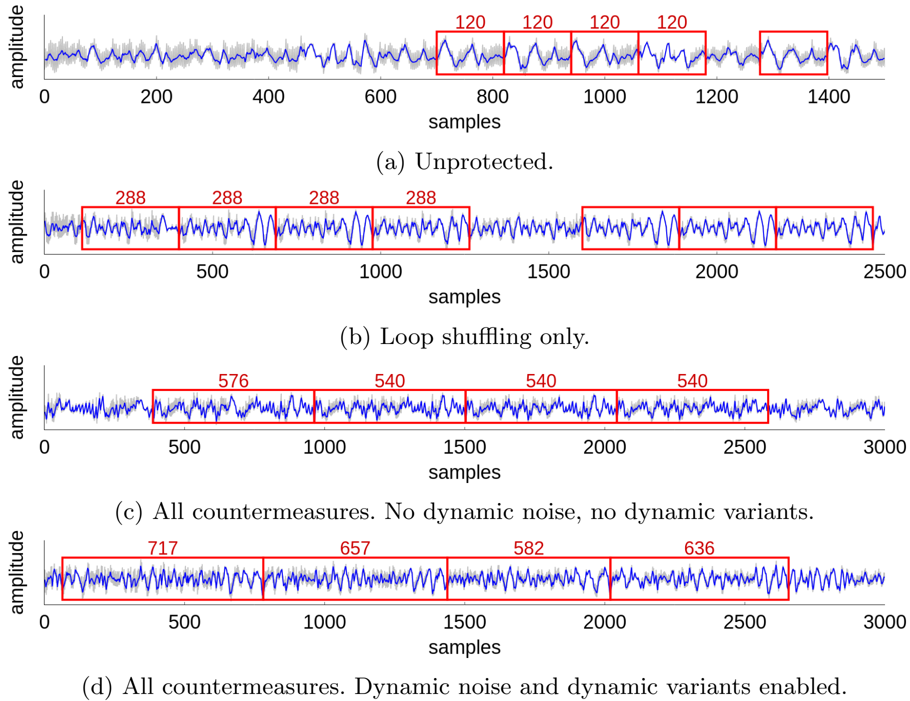
Configuration	Execution time		Size in bytes				
	Clock cycles	Overhead	`.text`	`.data`	`.bss`	Total	Overhead
Unprotected	2710	baseline	536	512	788	1836	baseline
Loop shuffling	3925	×1.45	884	540	788	2212	×1.20
Code polymorphism	9128	×3.37	23008	1364	7748	32120	×17.49
All countermeasures	11620	×4.29	27120	1392	9620	38132	×20.77

but is 3840 bytes smaller than T-table implementation. In addition, our implementation has the lowest code size. All in all, our implementation represents a performance vs. size trade-off well positioned w.r.t other implementations considered.

Countermeasures overhead. We pursue our experiments by measuring the impact of countermeasures on execution time and code size. Table 2 presents the measurements obtained. Execution time is averaged over 1000 executions. Both countermeasures have a significant overhead, reaching a 4.28× increase of execution time and 20.77× increase of size when combined. The large size increase is mainly due to code polymorphism that embeds the code generator and related functions in `.text` section and allocates a large buffer for the polymorphic instances in `.bss`. We note that even with such a large increase, the flash and RAM size in absolute remain low enough for many embedded systems, even for the implementation with combined countermeasures.

5.3 Security evaluation on STM32F7 traces

While each countermeasure's effect on security has already been studied, the effect of their combination is unknown. We ask whether the security obtained with combined countermeasures is higher than the security obtained with either countermeasure taken alone. As code polymorphism is slightly modified compared to previous work, we start by highlighting the effect of this modification. Then, we study how code polymorphism hides leakage of the loop shuffling permutation index. Finally, we assess how countermeasures resist to CPA with integration, and to the deep-learning attack from [21], that previously showed great success against code polymorphism.



(a) Unprotected.

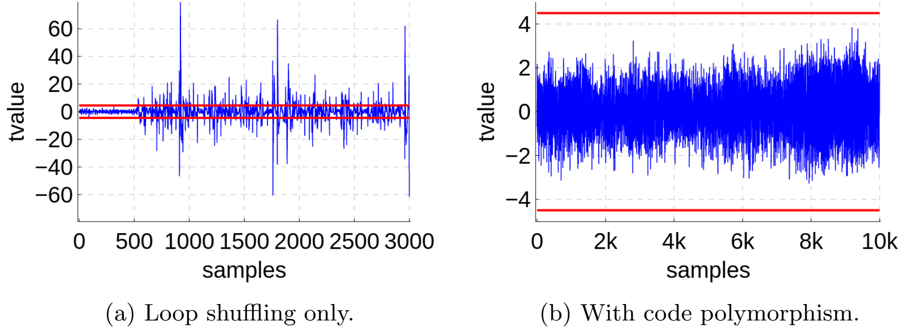
(b) Loop shuffling only.

(c) All countermeasures. No dynamic noise, no dynamic variants.

(d) All countermeasures. Dynamic noise and dynamic variants enabled.

Fig. 3: Electromagnetic traces obtained for different AES. The figure shows in grey in the background an average trace obtained by averaging 1000 executions with identical PRNG seed. The figure shows in blue the average trace post-processed using a moving average with a window of 3 samples. Repeating patterns are framed in rectangles, and their length in samples is indicated in red.

Impact of dynamic transformations on loop iterations. First, the interest of dynamic transformations put forward in this paper needs to be checked. As their goal is to induce variations in loop iterations, we inspect loop patterns in side-channel traces. For this purpose, a favourable setting is used: traces are averaged over 1000 executions. Randomness seeds are reset to ensure identical PRNG outputs for all the 1000 executions, and thus the identical code paths. In addition, traces undergo a moving average of 3 samples to further aid visualisation of repeating patterns. Figure 3 shows cropped parts of the resulting traces, for the different AES configuration we consider. Without dynamic transformation, we notice loop patterns whose size is constant, except for the first iteration with code polymorphism, which is a bit longer than the others. This may be due to cache misses. As expected, the implementation with code polymorphism with dynamic transformations enabled exhibits iterations with consistently varying size. Variance of iteration size could be further increased by choosing longer variants or longer dynamic noise sequences.



(a) Loop shuffling only. (b) With code polymorphism.

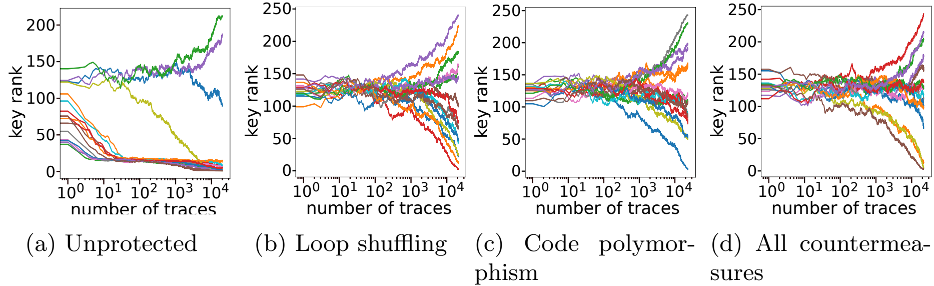
Fig. 4: Non specific t-tests (50k vs. 50k traces) targeting leakage from the permutation variable used to shuffle loops, with code polymorphism disabled (left) or enabled (right). Red lines indicate the 4.5 leakage detection threshold.

Impact of code polymorphism on loop shuffling security. Template attacks are frequently the attack of choice against loop shuffling, as explained in Section 2.3. Such attack starts with a point-of-interest extraction. We ask whether code polymorphism could harden such step, hiding the points of interest. To check this hypothesis, we perform a non-specific t-test targeting all permutation variables. For this purpose, we use a PRNG dedicated to the random permutation choice, whose seed is initialised in a fixed-vs.-random way. We collect 50k traces for both classes. Figure 4 shows the results of the t-tests, with and without code polymorphism. Without code polymorphism, strong leakage is visible: the t-value is above the 4.5 threshold for numerous samples, and reaches values as high as 79.30. With code polymorphism, the t-value is well contained in the $[-4.5, 4.5]$ range. As a result, no point of interest stands out from the analysed traces. As such, template attacks against the permutation index seem impractical.

Resistance of considered implementations against a CPA with integration. We first attack all implementations using a CPA with integration [25]. Integration is beneficial even for the unprotected implementation due to inherent jitter caused by our target platform. We determined an integration window of 24 samples was appropriate for this purpose. Changing the size of the integration window did not lead to any improvement, even in presence of code polymorphism. Hence, we kept the window of 24 for all implementations. The CPA results are presented in Table 3. The CPA easily finds all key bytes to the unprotected implementation and the implementation with shuffling only: $6,000$ traces and $50,000$ traces respectively are enough for the attack to succeed. However, it fails within $500,000$ traces for both implementations that feature code polymorphism.

Table 3: Results of CPA with integration on all considered implementations. Table reports number of traces to disclosure, for each of the target key byte. Table reports ✗ when the attack fails with 500k traces.

Configuration	Target key byte															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Unprotected	3k	6k	1k	2k	3k	2k	4k	1k	2k	3k	2k	4k	4k	3k	4k	2k
Loop shuffling	28k	32k	35k	18k	38k	28k	42k	45k	20k	18k	35k	20k	35k	40k	50k	45k
Code polymorphism	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
All countermeasures	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗



(a) Unprotected (b) Loop shuffling (c) Code polymorphism (d) All countermeasures

Fig. 5: Average rank of the right key vs. number of traces for all 16 key bytes using a deep learning attack against all considered implementations.

Resistance of considered implementations against a deep learning attack. As deep learning attacks are state-of-the-art attacks against code polymorphism [21], it is a natural candidate for a potential adversary against the enhanced version of code polymorphism combined with loop shuffling considered in this paper. As the target platform used in this paper slightly differs from the one used in [21], the neural network parameters require some modifications. Indeed, our clock frequency is much faster than the one of the STM32F3 used in [21], resulting in a lower number of samples per clock cycle (from 50 in [21] to 3 here). Accordingly, we remove the first layer that was specifically designed to handle the large number of samples per clock cycle. Apart from that, we keep the same hyper-parameters as in [21]: the filter size and the pooling size are respectively $w = 11, p = 5$, the number of convolution filters per layer is $k_0 = 10$ at the first layer, and doubles at every layer. Given the size of the traces, the number of convolutional layers is set to 3. The network is trained with the Adam optimiser [16] during 100 epochs on the Pytorch framework [24], with batch size equal to 32 traces. Given that a data-set of $100,000$ traces has been measured, we split it into $80,000$ training traces, and $20,000$ validation traces. Hence, there are $80,000/32 = 2500$ iterations per epoch. Once the training is finished, we run a key recovery attack by plugging the model's predictions into a maximum likelihood distinguisher. More precisely, we compute the cumulative sum of the log probabilities returned by the trained model for each trace from the valida-

tion set. Based on the cumulative scores, we may compute the rank of the right key. We repeat the procedure 50 times, by shuffling the order of the validation traces. This re-sampling (a.k.a. bootstrapping) method allows avoiding sampling 50 new validation datasets of $20,000$ traces each, although at the cost of a biased measure of the guessing entropy when the considered number of attack traces becomes close to the size of the validation set.

Figure 5 presents the attack results for all considered implementations. First, we notice that for 12 out of the 16 target bytes, the attacks succeeds in lowering the average rank of the right key significantly enough for a key enumeration, within 40 traces for the baseline (unprotected) implementation. Considering other implementations, none of them depicts a convincing key recovery. The curve diverging towards different key ranks on the right side of Figures 5b, 5c and 5d may be due to the bias induced by the re-sampling method. This suggests anyway that $20,000$ traces were not sufficient to observe the guessing entropy converging towards 0 with confidence. It is worth noticing the failed attack against loop shuffling alone, especially as we could succeed a CPA with integration to find all key bytes. We note some much more complex trained models have been shown efficient against shuffling [22]. Nevertheless, the latter models were not specifically designed against code polymorphism, hence we kept a neural network architecture close to the one of [21] as a baseline. Studying much heavier architectures that have been shown efficient against shuffling on polymorphic implementations remains an open problem, and is left for further works.

Likewise, code polymorphism seems to exhibit a better resistance than in [21]. This may be due to the difference of platform and implementation, including the storage of the SBox in RAM to avoid loads from flash. Finally, the combination of both countermeasures shows similar trend. Overall, these experiments are not sufficient to prove the security of the enhanced code polymorphism combined with loop shuffling with overwhelmingly high confidence. Yet, they provide significant empirical evidences that the baseline attacks from [21] may not straightforwardly apply, and that a successful adversary, should it exist, would be much more involved. In order to challenge this conjecture, we release our dataset along with the paper: `https://zenodo.org/records/10650737`.

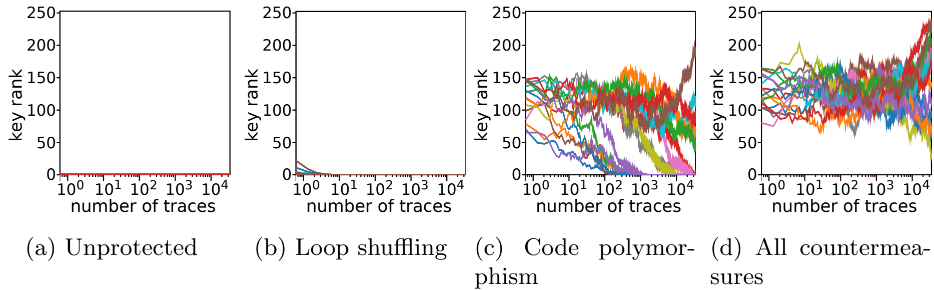
5.4 Security evaluation on simulated noiseless traces

As our security evaluation on real-world traces is insufficient to compare security of combined countermeasures w.r.t. code polymorphism alone, we pursue our security evaluation by simulating a perfect noiseless setting to ease attacks.

Traces simulation. Our simulation framework builds upon QEMU simulator [8]. We instruct QEMU to execute the program by steps of 1 assembly instruction. For each assembly instruction executed, our simulation framework outputs 2 samples: one that sums the Hamming Weights of the input operands before the instruction, and one that contains the Hamming Weight of the output operand after the instruction. No noise is added to the samples. We use this

Table 4: Results of CPA on simulated noiseless traces for all considered implementations. Table reports number of traces to disclosure, for each of the target key byte. Traces are simulated using the Hamming Weight model on registers used by each executed assembly instruction.

Config.	Target key byte															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Unprotect.	8	9	9	8	8	8	8	11	8	8	8	10	12	7	8	8
Loop shuf.	1165	921	707	713	1030	1186	1268	615	1104	1060	838	1047	891	604	234	1280
Code poly.	41k	41k	47k	61k	29k	25k	35k	55k	39k	42k	25k	85k	23k	24k	25k	50k
All	595k	215k	752k	369k	>1M	767k	998k	744k	>1M	592k	703k	747k	848k	588k	763k	735k



(a) Unprotected (b) Loop shuffling (c) Code polymorphism (d) All countermeasures

Fig. 6: Average rank of the right key vs. number of traces for all 16 key bytes using a deep learning attack against all considered implementations, on simulated traces.

framework to simulate traces for all the considered countermeasure configurations.

Resistance of considered implementations against a CPA on simulated noiseless traces. We perform a CPA [12] on the simulated traces. We note such setting is particularly favourable, as the employed Hamming weight model for simulation is the same as the one employed for the leakage estimation of the CPA. The results are presented in Table 4. As expected, we notice that the attack is much faster than in the real-world setting. The attack succeeds on all configurations. The added value of combined countermeasure is clearly visible, as the CPA requires in average 17× more traces for the implementation with combined countermeasures compared to the implementation with code polymorphism alone.

Resistance of considered implementations against a deep learning attack on simulated noiseless traces. In order to consolidate the CPA results on simulation, we also re-trained our deep learning models on the noiseless simulated traces. The training procedure remains the same, in particular we used the same number of training and attack traces as in Figure 5. The results are

given in Figure 6, which depicts the average rank of the right key byte with respect to the number of attack traces. We can see on the one hand in particular on Figures 6a and 6b that the deep-learning attack succeeds in recovering the right key within a few traces. On the other hand, on Figure 6c the key-rank curves are much slower to converge towards 0 (in a few hundreds of traces), and converge only for a few bytes, meaning that the attack did not always succeed. More interestingly, Figure 6d shows that no key recovery has been successful within $20,000$ attack traces after a profiling step. This empirically confirms the relevance of combining both shuffling and code polymorphism.

6 Conclusion

In this paper, we investigated the combination of two hiding countermeasures, namely loop shuffling and code polymorphism, to protect an AES implementation. Considering loop shuffling, we extended the countermeasure by shuffling as well the memory accesses within the loop, and we selected a random permutation among all possible ones using simple computation tricks to extend the number of possibilities w.r.t. classical random start index and random walk approaches. Considering code polymorphism, we stressed the interest of dynamic transformations in presence of loops, and proposed the use of dynamic variants, which is complementary to already existing dynamic noise. We considered both real-world traces and simulated noiseless traces for the experiments. Our security experimental evaluation showed that: (1) the use of dynamic transformation fulfils its role of making loop iteration length variable, but the use of longer variants may be needed to induce larger differences, (2) the loop shuffling permutation leakage are well mitigated by code polymorphism, (3) with sufficient noise or sufficiently low signal, code polymorphism resistance against deep learning attack seems greatly increased, (4) the combination of both countermeasures clearly increases security against CPA and against the considered deep learning attack, as shown in the noiseless simulated setting.

Such combination is thus worthy, especially in contexts where loop shuffling's main flaw is due to the leakage of its permutation variables. In particular, an interesting future work could be the combination with masking countermeasure, along several axes: (1) exploiting shuffling opportunities offered by masking as in [7], (2) better secure masked table computation, as shuffling alone was shown insufficient in this regard [26].

Acknowledgement

This work was partially funded as part of the TANGO project, which has received funding from the European Union's Horizon Europe Research and Innovation Programme under the grant agreement No. 101070052. The authors would like to thank Damien Couroussé for his helpful comments, and Simon Baissat-Chavent for the development of the trace simulator used in this paper.

References

1. Mbed TLS library. https://www.trustedfirmware.org/projects/mbed-tls/
2. Abdellatif, K.M., Couroussé, D., Potin, O., Jaillon, P.: Filtering-based CPA: A successful side-channel attack against desynchronization countermeasures. In: Proceedings of the Fourth Workshop on Cryptography and Security in Computing Systems. p. 29–32. CS2 '17, Association for Computing Machinery (2017)
3. Agosta, G., Barenghi, A., Pelosi, G.: A code morphing methodology to automate power analysis countermeasures. In: DAC. pp. 77–82 (2012)
4. Agosta, G., Barenghi, A., Pelosi, G., Scandale, M.: The MEET approach: Securing cryptographic embedded software against side channel attacks. IEEE TCAD **34**(8), 1320–1333 (2015)
5. Amarilli, A., Müller, S., Naccache, D., Page, D., Rauzy, P., Tunstall, M.: Can code polymorphism limit information leakage? In: Ardagna, C.A., Zhou, J. (eds.) Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication. pp. 1–21. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
6. Antognazza, F., Barenghi, A., Pelosi, G.: Metis: An integrated morphing engine cpu to protect against side channel attacks. IEEE Access **9**, 69210–69225 (2021). https://doi.org/10.1109/ACCESS.2021.3077977
7. Azouaoui, M., Bronchain, O., Grosso, V., Papagiannopoulos, K., Standaert, F.X.: Bitslice masking and improved shuffling:: How and when to mix them in software? IACR Transactions on Cryptographic Hardware and Embedded Systems **2022**(2), 140–165 (Feb 2022)
8. Bellard, F.: Qemu, a fast and portable dynamic translator. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference. p. 41. ATEC '05, USENIX Association, USA (2005)
9. Belleville, N., Couroussé, D., Heydemann, K., Charles, H.P.: Automated software protection for the masses against side-channel attacks. In: TACO. ACM (2018)
10. Benadjila, R., Khati, L., Prouff, E., Thillard, A.: ANSSI-FR/secaesstm32: Bibliothèque C et assembleur permettant le chiffrement/déchiffrement AES-128 de messages pour des composants grand public (famille STM32F3/STM32F4), https://github.com/ANSSI-FR/SecAESSTM32
11. Benadjila, R., Lomné, V., Prouff, E., Roche, T.: ANSSI-FR/secaes-atmega8515: Secure aes128 for atmega8515, https://github.com/ANSSI-FR/secAES-ATmega8515/blob/master/src/Version2/maskedAES128enc.S
12. Brier, E., Clavier, C., Olivier, F.: Correlation power analysis with a leakage model. In: Joye, M., Quisquater, J.J. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2004. pp. 16–29. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
13. Bronchain, O., Standaert, F.X.: Side-channel countermeasures' dissection and the limits of closed source security evaluations. IACR Transactions on Cryptographic Hardware and Embedded Systems **2020**(2), 1–25 (Mar 2020)
14. Couroussé, D., Barry, T., Robisson, B., Jaillon, P., Potin, O., Lanet, J.L.: Runtime code polymorphism as a protection against side channel attacks. In: Foresti, S., Lopez, J. (eds.) Information Security Theory and Practice. pp. 136–152. Springer International Publishing, Cham (2016)
15. Fumaroli, G., Martinelli, A., Prouff, E., Rivain, M.: Affine masking against higher-order side channel analysis. In: Selected Areas in Cryptography. Springer Berlin Heidelberg (2011)

16. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. In: 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings (2015), http://arxiv.org/abs/1412.6980
17. Luo, P., Zhang, L., Fei, Y., Ding, A.A.: Towards secure cryptographic software implementation against side-channel power analysis attacks. In: 2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP). pp. 144–148 (2015). https://doi.org/10.1109/ASAP.2015.7245722
18. Malagón, P., De Goyeneche, J.M., Zapater, M., Moya, J.M., Banković, Z.: Compiler optimizations as a countermeasure against side-channel analysis in msp430-based devices. Sensors **12**(6), 7994–8012 (2012). https://doi.org/10.3390/s120607994, https://www.mdpi.com/1424-8220/12/6/7994
19. Mangard, S., Oswald, E., Popp, T.: Power analysis attacks - revealing the secrets of smart cards. Springer (2007)
20. Masure, L.: Towards a better comprehension of deep learning for side-channel analysis. (Vers une meilleure compréhension de l'apprentissage profond appliqué aux attaques par observations). Ph.D. thesis, Sorbonne University, Paris, France (2020), https://tel.archives-ouvertes.fr/tel-03651269
21. Masure, L., Belleville, N., Cagli, E., Cornélie, M.A., Couroussé, D., Dumas, C., Maingault, L.: Deep learning side-channel analysis on large-scale traces. In: ESORICS (2020)
22. Masure, L., Strullu, R.: Side-channel analysis against ANSSI's protected AES implementation on ARM: end-to-end attacks with multi-task learning. J. Cryptogr. Eng. **13**(2), 129–147 (2023). https://doi.org/10.1007/S13389-023-00311-7, https://doi.org/10.1007/s13389-023-00311-7
23. Naccache, D., Nguyên, P.Q., Tunstall, M., Whelan, C.: Experimenting with faults, lattices and the DSA. In: Public Key Cryptography - PKC 2005. Springer Berlin Heidelberg (2005)
24. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E.Z., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S.: PyTorch: An imperative style, high-performance deep learning library. In: Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada. pp. 8024–8035 (2019)
25. Rivain, M., Prouff, E., Doget, J.: Higher-order masking and shuffling for software implementations of block ciphers. In: Cryptographic Hardware and Embedded Systems - CHES 2009. Springer Berlin Heidelberg (2009)
26. Tunstall, M., Whitnall, C., Oswald, E.: Masking tables—an underestimated security risk. In: Fast Software Encryption. pp. 425–444. Springer Berlin Heidelberg (2014)
27. Udvarhelyi, B., Bronchain, O., Standaert, F.X.: Security analysis of deterministic re-keying with masking and shuffling: Application to ISAP. In: Constructive Side-Channel Analysis and Secure Design. Springer International Publishing (2021)
28. Veyrat-Charvillon, N., Medwed, M., Kerckhof, S., Standaert, F.X.: Shuffling against side-channel attacks: A comprehensive study with cautionary note. In: Advances in Cryptology – ASIACRYPT 2012. Springer Berlin Heidelberg (2012)