Provable Secure Software Masking in the Real-World*

Arthur Beckers, Lennert Wouters, Benedikt Gierlichs, Bart Preneel, and Ingrid Verbauwhede

imec-COSIC, KU Leuven, Kasteelpark Arenberg 10, B-3001 Heverlee, Belgium
`firstname.lastname@esat.kuleuven.be`

Abstract. We evaluate eight implementations of provable secure side-channel masking schemes that were published in top-tier academic venues such as Eurocrypt, Asiacrypt, CHES and SAC. Specifically, we evaluate the side-channel attack resistance of eight open-source and first-order side-channel protected AES-128 software implementations on the Cortex-M4 platform. Using a T-test based leakage assessment we demonstrate that all implementations produce first-order leakage with as little as 10,000 traces. Additionally, we demonstrate that all except for two Inner Product Masking based implementations are vulnerable to a straightforward correlation power analysis attack. We provide an assembly level analysis showing potential sources of leakage for two implementations. Some of the studied implementations were provided for benchmarking purposes. We demonstrate several flaws in the benchmarking procedures and question the usefulness of the reported performance numbers in the face of the implementations' poor side-channel resistance. This work serves as a reminder that practical evaluations cannot be omitted in the context of side-channel analysis.

Keywords: Side-Channel Analysis, Leakage Assessment, Masking in Software

1 Introduction

Cryptographic primitives are designed to thwart cryptanalytic attacks such as differential and linear cryptanalysis. Even though these cryptographic primitives are deemed theoretically and cryptanalytically secure, their real-world implementations can still be vulnerable to attack. Side-channel attacks are one example of such implementation attacks. The field of Side-Channel Analysis (SCA) studies how unintentional side-channel leakage, produced by a cryptographic primitive implemented on a specific platform, can be used to extract secret information (e.g. the cryptographic key). To mount such a side-channel attack one typically executes the cryptographic operations several times while acquiring side-channel

* This preprint has not undergone peer review (when applicable) or any post-submission improvements or corrections. The Version of Record of this contribution is published in COSADE 2022, and is available online at https://doi.org/[insert DOI]

information. Side-channel information can come in many shapes and forms and can, for example, be acquired by passively monitoring execution time, power consumption and electromagnetic (EM) emanations.

SCA research was instigated by Kocher et al. through their seminal work on Differential Power Analys (DPA) in 1999 [19]. Here the attacker exploits the dependency between the secret data being processed on the device and its power consumption. To mitigate these SCA attacks an implementer generally tries to break the relation between the power consumption and the secret data being handled by the device. A common technique to achieve this is the use of masking [6, 14]. In a masked implementation the relation is broken by splitting up the sensitive intermediates in multiple random shares. Each of these shares is constructed such that on their own they are uncorrelated to the sensitive data. Depending on the masking scheme the implementation is given a security order. A masked implementation is said to be d^{th}-order secure if the implementation can withstand an attack exploiting up to d shares. Since the introduction of DPA different flavors of masking schemes have been proposed to counter SCA attacks. Masking schemes require randomness and the introduction of the shares comes with a large computational overhead especially when going to higher orders. The goal of many published schemes is therefore to minimize the randomness requirement and the execution time without compromising on security. Another aspect is to prove masking schemes secure in more realistic models. Many of the proposed schemes however focus on improving the timing and randomness requirement while neglecting to evaluate the practical side-channel security of their implementation. However, it has been shown many times that it is not easy to effectively protect an implementation with masking [3, 9]. The estimated execution-time overheads lose their meaning if benchmarking is not performed rigorously. If the benchmarked implementation does not provide the claimed side-channel resistance it becomes impossible to judge the additional overhead involved in resolving the leakage. Said differently, there is no point in comparing the performance of two insecure implementations for which the additional overhead to secure them is unknown.

1.1 Contributions

In this paper we benchmark and evaluate the side-channel resistance of multiple software masked AES implementations published in a wide range of academic venues including Eurocrypt, Asiacrypt, CHES and SAC. The evaluated implementations are listed in Table 1. The evaluations and benchmarks are performed on the same ARM Cortex-M4 target platform. The implementations were evaluated for their side-channel security using test vector leakage assessment (TVLA) [13] and correlation power analysis (CPA) [5]. During our leakage assessment all of the evaluated implementations showed TVLA leakage and nearly all of them could be broken with a straightforward CPA attack in our security evaluation. Additionally, all schemes were benchmarked using multiple configurations of the platform's clock tree. Our analysis reveals several discrepancies between cycle counts measured by us and the cycle counts reported by the authors. To reduce the risk of benchmarking mistakes and guarantee the relevance of the proposed implementation we propose a set of recommendations which should be followed when publishing side-channel secure software implementations.

1.2 Related work

In the academic literature a multitude of masking schemes and implementations have been proposed. We collected AES implementations for multiple of these masking schemes. An overview of the schemes for which we found an implementation either online or by contacting the authors can be seen in Table 1. These implementations will be the center of this work. These schemes were selected purely on the basis of a software implementation being available which could be ported to our target platform. All these schemes implement side-channel countermeasures which are solely based on masking. Implementations containing other countermeasures such as random delays or shuffling of the intermediates (e.g. the side-channel protected ANSSI implementation [4]) were not considered in this work.

Table 1. An overview of the evaluated implementations. Note that implementations for [24, 8, 10] are provided as part of [10].

Paper title	Published venue	Reference
Provably Secure Higher-Order Masking of AES	CHES 2010	[24]
Higher order masking of look-up tables	Eurocrypt 2014	[8]
All the AES You Need on Cortex-M3 and M4	SAC 2016	[25]
Consolidating Inner Product Masking	Asiacrypt 2017	[2]
First-Order Masking with Only Two Random Bits	CCS-TIS 2019	[15]
Side-channel Masking with Pseudo-Random Generator	Eurocrypt 2020	[10]
Detecting faults in inner product masking scheme	JCEN 2020	[7]
Fixslicing AES-like Ciphers	TCHES 2021	[1]

Masking aims at removing the dependency between the intermediate values and the secret key. This is achieved by splitting up the intermediate values into random shares. The number of shares determines the security order of the scheme. Ideally if one has d+1 shares an attacker needs to exploit leakage of d+1 shares in order to mount a successful attack. In their work [3] Balasch et al. showed how, if one does not pay close attention when implementing a theoretically secure masking scheme, a reduction in the security order can occur. This is because the leakage models on which the masking schemes are based assume independent leakage of the intermediate values. However, in software implementations the independent leakage assumption is often broken by transition based leakage. This for instance occurs when values stored in registers are overwritten leading to a recombination of the shares. The occurrence of transition based leakage can often be attributed to overwriting a register in the register file, but there are also other micro-architectural leakage mechanisms present in microcontrollers as was shown by McCann et al. [20].

Balasch et al. propose to increase the security order to compensate for these micro-architectural transition based leakages [3]. Alternatively, the masking scheme can be carefully implemented taking all potential sources of leakage into account. Such side-channel leakage simulators require meticulously engineered leakage models specific for each target platform. McCann et al. introduced ELMO [20], a leakage simulator with a specifically engineered leakage model for the Cortex-M0. This work was later extended by Shelton et al., which proposed Rosita [26] a tool that patches the underlying assembly instructions based on a target specific leakage model to reduce the side-channel leakage. Tools such as Elmo [20], Rosita [26] and CoCo [12] demonstrate that relatively basic microcontrollers have multiple hidden leakage sources that can be difficult to discover and compensate for.

The implementations evaluated in this work cover a wide variety of underlying masking schemes most of which are proven to be secure under the d-probing model introduced by Ishai-Sahai-Wagner [17]. In [24] the authors propose a generic higher-order boolean masking scheme for AES. The proposed scheme allows to construct masked implementations with an arbitrary security order. The software implementation for this scheme was provided by Coron et al. and served as a baseline to compare to their proposed schemes [10]. In [10] the authors also use boolean masking based on the ISW scheme. However, their main goal is to try and reduce the number of true random bits required by the scheme by using a combination of true and pseudo random number generators. The repository implementing [24] and [10] also contains implementations for the masking schemes introduced in [8]. Here a generalized table based masking scheme is proposed which can be extended to any security order. All the previous implementations were implemented in C in a straightforward and byte-oriented manner.

Schwabe and Stoffelen provide a highly optimized bitsliced AES-128 implementation protected with first-order boolean masking using Trichina AND gates [25]. Their optimised assembly implementation targets ARM Cortex-M3 and Cortex-M4 based microcontrollers. In [15] Gross et al. implement a first order boolean masking scheme. Their design includes a novel masked AND gate which allows for the reuse of randomness, reducing the number of true random bits required to two. Gross et al. provide a highly optimized assembly implementation for the ARM Cortex-M4 platform. Adomnicai and Peyrin further reduce the cycle count of this implementation by optimising the linear operations using a fixed-slicing construction [1]. The fixed-slicing implementation is based on the implementation provided in [15] and uses the same S-Box.

In addition to boolean masking based implementations we also evaluate two Inner Product Masking (IPM) based implementations. Out of all publications listed in Table 1 only the IPM work by Balasch et al. includes a practical side-channel evaluation [2]. The second IPM based implementation combines IPM with a fault attack countermeasure [7]. In this work we will only evaluate the side-channel resistance of the implementation. The implementation of [7] is publicly available and is written in C.

All the implementations evaluated in this work (see Table 1) were used as provided by the authors, without performing any modifications to the code besides adding GPIO triggers and cycle counters. The analysis performed on the collected traces is straightforward and straightforward to replicate using open-source side-channel toolboxes such as the ChipWhisperer project [21] or eShard's SCAred [11].

Note that most implementations are provided by the authors of the underlying masking scheme. Therefore, we assume that the respective authors verified the adherence of their implementation to their proposed masking scheme. Benchmarking results based on a flawed implementation would be meaningless and a thorough security evaluation requires auditing the code. Additionally, we note that some authors provide a disclaimer stating that the practical side-channel security of the provided implementation was not evaluated and that the implementation is provided for the purpose of benchmarking. However, throughout this work we will provide several examples of flawed benchmark results and argue that such results do not provide a realistic estimate for the additional overhead required to secure the implementation.

Finally, for one implementation the authors state in their paper: "We also provide [...] a masked implementation that is secure against first-order power analysis attacks" [25].

2 Side-Channel Analysis

This section covers SCA of open-source first order protected AES implementations. We start by detailing the used measurement setup followed by leakage assessment and CPA for each implementation.

2.1 Measurement setup

All of the studied implementations are compiled for, and executed on the same STM32F415 (Cortex-M4) microcontroller, with the Cortex-M4 being the target platform for most of the studied implementations. Each implementation is compiled using the same toolchain [1], compiler optimizations were mimicked from makefiles provided by the respective authors. Consequently, most C based implementations were compiled using `-O3`, except for the IPM based implementations that were compiled using `-O1`. These compiler optimizations can have an impact on the side-channel security of the implementations, but also impact the benchmarking results discussed in Sect. 3. It is thus important to evaluate both the side-channel security and the performance of an implementation using the

[1] `arm-none-eabi toolchain release 8-2019-q3`

same compiler flags. As later discussed in Sect. 3, the execution time of a side-channel protected implementation may be meaningless if the implementation leaks side-channel information.

Additionally, each implementation is evaluated using the same clock tree configuration using an 8 MHz external clock and 24 MHz core operating frequency.

For the acquisition setup we used a NewAE CW308 UFO board in combination with a STM32F415 target board, a Mini-Circuits ZFL-1000LN+ amplifier and a Tektronix DPO7254C oscilloscope. All measurements were acquired using a sample rate of 200 MS/s with the oscilloscope's internal 20 MHz lowpass filter enabled. The oscilloscope's vertical range was adjusted for each implementation to minimize quantization noise. Similar results are likely achievable using the low cost and open-source ChipWhisperer-Lite side-channel evaluation board [22].

2.2 Leakage assessment

We performed a fixed vs random TVLA for each of the evaluated implementations [13]. Each implementation was evaluated using the same fixed key, the same fixed plaintext was used for the fixed set during each evaluation. We used the fixed key and plaintext values suggested in [13].

We limit the scope of this evaluation to first order protected implementations as not all of the evaluated works include higher order implementations. Furthermore, Balasch et al. demonstrated that a straightforward implementation of an n-th order protected implementation protects against $n-1$ order attacks [3]. While higher order software implementations are less likely to leak in the first order they are likely to exhibit multivariate leakage at an order lower than intended by the design. Given that some of the higher order implementations require many millions of CPU cycles the evaluation would quickly become impractical.

Figure 1 shows the TVLA results using 10,000 measurements for each implementation. The TVLA based leakage assessment provides a high degree of confidence that these first-order masked implementations do in fact produce first-order leakage on the target platform (Cortex-M4) using the measurement setup documented in Sect. 2.1. At this point in the evaluation it is clear that the evaluated implementations do not live up to their claims. However, while TVLA based leakage assessment is a useful tool it does not allow us to compare the side-channel security of these implementations or to draw conclusions on how straightforward it would be to extract the secret key.

Note that the implementation provided as part of [2] (Consolidating Inner Product Masking) was deemed to be leakage free up to 1M traces. We consider determining the exact reason for this discrepancy out of scope as too many variables are unknown and beyond our control. Note that when compared to [2] we are using a different lab environment, physical side-channel, measurement setup and compiler version. Put differently, the only similarities between our leakage assessments are the used C source code and the use of a Cortex-M4 based microcontroller as evaluation target.

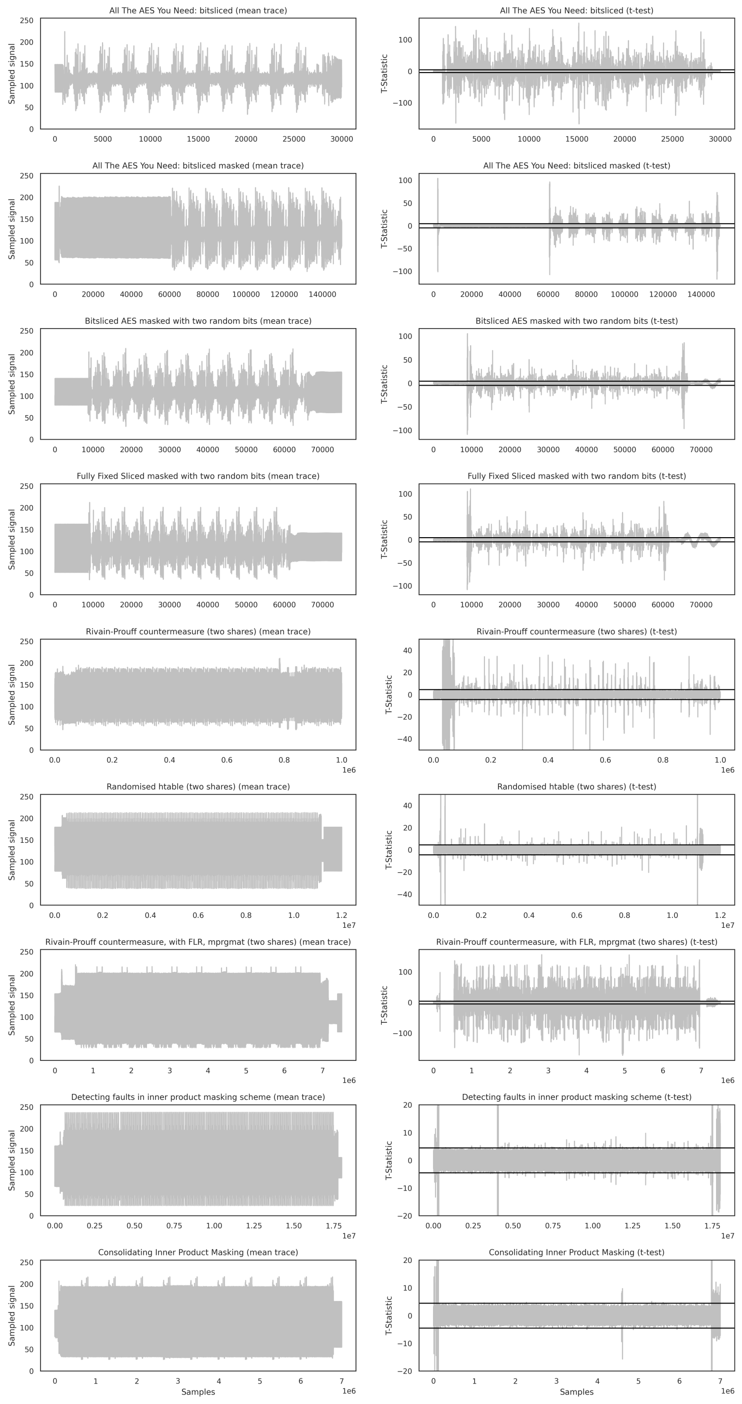

Fig. 1. TVLA results for the evaluated software implementations. The topmost plot contains the results for an unprotected bitsliced implementation. All other plots contain results for first order protected software implementations, yet exhibit clear first order leakage. This is evident from the T-statistic surpassing the ±4.5 boundary marked by black horizontal lines. Note that big peaks towards the start and end in the T-statistic trace likely correspond to input and output leakage respectively.

2.3 CPA attack results

The most straightforward attack on unprotected software AES implementations is a CPA attack in which the leakage model is assumed to be the Hamming weight (HW) of the first round S-Box output or last round S-box input.

A first-order protected implementation should not be susceptible to such a first-order attack. However, as demonstrated in Sect. 2.2, these first-order protected implementations leak first-order information. Therefore, it makes sense for us to try and mount a first-order attack using the classical S-box output as the target intermediate. This type of unprofiled attack would arguably be the first thing any attacker would try, making it the bare minimum side-channel attack to protect from.

To evaluate the studied implementations we mounted first-order CPA attacks targeting both the first round S-box output and the last round S-box input. For the byte-oriented implementations (those provided by [10], [7], [2]) we target the first key byte. Among the evaluated implementations there are also several bitsliced implementations for which the attack strategy needs to be slightly modified. The bitsliced implementations process two AES-block simultaneously. In the bitsliced representation each 32-bit state register contains one bit of each of the state bytes from each block. Unprotected bitsliced implementations can often be attacked by targeting a single state bit. Nevertheless, the nature of these implementations results in more algorithmic noise as a single bit (out of 32) is being targeted. The implementations provided in [25] and [15] assume the use of AES in counter mode, therefore we also use the implementation provided in [1] as a counter mode implementation. Because of the use of counter mode we are limited to attacking a single bit in the first round. However, when carrying out an attack on the last round we can target two bits (one bit from each block being processed).

Figure 2 provides the results for each CPA attack. Even though we were able to attack most implementations with relative ease it is important to note that this does not mean that the underlying masking scheme is flawed. Neither does it show that one masking scheme is more secure than another. It does demonstrate that a straightforward software implementation of a theoretically secure masking scheme is unlikely to live up to its expectations in the real world.

These results demonstrate that the correct key byte can be recovered for six out of eight masked implementations using a first-order attack. Notably only the implementations where inner product masking is applied cannot be attacked with a classical CPA attack. This is probably due to mechanism behind the accidental unmasking. In Sect. 2.4 we discuss one potential leakage source for some of the implementations.

CPA elaboration per implementation Figure 2 contains a few interesting and/or surprising results. Unsurprising is the fact that an unprotected, yet bitsliced implementation can be attacked with ease (topmost plot). This result is included as a reference. The protected bitsliced implementations require slightly

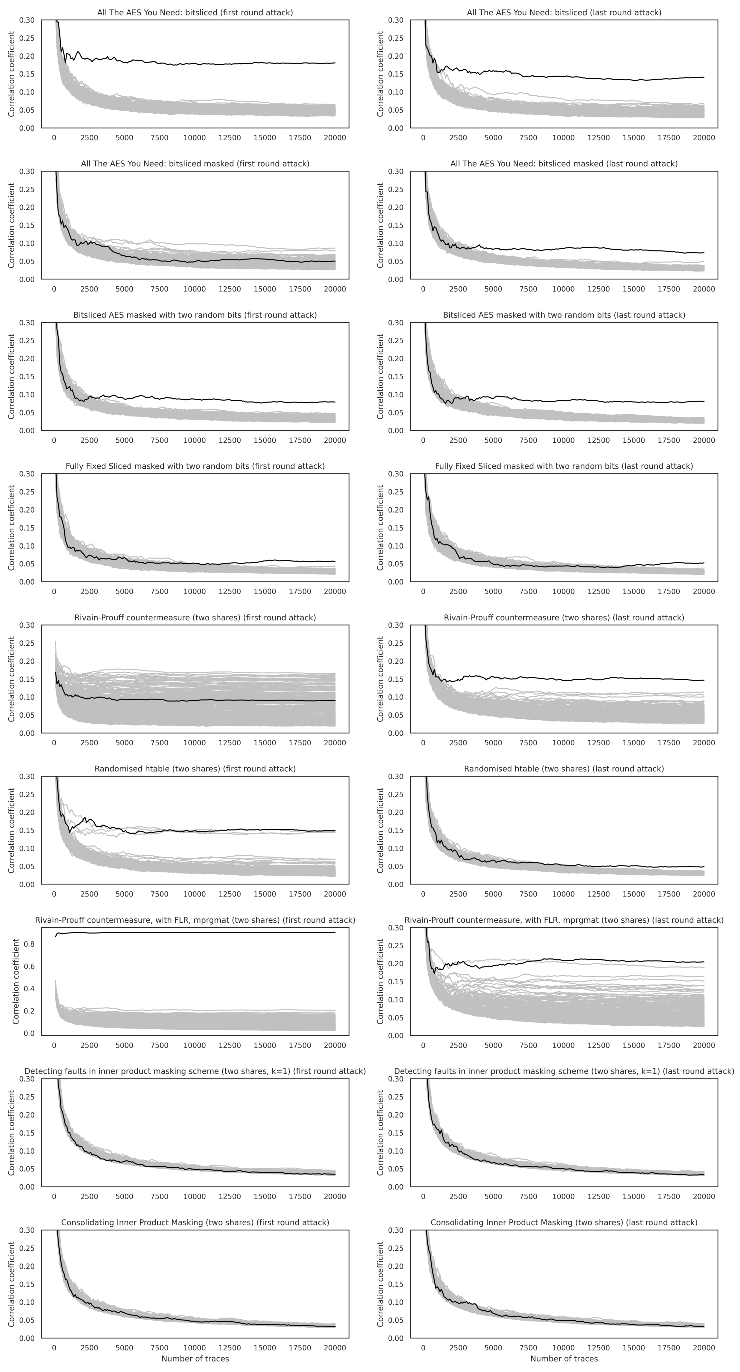

Fig. 2. CPA results for the evaluated software implementations. The plots show the evolution of the correlation coefficient (Y-axis) versus the number of traces (X-axis) used for the attack. The correlation coefficient for the correct key guess is shown in black.

more traces to result in a successful attack, but the exact same attack strategy applied for the unprotected implementation can be used.

Note that the attack on the bitsliced AES implementation from [25] (second row) was only successful (using at most 20,000 traces) while attacking the last round. This is most likely the result of transition based leakage occurring at the S-Box input as will be explained in Sect. 2.4. Additionally it is interesting to note that fixsliced implementations, which are based on the implementations masked with two random bits, require significantly more traces to recover the key. This may indicate that the original implementation provided in [15] contains an additional implementation mistake in the linear layers.

Interestingly, the SecMultFLR implementation provided by Coron et al. appears to not provide any side-channel protection in the first round (Fig. 2, 7th row). Figure 3 shows that the lack of side-channel protection can be partially attributed to the use of compiler optimizations.

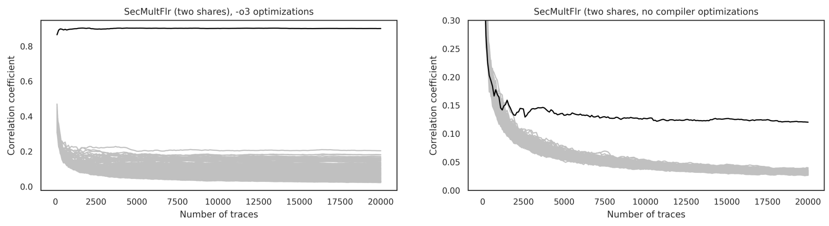

Fig. 3. CPA results when targeting the first round of the SecMultFlr implementation with (left) and without (right) compiler optimizations enabled.

Finally, neither of the IPM-based implementations appear to be vulnerable to the classical CPA attack up to 20,000 traces. This is likely because accidental recombinations of the shares, resulting in transition based leakage, do not directly reveal secret information in the IPM scheme. Note that both of these implementations are written in C and were compiled with optimizations. This shows that masking schemes implemented in software which do not suffer from security order reduction by transitional leakage are less prone to implementation mistakes and therefore an interesting field of study for future research.

2.4 Root cause analysis

From Fig. 2 it is clear that it is often easier to attack the last round S-box input compared to the first round S-box output. To investigate why this is the case we performed a manual root cause analysis for the boolean masked bitsliced AES implementation and the bitsliced AES implementation masked using two random bits.

In order to speed up our analysis we employed an emulator to pinpoint the different instructions which could potentially lead to a successful attack. We emulated the leakage of register updates with both the Hamming weight and Hamming distance leakage model. In practice there are more subtle sources of leakage present in these microcontrollers which are not covered by our emulation of the leakage through register updates. McCann et al. [20] for example showed that a microcontroller can leak through recombinations between pipeline stages of buffer registers in the arithmetic logic unit (ALU). Leakage free emulated traces therefore do not guarantee a leakage free implementation but they are a good starting point. In our case the simple model proved to be sufficient to find a cause of leakage in both the boolean masked bitsliced AES implementation and the bitsliced AES implementation masked using two random bits. The fixsliced AES uses exactly the same S-box implementation as the bitsliced AES implementation masked using two random bits and therefore has the same implementation flaw.

Unsurprisingly the Hamming weight model did not show any leaking instructions. Leakage in the Hamming weight model would only occur if an instruction operated directly on an unmasked piece of data or under a biased randomness distribution, indicating a severe flaw in the masking scheme. In the Hamming distance model however multiple unintentional unmaskings were discovered for both the boolean masked implementation and the bitsliced AES implementation masked using two random bits. These leakages are a common flaw in first order boolean masked software implementations and occur when a register containing one share is overwritten by the other share leading to an unintentional recombination of the shares.

Listing 1.1 shows the part of the S-box code containing the unintended unmasking for the boolean masked bitsliced implementation. The Listing was taken from the public Github repository of the implementation [28]. The unintentional unmasking happens at line 1502 in the aes_128_ctr_bs_masked.s file. The intermediate S-box value $y3$ stored in register r9 (line 1482) gets overwritten by the previously calculated $y3m$, which was stored on the stack. Since it is a two share implementation the recombination of two shares will result in an unmasking of the data. The leakage resulting from the register overwrite is written out in full in Equation (1)-(3) where y_{3p} is the unmasked intermediate S-box value.

$$HW\big[r9 \oplus [sp + 120]\big] \tag{1}$$
$$HW\big[y_{3m} \oplus y_3\big] \tag{2}$$
$$HW\big[y_{3p}\big] \tag{3}$$

Listing 1.2 shows the critical assembly instructions on line 1471 resulting in the observed leakage in the bitsliced AES implementation masked using two random bits. In this case the unmasking is more subtle in nature and one has to calculate back to the initial masking of the plaintext ($i_{0,P}$) and key ($k_{0,p}$) to demonstrate the accidental unmasking. The full backtracing of the unmasking is given by Equations (4)-(8).

```
eor   r11,  r7, r11    //Exec y8 = x0 ^ x5; into r11
eor   r9, r6, r11 //Exec y3 = y5 ∧ y8; into r9
eor   r2,  r7,  r2     //Exec y9 = x0 ^ x3; into r2
str   r11, [sp, #100 ] //Store r11/y8 on stack
str   r8, [sp, #96  ] //Store r8/y10 on stack
str.w r5, [sp, #92  ] //Store r5/y20 on stack
eor   r11,  r5,  r2    //Exec y11 = y20 ^ y9; into r11
eor   r8,  r8, r11    //Exec y17 = y10 ^ y11; into r8
eor   r0,  r0, r11    //Exec y16 = t0 ^ y11; into r0
str   r8, [sp, #88  ] //Store r8/y17 on stack
eor   r5,  r4, r11    //Exec y7 = x7 ^ y11; into r5
ldr   r8, [sp, #1496] //Exec t2 = rand() % 2; into r8
str   r9, [sp, #84 ] //Store r9/y3 on stack
eor   r10, r10,  r8    //Exec u1 = u0 ^ t2; into r10
eor   r1, r10,  r1    //Exec u3 = u1 ^ u2; into r1
eor   r3,  r1,  r3    //Exec u5 = u3 ^ u4; into r3
eor   r3,  r3, r14    //Exec t2m = u5 ^ u6; into r3
and   r1,  r9, r12    //Exec u0 = y3 & y6; into r1
ldr   r10, [sp, #112 ] //Load y6m into r10
str   r12, [sp, #80  ] //Store r12/y6 on stack
and   r14,  r9, r10    //Exec u2 = y3 & y6m; into r14
ldr   r9, [sp, #120 ] //Load y3m into r9
and   r12,  r9, r12    //Exec u4 = y3m & y6; into r12
```

Listing 1.1. Assembly snippet of All the AES You Need

```
    orr   r0, r12, r14   //Exec M1ORM2 = MASK1 | MASK2 into r0
    eor   r2,  r7,  r9   //Exec y14 = i4 ^ i2 into r2
    str.w r0, [sp, #112] //Store r0/M1ORM2 on stack
    eor   r0,  r4, r10   //Exec y13 = i7 ^ i1 into r0
    eor   r1,  r0, r14   //Exec hy13 = y13 ^ MASK2 into r1
    eor   r3,  r4,  r7   //Exec y9 = i7 ^ i4 into r3
    str.w r3, [sp, #108] //Store r3/y9 on stack
    eor   r3,  r3, r14   //Exec hy9 = y9 ^ MASK2 into r3
    str.w r1, [sp, #104] //Store r1/hy13 on stack
    eor   r1,  r4,  r9   //Exec y8 = i7 ^ i2 into r1
    eor   r6, r5, r6 //Exec t0 = i6 ∧ i5 into r6
    str.w r3, [sp, #100] //Store r3/hy9 on stack
    eor   r3, r6, r11 //Exec y1 = t0 ∧ i0 into r3
    str.w r6, [sp, #96 ] //Store r6/t0 on stack
    eor   r6, r3, r14 //Exec hy1 = y1 ∧ MASK2 into r6
    eor   r7,  r6,  r7   //Exec y4 = hy1 ^ i4 into r7
```

Listing 1.2. Assembly snippet of First-Order Masking with Only Two Random Bits

In both implementations the unintentional unmasking occurs towards the start of the S-Box computation. This observation explains why the CPA attack targeting the last round S-box input is more successful.

$$HW\big[(r6) \oplus (r3 \oplus r14)\big] \tag{4}$$
$$HW\big[(t_0) \oplus (t_0 \oplus i_{0,M} \oplus M2)\big] \tag{5}$$
$$HW\big[i_{0,M} \oplus M2\big] \tag{6}$$
$$HW\big[(i_{0,P} \oplus M2 \oplus k_{0,P} \oplus M1 \oplus M2 \oplus M1 \oplus M2) \oplus M2\big] \tag{7}$$
$$HW\big[i_{0,P} \oplus k_{0,P}\big] \tag{8}$$

3 Benchmarking

While some authors did provide a disclaimer stating that the side-channel security of their implementations was not practically verified they also state that the masked implementation is provided for benchmarking purposes. Nevertheless, during the side-channel evaluation of the different implementations we noticed several discrepancies between observed and reported cycle counts.

In this section we provide a comparison between measured and reported cycle counts for all implementations and analyse a few discrepancies. Additionally, we provide insight into how seemingly small configuration changes can have a big impact on the cycle count, a metric often optimised for in the academic literature.

All execution time measurements were taken using the same STM32F415 Cortex-M4 microcontroller that was used during the side-channel evaluation and the same toolchain. As the evaluated implementations aim to be side-channel resistant we disabled both the data and instruction cache. Leaving these enabled might result in unintentional leakage, especially at higher CPU frequencies. With caches disabled the number of flash wait cycles will have a significant impact on the execution time. The number of flash wait cycles indicates the latency between requesting data from flash and the data arriving in the registers of the microcontroller. We provide measurements at clock frequencies of 24 MHz and 168 MHz to demonstrate the effect of flash wait cycles on the reported cycle count. The used platform does not require any additional flash wait cycles at 24 MHz, but requires an additional 5 wait cycles at 168 MHz. Implementations suffering from more pipeline stalls will thus be penalized in their cycle counts when the clock frequency is increased.

The internal random number generator was always clocked at 48 MHz, its maximal operating frequency. We measured the cycle count for each implementation using the Data Watchpoint and Trace (DWT) unit. As in the side-channel evaluation, all software implementations were compiled using the flags provided by the respective authors. Specifically, this means that most C implementations are compiled using `-O3` with the exception of the IPM implementation provided by Balash et al. which is compiled using `-O1` [2].

To compare the benchmarking results we used the same implementation parameters (e.g. number of shares) as those that were used by the respective authors, in some cases these parameters differ from those used during side-channel evaluation. The resulting cycle counts can be found in Table 2. Note that these cycle counts correspond to the execution of a single call to the implemented primitive. For most of the implementations this corresponds to one block of AES-128. However, for the bitsliced implementations (i.e. [25], [1], [15]) these cycle counts correspond to the encryption of two AES-128 blocks. Additionally, note that this table reports the number of random words collected, this number does not necessarily correspond to the number of random bytes used in the implementation.

The authors of each implementation report execution times or cycle counts in their respective publications, but the used evaluation platforms vary. We omit reported cycle counts in Table 2 if the used platforms cannot be directly compared. For example Rivain and Prouff provided cycle counts on a 8051-based platform in their original work and report 271k cycles for their three share assembly implementation [24]. It is not clear how the randomness was generated in, or provided to, this implementation. Nevertheless, Coron et al. provide their own implementation of the scheme proposed in [24] as a baseline for comparison in [10] and report a cycle count of 20.6M cycles for the three share variant. The implementations provided in [10] were benchmarked using an emulated Cortex-M3 running at 44 MHz, it is not clear which emulator was used exactly. Results from emulator based benchmarks are included in Table 2 and marked with an asterisk (*). We used the implementation provided in [10] during our benchmarks as the target platform closely matches ours. In Sect. 3.1 we demonstrate the near 76 fold increase in reported cycle-count can be largely attributed to the unrealistically low throughput TRNG used during the emulation process.

Similarly, the difference in measured and reported cycle count for the masked and bitsliced implementation provided in [25] is too big to be attributed to a small configuration or platform difference. The difference is probably due to a misconfiguration of the random number generator (RNG) during the benchmarking of the implementation. This issue will be explained in more detail in Sect. 3.1.

Table 2. Measured and reported cycle counts for the evaluated implementations.

Implementation	cycles (measured/reported)	randomness (RNG/PRNG)	clock frequency
[25]	17.5k/14.8k	328/-	24 MHz
	62.8k/-	328/-	168 MHz
[15]	6.8k/6.8k	2/-	24 MHz
	9.5k/-	2/-	168 MHz
[1]	6.2k/6k	2/-	24 MHz
	8.9k/-	2/-	168 MHz
[24] (n=3)	651k/20.6M*	2880/-	24 MHz
	834k/-	2880/-	168 MHz
[8] (n=3, randomized table)	9.099M/-	164,160/-	24 MHz
	13.639M/-	164,160/-	168 MHz
[8] (n=3, randomized table word including common shares)	2.091M/-	34,032/-	24 MHz
	3.195M/-	34,032/-	168 MHz
[10] (n=3, multiple PRG, secmultFLR)	3.608M/12M*	52/5120	24 MHz
	4.576M/-	52/5120	168 MHz
[2]	819k/-	1632/-	24 MHz
	1.272M/-	1632/-	168 MHz
[7] (n=2, k=1)	1.650M/-	2432/-	24 MHz
	2.283M/-	2432/-	168 MHz

3.1 Randomness generation

Certain members of the STM32F4 family of microcontrollers have an internal TRNG. According to the public documentation the TRNG is based on multiple ring oscillators, the outputs of which are summed and used as the seed for a Linear Feedback Shift Register (LFSR) [27]. The seeded LFSR is clocked by a dedicated clock to generate 32-bit random words. The reference manual states that a new 32-bit random word is generated every 40 periods of the dedicated TRNG clock which operates at maximum 48 MHz. In addition to being independent of the main system clock, the TRNG will not operate correctly under certain clock tree configurations.

For the remainder of this discussion we assume that the TRNG is clocked at 48 MHz independent of the system clock. As the TRNG operates independently from the main clock the number of CPU clock cycles required to generate a random word can vary. When the operating frequency of the microcontroller is set to 24 MHz (i.e. half of the TRNG clock) it should take 20 MCU cycles to get a new random word. An increase in the CPU clock frequency will thus result in more CPU clock cycles to generate a random word.

This effect can be observed in Table 3, using a simple assembly loop we get a new random word every 21 CPU cycles, the same code will have to be executed more often at a higher operating frequency resulting in more CPU cycles to obtain a random word. Additionally, Table 3 provides an overview of the number of cycles required to generate a random word on different platforms running at multiple CPU clock frequencies.

Interestingly, Schwabe and Stoffelen report that their bitsliced and masked AES implementation takes 7422.6 cycles per block, of which 2132.5 cycles are used to collect the required randomness [25]. As the reported cycle counts are per block we can double them to obtain the number of cycles required for one call to their AES implementation which computes two blocks in parallel. Each call requires 328 words of randomness which, according to the reported numbers, requires 4265 cycles to collect. Looking back at Table 3 we would expect this process to take 6888 ($328 * 21$) cycles at the used 24 MHz CPU clock. Schwabe and Stoffelen made their git repository publicly available, allowing us to track down the issue that caused this cycle count discrepancy. Two recent commits (`910d446` and `56abc40`) slightly modified the clock tree configuration and the assembly code responsible for collecting randomness. Before those commits the TRNG could not operate as expected resulting in the the randomness collection loop simply reading the TRNG status register and data register 328 times without actually obtaining random data. This admittedly easy to make mistake resulted in an overestimation of the TRNG throughput and the use of all zero masks, effectively resulting in an unmasked implementation. Nevertheless, it is commendable that five years after the initial commit the authors are still maintaining their repository and fixing issues.

Coron et al. used an emulator to estimate the cycle count of their implementations on a 44 MHz ARM-Cortex M3 processor [10]. The exact processor or emulator is not mentioned, but the authors report that one 32-bit random word is generated every 6000 CPU cycles. We are not challenging the numbers reported by their emulator, but it is important to use realistic numbers when comparing the use of a TRNG to that of a software PRNG in terms of cycle counts. As mentioned earlier, the STM32F4 TRNG produces a new random word every 40 clock cycles at 48 MHz. Similarly, the Microchip SAM D5x microcontrollers produce a new random word every 84 clock cycles over a wide range of clock frequencies (up to 120 MHz) [16]. This would mean that the TRNG used in [10] is 150 times slower compared to the TRNG used in the STM32F4 microcontrollers or 71 times slower compared to the SAM D5x microcontrollers.

As can be seen from Table 3 the XorShift PRNG used by Coron et al. does produce more random bits per cycle and scales better when the clock frequency on the STM32F4 is increased. Nevertheless, the use of unrealistic TRNG performance estimates results in an overestimation in the performance gained from using a software PRNG. In the extreme case these unrealistic performance estimates resulted in implementations which are more efficient (in terms of cycle count) on paper, but in fact less efficient in practice. For example, in [10] the authors claim that their three share implementation with multiple PRGs (secmultFLR) requires roughly half the number of cycles compared to the their reference implementation of [24]. From Table 2 it is clear that in practice their provided implementation requires roughly seven times more cycles compared to the reference implementation.

Table 3. Randomness generation cycle counts for different platforms and CPU clock frequencies.

platform	function	word length	cycles	clock frequency
STM32F415 Cortex-M4	polling opencm3	32 bit	27	24 MHz
		32 bit	147	168 MHz
	polling assembly	32 bit	21	24 MHz
		32 bit	147	168 MHz
	PRNG XorShift 96	64 bit	39	24 MHz
		64 bit	63	168 MHz
LPC55S69 Cortex-M33	polling assembly	32 bit	104	25 MHz
		32 bit	361	150 MHz
SAM D5x Cortex-M4	according to datasheet	32 bit	84	24 MHz
		32 bit	84	140 MHz

3.2 Benchmarking: discussion and conclusion

Cycle counts or execution time are popular metrics to optimise for in software implementations. New masking schemes and implementations often serve the purpose of outperforming previous work in such metrics, regardless of the real-world side-channel security. Unfortunately, results reported in the academic literature are often not directly comparable to other works or in certain cases impossible to reproduce. The need for a detailed description of the benchmarking setup is also evidenced when compiling the same implementation using different compiler versions. For example, compiling the RP implementation provided in [10] with toolchain version 7-2018-q2 results in an implementation that requires roughly 50k cycles more. In general, the masking community would benefit from a unified benchmarking process. Successful deployments of such processes were demonstrated by Kannwischer et al. as part of the PQM4 project [18] and by Renner et al. as part of the NIST Lightweight cryptography competition [23].

Table 2 demonstrates the necessity of using realistic platforms, realistic microcontroller configurations and providing detailed descriptions of the benchmarking setup. This is also evident by comparing the measured cycle counts when the clock frequency of the used platform is increased.

The use of a hardware TRNG peripheral adds another dimension to the benchmarking process. The implementer can carefully interleave the collection of randomness with other useful instructions instead of polling the TRNG status register in a blocking manner, this comes at a cost of increased implementation complexity. Similarly, some implementations waste randomness by discarding three out of four bytes produced by the TRNG. Alternatively, if cycle-count is the metric to be optimised we can also simply reduce the CPU clock frequency. Specifically, for the STM32F415, we would be able to reduce the CPU clock frequency to 4 MHz while keeping the TRNG peripheral clocked at 48 MHz. As shown in Table 3, reducing the main CPU clock results in a trivial and meaningless reduction of the cycle count.

From this discussion it should be clear that benchmarking results depend on many factors, straightforward cycle-count comparisons without a detailed description can thus be considered meaningless. Furthermore, benchmark results for masked implementations that do not provide the claimed security level may not be meaningful, as securing the implementation will require additional unpredictable overhead.

4 Discussion and conclusions

In this work we benchmark and evaluate the side-channel security of multiple masked software AES implementations based on a variety of masking schemes. Only two of the evaluated implementations namely [7] and [2] seem to live up to their promises. All other implementations were not side-channel secure in the claimed security order, or reported skewed benchmarking results in their respective publications.

This comes to show that a thorough side-channel evaluation is required when implementing a masking scheme. The side-channel evaluation will reveal potential implementation mistakes like a wrongly configured TRNG or dramatic over estimations of the TRNG overhead. Additionally, it will highlight micro-architectural leakage mechanisms present in the evaluation platform which are not captured by most theoretical leakage models. Compensating for these unexpected leakage sources can introduce a significant overhead. Madura et al. [26] report an overhead of up to 60% when rewriting a straightforward implementation to be free of T-Test leakage on the Cortex-M0 platform.

Most of the analysed works did not contain strong claims regarding the security of their implementations, but do offer security proofs for the used masking scheme. In those cases the provided implementations are used for benchmarks and comparisons with related work. We question the relevance of benchmarking results which do not take into account the additional unpredictable overhead required to secure the implementation.

Our work was enabled through the availability of the evaluated implementations, we want to commend the respective authors. Similar works without published implementations undoubtedly suffer from the same issues. Unfortunately, the absence of the implementations makes it impossible for anyone to evaluate and improve them.

In general, published implementations of masking schemes would benefit from a more rigorous approach to benchmarking and side-channel evaluation. Therefore, we provide a set of guidelines to be used when publishing a new side-channel secure implementation.

4.1 Recommendations

In this section we lay out some recommendations that can be used as a checklist for side-channel evaluations and benchmarks of masked software implementations. While many of these recommendations may appear obvious to an experienced practitioner they seem to be rarely applied in academic literature, as evidenced by this work.

– **Describe the side-channel setup in detail:** oscilloscope settings, filters, measurement method, probe location etc. In other words, provide all of the required information for someone to reproduce your setup. Commercially available and easily reproducible measurement setups exist (e.g. ChipWhisperer and accompanying target boards).
– **Perform a convincing side-channel leakage assessment:** An assessment should consist of two parts. First, the soundness of the side-channel measurement setup should be demonstrated by for instance performing TVLA with the masking randomness (TRNG/PRNG) disabled or by checking the Signal to Noise Ratio (SNR) of a known intermediate variable's leakage. Secondly TVLA should be performed with the countermeasures enabled. Perform a second evaluation with different fixed inputs if no leakage was detected [13] and provide the used fixed inputs.
  Note that leakage assessment is not meant to replace provable security but rather to complement it.
– **List the randomness requirement of the masking scheme.** Carefully indicate the number of bytes used for masking and the total number of random bytes acquired in the implementation.
– **Benchmark the randomness sources** used by the implementation separately. This allows to establish a trade-off between the use of TRNG and PRNG.
– **Use a realistic benchmarking platform.** Emulators are very useful tools, but as with any tool it has to be used correctly. The use of a real-world platform will provide more realistic results. Use a widely available platform for evaluation.
– **Provide all relevant platform settings:** clock setup, configuration of the caching mechanisms, flash wait cycles, TRNG clock frequency, etc.
– **Document the toolchain and compiler settings** used during the development and evaluation of the implementation.

Note that many of these recommendations can be easily addressed by providing a **public implementation**.

Acknowledgements This work was supported in part by CyberSecurity Research Flanders with reference number VR20192203. In part by the Research Council KU Leuven C1 on Security and Privacy for Cyber-Physical Systems and the Internet of Things with contract number C16/15/058. In addition, this work was supported by the European Commission through the Horizon 2020 research and innovation programme under grant agreement Cathedral ERC Advanced Grant 695305, under grant agreement H2020-FETFLAG-2018-03-820405 QRANGE and under grant agreement H2020-DS-LEIT-2017-780108 FENTEC.

References

1. Adomnicai, A., Peyrin, T.: Fixslicing AES-like Ciphers: New bitsliced AES speed records on ARM-Cortex M and RISC-V. IACR Transactions on Cryptographic Hardware and Embedded Systems **2021**(1), 402–425 (Dec 2020). https://doi.org/10.46586/tches.v2021.i1.402-425, `https://tches.iacr.org/index.php/TCHES/article/view/8739`
2. Balasch, J., Faust, S., Gierlichs, B., Paglialonga, C., Standaert, F.: Consolidating inner product masking. In: Takagi, T., Peyrin, T. (eds.) Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10624, pp. 724–754. Springer (2017). https://doi.org/10.1007/978-3-319-70694-8_25, `https://doi.org/10.1007/978-3-319-70694-8_25`
3. Balasch, J., Gierlichs, B., Grosso, V., Reparaz, O., Standaert, F.: On the Cost of Lazy Engineering for Masked Software Implementations. In: Smart Card Research and Advanced Applications - 13th International Conference, CARDIS 2014, Paris, France, November 5-7, 2014. Revised Selected Papers. pp. 64–81 (2014). https://doi.org/10.1007/978-3-319-16763-3_5, `https://doi.org/10.1007/978-3-319-16763-3_5`
4. Benadjila, R., Khati, L., Prouff, E., Thillard, A.: Hardened Library for AES-128 encryption/decryption on ARM Cortex M4 Achitecture. `https://github.com/ANSSI-FR/SecAESSTM32`, [Online; accessed 13-July-2020]
5. Brier, E., Clavier, C., Olivier, F.: Correlation power analysis with a leakage model. In: Joye, M., Quisquater, J. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings. Lecture Notes in Computer Science, vol. 3156, pp. 16–29. Springer (2004). https://doi.org/10.1007/978-3-540-28632-5_2, `https://doi.org/10.1007/978-3-540-28632-5_2`
6. Chari, S., Jutla, C.S., Rao, J.R., Rohatgi, P.: Towards sound approaches to counteract power-analysis attacks. In: Wiener, M.J. (ed.) Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings. Lecture Notes in Computer Science, vol. 1666, pp. 398–412. Springer (1999). https://doi.org/10.1007/3-540-48405-1_26, `https://doi.org/10.1007/3-540-48405-1_26`
7. Cheng, W., Carlet, C., Goli, K., Danger, J.L., Guilley, S.: Detecting Faults in Inner Product Masking Scheme IPM-FD: IPM with Fault Detection. Journal of Cryptographic Engineering (May 2020). https://doi.org/10.1007/s13389-020-00227-6, `https://hal-cnrs.archives-ouvertes.fr/hal-02915673`

8. Coron, J.: Higher order masking of look-up tables. In: Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings. pp. 441–458 (2014). https://doi.org/10.1007/978-3-642-55220-5_25, https://doi.org/10.1007/978-3-642-55220-5_25
9. Coron, J., Giraud, C., Prouff, E., Renner, S., Rivain, M., Vadnala, P.K.: Conversion of security proofs from one leakage model to another: A new issue. In: Schindler, W., Huss, S.A. (eds.) Constructive Side-Channel Analysis and Secure Design - Third International Workshop, COSADE 2012, Darmstadt, Germany, May 3-4, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7275, pp. 69–81. Springer (2012). https://doi.org/10.1007/978-3-642-29912-4_6, https://doi.org/10.1007/978-3-642-29912-4_6
10. Coron, J., Greuet, A., Zeitoun, R.: Side-channel masking with pseudo-random generator. In: Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part III. pp. 342–375 (2020). https://doi.org/10.1007/978-3-030-45727-3_12, https://doi.org/10.1007/978-3-030-45727-3_12
11. eShard: SCAred. https://gitlab.com/eshard/scared, [Online; accessed 16-December-2021]
12. Gigerl, B., Hadzic, V., Primas, R., Mangard, S., Bloem, R.: Coco: Co-Design and Co-Verification of Masked Software Implementations on CPUs. In: 30th USENIX Security Symposium (USENIX Security 21). USENIX Association (Aug 2021), https://www.usenix.org/conference/usenixsecurity21/presentation/gigerl
13. Gilbert Goodwill, B.J., Jaffe, J., Rohatgi, P., et al.: A testing methodology for side-channel resistance validation. In: NIST non-invasive attack testing workshop. vol. 7, pp. 115–136 (2011)
14. Goubin, L., Patarin, J.: DES and differential power analysis (the "duplication" method). In: Koç, Ç.K., Paar, C. (eds.) Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings. Lecture Notes in Computer Science, vol. 1717, pp. 158–172. Springer (1999). https://doi.org/10.1007/3-540-48059-5_15, https://doi.org/10.1007/3-540-48059-5_15
15. Groß, H., Stoffelen, K., Meyer, L.D., Krenn, M., Mangard, S.: First-Order Masking with Only Two Random Bits. In: Proceedings of ACM Workshop on Theory of Implementation Security Workshop, TIS@CCS 2019, London, UK, November 11, 2019. pp. 10–23 (2019). https://doi.org/10.1145/3338467.3358950, https://doi.org/10.1145/3338467.3358950
16. Inc., M.T.: SAM D5x/E5x Family Data Sheet. https://ww1.microchip.com/downloads/en/DeviceDoc/SAM_D5xE5x_Family_Data_Sheet_DS60001507F.pdf (2020), [Online; accessed 3-December-2020]
17. Ishai, Y., Sahai, A., Wagner, D.: Private circuits: Securing hardware against probing attacks. In: Boneh, D. (ed.) Advances in Cryptology - CRYPTO 2003. pp. 463–481. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
18. Kannwischer, M.J., Rijneveld, J., Schwabe, P., Stoffelen, K.: PQM4: Post-quantum crypto library for the ARM Cortex-M4, https://github.com/mupq/pqm4
19. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M.J. (ed.) Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings. Lecture Notes in Computer Science, vol. 1666, pp. 388–397. Springer

(1999). https://doi.org/10.1007/3-540-48405-1_25, https://doi.org/10.1007/3-540-48405-1_25

20. McCann, D., Oswald, E., Whitnall, C.: Towards practical tools for side channel aware software engineering: 'grey box' modelling for instruction leakages. In: 26th USENIX Security Symposium (USENIX Security 17). pp. 199–216. USENIX Association, Vancouver, BC (Aug 2017), https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/mccann
21. NewAE Technology Inc.: ChipWhisperer. https://github.com/newaetech/chipwhisperer, [Online; accessed 16-December-2021]
22. O'Flynn, C., Chen, Z.D.: ChipWhisperer: An open-source platform for hardware embedded security research. In: Constructive Side-Channel Analysis and Secure Design - 5th International Workshop, COSADE 2014, Paris, France, April 13-15, 2014. Revised Selected Papers. pp. 243–260 (2014). https://doi.org/10.1007/978-3-319-10175-0_17, https://doi.org/10.1007/978-3-319-10175-0_17
23. Renner, S., Pozzobon, E., Mottok, J.: A hardware in the loop benchmark suite to evaluate NIST LWC ciphers on microcontrollers. In: Meng, W., Gollmann, D., Jensen, C.D., Zhou, J. (eds.) Information and Communications Security - 22nd International Conference, ICICS 2020, Copenhagen, Denmark, August 24-26, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12282, pp. 495–509. Springer (2020). https://doi.org/10.1007/978-3-030-61078-4_28, https://doi.org/10.1007/978-3-030-61078-4_28
24. Rivain, M., Prouff, E.: Provably Secure Higher-Order Masking of AES. In: Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings. pp. 413–427 (2010). https://doi.org/10.1007/978-3-642-15031-9_28, https://doi.org/10.1007/978-3-642-15031-9_28
25. Schwabe, P., Stoffelen, K.: All the AES You Need on Cortex-M3 and M4. In: Selected Areas in Cryptography - SAC 2016 - 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers. pp. 180–194 (2016). https://doi.org/10.1007/978-3-319-69453-5_10, https://doi.org/10.1007/978-3-319-69453-5_10
26. Shelton, M.A., Samwel, N., Batina, L., Regazzoni, F., Wagner, M., Yarom, Y.: Rosita: Towards automatic elimination of power-analysis leakage in ciphers. In: 28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021. The Internet Society (2021), https://www.ndss-symposium.org/ndss-paper/rosita-towards-automatic-elimination-of-power-analysis-leakage-in-ciphers/
27. STMicroelectronics: RM0090 Reference manual. https://www.st.com/resource/en/reference_manual/dm00031020-stm32f405-415-stm32f407-417-stm32f427-437-and-stm32f429-439-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf (2019), [Online; accessed 3-December-2020]
28. Stoffelen, K.: aes-armcortexm. https://github.com/Ko-/aes-armcortexm, [Online; accessed 30-September-2020]