Vectorizing Higher-Order Masking

Benjamin Grégoire, Kostas Papagiannopoulos, Peter Schwabe, Ko Stoffelen

Vectorizing Higher-Order Masking

Benjamin Grégoire[1] and Kostas Papagiannopoulos[2] and Peter Schwabe[2] and Ko Stoffelen[2]

[1] INRIA Sophia Antipolis, France
[2] Digital Security Group, Radboud University, Nijmegen, The Netherlands

Abstract. The cost of higher-order masking as a countermeasure against side-channel attacks is often considered too high for practical scenarios, as protected implementations become very slow. At Eurocrypt 2017, the bounded moment leakage model was proposed to study the (theoretical) security of parallel implementations of masking schemes [5]. Work at CHES 2017 then brought this to practice by considering an implementation of AES with 32 shares [26], bitsliced inside 32-bit registers of ARM Cortex-M processors. In this paper we show how the NEON vector instructions of larger ARM Cortex-A processors can be exploited to build much faster masked implementations of AES. Specifically, we present AES with 4 and 8 shares, which in theory provide security against 3rd and 7th-order attacks, respectively. The software is publicly available and optimized for the ARM Cortex-A8. We use refreshing and multiplication algorithms that are proven to be secure in the bounded moment leakage model and to be strongly non-interfering. Additionally, we perform a concrete side-channel evaluation on a BeagleBone Black, using a combination of test vector leakage assessment (TVLA), leakage certification tools and information-theoretic bounds.

Keywords: Higher-order masking, side-channel analysis, AES, ARM Cortex-A8

1 Introduction

There is a long history of protecting AES [15] implementations against side-channel analysis (SCA) attacks. Side-channel attacks exploit physical information, such as power consumption or electromagnetic radiation of devices running some cryptographic primitive, to learn information about secret data, typically cryptographic keys. Higher-order masking is a well-studied countermeasure against such attacks [11,22]; unfortunately, it comes at a rather high cost in terms of performance. This is a reason why in practice, well-protected implementations are not as ubiquitous as one would hope. In software, higher-order masked implementations are typically orders of magnitude slower compared to unprotected implementations, as was explored at Eurocrypt 2017 by Goudarzi and Rivain [23].

Simultaneously at Eurocrypt 2017, a theoretical model was proposed to study the security of *parallel implementations* of masking schemes, called the bounded moment leakage model [5]. As parallelization is a very powerful tool to increase

performance, this model gives the foundation for faster protected implementations. One common way to parallelize software implementations is through vectorization. In a vectorized implementation, a single instruction operates on multiple data elements inside one vector register at the same time. For vectorization to be useful, data parallelism is required, which in the case of higher-order masking is trivially provided by the availability of multiple shares.

Precisely this approach of vectorization with data-level parallelism coming from multiple shares was used in a CHES 2017 paper by Journault and Standaert [26]. That paper studies a parallel bitsliced (i.e., vectorized with 1-bit vector elements) implementation using 32 shares on the ARM Cortex-M4. The reason for using 32 shares was the fact that the Cortex-M4 has 32-bit registers and bitslicing thus needs 32× data-level parallelism. Empirical tests in this paper confirmed that the bounded moment model is useful also in practice. Specifically, these tests showed that a 4-share version of their implementation yielded no leakage of order less than 4. It is of course still possible that the actual security order is lower, but it can at least be viewed as an optimistic result. They conclude their evaluation by performing an information-theoretic analysis of the leakage in order to bound the attack complexity for the 32-share implementation.

In this paper we study how the powerful NEON vector unit on larger ARM Cortex-A processors can be used to obtain efficient masked AES implementations. Straight-forwardly adapting the approach from [26] to obtain data-level parallelism would result in implementations with 64 or 128 shares (for 64-bit or 128-bit vector registers), which would be a security overkill and result in terrible performance. Instead we follow the approach of the bitsliced AES implementations presented in [27,30], which exploit the data-level parallelism of 16 independent S-Box computations. As a result, we present implementations using 4 and 8 shares, which in theory offer security at the 3rd and 7th order. We use refreshing and multiplication algorithms that are based on the algorithms in [5] and even slightly improve on some of them by requiring less randomness. They are proven secure in the bounded moment model and also proven to be strongly non-interfering [3]. We provide a concrete evaluation of our implementations on a BeagleBone Black, which has been used successfully before to perform differential electromagnetic analysis at 1 GHz [2]. Using nearly the same setup, we employ the popular TVLA methodology [13] in conjunction with leakage certification [18] and we show that there is actually some leakage in the 3rd order of our 4-share implementation, but not in the 2nd order. We then continue to bound the measurement complexity of the 8-share implementation using an information-theoretic approach [17].

To summarize, the contributions of this paper are that

- we provide the first vectorized instantiation of the bounded moment leakage model published at Eurocrypt 2017 [5] with strong non-interference [3];
- we provide the fastest publicly available higher-order masked AES implementations with 4 and 8 shares for the ARM Cortex-A8; and that
- we perform a practical side-channel evaluation of the 4-share AES implementation and derive security bounds for the 8-share implementation.

Source Code. The source code of our implementations is available in the public domain. It can be downloaded at https://github.com/Ko-/aes-masked-neon.

2 Preliminaries

2.1 Higher-Order Masking of AES

Implementations of cryptographic primitives such as block ciphers are typically vulnerable to attacks that use side-channel analysis (SCA). Information about physical characteristics, such as the electromagnetic radiation, of a device that executes a block cipher can be used to recover the secret key [21,29].

A well-studied countermeasure against this class of attacks is (higher-order) masking. It works by splitting each secret variable x into d shares x_i that satisfy $x_0 \oplus x_1 \oplus x_2 \oplus \cdots \oplus x_{d-1} = x$. When $\oplus$ denotes the Boolean `XOR` operation, this is called Boolean masking. Any $d-1$ of these shares should be statistically independent of x and should be uniformly randomly distributed. If this is the case, then this masking scheme provides privacy in the $(d-1)$-probing model, as put forward by Ishai, Sahai, and Wagner [25]. The idea is that an attacker applying $d-1$ probes to learn intermediate values of the computation will not be able to learn anything about the secret value. We call the value $d-1$ the *order* of the masking scheme.

When masking is applied, operations on x are to be performed on its shares. For linear operations f, those that satisfy $f(x+y) = f(x)+f(y)$ and $f(ax) = af(x)$, it holds that they can trivially be computed on the shares of x individually. For nonlinear operations, several algorithms have been suggested to retrieve the correct result. In [25] it was shown how to compute a masked `AND` gate and, together with the linear `NOT`, this is functionally complete.

AES [15] in particular has received a lot of attention when it comes to protected implementations. The round function of AES consists of AddRoundKey, SubBytes, ShiftRows, and MixColums. AddRoundKey, ShiftRows, and MixColumns are all linear. SubBytes is not. Much research has therefore been aimed at finding efficient representations of a masked variant of the AES S-box [10,23,28,33].

2.2 Strong Non-interference

Strong non-interference (SNI) is a security notion, formalized in [3], that is slightly stronger than probing security. It currently seems to be the right security notion when considering practical security. The problem with probing security is that, given two algorithms that are secure at order $d-1$ in the probing model, the composition of these algorithms is not necessarily secure at order $d-1$. SNI, on the other hand, means that an algorithm is *composable*, guaranteeing that one can verify the security of the composition of multiple secure algorithms.

As an example to see why SNI is desirable, consider the provably secure masking scheme by Rivain and Prouff from CHES 2010 [33]. Three years later,

an attack was found against the composition of the refreshing of masks and the masked multiplication [14]. The scheme was fixed subsequently. It was shown in [3] that the main difference between the original and the fixed algorithms is exactly this notion of strong non-interference.

Automated verification tools exist to formally prove strong non-interference. This gives stronger guarantees on the theoretical security of a masking scheme.

2.3 Bounded Moment Leakage Model

The probing model and its variants are not always straightforward to interpret. The fact that $d-1$ shares should be statistically independent is based on the idea that an attacker can inspect the leakage of intermediate computations on the shares separately. In software, it therefore applies better to serial implementations. When computations are performed on multiple shares in parallel, it is not immediately clear what the relation with the probing security model is.

To handle this, the bounded moment model has been proposed in [5]. It is more targeted towards parallel implementations and can deal with the concept that multiple shares are manipulated simultaneously. Barthe, Dupressoir, Faust, Grégoire, Standaert, and Strub proved that probing security of a serial implementation implies bounded moment security for its parallel counterpart. It is a weaker security notion than the noisy leakage model [11,32].

Security in the bounded moment model is defined using *leakage vectors* and *mixed moments*. For every clock cycle c, there is a leakage vector L_c. The leakage vector is a random variable that is computed as the sum of a deterministic part that depends on the shares that are manipulated, and on the noise R_c. The mixed moment of a set $\{Y_1, \ldots, Y_r\}$ of r random variables at orders $o_1, \ldots, o_r$ can be defined as $\mathrm{E}\left[\prod_{i=1}^{r} Y_i\right]$, where E denotes the expected value. Now, consider an N-cycle cryptographic implementation that manipulates a secret variable x. This results in a set $\{L_1, \ldots, L_N\}$ of N leakage vectors. The implementation is said to be secure at order o in the bounded moment model if all the mixed moments of order $\leq o$ of $\{L_1, \ldots, L_N\}$ are statistically independent of x.

2.4 Vectorization with NEON

The ARM Cortex-A8 is a 32-bit processor that implements the ARMv7-A microarchitecture. It is used in smartphones, digital TVs, and printers, among others. It was first introduced in 2005 and is currently widely deployed. Its main core can run at 1 GHz and implements features such as superscalar execution, an advanced branch prediction unit, and a 13-stage pipeline. There are 16 32-bit `r` registers, of which 14 are generally available to the programmer.

The Cortex-A8 comes with the so-called Advanced SIMD extension, better known as NEON, that add another 16 128-bit `q` registers. These vector registers can also be viewed as 32 64-bit `d` registers. For example, `q0` consists of `d0` and `d1`, `q1` consists of `d2` and `d3`, et cetera. Operations can typically be performed on 8-, 16-, or 32-bit elements in a SIMD fashion. While 128-bit registers are supported, the data path of the Cortex-A8 is actually only 64 bits wide, which

means that operations on 128-bit registers will be performed in two steps. NEON has a separate 10-stage pipeline. In particular, it has a load/store unit that runs next to an arithmetic unit. This means that an aligned load and an arithmetic instruction can be executed in the same cycle.

NEON has been used successfully in the past to vectorize and optimize implementations of cryptographic primitives [8], but its power has to the best of our knowledge not yet been exploited for higher-order masking in the way that we propose here.

3 Vectorizing Masking of AES

3.1 Representing the Masked State

The AES state [15] is usually pictured as a square matrix of 4 by 4 byte elements. This representation leads to efficient software implementations when SubBytes is implemented using lookup tables. However, such implementations are also prone to cache-timing attacks [7], as the memory location of the value that is looked up depends on some secret intermediate value. An alternative bitsliced representation avoids these attacks. In this bitsliced representation, all the first bits of every byte are put in one register, all the second bits in the next register, etc. For SubBytes, one can now compute the S-box on the individual bits and do that for all 16 bytes in parallel. The S-box parallelism of AES for bitslicing was first exploited by Könighofer in [30] and it was also used in the speed-record-setting AES implementation targeting Intel Core 2 processors by Käsper and Schwabe [27]. At a small cost, the other (linear) operations of AES are modified to operate on this bitsliced representation as well.

However, on most devices registers are longer than 16 bits, so it would be a waste to not utilize this. AES implementations without side-channel protections choose to process multiple blocks in parallel, by simply concatenating multiple 16-bit chunks from independent blocks in one register. For example, the AES implementation of [27] processes 8 blocks in parallel in a 128-bit XMM register. When the vector registers become larger, this trivially leads to higher throughputs for parallel modes of operation.

In this section we present three implementations that, instead of multiple blocks, process multiple shares in parallel. The first implementation fills a 64-bit `d` register with 4 shares. The second has 8 shares, that are used to fill a 128-bit `q` register. The third combines 2 blocks with each 4 shares, and also utilizes the 128-bit `q` registers. It interleaves the shares of the 2 blocks for efficiency reasons. Note that this third implementation requires a parallel mode of operation.

3.2 Parallel Multiplication and Refreshing

In [5], new algorithms for parallel multiplication (including the `AND` operation) and parallel refreshing were proposed. They are proven to be secure in the bounded moment model and proven to be strongly non-interfering using techniques from

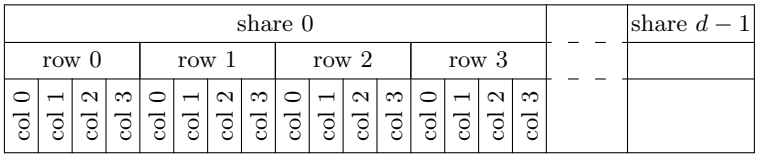


Fig. 1: Register lay-out for the single-block implementations. There are 8 of these $16d$-bit vector registers. The cells on the bottom row represent individual bits.

automated program verification [4]. Correct implementations of these algorithms are critical for the security of our implementations. We suggest slightly improved algorithms for $d = 4$ and $d = 8$ that require less randomness, but we could not generalize them to an improvement for all orders. As with the original algorithms, they are proven secure using the same automatic verification tools. NEON code that implements these algorithms can be found in Appendix A.

Refreshing. Refreshing can be necessary to make sure that values in registers are again statistically independent. The refreshing algorithm in [5] requires $2d$ bytes of fresh uniform randomness. Let $\boldsymbol{x}$ (in boldface) denote a vector register that contains $[x_0, \ldots, x_{d-1}]$, where $\bigoplus_{i=0}^{d-1} x_i = x$, and let $\boldsymbol{r}$ be a vector of the same length that contains uniformly random values. In the case of AES, a single share would be 16 bits long, so a randomness vector $\boldsymbol{r}$ will be $2d$ bytes.

Then $\boldsymbol{x}' = \boldsymbol{r} \oplus \mathrm{rot}(\boldsymbol{r}, 1) \oplus \boldsymbol{x}$ is a secure way to refresh x, where $\mathrm{rot}(\boldsymbol{a}, n)$ rotates $\boldsymbol{a}$ to either left or right by n shares. Note that in the case of AES, this is equal to applying a rotation by $2n$ bytes.

For 4 shares, this algorithm additionally achieves SNI. However, to reach this with 8 shares, in [5] it turned out to be necessary to iterate the refreshing algorithm 3 times. In other words, one would need to compute

$$\boldsymbol{r} \oplus \mathrm{rot}(\boldsymbol{r}, 1) \oplus \boldsymbol{r}' \oplus \mathrm{rot}(\boldsymbol{r}', 1) \oplus \boldsymbol{r}'' \oplus \mathrm{rot}(\boldsymbol{r}'', 1) \oplus \boldsymbol{x}$$

to achieve SNI at order 7. This requires 3 vectors of uniform randomness, or 48 bytes with AES. We improve this algorithm by computing:

$$\boldsymbol{r} \oplus \mathrm{rot}(\boldsymbol{r}, 1) \oplus \boldsymbol{r}' \oplus \mathrm{rot}(\boldsymbol{r}', 2) \oplus \boldsymbol{x}\,.$$

We verified with the current version of the tool of [4] that this also achieves SNI at order 7. Moreover, it requires one less randomness vector. In the case of AES, we now require 32 bytes of uniform randomness.

Multiplication. Multiplication in a finite field, or an `AND` gate in the case of $\mathbb{F}_2$, is trickier to perform in a secure way. Consider the case where one wants to compute $z = x \cdot y$. Let $\boldsymbol{r}$ and $\boldsymbol{r}'$ be uniformly random vectors. Then, with 4 shares, the algorithm suggested in [5] computes the following to achieve SNI at order 3:

$$\begin{aligned}\boldsymbol{z} = {} & \boldsymbol{x} \cdot \boldsymbol{y} \oplus \boldsymbol{r} \oplus \boldsymbol{x} \cdot \mathrm{rot}(\boldsymbol{y}, 1) \oplus \mathrm{rot}(\boldsymbol{x}, 1) \cdot \boldsymbol{y} \oplus \mathrm{rot}(\boldsymbol{r}, 1) \\ & \oplus \boldsymbol{x} \cdot \mathrm{rot}(\boldsymbol{y}, 2) \oplus \boldsymbol{r}' \oplus \mathrm{rot}(\boldsymbol{r}', 1)\,.\end{aligned}$$

However, we can again improve this slightly such that less randomness will be necessary. Let r_4 be a uniformly random value. Then we proved using the tool of [4] that the following is also 3rd-order SNI-secure. For AES, this requires 10 fresh uniformly random bytes (8 for $\boldsymbol{r}$ and 2 for r_4) instead of 16:

$$\begin{aligned}\boldsymbol{z} = \boldsymbol{x} \cdot \boldsymbol{y} \oplus \boldsymbol{r} \oplus \boldsymbol{x} \cdot \mathrm{rot}(\boldsymbol{y}, 1) \oplus \mathrm{rot}(\boldsymbol{x}, 1) \cdot \boldsymbol{y} \oplus \mathrm{rot}(\boldsymbol{r}, 1) \\ \oplus \boldsymbol{x} \cdot \mathrm{rot}(\boldsymbol{y}, 2) \oplus [r_4, r_4, r_4, r_4]\,.\end{aligned}$$

With 8 shares, we use the original algorithm of [5] that is SNI at order 7. This requires 3 randomness vectors, which in the case of AES amounts to 48 bytes:

$$\begin{aligned}\boldsymbol{z} = \boldsymbol{x} \cdot \boldsymbol{y} \oplus \boldsymbol{r} \oplus \boldsymbol{x} \cdot \mathrm{rot}(\boldsymbol{y}, 1) \oplus \mathrm{rot}(\boldsymbol{x}, 1) \cdot \boldsymbol{y} \oplus \mathrm{rot}(\boldsymbol{r}, 1) \\ \oplus \boldsymbol{x} \cdot \mathrm{rot}(\boldsymbol{y}, 2) \oplus \mathrm{rot}(\boldsymbol{x}, 2) \cdot \boldsymbol{y} \oplus \boldsymbol{r}' \\ \oplus \boldsymbol{x} \cdot \mathrm{rot}(\boldsymbol{y}, 3) \oplus \mathrm{rot}(\boldsymbol{x}, 3) \cdot \boldsymbol{y} \oplus \mathrm{rot}(\boldsymbol{r}', 1) \\ \oplus \boldsymbol{x} \cdot \mathrm{rot}(\boldsymbol{y}, 4) \oplus \boldsymbol{r}'' \oplus \mathrm{rot}(\boldsymbol{r}'', 1)\,.\end{aligned}$$

We attempted to reduce this by replacing the last randomness vector by a vector with a single random value, as in the algorithm for 4 shares, but we found that this does not achieve SNI at order 7.

Randomness. Implementations that are protected using higher-order masking require a lot of randomness. To be able to prove statistical independence, this randomness should be fresh and uniformly distributed. For resisting attacks in practice, it is not so clear whether the exact requirements are this strict. For instance, it might also be fine to expand a random seed using a pseudo-random number generator, or even to re-use randomness [1]. We consider this discussion to be out of scope of this work. However, because the impact on the performance can be very significant, we consider various approaches that occur in the literature. The first is to read all the randomness that we require from `/dev/urandom` using `fread`, like in [2]. This is the most conservative approach, but it is rather slow. Second, we also consider the case where all required randomness is already in a file that needs to be read into memory. The third approach assumes that there exists a fast true random-number generator and only considers the cost of a normal load instruction (`vld1`), like in [23].

The AES implementation with 4 shares requires 8 bytes per refresh and 10 bytes per masked `AND`. In the next section we will see that this amounts to $10 \cdot 32 \cdot (8 + 10) = 5760$ random bytes in total for the full AES, excluding the randomness used to do the initial masking of the input and the round keys. Naturally, the implementation that computes two blocks in parallel requires double the amount of random bytes. For 8 shares, refreshing takes 32 bytes and a masked `AND` uses 48 bytes, which makes the total $10 \cdot 32 \cdot (32 + 48) = 25600$ bytes.

3.3 SubBytes

Using the masked `AND` and refreshing algorithms, we can build our bitsliced SubBytes. Several papers have presented optimized bitsliced representations of

the AES S-box. The smallest known to us is by Boyar and Peralta [9]. It uses 83 `XOR`s/`XNOR`s and 32 `AND`s, which was later improved to 81 `XOR`s/`XNOR`s and 32 `AND`s. The few `NOT`s can be moved into the key expansion, so we only need to consider `XOR`s and `AND`s. We use this implementation as our starting point, as this is also the implementation with the smallest number of binary `AND`s, and an `AND` will be much slower to compute than a `XOR`.

We have used the compiler provided in [3] to generate a first masked implementation of SubBytes. This tells us when it is necessary to refresh a value, making sure that we do not refresh more often than strictly necessary. For our version of SubBytes, however, the compiler adds a refresh on one of the inputs for every `AND`. Then we implement an `XOR` on multiple shares in parallel with a `veor` instruction. For an `AND`, we use the algorithms of the previous section. Finally, the code has been manually optimized to limit pipeline stalls.

The S-box implementation has many intermediate variables. With 4 shares and a single block, the `d` registers are used. There are 32 of them and this turns out to be sufficient to store all the intermediate values. With two blocks or with 8 shares, however, we can use only 16 `q` registers. This implies that values have to be spilled to the stack. Of course, we want to minimize the overhead caused by this. In [36], an instruction scheduler and register allocator for the ARM Cortex-M4 was used to optimize the number of pushes to the stack. We modified this tool to handle the NEON instructions that we need, and use it to obtain an implementation with 18 push instructions and 18 loads.

According to a cycle-count simulator [38], our SubBytes implementation takes 1035 cycles with one block and 4 shares and 2127 cycles with 8 shares.

3.4 Linear Layer

We now discuss the linear operations of AES. We manually optimized them using a cycle-count simulator to hide as many latencies as possible [38].

AddRoundKey. AddRoundKey loads the round key with the `vld1` instruction and adds it to the state using `veor`. The loads and arithmetic instructions can be interleaved. This helps because they go into separate NEON pipelines. An arithmetic instruction can than be executed in parallel with the load of the next part of the round key. For the loads, we make sure that they are aligned to at least 64 bits. AddRoundKey then only takes 10 cycles.

ShiftRows. With ShiftRows, rotations by fixed distances over 16 bits need to be computed. This can be implemented using `vand`, `vsra`, `vshl`, and `vorr` instructions. The arithmetic pipeline is now clearly the bottleneck. According to the simulator, our ShiftRows takes 150 cycles.

MixColumns. MixColumns requires more rotations by 4 or by 12 over 16 bits. This takes 106 cycles as measured by the simulator.

3.5 Performance

We benchmark our implementations on the BeagleBone Black with the clock frequency fixed at 1 GHz. In other words, we disabled frequency scaling. For the rest, we did not apply any changes to a standard Debian Linux 9 installation. In particular, we did not disable background processes and did not give our process special priority or CPU core affinity. The implementations are run 10000 times and the median cycle counts are given in Table 1.

	4 shares 1 block	4 shares 2 blocks	8 shares 1 block
Clock cycles (randomness from `/dev/urandom`)	1,598,133	4,738,024	9,470,743
Clock cycles (randomness from normal file)	14,488	17,586	26,601
Clock cycles (pre-loaded randomness)	12,385	15,194	23,616
Random bytes	5,760	11,520	25,600
Stack usage in bytes	12	300	300
Code size in bytes	39,748	44,004	70,188

Table 1: Performance of our masked AES implementations.

When using `/dev/urandom`, more than 99% of the time is spent on waiting for randomness, which is delivered at a rate of only 369 cycles per byte in the 8-share case. With a faster RNG, it becomes clear that our implementations are very fast and practical. We reach 474 cycles/byte with 4 shares and 1476 cycles/byte with 8 shares with pre-loaded randomness. Note that all implementations are fully unrolled, so the code size can trivially be decreased to roughly a tenth when this is a concern. However, we do not expect this to be an issue for devices with a Cortex-A8 or similar microprocessors, as they are relatively high-end.

Comparison to Related Work. In the following we discuss how our implementation compares to related work. We note that one should be cautious when it comes to comparing cycle counts, in particular when benchmarks were obtained on different microarchitectures or from simulators.

Goudarzi and Rivain [23] compared the performance of different higher-order masking approaches on ARM architectures. A simplified model is assumed for the number of cycles that specific instructions take, without referring to a specific microarchitecture. Private communication made clear that they are derived from the Keil simulator based on an ARM7TDMI-S. Their fastest bitsliced implementation is claimed to take 120,972 cycles with 4 shares and 334,712 cycles with 8 shares. To achieve this performance, the presence of a fast TRNG is assumed that delivers fresh randomness at 2.5 cycles per byte. Only the cost of a normal `ldr` instruction it taken into account, which corresponds to our performance with pre-loaded randomness. Despite the differences between ARMv4T and ARMv7-A, it is clear that there is quite a performance gap.

Wang, Vadnala, Großschädl, and Xu [43] presented a masked AES implementation for NEON that appears to run in 14,855 cycles with 4 shares and 77,820 with 8 shares on a Cortex-A15 simulator. This uses a cheap LFSR-based PRNG to provide randomness of which the authors already say that it should be replaced by a better source of randomness. We require less randomness due to a different masking scheme and apply bitslicing instead of computing SubBytes with tower-field arithmetic. The Cortex-A15 is more modern and powerful than the Cortex-A8. It can decode 3 instructions instead of 2, has out-of-order execution, and its NEON unit has a 128-bit wide datapath instead of 64-bit. However, it has longer pipelines which means that the penalty for, for instance, wrong branch predictions will be higher. We ran their code on our Cortex-A8-based benchmarking device and measured 34,662 cycles for the 4-share implementation and 158,330 cycles for the 8-share implementation, but we cannot fully explain the difference due to the amount of possible causes and the unavailability of more detailed information.

Balasch, Gierlichs, Reparaz, and Verbauwhede [2] do use the same microarchitecture, but not the NEON SIMD processor. They do not mention the performance of their implementation. They explicitly say that they focus on the security evaluation and do not aim to achieve a high-throughput implementation.

Finally, Journault and Standaert [26] consider a bitsliced AES implementation with up to 32 shares on an ARM Cortex-M4. They exploit the parallelism of the shares, but not of AES itself as there are only 32-bit registers. An on-board TRNG is used to provide randomness at a reported speed of 20 cycles per byte. They use the refreshing and multiplication algorithms of [5] and almost the same S-box baseline implementation. Eventually, they report that 2,783,510 cycles are required to compute AES with 32 shares, of which 73% are spent on generating randomness. While this is certainly a very interesting idea, we show how the parallelism in SubBytes can additionally be exploited on a higher-end CPU with vector registers when using less shares might be sufficient.

Compared to unmasked implementations, there is of course still a noticeable performance penalty for adding side-channel protections. The unmasked bitsliced AES implementation of Bernstein and Schwabe [8] also exploits NEON to run at 19.12 cycles per byte (i.e., 306 cycles per block) in CTR mode, but that uses counter-mode caching and processes 8 blocks in parallel.

4 Side-Channel Evaluation

4.1 Measurement Setup

Balasch, Gierlichs, Reparaz, and Verbauwhede [2] described in detail how they performed DPA attacks on a BeagleBone Black running at 1 GHz. Our experimental setup and measuring environment follow their approach. The board is running Debian Jessie and has several processes running in the background. We power the board using a standard AC adapter and connect it to the measurement PC over Ethernet. A few lines of Python on the BeagleBone open a TCP socket

and spawn a new AES process for every input that it receives. The measurement PC connects to the socket and sends inputs over Ethernet.

We use a LeCroy WaveRunner 8404M-MS oscilloscope with a bandwidth of 4 GHz, operating at a sampling rate of 2.5 GSamples/sec. The AES process sets a GPIO port high before the execution of AES and sets it low after AES is finished, so that it can be used as the trigger signal. We place a magnetic field probe from Langer, model RF-B 0.3-3, with a small tip on the back of the BeagleBone board, near capacitor 66. The probe is connected to a Langer amplifier, model PA 303 SMA. The acquired traces were post-processed in order to perform signal alignment. We note that OS-related interrupts in conjunction with time-variant cache behavior result in a fairly unstable acquisition process. Thus, the evaluator has to either discard a large portion of the acquired trace set or resort to more sophisticated alignment techniques such as elastic alignment [41].

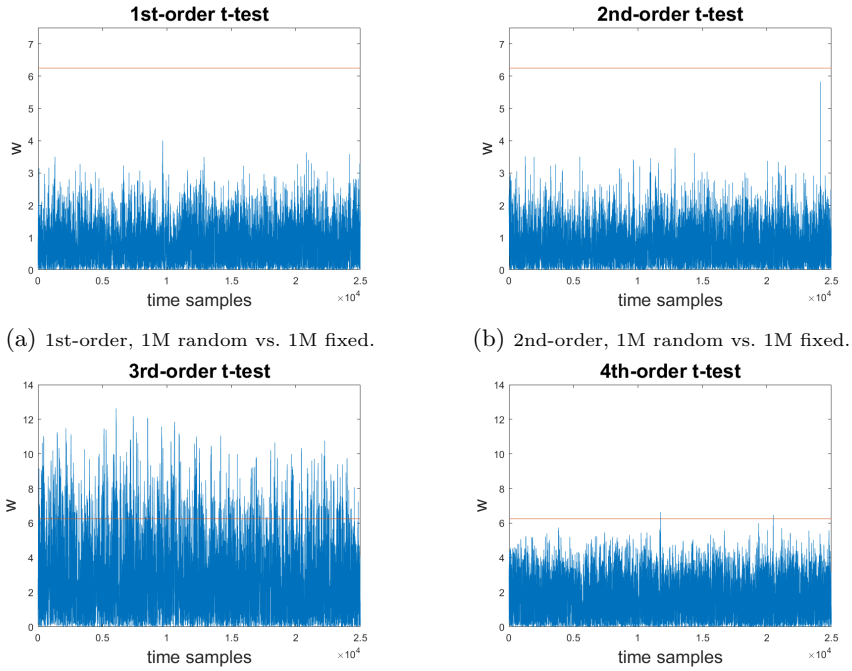
4.2 Security Order Evaluation

Since our implementation uses SNI gadgets, it maintains theoretical security against probing attacks of order $d-1$ or less. The natural starting point of our side-channel evaluation is to identify any discrepancy between the theoretical and the actual security order, i.e., to determine the real-world effectiveness of the masking scheme. To achieve that goal, we need to assess whether the shares leak independently or whether the leakage function recombines them. Such recombinations can be captured by evaluating the security order in the bounded moment model [5] using, e.g., the leakage detection methodology [13,34,44].

Several lines of work have observed divergence between the theoretical order of a masking scheme and its real-world counterpart. Initially, Balasch, Gierlichs, Grosso, Reparaz, and Standaert [1] put forward the issue of distance-based leakages, which can result in the order reduction of a scheme. Specifically, if a $(d-1)$th-order scheme is implemented on a device that exhibits distance-based leakages, its actual order will reduce to $\lfloor (d-1)/2 \rfloor$, damaging its effectiveness w.r.t. noise amplification. Such effects have been observed in numerous architectures such as AVR, 8051 [1], ARM Cortex-M4 [16], FPGAs [12] and stem from both architectural choices and physical phenomena. To some extent, they can be mitigated by either increasing the order of the scheme or by "hardening" the implementation against effects that breach the independence of shares [31].

We evaluate the security order using the leakage detection methodology known as TVLA [13], which emphasizes detection over exploitation in order to speed-up the procedure. To make the evaluation feasible w.r.t. data complexity, we focus on the first round of our single-block 4-share implementation and employ the random vs. fixed Welch t-test, which uses random and fixed plaintexts acquired in a non-deterministic and randomly interleaved manner. Consecutively, we perform univariate t-tests of orders 1 through 4 using the incremental, one-pass formulas of Schneider and Moradi [34] at a level of significance $\alpha = 0.00001$. The results are plotted in Figure 2. Note that the number of samples per trace is fairly high due to the lengthy computation of the 4-share masked AES round and due to the high sampling rate dictated by the clock frequency (1 GHz) and the

Nyquist theorem. As a result, the t-test methodology faces the issue of multiple comparisons and we need to control the familywise error rate using the Šidák correction $\alpha_{SID} = 1 - (1 - \alpha)^{1/\#samples}$ [37]. The leakage detection threshold th is then computed using the formula $th = CDF^{-1}_{\mathcal{N}(0,1)}(1 - \alpha_{SID}/2)$, which equals to 6.25 when testing 25k samples per trace [44].



(a) 1st-order, 1M random vs. 1M fixed.

(b) 2nd-order, 1M random vs. 1M fixed.

(c) 3rd-order, 1M random vs. 1M fixed.

(d) 4th-order, 1M random vs. 1M fixed.

Fig. 2: Univariate leakage detection of orders 1 until 4.

In Figure 2 we observe that for orders 1 and 2, a 1M random vs. 1M fixed t-test does not reject the null hypothesis, thus no leakage is detected in the first two statistical moments. The situation is different for higher orders: both the 3rd and the 4th-order univariate t-tests are able to detect leakage. This demonstrates that the actual security order of the implementation is less than the theoretical one and detecting the presence of 3rd-order leakage is in fact easier than detecting 4th-order leakage. Interestingly, the experimental results are not in direct accordance with the order reduction suggested by [1], i.e., our 3rd-order (4-share) implementation achieves practical order of 2, while the theorized reduction suggests $\lfloor 3/2 \rfloor = 1$st-order security.

An additional way to approach the order reduction issue is to phrase it as a leakage certification problem [18,19]. The leakage certification procedure allows us to assess the quality of a leakage model w.r.t. estimation and assumption errors. Gauging the effect of estimation errors, i.e., those that arise from insufficient profiling, is straightforward and can be carried out via cross-validation tech-

niques [20]. Assumption errors are more difficult to assess, since they arise from incorrect modeling choices and would ideally require the comparison between the chosen model and an unknown perfect model. To tackle this, the indirect approach of Durvaux, Standaert and Veyrat-Charvillon [19] observes the relation between estimation and assumption errors and if the latter are negligible in comparison, they conclude that the chosen model is adequate.

In our approach, we use the t-test-based certification toolset of Durvaux, Standaert, and Del Pozo [18], which focuses on the assumption and estimation errors for each statistical moment. Initially, we start with an erroneous model for our 4-share implementation: we assume that the leakage is sufficiently captured by a Gaussian template, i.e., a normal distribution that is fully described by the first two statistical moments. The results are visible in the upper part of Figure 3, using a trace set of size 900,000. In particular, we plot the p-value of a t-test that compares an actual statistical moment (estimated from the trace set) with a simulated statistical moment (estimated by sampling the profiled model). A high p-value (i.e., a mostly white image) indicates that estimation errors overwhelm assumption errors and that the chosen model is adequate. A small p-value indicates that assumption errors are larger than estimation errors, thus the chosen model is erroneous. The process is repeated for all first four statistical moments (mean, variance, skewness, kurtosis) using cross-validation.

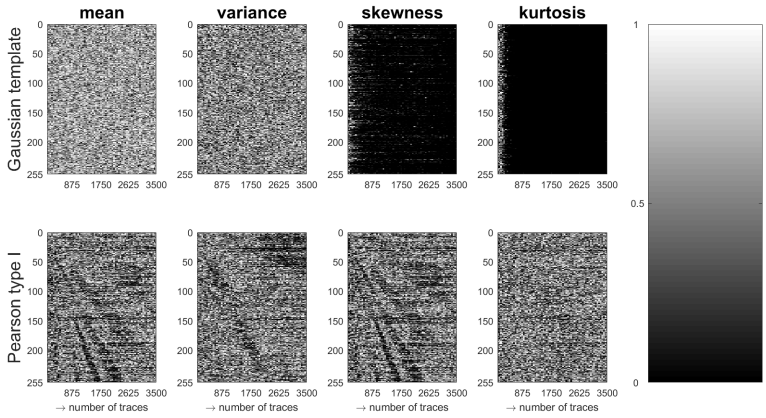


Fig. 3: Leakage certification p-values for Gaussian templates and Pearson type I.

In the first two images of Figure 3 (upper part, mean and variance), the high p-values indicate that these moments are well-captured by the model. Naturally, the fourth image (upper part, kurtosis) is black, indicating that the model disregards the 4th moment of a parallel 4-share implementation which should (in theory) contain useful information. Interestingly, the third image (skewness) is also black, penalizing any model that does not include the 3rd statistical moment, although in a perfect scheme it should not convey any information. We continue this approach with a more adequate model for the 4-share implementation: we

assume that the leakage is captured by a Pearson type I distribution [35], i.e., a 4-moment Beta distribution. The results are visible in the lower part of Figure 3 and show that the assumption errors in the 3rd and 4th moments tend to be smaller than the corresponding estimation errors.

As demonstrated by both the t-test methodology and the leakage certification process, the NEON-based implementations on ARM Cortex-A8 are likely to be subject to order reduction and may require further hardening to prevent dependencies between shares. The potential causes of the order reduction remain unexplored since they may stem from bus/register/memory transitions, pipelined data processing or even electrical coupling effects. Pinpointing the origin of the security reduction remains an open problem in the side-channel field since it essentially requires the countermeasure designer to access/modify the hardware architecture and chip layout, a task that is not possible with proprietary designs.

4.3 Information-Theoretic Evaluation

Having investigated the security order of the single-block 4-share AES implementation, we turn to the evaluation of its 8-share counterpart. The core feature of a masking scheme is the noise amplification stage. Assuming sufficient noise, it has been shown that the number of traces required for a successful attack grows exponentially w.r.t. the order $d-1$ [11]. As a result, the evaluation of the proposed 8-share implementation can be beyond the measurement capability of most evaluators. To tackle this issue, we will rely on an information-theoretic approach used by Standaert et al. and Journault et al. [26,39,40], assisted by the bound-oriented works of Prouff and Rivain [32], Duc, Faust, and Standaert [17], and Grosso and Standaert [24].

Analytically, we start with an unprotected (single-share) AES implementation and estimate the device/setup signal to noise ratio (SNR). We define the random variable S to correspond to the sensitive (key-dependent) intermediate values that we try to recover. Likewise, we define the random variable L to correspond to the time sample that exhibits high leakage (heuristically chosen as the sample with the highest t-test value). Subsequently, we profile Gaussian templates for all sensitive values s that are instances of variable S. In other words, we estimate $\hat{Pr}[L|s]_{model} \sim \mathcal{N}(\hat{\mu}_s, \hat{\sigma}_s^2)$ for all s. Using the estimated moments, we compute the SNR as the ratio $\hat{var}_s(\hat{\mu}_s)/\hat{E}_s(\hat{\sigma}_s^2)$, resulting in SNR ≈ 0.004. We continue to compute the Hypothetical Information (HI) which shows the amount of information leaked if the leakage is adequately represented by the estimated model $\hat{Pr}_{model}$.

$$\mathrm{HI}(S;L) = H[S] + \sum_{s\in\mathcal{S}} Pr[s] \cdot \int_{l\in\mathcal{L}} \hat{Pr}_{model}[l|s] \cdot log_2 \hat{Pr}_{model}[s|l] \;\; dl,$$

$$\text{where } \hat{Pr}_{model}[s|l] = \frac{\hat{Pr}_{model}[l|s]}{\sum_{s^*\in\mathcal{S}} \hat{Pr}_{model}[l|s^*]}.$$

To simplify the evaluation process, we employ the independent shares' leakage assumption so as to extrapolate the information of a single share to the information of a d-tuple of shares. Thus, in order to obtain the HI bounds for security orders 3 and 7, we raise $\mathrm{HI}(S; L)$ to the security order. In addition, the evaluator should take special consideration w.r.t. horizontal exploitation [6,42], which can be particularly hazardous, e.g., in the context of lengthy masked multiplications. To showcase such a scenario, we employ the bound of Prouff and Rivain [32], stating that the multiplication leakage is roughly $1.72d + 2.72$ times the leakage of a d-tuple of shares. The results of the information-theoretic evaluation are visible in Figure 4.

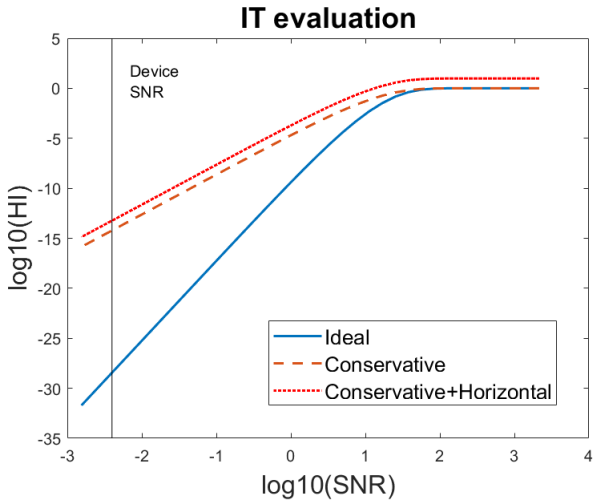


Fig. 4: Information-theoretic evaluation for the 8-share masked implementation.

Figure 4 assesses the performance of the proposed 8-share AES implementation, using information-theoretic bounds. The solid line shows the ideal masking performance, while the dashed line shows a conservative masking performance due to order reduction from order 7 to order 3. Last, the dotted line demonstrates the scenario where the adversary exploits the order-reduced (conservative) version in a horizontal fashion, i.e., (s)he incorporates all intermediate values computed during a masked AES multiplication. For the current SNR of the device, the measurement complexity is bounded by approximately 2^{91} measurements (ideal case), 2^{45} (conservative case) and 2^{42} (conservative horizontal case) [17].

5 Conclusion and Outlook

We have shown how higher-order masking of AES can be sped up using NEON vector registers. With a good randomness source, such implementations are very fast and practical. We also performed a side-channel evaluation to study the

security order of the single-block 4-share implementation and an information-theoretic analysis to bound the measurement complexity w.r.t. the 8-share implementation.

Future SCA work can delve deeper into order-reduction issues, in conjunction with multivariate and horizontal exploitation. For instance, with our high-order univariate methodology, it is implicitly assumed that all the shares are manipulated in parallel. While this appears to hold when looking at the NEON assembly specifications, full parallelism may not be enforced on a hardware level. A deeper inspection of the circuitry could potentially clarify the actual parallelism and identify the underlying issues behind order reduction. Moving towards multivariate exploitation, practical horizontal attacks such as soft-analytical attacks need to be carried out such that we can gauge in practice the detrimental effects of lengthy leaky computations and establish a fairer evaluation procedure.

References

1. Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. On the cost of lazy engineering for masked software implementations. In Marc Joye and Amir Moradi, editors, *Smart Card Research and Advanced Applications — CARDIS 2014*, volume 8968 of *LNCS*, pages 64–81. Springer, 2014. http://eprint.iacr.org/2014/413.pdf.
2. Josep Balasch, Benedikt Gierlichs, Oscar Reparaz, and Ingrid Verbauwhede. DPA, bitslicing and masking at 1 GHz. In Tim Güneysu and Helena Handschuh, editors, *Cryptographic Hardware and Embedded Systems — CHES 2015*, volume 9293 of *LNCS*, pages 599–619. Springer, 2015. http://eprint.iacr.org/2015/727.pdf.
3. Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong non-interference and type-directed higher-order masking. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 116–129. ACM, 2016. http://eprint.iacr.org/2015/506.pdf.
4. Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. Verified proofs of higher-order masking. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology — EUROCRYPT 2015*, volume 9056 of *LNCS*, pages 457–485. Springer, 2015. https://eprint.iacr.org/2015/060.pdf.
5. Gilles Barthe, François Dupressoir, Sebastian Faust, Benjamin Grégoire, François-Xavier Standaert, and Pierre-Yves Strub. Parallel implementations of masking schemes and the bounded moment leakage model. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology — EUROCRYPT 2017*, volume 10210 of *LNCS*, pages 535–566. Springer, 2017. http://eprint.iacr.org/2016/912.pdf.
6. Albertoa Battistello, Jean-Sébastien Coron, Emmanuel Prouff, and Rina Zeitoun. Horizontal side-channel attacks and countermeasures on the ISW masking scheme. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems — CHES 2016*, volume 9813 of *LNCS*, pages 23–39. Springer, 2016. https://eprint.iacr.org/2016/540.pdf.
7. Daniel J. Bernstein. Cache-timing attacks on AES. https://cr.yp.to/antiforgery/cachetiming-20050414.pdf, 2005.

8. Daniel J. Bernstein and Peter Schwabe. NEON crypto. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems — CHES 2012*, volume 7428 of *LNCS*, pages 320–339. Springer, 2012. https://cr.yp.to/highspeed/neoncrypto-20120320.pdf.
9. Joan Boyar and René Peralta. A new combinational logic minimization technique with applications to cryptology. In Paola Festa, editor, *Experimental Algorithms*, volume 6049 of *LNCS*, pages 178–189. Springer, 2010. http://eprint.iacr.org/2009/191.pdf.
10. D. Canright and Lejla Batina. A very compact "perfectly masked" S-box for AES. In Steven M. Bellovin, Rosario Gennaro, Angelos Keromytis, and Moti Yung, editors, *Applied Cryptography and Network Security — ACNS 2008*, volume 5037 of *LNCS*, pages 446–459. Springer, 2008. https://eprint.iacr.org/2009/011.pdf.
11. Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO '99*, volume 1666 of *LNCS*, pages 398–412. Springer, 1999. http://dx.doi.org/10.1007/3-540-48405-1_26.
12. Thomas De Cnudde, Begül Bilgin, Benedikt Gierlichs, Ventzislav Nikov, Svetla Nikova, and Vincent Rijmen. Does coupling affect the security of masked implementations? In Sylvain Guilley, editor, *Constructive Side-Channel Analysis and Secure Design — COSADE 2017*, volume 10348 of *LNCS*, pages 1–18. Springer, 2017. http://eprint.iacr.org/2016/1080.pdf.
13. Jeremy Cooper, Elke DeMulder, Gilbert Goodwill, Joshua Jaffe, Gary Kenworthy, and Pankaj Rohatgi. Test vector leakage assessment (TVLA) methodology in practice, 2013. http://icmc-2013.org/wp/wp-content/uploads/2013/09/goodwillkenworthtestvector.pdf.
14. Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain, and Thomas Roche. Higher-order side channel security and mask refreshing. In Shiho Moriai, editor, *Fast Software Encryption — FSE 2013*, volume 8424 of *LNCS*, pages 410–424. Springer, 2014. https://eprint.iacr.org/2015/359.pdf.
15. Joan Daemen and Vincent Rijmen. *The design of Rijndael — AES, The Advanced Encryption Standard*. Springer, 2002. http://jda.noekeon.org/JDA_VRI_Rijndael_2002.pdf.
16. Wouter de Groot, Kostas Papagiannopoulos, Antonio de la Piedra, Erik Schneider, and Lejla Batina. Bitsliced masking and ARM: friends or foes? In *Lightweight Cryptography for Security and Privacy — LightSec 2016*, volume 10098 of *LNCS*, pages 91–109. Springer, 2016. https://eprint.iacr.org/2016/946.pdf.
17. Alexandre Duc, Sebastian Faust, and François-Xavier Standaert. Making masking security proofs concrete — Or how to evaluate the security of any leaking device. In *Advances in Cryptology — EUROCRYPT 2015*, volume 9056 of *LNCS*, pages 401–429. Springer, 2015. https://eprint.iacr.org/2015/119.pdf.
18. François Durvaux, François-Xavier Standaert, and Santos Merino Del Pozo. Towards easy leakage certification. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems — CHES 2016*, volume 9813 of *LNCS*, pages 40–60. Springer, 2016. https://eprint.iacr.org/2015/537.pdf.
19. François Durvaux, François-Xavier Standaert, and Nicolas Veyrat-Charvillon. How to certify the leakage of a chip? In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology — EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 459–476. Springer, 2014. http://eprint.iacr.org/2013/706.pdf.
20. Bradley Efron and Robert J. Tibshirani. *An Introduction to the Bootstrap*. CRC press, 1994.

21. Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. In Çetin K. Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2001*, volume 2162 of *LNCS*, pages 251–261. Springer, 2001. http://dx.doi.org/10.1007/3-540-44709-1_21.
22. Louis Goubin and Jacques Patarin. DES and differential power analysis — The "duplication" method. In Çetin K. Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES'99*, volume 1717 of *LNCS*, pages 158–172. Springer, 1999. http://www.goubin.fr/papers/dpafinal.pdf.
23. Dahmun Goudarzi and Matthieu Rivain. How fast can higher-order masking be in software? In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology — EUROCRYPT 2017*, volume 10210 of *LNCS*, pages 567–597. Springer, 2017. https://eprint.iacr.org/2016/264.pdf.
24. Vincent Grosso and François-Xavier Standaert. Masking proofs are tight (and how to exploit it in security evaluations). Cryptology ePrint Archive, Report 2017/116, 2017. http://eprint.iacr.org/2017/116.
25. Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *Advances in Cryptology — CRYPTO 2003*, volume 2729 of *LNCS*, pages 463–481. Springer, 2003. http://iacr.org/archive/crypto2003/27290462/27290462.pdf.
26. Anthony Journault and François-Xavier Standaert. Very high order masking: Efficient implementation and security evaluation. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems — CHES 2017*, volume 10529 of *LNCS*, pages 623–643. Springer, 2017. https://eprint.iacr.org/2017/637.pdf.
27. Emilia Käsper and Peter Schwabe. Faster and timing-attack resistant AES-GCM. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems — CHES 2009*, volume 5747 of *LNCS*, pages 1–17. Springer, 2009. http://cryptojedi.org/users/peter/#aesbs.
28. HeeSeok Kim, Seokhie Hong, and Jongin Lim. A fast and provably secure higher-order masking of AES S-box. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems — CHES 2011*, volume 6917 of *LNCS*, pages 95–107. Springer, 2011. http://dx.doi.org/10.1007/978-3-642-23951-9_7.
29. Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO '99*, volume 1666 of *LNCS*, pages 388–397. Springer, 1999. http://dx.doi.org/10.1007/3-540-48405-1_25.
30. Robert Könighofer. A fast and cache-timing resistant implementation of the AES. In Tal Malkin, editor, *Topics in Cryptology — CT-RSA 2008*, volume 4964 of *LNCS*, pages 187–202. Springer, 2008. https://doi.org/10.1007/978-3-540-79263-5_12.
31. Kostas Papagiannopoulos and Nikita Veshchikov. Mind the gap: Towards secure 1st-order masking in software. In Sylvain Guilley, editor, *Constructive Side-Channel Analysis and Secure Design — COSADE 2017*, volume 10348 of *LNCS*, pages 282–297. Springer, 2017. http://eprint.iacr.org/2017/345.pdf.
32. Emmanuel Prouff and Matthieu Rivain. Masking against side-channel attacks: A formal security proof. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology — EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 142–159. Springer, 2013. http://www.iacr.org/archive/eurocrypt2013/78810139/78810139.pdf.

33. Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of AES. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems — CHES 2010*, volume 6225 of *LNCS*, pages 413–427. Springer, 2010. https://eprint.iacr.org/2010/441.pdf.

34. Tobias Schneider and Amir Moradi. Leakage assessment methodology — A clear roadmap for side-channel evaluations. In *Cryptographic Hardware and Embedded Systems — CHES 2015*, volume 9293 of *LNCS*, pages 495–513. Springer, 2015. http://eprint.iacr.org/2015/207.pdf.

35. Tobias Schneider, Amir Moradi, François-Xavier Standaert, and Tim Güneysu. Bridging the gap: Advanced tools for side-channel leakage estimation beyond Gaussian templates and histograms. In Roberto Avanzi and Howard Heys, editors, *Selected Areas in Cryptography — SAC 2016*, volume 10532 of *LNCS*, pages 58–78. Springer, 2017. https://eprint.iacr.org/2016/719.pdf.

36. Peter Schwabe and Ko Stoffelen. All the AES you need on Cortex-M3 and M4. In *Selected Areas in Cryptography — SAC 2016*, LNCS. Springer, 2016. https://eprint.iacr.org/2016/714.pdf.

37. Zbynek Sidak. Rectangular confidence regions for the means of multivariate normal distributions. *Journal of the American Statistical Association*, 62(318):626–633, 1967.

38. Etienne Sobole. Cycle counter for Cortex-A8. http://pulsar.webshaker.net/ccc/index.php?lng=us. Retrieved June 2017.

39. François-Xavier Standaert, Tal Malkin, and Moti Yung. A unified framework for the analysis of side-channel key recovery attacks. In *Advances in Cryptology — EUROCRYPT 2009*, volume 5479 of *LNCS*, pages 443–461. Springer, 2009. http://www.iacr.org/archive/eurocrypt2009/54790443/54790443.pdf.

40. François-Xavier Standaert, Nicolas Veyrat-Charvillon, Elisabeth Oswald, Benedikt Gierlichs, Marcel Medwed, Markus Kasper, and Stefan Mangard. The world is not enough: Another look on second-order DPA. In Masayuki Abe, editor, *Advances in Cryptology — ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 112–129. Springer, 2010. http://www.iacr.org/archive/asiacrypt2010/6477112/6477112.pdf.

41. Jasper G. J. van Woudenberg, Marc F. Witteman, and Bram Bakker. Improving differential power analysis by elastic alignment. In Aggelos Kiayias, editor, *Topics in Cryptology — CT-RSA 2011*, LNCS, pages 104–119. Springer, 2011. https://doi.org/10.1007/978-3-642-19074-2_8.

42. Nicolas Veyrat-Charvillon, Benoît Gérard, and François-Xavier Standaert. Soft analytical side-channel attacks. In *Advances in Cryptology — ASIACRYPT 2014*, volume 8873 of *LNCS*, pages 282–296. Springer, 2014. https://eprint.iacr.org/2014/410.pdf.

43. Junwei Wang, Praveen Kumar Vadnala, Johann Großschädl, and Qiuliang Xu. Higher-order masking in practice: A vector implementation of masked AES for ARM NEON. In Kaisa Nyberg, editor, *Topics in Cryptology — CT-RSA 2015*, volume 9048 of *LNCS*, pages 181–198. Springer, 2015. http://dx.doi.org/10.1007/978-3-319-16715-2_10.

44. Liwei Zhang, A. Adam Ding, Francois Durvaux, Francois-Xavier Standaert, and Yunsi Fei. Towards sound and optimal leakage detection procedure (extended version). In *Smart Card Research and Advanced Applications — CARDIS 2017*, LNCS. Springer, 2017. https://eprint.iacr.org/2017/287.

A NEON Implementations

A.1 Refreshing, 4 Shares

```
//param rand is r register with address of randomness
//param a is d register to refresh
//param tmp is d register that gets overwritten
.macro refresh rand a tmp
  vld1.64 {\tmp}, [\rand]! //get 8 bytes of randomness
  veor \a, \tmp
  vext.16 \tmp, \tmp, #1
  veor \a, \tmp
.endm
```

A.2 Refreshing, 8 Shares

```
//param rand is r register with address of randomness
//param a is q register to refresh
//param tmp is q register that gets overwritten
.macro refresh rand a tmp
  vld1.64 {\tmp}, [\rand:128]! //get 16 bytes of randomness
  veor \a, \tmp
  vext.16 \tmp, \tmp, #1
  veor \a, \tmp

  vld1.64 {\tmp}, [\rand:128]! //get 16 bytes of randomness
  veor \a, \tmp
  vext.16 \tmp, \tmp, #2
  veor \a, \tmp
.endm
```

A.3 Multiplication, 4 Shares

```
//param rand is r register with address of randomness
//param c is d register where result gets stored
//param a and b are d registers to and, remain unchanged
//param tmp and tmpr are d registers that get overwritten
.macro masked_and rand c a b tmp tmpr
  vand \c, \a, \b //z = x.y
  vld1.64 {\tmpr}, [\rand]! //get 8 bytes of randomness
  vext.16 \tmp, \b, \b, #1
  veor \c, \tmpr  // + r
  vand \tmp, \a
  veor \c, \tmp   // + x.(rot y 1)
  vext.16 \tmp, \a, \a, #1
  vand \tmp, \b
  veor \c, \tmp   // + (rot x 1).y
  vext.16 \tmpr, \tmpr, #1
  veor \c, \tmpr  // + (rot r 1)
  vext.16 \tmp, \b, \b, #2
```

```
  vand \tmp, \a
  veor \c, \tmp    // + x.(rot y 2)
  vld1.16 {\tmp[]}, [\rand]! //get 2 bytes of randomness
  veor \c, \tmp    // + (r4,r4,r4,r4)
.endm
```

A.4 Multiplication, 8 Shares

```
//param rand is r register with address of randomness
//param c is q register where result gets stored
//param a and b are q registers to and, remain unchanged
//param tmp and tmpr are q registers that get overwritten
.macro masked_and rand c a b tmp tmpr
  vand \c, \a, \b //K = A.B
  vld1.64 {\tmpr}, [\rand:128]! //get 16 bytes of randomness
  vext.16 \tmp, \b, \b, #1
  veor \c, \tmpr  // + R
  vand \tmp, \a
  veor \c, \tmp    // + A.(rot B 1)
  vext.16 \tmp, \a, \a, #1
  vand \tmp, \b
  veor \c, \tmp    // + (rot A 1).B
  vext.16 \tmpr, \tmpr, #1
  veor \c, \tmpr  // + (rot R 1)
  vext.16 \tmp, \b, \b, #2
  vand \tmp, \a
  veor \c, \tmp    // + A.(rot B 2)
  vext.16 \tmp, \a, \a, #2
  vand \tmp, \b
  veor \c, \tmp    // + (rot A 2).B
  vld1.64 {\tmpr}, [\rand:128]! //get 16 bytes of randomness
  vext.16 \tmp, \b, \b, #3
  veor \c, \tmpr  // + R'
  vand \tmp, \a
  veor \c, \tmp    // + A.(rot B 3)
  vext.16 \tmp, \a, \a, #3
  vand \tmp, \b
  veor \c, \tmp    // + (rot A 3).B
  vext.16 \tmpr, \tmpr, #1
  veor \c, \tmpr  // + (rot R' 1)
  vext.16 \tmp, \b, \b, #4
  vand \tmp, \a
  veor \c, \tmp    // + A.(rot B 4)
  vld1.64 {\tmpr}, [\rand:128]! //get 16 bytes of randomness
  veor \c, \tmpr  // + R''
  vext.16 \tmpr, \tmpr, #1
  veor \c, \tmpr  // + (rot R'' 1)
.endm
```