Masking the AES with Only Two Random Bits

Hannes Gross[1], Lauren De Meyer[2], Martin Krenn[1], and Stefan Mangard[1]

[1] Institute for Applied Information Processing and Communications (IAIK), Graz University of Technology, Inffeldgasse 16a, 8010 Graz, Austria
hannes.gross@iaik.tugraz.at
[2] KU Leuven, imec - COSIC, Belgium
lauren.demeyer@esat.kuleuven.be

Abstract. Masking is the best-researched countermeasure against side-channel analysis attacks. Even though masking was invented almost 20 years ago, research on the efficient implementation of masking continues to be an active research topic. Many of the existing works focus on the reduction of randomness requirements since the production of fresh random bits with high entropy is very costly in practice. Most of these works rely on the assumption that only so-called online randomness results in additional costs. In practice, however, it shows that the distinction between randomness costs to produce the initial masking and the randomness to maintain security during computation (online) is not meaningful. In this work, we thus study the question of minimum randomness requirements for first-order Boolean masking when taking the costs for initial randomness into account. We demonstrate that first-order masking can always be performed by just using two fresh random bits and without requiring online randomness. We first show that two random bits are enough to mask linear transformations and then discuss prerequisites under which nonlinear transformations are securely performed likewise. Subsequently, we introduce a new masked AND gate that fulfills these requirements which form the basis for our synthesis tool that automatically transforms an unmasked circuit into a first-order secure masked circuit. We demonstrate the feasibility of this approach by implementing an AES circuit with only two bits of randomness.

Keywords: masking · AES · first-order masking · hardware security · side-channel analysis

1 Introduction

Ever since the findings of Kocher et al. [24] and Quisquater [27] on differential power analysis and electromagnetic emanation analysis, respectively, the efficient protection against so-called side-channel analysis attacks has been eagerly studied. Masking has been proven over the years to be a countermeasure with high and formally well-understood security guarantees [23, 17] as well as good scalability [13]. Despite its popularity, the research on more efficient approaches to mask security-critical implementations does not seem to come to an end soon [6, 19, 20, 22, 26, 28].

One drawback of masking, however, is its implementation costs not least because of its high demand for fresh randomness. Since the creation of large amounts of fresh randomness requires additional time, chip area, energy, et cetera., a lot of research has been spent on more randomness efficient masking [3–5, 9, 18, 21, 22]. Most of the existing work, however, focuses on the randomness optimization for specific masking gadgets, like masked AND gates, and do not consider the minimization of the overall randomness costs. Even more of the masking implementation papers only consider the so-called online randomness costs spent on producing fresh randomness to secure the computation once the initial sharing of the input data, e.g., plaintext ciphertext or data and key material, has been performed. There is, to the best of our knowledge, no paper that considers the minimization of randomness costs when taking the masking of the input data into account or tries to minimize the overall randomness costs. Furthermore, no work states a lower bound for the minimal randomness costs for masking apart from specific masked gates [5].

Our contribution. We start of this work in Section 2 by taking a step away from the modern sharing-based perspective of masking back to the classical Boolean masking perspective. From this masking point of view, we then demonstrate that first-order masking is theoretically possible by just using two random one-bit masks on the basis of linear transformations. We then discuss what properties need to be fulfilled such that this approach also works for masked nonlinear transformations and show that existing approaches of masked AND gates do not fulfill these criteria. As a first practical contribution, we design a masked AND gate that allows reusing randomness from its inputs safely.

Based on our findings we introduce in Section 3 a simple rule-based system. These rules can be encoded in SMT2 statements and then used to automatically check whether the masking approach is directly applicable to an unprotected circuit or if modifications (mask changes) are required. Upon acceptance, our tool synthesizes a securely masked circuit for a given set of additional constraints like the used mask encoding.

We then show how our approach can be applied on larger circuits (Section 4) and demonstrate its feasibility and its impact on software and hardware on a full AES-128 encryption-only implementation in Section 5. It shows that our approach has a comparable hardware footprint (when neglecting pipelining registers) than state-of-the-art AES S-box designs our approach generates more latency. Nevertheless, with our approach, we successfully designed the first formally verified AES S-box design that requires only two random bits for the initial sharing of its inputs and requires no online randomness to achieve first-order security in the probing model. Even when going for a full AES implementation the randomness requirements do not increase further. However, since existing formal tools are not yet efficient enough to digest a full round unrolled AES implementation, we instead verified each building block of our design using the maskVerif tool of Barthe et al. [2] for a predefined mask encoding of its inputs and outputs. Ensuring the same masks encoding for each input and output al-

lows us to argue the security when putting the components together in the full AES implementation. Details on the formal verification are given in Section 6.

2 Masking without Online Randomness

The goal of masking is to make the power consumption (and other side-channels that depend on the power consumption) independent from security-sensitive information. For this purpose, the security-sensitive information is first combined with a uniformly random sampled data in an invertible masking function, such that the representation of the data itself becomes uniformly random distributed. In the case of Boolean masking, the sensitive information s, for instance, is combined with a random mask m by using the Boolean exclusive-or (XOR) operation. The resulting masking value $s_0 = s \oplus m_0$ thus becomes statistically independent from s, i.e., the mutual information between s and s_0 becomes zero. For this reason, any computation on s_0 trivially results in power consumption that is statistically independent of s as long as m_0 is not recombined with s_0. For convenience reasons, the security is often expressed in the so-called t-probing model [23] which assumes that an attacker can make up to t observations in the circuit (place up to t probe on the circuit). It has been verified in the past that this formal model accurately models the abilities of a differential side-channel analysis attacker that has access to noisy side-channel leakage traces [17]. We assume in the following a first-order attacker, i.e., an attacker that can place a single probe on the circuit.

Often in the present literature, the relation between the masked data s_0 and the mask(s) m_0 is expressed by using a sharing-based notation for convenience reasons. For first-order masking (i.e., only one mask is used to protect s) the information is assumed to be split into two shares (e.g., s_0 and s_1) such that again the additive relation $s = s_0 \oplus s_1$ is fulfilled. While it is trivial to convert from a masking representation to a sharing representation by setting $s_0 = s \oplus m$ and $s_1 = m$, the sharing representation inherently hides the relation between secret information and masks.

For brevity reasons, we use the sharing based representation in most parts of the paper. Since in the following we are sometimes particularly interested in the relation between secrets (or shares) and masks we often switch to the masking form. To make the used notation clearer, we always use the prefix m for masks followed by a number in the subscript. Any other variable name with a suffix subscript number denotes a specific share of the variable. Most of the time we just use 0 or 1 in the subscript (e.g., a_0 or a_1) to refer to the first or second share of a first-order masked variable a, respectively. Without any subscript notation we always refer to the plain secret variable (a, b, $q \ldots$).

Computation on masked data. To realize computations that are not only secure against side-channel analysis but also correct, the computed masked function needs to take the mask into account but without demasking the data. For example, when calculating the XOR of two masked sensitive variables as $q = a \oplus b$,

where a is shared in the two shares a_0 and a_1 and b is shared as b_0 and b_1, the correct and securely masked realization is very simple.

$$\begin{aligned} q_0 &= a_0 \oplus b_0 \\ q_1 &= a_1 \oplus b_1 \end{aligned} \tag{1}$$

When observing the masked representation of this equation with $a_0 = a \oplus m_0$, $a_1 = m_0$ and $b_0 = b \oplus m_1$, $b_1 = m_1$ the correctness can be easily observed when considering the addition of the shares of q, because both shares added together result in the desired operation in the sensitive variables a and b.

$$\begin{aligned} q &= q_0 \oplus q_1 \\ &= (a \oplus m_0) \oplus (b \oplus m_1) \oplus m_0 \oplus m_1 \\ &= a \oplus b \end{aligned}$$

To demonstrate the security of the masked realization of the XOR in Equation 1 it needs to be shown that each intermediate product is statistically independent of a and b which in this case are only the output shares q_0 and q_1. Statistical independence is given because we assume that each of the two masks m_0 and m_1 is uniformly random and statistically independent from each other. By looking on the truth table for both shares of q in Table 1, one can observe that when subdividing the truth table into the four possible combinations of values for a and b, the count of 1 appearances (Hamming weight) for q_0 and q_1 in each case are equal.

This equal distribution for each possible combination of secrets results in power consumption that is in average equal for all cases of a and b. We note that an attacker with the ability to probe more signals could observe differences by combining multiple probed signals. Higher-order leakages, however, are more difficult to exploit than the average power consumption (exponentially more observations are required [13]) and are not considered in this paper.

The situation changes when assuming that both masked variables use the same mask $m_0 = m_1$, which trivially reveals a and b in the equation of q_0.

$$q_0 = a_0 \oplus b_0 = (a \oplus m_0) \oplus (b \oplus m_0) = a \oplus b$$

Most state-of-the-art masking works assume that shares are produced using independent random masks which helps to avoid such situations. So when multiple XOR operations are chained together (e.g., $a \oplus b \oplus c \oplus \cdots \oplus z$) a lot of random masks are accumulated.

$$\begin{aligned} q_0 &= a_0 \oplus b_0 \oplus c_0 \oplus \cdots \oplus z_0 = (a \oplus m_0) \oplus (b \oplus m_1) \oplus (c \oplus m_2) \oplus \ldots (z \oplus m_{25}) \\ q_1 &= a_1 \oplus b_1 \oplus c_1 \oplus \cdots \oplus z_1 = m_0 \oplus m_1 \oplus m_2 \oplus \cdots \oplus m_{25} \end{aligned}$$

Please note that we assume here and in the remainder of the paper that the masked equations are evaluated from left to right, and parentheses indicate

Table 1. Truth table and security of the masked XOR from Equation 1

Shares				*Secrets*			*Hamming weights*	
a_0	a_1	b_0	b_1	a	b	$a \oplus b$	q_0	q_1
0	0	0	0	**0**	**0**	**0**	**2**	**2**
0	0	1	1					
1	1	0	0					
1	1	1	1					
0	0	0	1	**0**	**1**	**1**	**2**	**2**
0	0	1	0					
1	1	0	1					
1	1	1	0					
0	1	0	0	**1**	**0**	**1**	**2**	**2**
0	1	1	1					
1	0	0	0					
1	0	1	1					
0	1	0	1	**1**	**1**	**0**	**2**	**2**
0	1	1	0					
1	0	0	1					
1	0	1	0					

atomic operations that do not produce further intermediate results (often to indicate the result of the evaluation of a sharing function or initial sharings). Our first and admittedly rather trivial observation is that the amount of accumulated randomness is unnecessarily high. To realize the same function in a secure and correct shared way, two random masks m_0 and m_1 can be simply alternately used in such a way that at no time any intermediate result is formed that consists without a mask. One possible realization is thus that to use m_0 to mask a and use m_1 for the remaining variables.

$$\begin{aligned} q_0 &= a \oplus m_0 \oplus b \oplus m_1 \oplus c \oplus m_1 \oplus \ldots z \oplus m_1 \\ &= (a \oplus b \oplus c \oplus \cdots \oplus z \oplus m_0 \oplus m_1) \\ q_1 &= m_0 \oplus m_1 \oplus m_1 \oplus \cdots \oplus m_1 \\ &= (m_0 \oplus m_1) \end{aligned}$$

There exist also many other possible and secure realizations for this function. Depending on the order of the mask appliance, the resulting mask can be either m_0 or m_1 or their combination $m0 \oplus m_1$. With these findings, we can secure any linear function likewise. However, there is no straight-forward extension to nonlinear functions.

Application to nonlinear gates. There exists a vast variety of first-order masked AND gates in the literature which form the simplest class of nonlinear functions and are used to form more complex functions. These realizations of

masked AND gates usually vary regarding online randomness requirements and the numbers of used input and output shares. The underlying functionality is of course always the same and in the case of a realization with two shares requires the secure evaluation of the following four multiplication terms.

$$q = a \wedge b = (a_0 \oplus a_1)(b_0 \oplus b_1) = a_0 \wedge b_0 \oplus a_0 \wedge b_1 \oplus a_1 \wedge b_0 \oplus a_1 \wedge b_1 \tag{2}$$

Any direct combination of either two multiplications terms (e.g., $a_0 b_0 \oplus a_0 b_1$) is insecure because it leads to a function that statistically depends on the secret a or b. Most of the existing masked AND gadgets thus use fresh random masks to realize the secure evaluation like m_2 is used in the following example.

$$\begin{aligned} q_0 &= a_0 \wedge b_0 \oplus m_2 \oplus a_0 \wedge b_1 \\ q_1 &= a_1 \wedge b_0 \oplus m_2 \oplus a_1 \wedge b_1 \end{aligned} \tag{3}$$

This masked AND gate is indeed secure as long as the order of execution is from left to right and the masks including the one used for sharing a and b are statistically independent and uniformly distributed. Another advantage of this realization is that it inherently refreshes the sharing which makes the result independent of a and b. Any linear or nonlinear combination of q with the sharing of a or b is thus still possible, as long as the transformation itself is secure under the assumption of independently shared inputs.

There also exist realizations of a masked AND gate that does not require any fresh randomness. As an example, we consider the following equations from Biryukov et al. [9].

$$\begin{aligned} q_0 &= a_0 \wedge b_0 \oplus (a_0 \vee \neg b_1) \\ q_1 &= a_1 \wedge b_0 \oplus (a_1 \vee \neg b_1) \end{aligned} \tag{4}$$

A closer look on the properties of this realization from Biryukov et al. in Table 2 reveals that while the masking itself is secure, a further (linear) combination with shares or combinations of shares from a and b (except for a_0 and a_1) can make the sharing insecure again. Because this masked AND gate is insensitive to combinations with a single share from a masked by m_0 (cf. column $q_0 \oplus a_0$ in Table 2), one could assume that q is similarly protected as an XOR gate protected by the mask of m_1 of b. The problem is that this masked AND gate behaves entirely different than the masked XOR gate from Equation 1 or the masked AND from Equation 3. For the output of a masked XOR gate where $q_0 = a \oplus b \oplus m_1$, we would assume that an XOR with m_0 followed by the addition of m_1 would result in a secure sharing masked by m_0, since $(a \oplus b \oplus m_1) \oplus m_0 \oplus m_1$ results in $a \oplus b \oplus m_0$. However, for the output q_0 of the masked AND gate, the combination of with m_0 followed by m_1 results in an insecure sharing (different Hamming weights for different cases of b). Chaining of masked AND operations by carefully selecting (or changing) between two different masks is thus not possible with this masked AND gate.

Table 2. Truth table for q_0 of Biryukov et al.'s masked AND (or for q_1 if a_0 is replaced by a_1)

Shares			*Secrets*	*Hamming weights*		
a_0	b_0	b_1	b	q_0	$q_0 \oplus a_0$	$(q_0 \oplus a_0) \oplus b_0$
0	0	0	**0**	**2**	**2**	**2**
0	1	1				
1	0	0				
1	1	1				
0	0	1	**1**	**2**	**2**	**0**
0	1	0				
1	0	1				
1	1	0				

Construction of a new masked AND. The design idea to ensure that the resulting sharing behaves similarly to the masked XOR gate is to first securely combine all multiplication terms (Equation 2) in a single share of q and then add a single mask. For this reason, we first transform the secure equations of Biryukov et al. such that we can directly observe what happens to the multiplication terms.

$$\begin{aligned} q_0 &= a_0 \wedge b_0 \oplus (a_0 \vee \neg b_1) \\ &= a_0 \wedge b_0 \oplus \neg(\neg a_0 \wedge b_1) \\ &= a_0 \wedge b_0 \oplus (a_0 \wedge b_1 \oplus b_1) \oplus 1 \\ q_1 &= a_1 \wedge b_0 \oplus (a_1 \vee \neg b_1) \\ &= a_1 \wedge b_0 \oplus \neg(\neg a_1 \wedge b_1) \\ &= a_1 \wedge b_0 \oplus (a_1 \wedge b_1 \oplus b_1) \oplus 1 \end{aligned}$$

From this transformation, we can see that the terms $a_0 \wedge b_0 \oplus (a_0 \wedge b_1 \oplus b_1)$ from q_0 and $a_1 \wedge b_0 \oplus (a_1 \wedge b_1 \oplus b_1)$ from q_1, considered separately are securely masked by b_1 ($= m_1$, in the masking representation). Adding them directly together, however, is insecure because this results in $a \wedge b$. By adding q_1 ($= m_0$) to the second term, both terms can be added without leaking information. The result is only masked with a single mask from m_0. To achieve correctness the second share is set to m_0 or a_1, respectively.

$$\begin{aligned} q_0 &= \overbrace{(\underbrace{a_0 \wedge b_0}_{t1} \oplus \underbrace{(a_0 \wedge b_1 \oplus b_1)}_{t2})}^{t3} \oplus \underbrace{(\overbrace{(\underbrace{a_1 \wedge b_0}_{t4} \oplus \underbrace{(a_1 \wedge b_1 \oplus b_1)}_{t5})}^{t6} \oplus a_1)}_{t7} \\ &= (a \wedge b) \oplus m_0 \\ q_1 &= a_1 = m_0 \end{aligned} \tag{5}$$

Table 3. Security of the masked AND from Equation 5

Shares				*Secrets*			*Hamming weights*							
a_0	a_1	b_0	b_1	a	b	$a \wedge b$	$t1$	$t2$	$t3$	$t4$	$t5$	$t6$	$t7$	q_0
0	0	0	0											
0	0	1	1	**0**	**0**	**0**	**1**	**1**	**1**	**1**	**2**	**2**	**2**	**2**
1	1	0	0											
1	1	1	1											
0	0	0	1											
0	0	1	0	**0**	**1**	**0**	**1**	**1**	**1**	**1**	**2**	**2**	**2**	**2**
1	1	0	1											
1	1	1	0											
0	1	0	0											
0	1	1	1	**1**	**0**	**0**	**1**	**1**	**1**	**1**	**2**	**2**	**2**	**2**
1	0	0	0											
1	0	1	1											
0	1	0	1											
0	1	1	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**2**	**2**	**2**	**2**
1	0	0	1											
1	0	1	0											

The security of the masked AND gate can be easily verified by hand as shown in Table 3 where t values denote intermediate results. Therefore, we again denote all possible input share combinations in a truth table and sort them by the unshared secrets a and b. For each possible intermediate ($t1$ to $t7$, and q) we sum up the number of ones in the intermediates corresponding to the same value for a and b. Since for all intermediate value combinations, the number of ones is equal; a first-order attacker does not gain any sensitive information by probing either one of the intermediates. In addition to the manual inspection of the masked AND gate, we also performed a formal verification by using the tools from Bloem et al. [10] and Barthe et al. [2] which gave us the same results.

By combining the findings for the XOR and the AND gates we can mask arbitrary circuits, and as we will show in the next section, also derive simple rules to synthesize securely masked circuits from unprotected ones.

3 Synthesis of First-Order Secure Circuits

Manually tracking the masks as they propagate through the circuits soon becomes a very complex task as the circuit size increases. We thus decided on creating an automated approach to create a masked circuit when possible, or otherwise, indicate which signals need to be changed. As a first step, the tool reads the description of a Boolean circuit in static single assignment (SSA) form in Verilog syntax such that each instruction is either a one-bit signal assignment or a two-bit XOR, XNOR or AND gate. The Boolean circuit is then represented

as an SMT problem which is fed to the Z3 [16] theorem prover The Z3 searches for a possible solution for the mask encoding of the input signals so that for each gate the inputs have different masks. Furthermore, it allows ensuring the desired input and output signal mask encoding. A more detailed description of how the circuit is encoded in SMT2 in which steps are necessary is given in the following.

Input mask encoding. Each circuit is fed with two masks m_0 and m_1. As a result there are three possible mask combinations for each signal and thus three possible encodings for the input signals: $1 = m_0$, $2 = m_1$, $3 = m_0 \oplus m_1$. Depending on whether we target a specific encoding or we let the theorem prover decide on the encoding we adjust the assertions accordingly. With the following SMT2 code snippet the input signal a is mapped to either of the three masking combinations.

```
; Input encoding definition and constraints
(declare-const a Int)
(assert (> a 0))
(assert (< a 4))
```

The same rules are also applied to every output of a gate to restrict the mask encoding two the three possibilities.

Gate connections and encoding rules. For each of the four possible instruction classes (assignment, XOR, XNOR, and AND) of the SSA encoded input file we create specific rules according to which masks appear on the output q for the given input combination. In the example encoding it is always assumed that the signals a and optionally b form the operands. The encoding of the signal assignment $q = a$ just results in a copy of the mask encoding in the SMT2 rules.

```
# Signal assignment rule
(assert (= q a))
```

To encode the output of the XOR (and XNOR) instructions we utilize the fact that for different input encoding for a and b the third encoding is always calculated for the output.

$$\begin{aligned} m_0 \oplus m_1 &= (m_0 \oplus m_1) \\ m_0 \oplus (m_0 \oplus m_1) &= m_1 \\ m_1 \oplus (m_0 \oplus m_1) &= m_0 \\ &\dots \end{aligned}$$

When neglecting the case that both inputs could have the same mask encoding, which is covered by the safety rules for the gates in the next step which ensure the security of the circuit, the following SMT2 encoding can be used.

```
; XOR/XNOR gate rule
(assert (not (or (= q a) (= q b))))
```

Note that the negation of the output in case of the XNOR has no influence on the encoding because it is a simple addition of a constant value 1.

Finally, for the AND gate, the mask encoding can be the same as either operand since the operands can be simply swapped. We thus let the theorem prover decide which signal is used as the first operand which defines the mask encoding of the output (see Equation 5). The information which masks appear on the output is later on taken into account when the masked circuit is created to decide on the first operand.

```
; AND gate rule
(assert (or (= q a) (= q b)))
```

Again the AND gate rule does no cover the cases of both operands having the same mask encoding.

Safety rules. For each two-input gate we additionally define that both operands are required to have a different mask encoding which otherwise would create a flaw in the masked circuit.

```
; Safety rule or two input gates
(assert (not (= a b)))
```

Output constraints (optional). To make the design and verification of separate modules easier we decide on using the same input and output mask encoding on byte-level for all our modules. We can restrict the output encoding by setting the input and output signals equal.

```
; Equal input and output byte-encoding
(assert (= o0 i0)) ;
(assert (= o1 i1)) ;
(assert (= o2 i2)) ;
...
(assert (= o7 i7)) ;
```

Checking of the model and creating the masked circuit. When the Z3 theorem solver finds a secure model that fulfills our constraints the tool receives a model containing all mask assignments. The translation of the unprotected circuit to a secure masked circuit is then rather straightforward. At first, we duplicate all input and output ports of the module and additionally add the two masks m_0 and m_1 as input signals. For each instruction of the SSA input file we replace the original code by its masked variant according to the masked gates introduced in Section 2. As a further optimization, the second share of each instruction is (optionally) replaced by the resulting mask of the output signal which helps to save unnecessary instructions that would result in one of the three mask encodings anyway. We do not give a more detailed description of the tool at this point since the rest of the functionality follows from the description of the masked gates above and is mostly engineering work.

4 Masking the AES

To demonstrate the practicality of our approach, we decided on targeting the AES-128 (encryption-only) variant as an example. Since none of the existing formal verification tools are yet powerful enough to verify a full AES encryption, we decide on using a module wise implementation and verification approach. To justify the security of the overall design when bringing the modules together, we restricted the mask encoding for each input and output byte of every function to be equal. The starting point of our iterative design approach is the S-box construction used in the SubBytes transformation of the AES.

SubBytes. The most complicated part of the AES is its S-box layer which is implemented as 16 instances of SubByte modules. Most of the masked AES designs published over the last years are based on the S-box construction of Canright [12]. A more suitable design for our approach, however, is the design of Boyar and Peralta [11] which targets on minimizing the logic depth and is already constructed in SSA form. The original code of the forward S-box consists of 128 SSA instructions. In total there are 34 AND, 90 XOR, and 4 XNOR instructions for the unmasked circuit. Each instruction takes two one-bit variables as input.

After running our synthesis tool without any further optimizations of the resulting circuit, the masked design consisted of 136 AND gates, 353 XOR gates, and 4 NOT gates (because XNORs are decomposed to one XOR followed by a not gate in Yosys' ILANG). The 136 AND gates result from the fact that the masked AND quadruples the number of AND gates compared to the unmasked design. Also, each masked AND gate introduces 6 XOR gates which in total results in 204 additional XOR gates. The masking of the XOR and XNOR gates, on the other hand, do not introduces additional circuitry sine we already replaced the second share of each masked XOR output with its mask. The majority of the remaining 55 XOR gates result from the fact that at 38 occasions we use the combination of $m_0 \oplus m_1$ which are mapped to 38 individual gates. The rest of the XOR gates are required because at some points we needed to change the masking of a signal by introducing additional XOR instructions to receive a satisfiable Z3 model and thus a securely synthesizable circuit, and to ensure that the input and output mask encoding is equal.

After running an optimization pass in Yosys which only maps gates which implement the same function to a single gate, the number of gates could be reduced to 97 AND gates, 284 XOR gates and 4 NOT gates. We rerun the verification after this optimization to ensure that the circuit remains secure. The total overhead for the masking of the S-box is thus about a factor 2.92 regarding instructions.

From the design of the S-box, we also gathered the byte encoding that is used for the rest of the AES modules to ensure security and correctness of the code. The byte encoding is set to be $\{2, 3, 3, 1, 1, 2, 1, 2\}$ in the SMT encoding which corresponds to $\{m_1, m_0 \oplus m_1, m_0 \oplus m_1, m_0, m_0, m_1, m_0, m_1\}$ in the mask encoding for the inputs from $i0$ to $i7$.

ShiftRows. Since the ShiftRows transformation only changes the order of the bytes in the state rows, no special modifications are required for the round transformation compared to an unprotected design. Furthermore, all of the masked AES modules (in the final design) do not explicitly carry the second share of each signal but instead just assume the byte mask encoding as used by the S-box for inputs and outputs as the second share. The ShiftRows transformation thus just consists of rewiring (or readdressing) of the state byte and can be implicitly combined with the MixColumns transformation without adding any overhead.

MixColumns. For the MixColumns module, we again used the synthesis tool for converting the unprotected circuit of a single-column transformation (MixColumn) to a masked circuit, and also to ensure the byte mask encoding of the inputs and outputs. One MixColumn module operates on a four-byte state column. The unprotected module requires 140 XOR gates. Because at some point remasking of some signals is required to ensure security, the number of XOR gates increases to 176 in the final design. However, compared to other first-order masked designs this number quite low because for sharing based designs one would expect that the number of linear gates scale with the number of shares. One would thus expect 280 XOR gates for first-order protection. Note that this saving is only possible because at no point we carry secret variables in the second share of any signal.

AddRoundKey. In AddRoundKey the state or the plaintext is added byte-wise to the round key (AddByte module). Since we enforce the same byte encoding for all bytes in our design, the key byte first needs to be remasked before it can be added to the state or plaintext byte and the result the again require remasking. Instead of 8 XOR gates as in the unprotected design, we thus require 24 XOR gates. The number of XOR gates could be reduced to only eight again if the state and key bytes are shared such that their sum again results in the assumed byte sharing for the S-box. Due to the lack of the possibility to formally verify the whole circuit, we decide on keeping the byte encoding for both key and state bytes the same. This makes arguing of the security of the overall implementation for individually verified modules easier because there are fewer issues to oversee.

Key schedule. In the calculation of the round keys, we reused the masked SubByte and AddByte modules from the round transformation of the state. For the SubWord transformation, the same masked S-box design as for the state is used since the key has the same byte encoding, and also for the addition of the state columns, the same AddByte function as for AddRoundKey is used. The remaining key schedule components are the addition of the round constant which does not require masking of the publicly known constants, and the RotWord transformation which is just rewiring or readdressing of the key bytes and can thus be combined with the SubWord transformation.

Table 4 summarizes the costs for the individual transformations that are required to implement the full AES. More implementation details are given in the next section for the full AES implementation.

Table 4. Summary of the costs for the AES transformations

Module	*Instances*	*AND*	*XOR*	*NOT*
SubBytes		1,552	4,544	64
▶ SubByte	4	97	284	4
ShiftRows		-	-	-
AddRoundKey		-	384	-
▶ AddByte	16	-	24	-
MixColumns		-	704	-
▶ MixColumn	4	-	176	-
Key schedule		388	1,528	16
▶ RotWord	1	-	-	-
▶ SubWord	1	388	1,136	16
▶ AddRcon	1	-	8	-
▶ AddWords	1	-	384	-

5 Implementation Results and Discussion

Table 5 provides an overview of the implementation costs for a full AES-128 encryption regarding required one-bit AND, XOR and NOT operations. Depending on whether pre-computed keys or an on-the-fly key scheduling is assumed the total amount of instructions is between 72 and 91.5 thousand single-bit instructions. Since these are the counts for one-bit instructions, the required number of instructions for different architecture sizes can be easily estimated for a bit-slice software implementation by dividing these numbers by the data width. Accordingly, on an 8-bit machine, the instruction count is between 9 and 11.4 thousand instruction, for a 16-bit machine between 4.5 and 5.7 thousand instruction, for a 32-bit machine between 2.3 and 2.9 thousand instructions, and for a 64-bit machine between 1.1 and 1.4 thousand instruction depending on which round key strategy is performed. We also note that these numbers should be only considered as rough estimates for comparison with the state-of-the-art. The actual number of instructions highly depends on the target platform specifics, e.g., to ensure no leakage is caused by register overwrites.

The most time is spent on the S-box calculations with 84.2% or 85.4%, respectively. The secure byte addition, as used for they AddRoundKey and the AddWords functions, consumes between 5.85% and 8.82% of the overall time. The remaining time is mostly spent on the MixColumns transformation with about 6.93% to 8.78%.

Comparison with masked software implementations. Unfortunately, there does not exist many openly available masked software implementations. There is, however, some great work by Schwabe et al. [29] which implemented various variants of the AES for a 32-bit ARM Cortex M3 and M4. The only masked variant, however, is an AES-128 CTR mode bit-sliced implementation on the

M4. This work also includes performance figures for the generation of random bits using an onboard random number generator (RNG).

Schwabe et al.'s implementation requires in average 7,423 cycles for the encryption of one plaintext block. 2,133 cycles of which are spent on generating the required 10.5 thousand random bits which are almost 29% percent of the overall instruction count. The remaining 5,290 cycles are required for the actual encryption.

For a fairer comparison, it has to be kept in mind that Schwabe et al.'s masked implementation calculates two AES encryptions in parallel. This is possible without a notable performance loss over a single encryption since it is a bit-sliced design and the registers fit in more data than would be required by one AES encryption. The performance numbers are then calculated over an average of 256 blocks. The same technique, however, is also applicable to our design but is not yet reflected by the pure count of instructions in our result tables for which we could about half these numbers. In the following, we consider both the original instruction count numbers and the corrected estimates for parallel execution of encryptions.

We also note that our instruction counts so far do not consider cycles spent on additional loading and storing data operations. We thus estimate this overhead by looking at the ratio between logic operations and the loading-and-storing operations for the S-box in [29] which adds about 58.6% overhead to the pure logic operations. Assuming the same overhead for the whole implementation gives an estimate of 4,601 cycles when not taking two parallel encryptions into account which is still only 62% of the 7,423 cycles of Schwabe et al.'s implementation. When correction our numbers to also reflect the two parallel encryptions, the cycle count changes to about 2,301 cycles per encryption which is then just 31% percent of Schwabe et al.'s implementation.

Please note that this is just a rough estimation and we do not claim that our approach automatically leads to faster software implementations than related work, for which a more detailed comparison on the same platform with the same assumptions would be required. Our main goal is to show the feasibility of our approach to lower the randomness costs and argue that this has the potential to pay-off in practice.

5.1 Discussion and Comparison for Hardware

For the side-channel analysis resistant implementation in hardware, security in the probing model with glitches needs to be ensured. Our approaches security critically depends on the correct order in which the signals are combined. For this reason, registers are required after every XOR gate. Registers are not required after the AND gates (see Equation 5) and also are in general not required for NOT gates since they do not provide more information than probing a single wire. To estimate the size of the implementation in hardware the UMC 90 nm from Faraday serves as the basis. Accordingly an AND gate corresponds to 1.25 gate equivalents (GE), and XOR gate to 2.5 GE, a NOT gate to 0.5 GE, and a D-FF with asynchronous reset to 4.5 GE.

Table 5. AES-128 Implementations results, parentheses indicate gate counts including the key schedule (cf. Table 4)

Module	*Instances*	*AND*	*XOR*	*NOT*
AES		15,520 (19,400)	56,000 (71,280)	640 (800)
PreRound	1	- (388)	384 (1,912)	- (16)
▶ AddRoundKey	1	-	384	-
Round	9	1,552 (1,940)	5,632 (7,160)	64 (80)
▶ SubBytes	1	1,552	4,544	64
▶ ShiftRows	1	-	-	-
▶ MixColumns	1	-	704	-
▶ AddRoundKey	1	-	384	-
LastRound	1	1,552	4,928	64
▶ SubBytes	1	1,552	4,544	64
▶ ShiftRows	1	-	-	-
▶ AddRoundKey	1	-	384	-

S-box. Without the registers, the size of a single S-box (consisting of 97 ANDs, 284 XORs, and 4 NOTs) would be 833.25 GE. Due to the required amount of registers (284 D-FFs) to resist glitches, the size increases to 2.1 kGE which is about the same size as of related work when neglecting pipe-lining registers (cf. Table 6). Our S-box implementation is, however, the only variant that requires no online randomness when using two shares. Furthermore, instead of requiring each input bit to be independently shared (which requires at least $8 \cdot (t-1)$ random bits, where t is the number of shares), our inputs are shared with only two random bits in total. The cost for the randomness reduction is paid by the increased amount of latency. The unmasked Boyar-Peralata S-box has a maximum logic depth of 16 and an AND depth of 4. Every masked AND gate requires four cycles to securely calculate the result. Accordingly, the total latency of the S-box is 12 cycles $(16-4)$ plus 16 $(4 \cdot 4)$ which in total amounts to 28 cycles.

Full AES estimates. For the overall AES implementation in hardware the implementation of a round-unrolled variant, even though the production of fresh random bits would not become the bottleneck in our design, is probably not the first choice. Even when considering a precomputation of the keys, the size would be more than 411 kGE. With an on-the-fly key scheduling the AES would have a total size of 524 kGE.

A bit more practical choice would be to implement one full round and perform iterative encryption over ten rounds. One full round requires 41.4 kGE or 52.6 kGE, respectively (depending on the key management).

The more common approach in the literature of masked AES implementations is to also calculate the SubBytes transform in a serialized way by using online one S-box instance. This reduces the costs for one full round transforma-

Table 6. AES S-box results of related hardware implementations

Design	Size *[kGE]*	Cycles */S-box*	Randomness *[bits] (online)*	Shares *[Input/Output]*
[7]	3.71	3	44	4/3
[8]	2.84	3	32	3/3
[14]	1.98	6	54	2/2
[15]	1.69	2-5	19	2/2
[18]	4.61	4	0	4/4
[18]	3.63	4	68	3/3
[18]	3.8	4	34	3/3
[18]	3.34	3	24	3/3
[18]	2.91	3	20	3/3
[22]	2.2	8	18	2/2
[25]	4.24	4	48	3/3

tion to about 9.7 kGE and 11.8 kGE (also iterative SubWord transformation for key schedule), respectively.

Discussion on the randomness costs. Independent from the targeted implementation variant, our design requires no online randomness and only two random bits for the initial sharing. For other implementations in the literature at least 128 bits for the sharing of the key and 128 bits for the sharing of the plaintext are required for a two-share implementation, or $2 \cdot 256$ bits for a three-share threshold implementation, respectively. From this perspective, our implementation saves at least 254 random bits for the initial sharing.

Regarding online randomness, the most randomness efficient first-order two-share implementation is reported for the DOM AES design by Gross et al. [22] which requires 18 bits of fresh randomness for each S-box. One SubBytes transformation thus requires at least 288 random bits, and one round with key scheduling requires 360 bits. The total amount of online randomness for one AES-128 encryption is thus between 2.88 kbits and 3.6 kbits of fresh randomness.

For the sake of completeness, we note that there exist more online randomness efficient S-box implementations which, however, require an increased amount of input shares for the S-box (e.g., the four-share S-box of Ghoshal et al. [18]). Since there is no full AES implementation or estimation given in [18], one has to decide whether to use multiple shares for each masked input signal or to dynamically scale the number of shares before and after the S-box. The first case results in an increased number of randomness for the initial sharing (e.g., 768 bits for four shares of plaintext and key as in [18]), and requires a lot of additional registers and logic for storing and calculating on these four shares. For the dynamic conversion from two to multiple shares, on the other hand, the area overhead would mainly result from the larger size of the S-box (4.7 kGE) and the logic required for expansion and compression of the shares. Regarding randomness, at least the same amount of online randomness as for the initial

sharing would be required in each round for the expansion and compression from two to four shares.

Summary. Comparing our design with others is quite difficult because most of the existing implementations do not consider the amount of required initial randomness for sharing the key and plaintext data. Nevertheless, as our hardware comparison shows the size of our AES (S-box) is similar to state-of-the-art Boolean masked designs while not only saving online randomness but also saving randomness required for the initial sharing. Requiring only two bits of randomness for each masked encryption could thus make the difference in practice between deciding on requiring an additional PRNG or using an already on-board TRNG, and could thus make first-order masking cheap enough to be used for highly constraint devices like low-cost RFID tags.

6 Formal Verification of the SCA Resistance

For the verification of the side-channel security of our approach, we used the formal verification tool maskVerif of Barthe et al. [2] on the synthesized modules. Since maskVerif is originally designed to verify sharing-based circuits, the outcome of our synthesis tool creates a verification wrapper that is later on modified to represent the correct masking for the input signals of the actual masked circuit. The verification wrapper thus takes two shares per input of the masked module and creates the correct masking by first adding the mask as defined by the mask encoding and subsequently the second share of the input. The input bits $\{i0, i1, \ldots i7\}$ of the masked AES S-box module for example, uses the same SMT mask encoding $\{2, 3, 3, 1, 1, 2, 1, 2\}$ (where 1 denotes m_0, 2 denotes m_1, and 3 denotes $m_0 \oplus m_1$) as any other module for both inputs and outputs. We take the input shares of the wrapper (indicated by the suffix "_0" for the first share or "_1" for the second share) and create the actual masking as follows.

Table 7. SCA resistance verification results

*M*odule	*N*umber of tuples	*V*erification time	*R*esult
AddByte	95	16 ms	probing secure
MixColumns	315	108 ms	probing secure
SubByte	429	22 s	probing secure

```
module verification_wrapper(
        input   i0_0, i1_0, ..., i7_0,
        input   i0_1, i1_1, ..., i7_1,
        input   m0, m1,
        output  o0, o1, ..., o7);

        // Mask encoding
        assign i0 = (i0_0 ^ m1)       ^ i0_1; // 2
        assign i1 = (i1_0 ^ m0 ^ m1) ^ i1_1; // 3
        assign i2 = (i2_0 ^ m0 ^ m1) ^ i2_1; // 3
        assign i3 = (i3_0 ^ m0)       ^ i3_1; // 1
        assign i4 = (i4_0 ^ m0)       ^ i4_1; // 1
        assign i5 = (i5_0 ^ m1)       ^ i5_1; // 2
        assign i6 = (i6_0 ^ m0)       ^ i6_1; // 1
        assign i7 = (i7_0 ^ m1)       ^ i7_1; // 2

        // DUV
        aes_sbox sbox_inst(i0, i1, i2, ..., i7, ...);
endmodule
```

For the input in the maskVerif tool, the circuit is read by the Yosys [30] open synthesis tool. The circuit is then mapped to Yosys' internal gate representation (ILANG) and subsequently flattened such that a single module is created that contains all gates of the circuits. The resulting circuit is then returned in ILANG format for which input, output and mask signals are annotated before the circuit is fed into maskVerif. The circuits are validated for the probing model of Ishai et al. [23].

As the results show, all of the modules on which our entire AES-128 encryption relies on is probing secure as intended. The remaining parts of our AES circuit are only rewiring (readdressing) which does not influence the probing security. With the input and output constraints for our synthesis tool, we also ensured that the mask encoding for each byte is the same, and we can thus safely compose these modules without creating flaws in the probing model for first-orders. However, we note that this composition argument is only true for first-order circuits for which a probing attacker is restricted to a single probe.

7 Conclusions

In this paper, we have demonstrated that first-order masking does not require more than two bits of randomness in both software and hardware. These two bits of randomness include the initial randomness for masking of secret data as well as the so-called online randomness that is usually required by other masking approaches to keep the first-order probing security. In practice, the differentiation of randomness spend on masking the input data and the randomness spent on keeping this independence during the computation is not very meaningful for

most applications. We have also shown that our approach not only leads to first-order probing secure circuits (which we verified using formal tools as well as manual verification) but also that this approach can be automated easily. The downside of our approach, which is more noticeable in hardware, is an increased latency behavior due to the required control of the order in which operations are performed. However, we want to emphasize that the main idea of this work was to demonstrate that two bits of randomness not only pose the intuitive theoretical lower bound for first-order masking but that this bound is achievable in practical circuits.

Future Work Our findings not only give answers to intriguing research questions but also lead the way to some follow-up questions.

- We demonstrated that when sacrificing latency, a lot of random bits can be saved and therefore the costs involved with the production of randomness. At the same time, there exists work like the low-latency masking approach of Gross et al. [19], that show that arbitrary circuits can be calculated in a securely masked way and in a single cycle when randomness considerations are not taken into account. A consequent next step is thus to research concepts to design masked circuits which achieve a better trade-off regarding latency and area for a give randomness budget.
- Another open question is if and how the introduced concepts can be extended to higher-order masking. While for first-order masking, where an attacker is bound to a single observation, masks can be reused in the same form and combination at different points in the circuit. For higher-order masking, the same combination of masks at different positions automatically lead to a violation of the probing security. This does not mean that mask reuse is not possible but only that more aspects need to be taken into account like the encoding of the masks at multiple positions.
- While two random bits are enough to achieve first-order probing security, it does not mean that its actual security level (e.g., in terms of required leakage traces to extract a secret) is the same as for an implementation that uses a lot of randomness on mask refreshing of intermediate values. Using less randomness clearly provide a larger attack surface to horizontal attacks and most likely also increases the signal-to-noise ratio for an DPA attacker. However, what is less obvious is the question whether or not the saved randomness could be more effectively used, e.g., for an additional hiding countermeasures that lowers the signal-to-noise ration by a higher extend then by spending more randomness on masking itself.
- Currently, there exist two strong trends in the masking community: 1) there exist more and more work on how to make the formal analysis of masked implementations faster and easier for general circuits, 2) there is a lot of research published on making masking more randomness efficient and to provide better trade-offs. These two trends are diverging, since faster formal verification for general masked circuits seems to require stronger randomness requirements and more circuitry to achieve these easier verifiable security notions like NI and SNI [1]. However, we think that by more careful analysis

of the properties of masked gates and masking schemes, faster (formal) verification methods can be designed that are tailored for a specific masking approach.

Acknowledgements.

This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402). This work has been supported by the Austrian Research Promotion Agency (FFG) via the project IoT4CPS. Lauren De Meyer is funded by a PhD fellowship of the Fund for Scientific Research - Flanders (FWO).

References

1. G. Barthe, S. Belaïd, F. Dupressoir, P. Fouque, B. Grégoire, P. Strub, and R. Zucchini. Strong non-interference and type-directed higher-order masking. In *ACM Conference on Computer and Communications Security*, pages 116–129. ACM, 2016.
2. G. Barthe, S. Belaïd, P. Fouque, and B. Grégoire. maskverif: a formal tool for analyzing software and hardware masked implementations. *IACR Cryptology ePrint Archive*, 2018:562, 2018.
3. G. Barthe, F. Dupressoir, S. Faust, B. Grégoire, F. Standaert, and P. Strub. Parallel implementations of masking schemes and the bounded moment leakage model. In *EUROCRYPT (1)*, volume 10210 of *Lecture Notes in Computer Science*, pages 535–566, 2017.
4. S. Belaïd, F. Benhamouda, A. Passelègue, E. Prouff, A. Thillard, and D. Vergnaud. Randomness complexity of private circuits for multiplication. In *EUROCRYPT (2)*, volume 9666 of *Lecture Notes in Computer Science*, pages 616–648. Springer, 2016.
5. S. Belaïd, F. Benhamouda, A. Passelègue, E. Prouff, A. Thillard, and D. Vergnaud. Private multiplication over finite fields. In *CRYPTO (3)*, volume 10403 of *Lecture Notes in Computer Science*, pages 397–426. Springer, 2017.
6. B. Bilgin, B. Gierlichs, S. Nikova, V. Nikov, and V. Rijmen. Higher-order threshold implementations. In *ASIACRYPT (2)*, volume 8874 of *Lecture Notes in Computer Science*, pages 326–343. Springer, 2014.
7. B. Bilgin, B. Gierlichs, S. Nikova, V. Nikov, and V. Rijmen. A more efficient AES threshold implementation. In *AFRICACRYPT*, volume 8469 of *Lecture Notes in Computer Science*, pages 267–284. Springer, 2014.
8. B. Bilgin, B. Gierlichs, S. Nikova, V. Nikov, and V. Rijmen. Trade-offs for threshold implementations illustrated on AES. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 34(7):1188–1200, 2015.
9. A. Biryukov, D. Dinu, Y. L. Corre, and A. Udovenko. Optimal first-order boolean masking for embedded iot devices. In *CARDIS*, volume 10728 of *Lecture Notes in Computer Science*, pages 22–41. Springer, 2017.
10. R. Bloem, H. Groß, R. Iusupov, B. Könighofer, S. Mangard, and J. Winter. Formal verification of masked hardware implementations in the presence of glitches. In *EUROCRYPT (2)*, volume 10821 of *Lecture Notes in Computer Science*, pages 321–353. Springer, 2018.

11. J. Boyar and R. Peralta. A Small Depth-16 Circuit for the AES S-Box. In *SEC*, volume 376 of *IFIP Advances in Information and Communication Technology*, pages 287–298. Springer, 2012.
12. D. Canright. A very compact s-box for AES. In *CHES*, volume 3659 of *Lecture Notes in Computer Science*, pages 441–455. Springer, 2005.
13. S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 1999.
14. T. D. Cnudde, O. Reparaz, B. Bilgin, S. Nikova, V. Nikov, and V. Rijmen. Masking AES with d+1 shares in hardware. In *CHES*, volume 9813 of *Lecture Notes in Computer Science*, pages 194–212. Springer, 2016.
15. L. De Meyer, O. Reparaz, and B. Bilgin. Multiplicative masking for aes in hardware. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(3):431–468, Aug. 2018.
16. L. de Moura and N. Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
17. A. Duc, S. Dziembowski, and S. Faust. Unifying leakage models: From probing attacks to noisy leakage. In *EUROCRYPT*, volume 8441 of *Lecture Notes in Computer Science*, pages 423–440. Springer, 2014.
18. A. Ghoshal and T. D. Cnudde. Several masked implementations of the boyar-peralta AES s-box. In *INDOCRYPT*, volume 10698 of *Lecture Notes in Computer Science*, pages 384–402. Springer, 2017.
19. H. Groß, R. Iusupov, and R. Bloem. Generic low-latency masking in hardware. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(2):1–21, 2018.
20. H. Groß and S. Mangard. Reconciling d+1 masking in hardware and software. In *CHES*, volume 10529 of *Lecture Notes in Computer Science*, pages 115–136. Springer, 2017.
21. H. Groß and S. Mangard. A unified masking approach. *J. Cryptographic Engineering*, 8(2):109–124, 2018.
22. H. Groß, S. Mangard, and T. Korak. Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. *IACR Cryptology ePrint Archive*, 2016:486, 2016.
23. Y. Ishai, A. Sahai, and D. A. Wagner. Private circuits: Securing hardware against probing attacks. In *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 463–481. Springer, 2003.
24. P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.
25. A. Moradi, A. Poschmann, S. Ling, C. Paar, and H. Wang. Pushing the limits: A very compact and a threshold implementation of AES. In *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Science*, pages 69–88. Springer, 2011.
26. S. Nikova, C. Rechberger, and V. Rijmen. Threshold implementations against side-channel attacks and glitches. In *ICICS*, volume 4307 of *Lecture Notes in Computer Science*, pages 529–545. Springer, 2006.
27. J. Quisquater and D. Samyde. Electromagnetic analysis (EMA): measures and counter-measures for smart cards. In *E-smart*, volume 2140 of *Lecture Notes in Computer Science*, pages 200–210. Springer, 2001.
28. O. Reparaz, B. Bilgin, S. Nikova, B. Gierlichs, and I. Verbauwhede. Consolidating masking schemes. In *CRYPTO (1)*, volume 9215 of *Lecture Notes in Computer Science*, pages 764–783. Springer, 2015.

29. P. Schwabe and K. Stoffelen. All the AES you need on cortex-m3 and M4. In *SAC*, volume 10532 of *Lecture Notes in Computer Science*, pages 180–194. Springer, 2016.
30. C. Wolf. Yosys Open SYnthesis Suite. http://www.clifford.at/yosys/.