Low-Cost First-Order Secure Boolean Masking in Glitchy Hardware - full version*

Dilip Kumar S.V., Josep Balasch, Benedikt Gierlichs, Ingrid Verbauwhede *Fellow, IEEE,*

***Abstract*—We describe how to securely implement the masked logical AND of two bits in hardware in the presence of glitches without the need for fresh randomness, and we provide guidelines for the composition of circuits. As a case study, we design, implement, and evaluate masked DES cores. We focus on first-order secure Boolean masking and do not aim for provable security. Our goal is a practically relevant trade-off between area, latency, randomness cost, and security. We provide two low-cost solutions. Our first solution focuses on strong security while simultaneously aiming for low implementation costs. The resulting DES engine shows no evidence of first-order leakage in a non-specific leakage assessment with 50M traces. Our second solution follows the opposite approach: we focus on lowering implementation costs, latency to be specific, while not sacrificing much on security. Our low-latency DES engine exhibits signs of first-order leakage only after approximately 15M traces.**

***Index Terms*—Side-channel analysis, Masking, Glitches.**

I. Introduction

Over the last few decades, much attention has been dedicated to researching and developing fast and efficient cryptographic implementations that are secure against power analysis attacks [2]. Masking [3], [4] is a well-known technique that can be used to protect both hardware and software implementations. Its core idea is to split the data being processed by an implementation into random shares, effectively eliminating its correlation with the device's power consumption.

In this work, we focus on first-order Boolean masking, where each sensitive (intermediate) value x is randomly split into two shares x_0 and x_1 such that $x = x_0 \oplus x_1$. First-order masked implementations can, in theory, be broken with higher-order attacks, which combine leakage of multiple (or all) shares to derive sensitive values. We nevertheless focus on first-order masking because performing a successful higher-order attack can be made very difficult by adding noise (the number of traces needed increases exponentially in the attack order, with the noise factor in the basis) [3].

In hardware, masking is commonly applied at the gate level. As logic gates are used as a fundamental building block in gate-level masking, any cost reduction in building a masked logic gate significantly benefits the overall cost of a masked circuit. A significant hurdle in hardware masking is to overcome the effect of glitches, i.e. undesired signal transitions in the circuit, as they are known to temporarily reveal unmasked sensitive values [5].

A methodology for implementing a masked circuit in hardware requires at least a masked AND gadget, a masked XOR gadget and rules for the composition of gadgets to build a masked circuit. A masked XOR is easy because one can simply apply the XOR to each share separately. A masked AND is more difficult as the computation needs to involve all shares of all variables at some point. One needs to be very careful not to reveal any unmasked sensitive intermediate values, as demonstrated in many previous works. Composition is also difficult because circuit effects, uniformity of inputs and outputs, and their dependencies need to be tracked and corrected as necessary.

Modern hardware masking techniques, such as Threshold Implementation (TI) [6] and Domain-oriented Masking (DOM) [7], have been designed to address the problem caused by glitches. In contrast to classical Boolean masking, they control the propagation of glitches through register layers and maintain the uniformity of the intermediate values by injecting fresh randomness. As a result, they achieve *provable* security against first-order attacks. Threshold Implementation is shown to be provable secure [8] under the glitch-robust probing model [9]. Although DOM does not enjoy a security proof, it has shown many times to be secure in practice. Some of them further generalize to higher-orders. However provable security comes with higher costs. Protected implementations using modern masking schemes require a lot more resources in terms of area, latency, and randomness than classical Boolean masking [10].

In this work, we develop a low-cost Boolean masked AND gate to build masked circuits that provide *practical* security in the presence of glitches. Our contributions are as follows:

- Starting from the software-oriented masked AND construction by Biryukov et al. [11], we derive a low-cost AND gadget suitable for hardware implementations which requires no fresh randomness. We propose two solutions to prevent glitches by controlling the arrival time of input operands.
- We provide guidelines for composition and exemplary circuits for securely computing the logical AND of more than two terms and circuits with AND and XOR gadgets. We pay particular attention to the need to remask and explain when and how to do it.
- We design and implement two masked DES encryption engines building on the proposed low-cost AND gadget and guidelines for composition. We add security measures only where needed for practical security.
- We evaluate the performance of our designs both in terms of cost (area, latency, randomness) and first-order side-channel leakage on an FPGA platform.

* Full version of an extended abstract published at DATE 2023 [1].

II. Low-Cost Masked AND2 Gadget

If a regular AND2 computes $z = x \cdot y$, a straightforward masked AND2 could for instance compute $z_0 = x_0 \cdot y_0 \oplus x_0 \cdot y_1$ and $z_1 = x_1 \cdot y_0 \oplus x_1 \cdot y_1$ such that $z = z_0 \oplus z_1$. This would, however, not be secure because $z_0 = x_0 \cdot (y_0 \oplus y1) = x_0 \cdot y$ depends on unmasked y, and similar for z_1. A simple solution to this problem, as first proposed by Trichina [10] consists in the introduction of a fresh random bit r in the equations:

$$\begin{aligned} z_0 &= r \oplus (x_0 \cdot y_0) \oplus (x_0 \cdot y_1) \oplus (x_1 \cdot y_1) \oplus (x_1 \cdot y_0) \\ z_1 &= r \end{aligned} \tag{1}$$

This construction is secure only if the order of evaluation is from left to right. A well-known problem arises when implementing such a gadget in hardware, because the order of evaluation is unknown and glitches in the combinational circuit can happen. Previous work, such as TI and DOM, provide solutions for this problem. They require fresh random bits, too. The cost in terms of the number of random bits is an important criterion when comparing masked implementations. Gross et al. [12] propose an AND2 gadget and rules for composition which allow implementing, e.g. an entire masked AES-128 using only two bits of randomness. The security of their approach was proven in the t-probing model [13]. However, in hardware, this approach leads to a significant penalty in latency. The software implementation provided by Gross et al. was found to be insecure [14]. Like Gross et al. [12] we start our work from the masked AND2 gadget proposed by Biryukov et al. [11] for software implementations:

$$\begin{aligned} z_0 &= (x_0 \cdot y_0) \oplus (x_0 + \overline{y_1}) \\ z_1 &= (x_1 \cdot y_0) \oplus (x_1 + \overline{y_1}) \end{aligned} \tag{2}$$

$\cdot, \oplus, +$ denote AND, XOR and OR, respectively. We refer to this gadget as `secAND2` from now on. A remarkable property of this gadget is that it does not require fresh randomness to be secure. Yet, due to the lack of a fresh mask, the output is not independent of the input, which needs to be considered during composition. Another advantage of the `secAND2` gadget over the one by Trichina is that it requires fewer elementary logic operations (AND, XOR, etc.) and will thus lead to a faster implementation in software or a smaller implementation in hardware. While Gross et al. aimed for provable security in the presence of glitches with minimal randomness requirements, we strive for an overall practically relevant tradeoff between area, latency, randomness cost and security.

A. Secure Hardware Implementation of `secAND2`

A straightforward ASIC implementation of `secAND2` using a common standard cell library, i.e. using AND2, XOR2, OR2 and INV gates, will be insecure due to glitches in the circuit. A similar problem occurs when implementing the logic equations on FPGAs using Look Up Tables (LUT). We have verified this by performing leakage assessment tests on a Spartan6 FPGA. Our results clearly show that programming the equations for the outputs of `secAND2` (z_0 and z_1) directly into LUTs leaks, which we attribute to glitches.

Glitches on the output of a logic gate are created by different arrival times of its input signals. Predicting the order in which the inputs arrive in a large circuit is impossible. But if we have control over the order of and the delay between the arrival of input signals, it might be possible to send the inputs in a safe sequence such that there are no glitches and thus no leakage of any information about the sensitive inputs or intermediate values. In the next subsection, we investigate the existence of such safe sequences for `secAND2`.

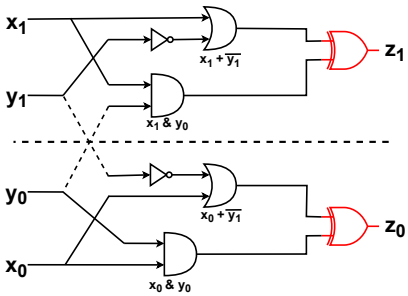


Fig. 1: `secAND2` gate schematic.

B. Identifying Safe Input Sequences

We experiment with the issue of glitches on the same Spartan6 FPGA by forcibly sending the inputs of `secAND2` (x_0, x_1, y_0, y_1) at different time instances. Sending one input after another can be executed in a controlled manner with the help of registers by connecting them directly to the inputs of `secAND2`. First, we reset all four registers to 0. Then, we update one register at a time over four consecutive clock cycles with the desired input sequence. We exhaust all $4! = 24$ possible input sequences and observe any potential leakage using TVLA methodology [15] for each sequence across the four clock cycles by collecting half a million traces. To improve the signal-to-noise ratio (SNR), we replicated multiple parallel instances of `secAND2` on the FPGA, each receiving identical inputs. Inputs x and y are independently shared with uniformly distributed random bits.

Clock Cycle	1	2	3	4	
Input	*	*	*	x_1 or x_0	→ Sequence leaks
Input	*	*	*	y_1 or y_0	→ Sequence does not leak

TABLE I: Leakage behaviour of `secAND2` for different input sequences. '*' denotes any of the remaining input shares.

The results of this experiment are summarized in Table I. In short, we observe leakage in sequences where either x_0 or x_1 arrive in the last clock cycle, but not in sequences where either y_0 or y_1 arrive last. These results can be explained from the `secAND2` equations in (2). Our `secAND2` gadget is not non-complete with respect to y (refer to the non-completeness property of TI), as the equations involve both shares y_0 and y_1. The equation for z_0 depends on x_0, y_0 and y_1 whereas z_1 depends on x_1, y_0 and y_1. Assume x_0 arrives in the last cycle, as all input registers are initially reset, if the input x_0 is 1, then the XOR gate outputting z_0(highlighted in red in Figure 1) toggles from $\overline{y_1}$ to $y_0 \oplus 1$, a hamming distance of $y_0 \oplus y_1$. Therefore, if a glitch occurs, the late arrival of x_0 can reveal information about the unshared input $y(= y_0 \oplus$

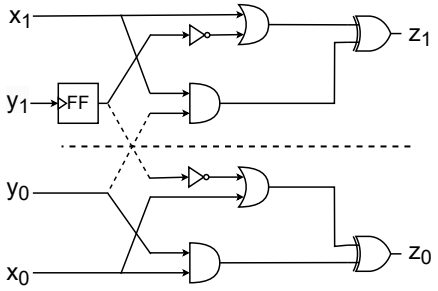


Fig. 2: secAND2 gate with internal FF or secAND2−FF.

y_1) and similar for x_1. By forcing y_0 or y_1 to arrive last, we essentially make x_0 and x_1 arrive early. We observe no leakage in sequences where y_0 or y_1 arrive last because x_0 and x_1 do not evaluate on the combined value of y_0 and y_1, which would leak the unshared input $y(= y_0 \oplus y_1)$. Looking at the secAND2 circuit in Figure 1, in the first three clock cycles, no signal or gate has enough information to be able to leak anything about either sensitive unshared inputs, *x* and *y*. And as a result, we can achieve a *temporary* non-completeness property for both output bits during the evaluation in the first three clock cycles. In the fourth clock cycle, only a single input bit arrives straight from a register. Any signal in the secAND2 circuit, see Figure 1, will toggle at most once. In other words, glitches cannot occur in the last clock cycle. Thus, no sensitive information can be leaked even though secAND2 is no longer non-complete. The Hamming Distance of the outputs, z_0 and z_1, before and after the fourth clock cycle does not depend on either sensitive inputs, *x* or *y*. Therefore, the final cycle does not leak either sensitive input, and any sequence that ends with y_0 or y_1 can securely compute a product of two shared variables.

Although we have identified safe sequences, we have to address a few issues related to our initial assumption and proposed solution. We began with the assumption that the four registers connected to secAND2 to provide inputs are reset to 0. But this is hardly the case in practice. Our secAND2 gadgets are typically expected to be reused for computation as part of a cryptographic circuit. It might not always be feasible to reset the input registers between computations, for example, if the circuit is pipelined. Additionally, this approach would significantly increase latency as each secAND2 evaluation would take four clock cycles to compute instead of one. Lastly, a masked cryptographic circuit typically contains several AND gates connected to one another; sending the inputs in four clock cycles for every multiplication would require extra registers to temporarily buffer the intermediate values, which also increases the area cost. In what follows, we propose two solutions to tackle these problems.

C. Solution 1: secAND2 with a flip-flop (secAND2−FF)

To create a secure low-cost masked AND gate, any circuit in which one of the two shares of y, i.e. y_0 or y_1, arrives last will guarantee no leakage. This can be achieved by delaying the processing of either of the two inputs, for example, y_1. Our secAND2 could hence be constructed as illustrated in Figure 2, using an internal delay flip flop (FF). The flip-flop delays the input y_1 and ensures secure computation. This optimization reduces the number of clock cycles to calculate a multiplication from four to two. We shall refer to this faster two-cycle secAND2 as secAND2−FF from now on. We also verified its security with leakage assessment experiments.

In subsection II-B, we explained that the order in which inputs arrive could determine whether the computation is secure or not and that late arrival of x_0 (or x_1) has the potential to reveal information about the unshared input $y(= y_0 \oplus y_1)$. Suppose we compute two multiplications consecutively on the same secAND2−FF gadget: let the inputs for the first multiplication be (m_0, m_1, n_0, n_1) and the inputs for the second multiplication (a_0, a_1, b_0, b_1). If we did not reset the inputs between the multiplications and a_0 arrives before b_0 and b_1, the existing inputs of the secAND2−FF, n_0 and n_1 would remain unchanged when a_0 arrives. If m_0 = 0 and a_0 = 1, then the output z_0(in Figure 2) toggles from $\overline{n_1}$ to $n_0 \oplus 1$, or if m_0 = 1 and a_0 = 0, then the output z_0 toggles from $n_0 \oplus 1$ to $\overline{n_1}$. Hence, a_0 would leak information about the previous computation, $n(= n_0 \oplus n_1)$. Our first solution, secAND2−FF, reduces latency, but it must be reset between successive computations.

D. Solution 2: secAND2 with path delay (secAND2−PD)

We propose our second solution to address the issues of secAND2. We eliminate the need for resetting between consecutive multiplications while also reducing the latency. Using a flip-flop as a delay element in our previous solution guarantees that one of the inputs arrives late. Instead of using a flip-flop, we now propose using path delay to achieve the same result, for instance, by making one of the input signals travel through a longer path so it arrives late. This solution follows a more practical approach and comes with certain constraints, such as placement and routing, which might not be as straightforward to implement. We shall later explain in more detail in Section V that this solution can indeed be achieved in practice. Using path delay as a delay element, instead of a flip-flop, eliminates the critical need to reduce the number of cycles required to send the inputs. Unlike our previous solution, we could send our inputs one after another, each input with a different amount of delay, while not increasing the cycle count of our implementation. In fact, we could compute secAND2 in a single clock cycle. This may of course increase the critical path delay and thus reduce the circuit's maximum clock frequency.

Consequently, this approach of sending each input with a different amount of delay would help us compute consecutive multiplications without the need to reset the inputs between computations. We propose such a cost-efficient delayed sequence in Figure 3, which we shall call secAND2−PD. Each input is either delayed by zero, one, or two DelayUnits. We refer to a replicable amount of delay as a DelayUnit, and we shall later, in Section V, explain how this can be realized in practice. Input y_0 is not delayed and arrives first in

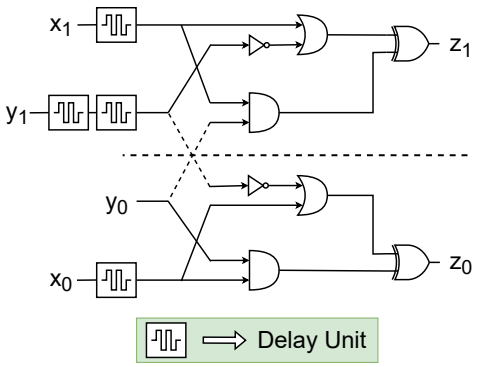


Fig. 3: secAND2 gate with path delay or secAND2−PD.

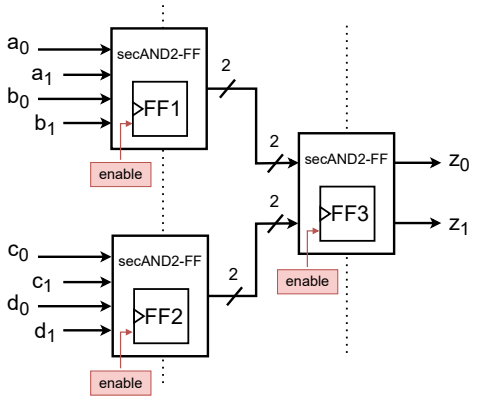


Fig. 4: Product of four masked variables using secAND2−FF.

order to protect against information leakage about the previous computation. It is followed by the delayed x_0 and x_1. And finally, y_1 arrives as the last input as explained above.

Referring to our previous example, in Section II-C, b_0 arrives before a_0 (or a_1) does. Therefore a_0 (or a_1) cannot leak information about $n(= n_0 \oplus n_1)$ from the previous computation, as n_0 is replaced by b_0. And the final input b_1 arrives after a_0 and a_1, thereby protecting information leakage about the current computation, i.e. a_0 (or a_1) cannot leak information about $b(= b_0 \oplus b_1)$. In conclusion, secAND2−PD does not require an input reset and also decreases the latency of our secAND2 gadget to a single cycle.

III. Composing Secure Masked Circuits

secAND2−FF and secAND2−PD gadgets can be used as a building block to securely implement more complex masked circuits. In this section, we provide guidelines on two important steps which are generally required to build circuits. We first show how to compute products of more than two variables securely. And then, we explain how to add *dependent* variables securely.

A. Computing Product Terms using secAND2−FF

As we start building a circuit, it is common to have situations where we need to compute a product of more than two variables. To illustrate, we compute a product of four variables, $z = a \cdot b \cdot c \cdot d$, which we assume here to be independently shared. Implementing this expression securely can be done with the circuit in Figure 4, which evaluates $z =$ secAND2−FF(secAND2−FF(a, b), secAND2−FF(c, d)) using three secAND2−FF gadgets and has a latency of three clock cycles.

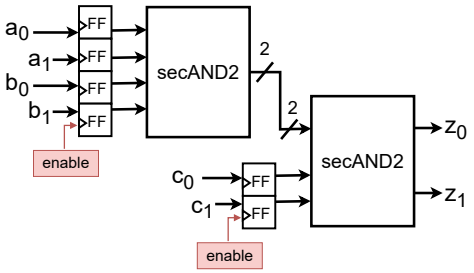


Fig. 5: secAND2 with input registers.

By carefully controlling when we sample the internal FFs, we can achieve a secure construction with no additional (i.e. external) FFs, which also helps us keep the area footprint low. All the inputs arrive in the first clock cycle, and in the second clock cycle, the enable signal corresponding to flip-flops FF1 and FF2 is set to high. The enable signal controls when the FF samples the input. FF1 and FF2 sample b_1 and d_1 respectively, therefore secAND2−FF(a, b) and secAND2−FF(c, d) are computed securely. The enable signal of FF3 remains disabled during the second clock cycle, so the secAND2−FF computing $z =$ secAND2−FF(secAND2−FF(a, b), secAND2−FF(c, d)) is inactive. And in the third and final clock cycle, the enable signal of FF3 is toggled to high, thereby securely completing the computation of output (z_0, z_1) in three clock cycles.

In the general case, implementing a product of n independent variables requires $n-1$ secAND2−FF gadgets arranged into $log_2(n)$ layers, such that all different sub-products are cascaded. The latency of the circuit becomes $log_2(n) + 1$ cycles.

In specific cases, it might be advantageous to take the internal FFs out of the secAND2−FF gadgets and instead place them at the beginning. For instance, when we compute low-degree products. The flip-flop inside secAND2−FF serves the purpose of delaying one of the input signals. Equivalently, we can replace secAND2−FF with a secAND2 and place registers before the gate to buffer the input shares, as shown in Figure 5. We can then use a Finite State Machine (FSM) to control when the FFs sample, thus guaranteeing a safe arrival sequence of the input operands to the secAND2 gadgets. Unlike internal FFs inside secAND2−FF gadgets, which solely belong to that gadget, the input registers we now use can be commonly shared by multiple secAND2 gadgets. For instance, consider the two multiplications $a*b*c$ and $a*b*d$. The input registers used to store a_0, a_1, b_0, b_1, can be shared for the two multiplications. This is usually the case when we are computing polynomials. We would have to compute several different products with common inputs. The resulting circuit can have a slightly larger area due to extra input FFs, but it can be beneficial for evaluation purposes. It allows us, for instance, to test and compare different input sequences or to

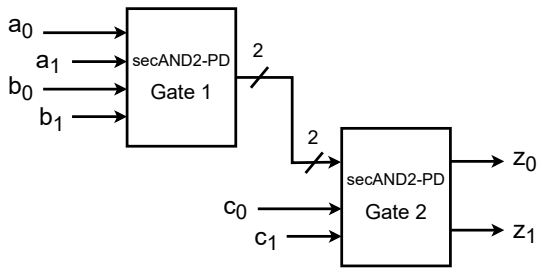


Fig. 6: Product of three masked variables using `secAND2-PD`.

Product of 3 variables $z = a \cdot b \cdot c$	$c_0 \rightarrow b_0 \rightarrow a_0, a_1 \rightarrow b_1 \rightarrow c_1$
Product of 4 variables $z = a \cdot b \cdot c \cdot d$	$d_0 \rightarrow c_0 \rightarrow b_0 \rightarrow a_0, a_1 \rightarrow b_1 \rightarrow c_1 \rightarrow d_1$

TABLE II: Delay sequence for a product 3 or 4 variables

reset the FFs at any given time.

B. Computing Product Terms using `secAND2-PD`

Our construction for computing a product of more than two variables using `secAND2-PD` resembles a chain-like structure in contrast to the tree structure we illustrated in the previous subsection. This decision was made for practical reasons, as it helped implement hardware delays easier. In our experience, it is relatively easy to enforce delays on inputs of `secAND2-PD` that arrive directly from registers. But in a typical tree structure which is organized in layers, the outputs of `secAND2-PD` gadgets of one layer are fed as inputs to the next layer of `secAND2-PD` gadgets. Based on our experience, it was not as easy to enforce delays on outputs of `secAND2-PD` gadgets. A construction for a product of three variables, $z = a \cdot b \cdot c$, which we assume here to be independently shared, is shown in Figure 6. An appropriately delayed input sequence, as shown in Table II, can be used to compute the product of three variables in a single clock cycle. To create the delayed sequence, we use a `DelayUnit`, like the one we used in Section II-D. c_0 is not delayed, b_0 is delayed by one `DelayUnit`, a_0 and a_1 are delayed by two `DelayUnits`, b_1 is delayed by three `DelayUnits` and c_1 is delayed by four `DelayUnits`. The reasoning behind the input sequence remains the same as before, a_0 (and a_1) not only has the potential to leak information about values $b(= b_0 \oplus b_1)$ in Gate 1, see Figure 6, but also $c(= c_0 \oplus c_1)$ in Gate 2. Hence, c_0 is sent first to protect against information leakage about the value of $c(= c_0 \oplus c_1)$ from any previous computation and c_1 arrives last to protect against leakage about the current computation. The rest of the sequence, between c_0 and c_1, is identical to what was discussed in subsection II-D. Similarly, we can construct an input sequence for a product of four variables, see Table II.

To generalize, implementing a product of n independent variables requires $n-1$ `secAND2-PD` gadgets arranged into $n-1$ layers. In theory, with a proper input sequence, a product of any number of variables can be computed in a single clock cycle. It is up to the designer to realise such a sequence in practice. In this work, we successfully and securely computed a product of three variables in a single clock cycle. We have not explored computing the product of more than three variables in a single cycle, as it was not needed for our secure DES implementation.

C. Addition of Product Terms

Masked AND and XOR gates are fundamental to building a masked circuit. So far, we concentrated on masking AND gates. But it is also essential to ensure no loss of security during XOR. Both our `secAND2-FF` and `secAND2-PD` gadgets do not consume fresh randomness. Instead, the uniformity of the output is achieved by reusing the randomness of the inputs. This characteristic becomes critical when implementing circuits that combine several terms, for instance, through addition. It can lead to decreased security if the added terms are not independent.

Consider the function $f = x \oplus y \oplus x \cdot y$, where the product term $z = x \cdot y$ is computed with either a `secAND2-FF` or `secAND2-PD` gadget. In this situation, the masked output z is not independent of x and y, leading to a data-dependent distribution of the masked inputs of the XOR plane. Securing this function, as well as any other which combines dependent shares, requires selectively *refreshing* the (intermediate) dependent variables. Figure 7 depicts a circuit to compute f securely. It requires 1 bit of randomness m to refresh the shares of z and guarantee a uniform output distribution.

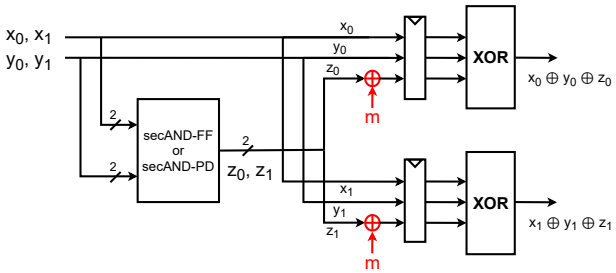


Fig. 7: $f = x \oplus y \oplus x \cdot y$ (secure).

IV. Case Study: DES

In this section, we use both `secAND2-FF` and `secAND2-PD` gadgets as fundamental building blocks to build a secure cryptographic circuit, following the rules for composition defined in Section III. We choose the Data Encryption Standard (DES) as our target algorithm because it is the main building block of Triple-DES(TDES), which is still widely used today.

A. Deconstructing DES

We begin with the (unprotected) design, which is a classical round-based DES architecture including key schedule. The main difficulty in protecting the DES implementation lies in the S-Boxes, which are the only non-linear components. The DES cipher has 8 different S-boxes in the substitution layer, each taking six input bits $(x_0, x_1, x_2, x_3, x_4, x_5)$ and returning four output bits (y_1, y_2, y_3, y_4). A hardware-friendly way to implement them is to represent them as four 4-bit permutations

(the so-called *mini S-boxes*) and a multiplexer (MUX). As we follow a bottom-up approach, we represent each mini S-box through equations in Algebraic Normal Form (ANF) that consist only of AND and XOR terms. Equation 3 shows the representation of a mini S-box.

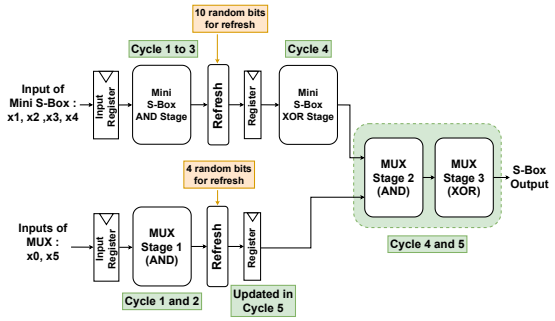
$$\begin{aligned} y_1 &= 1 \oplus x_1 \oplus x_2 \oplus x_1x_2 \oplus x_2x_3 \oplus x_1x_2x_3 \oplus x_4 \oplus x_2x_3x_4 \\ y_2 &= 1 \oplus x_1 \oplus x_2 \oplus x_1x_3 \oplus x_2x_4 \oplus x_3x_4 \oplus x_1x_3x_4 \\ y_3 &= 1 \oplus x_1x_2 \oplus x_3 \oplus x_1x_3 \oplus x_2x_3 \oplus x_1x_2x_3 \oplus x_4 \oplus x_1x_4 \\ &\quad \oplus x_2x_4 \oplus x_1x_2x_4 \oplus x_3x_4 \\ y_4 &= x_1 \oplus x_3 \oplus x_1x_4 \oplus x_2x_4 \oplus x_1x_3x_4 \end{aligned} \tag{3}$$

As visible in the equations, there are at most six distinct terms of degree 2 and four terms of degree 3. Additionally, all four mini S-boxes of an S-box share common inputs, which means it is sufficient to compute the ten possible product terms only once. The equations in ANF form can be split into two stages: AND stage and XOR stage. The AND stage can be implemented with 10 AND gates. In the XOR stage, the product terms from the AND stage are XORed together with the inputs to form the outputs of the mini S-boxes, as shown in Equation 3. Since the outputs of the AND stage are not independent of the inputs, it is necessary to add a refresh layer in front of the XOR layer, as motivated in Section III-C. The outputs of the 10 AND gates in the AND stage are refreshed with 10 bits of fresh randomness before the XOR layer. It is possible to further optimize the refresh step by selectively refreshing only some of the ten terms instead of refreshing all of them while maintaining uniformity, but we leave this optimization for future work.

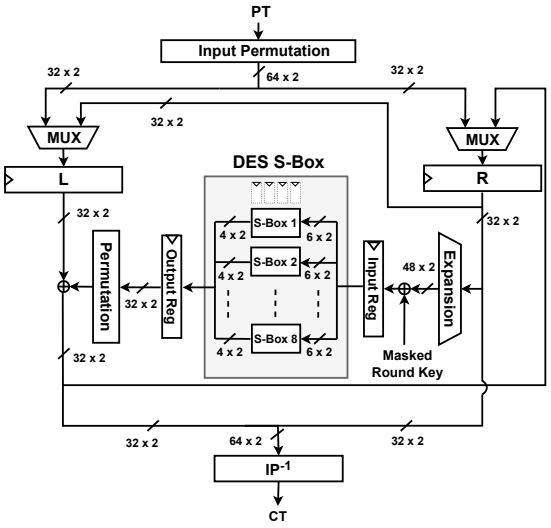
The purpose of the MUX is to select which one of the four mini S-boxes to output. Similar to the mini S-boxes, computations can also be split into three stages. Equation 4 is in ANF and shows how each output bit of the 4×1 MUXes is computed. First, we calculate the four multiplications of the select bits, x_0 and x_5, that is: $x_0x_5, x_0\overline{x_5}, \overline{x_0}x_5, \overline{x_0}\ \overline{x_5}$. Second, each of these products is multiplied with the respective output of the mini S-boxes. Since the MUX receives 16 inputs from the four mini S-boxes, there are 16 multiplications in the second stage. Lastly, in the third stage, the outputs of the second stage are XORed together to produce the four S-Box output bits.

$$\begin{aligned} y1 &= (x_0x_5)y_1^{(1)} \oplus (x_0\overline{x_5})y_1^{(2)} \oplus (\overline{x_0}x_5)y_1^{(3)} \oplus (\overline{x_0}\ \overline{x_5})y_1^{(4)} \\ y2 &= (x_0x_5)y_2^{(1)} \oplus (x_0\overline{x_5})y_2^{(2)} \oplus (\overline{x_0}x_5)y_2^{(3)} \oplus (\overline{x_0}\ \overline{x_5})y_2^{(4)} \\ y3 &= (x_0x_5)y_3^{(1)} \oplus (x_0\overline{x_5})y_3^{(2)} \oplus (\overline{x_0}x_5)y_3^{(3)} \oplus (\overline{x_0}\ \overline{x_5})y_3^{(4)} \\ y4 &= (x_0x_5)y_4^{(1)} \oplus (x_0\overline{x_5})y_4^{(2)} \oplus (\overline{x_0}x_5)y_4^{(3)} \oplus (\overline{x_0}\ \overline{x_5})y_4^{(4)} \end{aligned} \tag{4}$$

Finally, we add the MUX and the four mini S-box circuits to construct the DES S-box. Our final DES implementation is constructed by adapting the round-based architecture to incorporate first-order Boolean masking. All original variables (plaintext, key, round state) are split into two shares and operations are performed in the masked domain. Datapaths for linear operations are simply duplicated to operate on individual shares. Our design also includes a masked key schedule that runs parallel to the DES operation. The substitution layer is adapted to incorporate our protected S-Boxes.



(a) S-box architecture

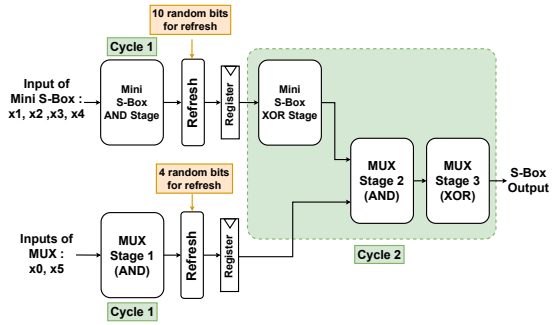


(b) High-level DES architecture (protected)

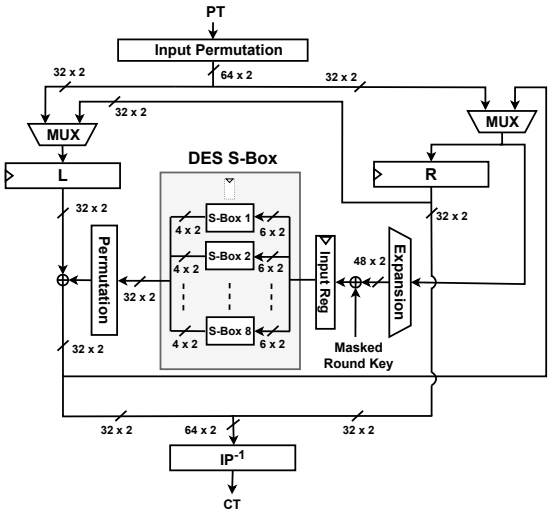
Fig. 8: Building DES using `secAND2-FF`.

B. Building DES with `secAND2-FF`

Figure 8a visually represents all stages of the S-Box using `secAND2-FF`. The AND stage of the mini S-boxes has product terms with a maximum degree of 3. Hence this stage can be computed in 3 clock cycles using ten `secAND2-FF` gadgets. As the mini S-boxes are composed of low-degree polynomials, we follow the approach we suggested in section III-A. We remove the internal FFs from `secAND2-FF` and place an input register layer. The calculation of the mini S-box outputs takes four clock cycles, three for the AND stage and one for the XOR stage. The calculation of the MUX AND Stage 1 (variables of the form (x_0x_5)) is done in parallel and takes two cycles. Both outputs are then sent to the MUX AND Stage 2. As the inputs for the MUX AND Stage 2 come from two parts of the S-Box, they must be synchronized to guarantee a safe multiplication sequence. This is why a register is placed after the MUX AND Stage 1. Recall that we explained the need to refresh dependent variables in Section III-C. Therefore, we need to refresh the output of the MUX AND Stage 2 to maintain uniformity before going through the MUX XOR Stage 3. To save costs, we move the refresh stage inside the MUX and safely apply it directly after the MUX AND Stage 1. By doing so, we save one clock cycle in the overall S-box computation while also reducing the



(a) S-box architecture



(b) High-level DES architecture (protected)

Fig. 9: Building DES using `secAND2-PD`.

randomness requirements, as fewer terms need to be refreshed. Given that a register is already placed after the MUX AND Stage 1 (to ensure a safe sequence for the MUX AND Stage 2), this optimization also saves some flip-flops. The final S-box architecture using `secAND2` with input register layer has a latency of 5 clock cycles, and it is the one we use in our evaluation and experiments.

The final DES architecture is shown in Figure 8b. After adding an S-box output register and a state register (L and R) between successive rounds, the design yields a final latency of 7 cycles per round. Whether the S-box output register can be removed and reduce the latency to 6 cycles, without affecting the security of the implementation, is a question we leave for future work.

C. Building DES with `secAND2-PD`

Using `secAND2-PD`, every AND stage in the circuit can be executed in a single cycle. The S-box architecture for the design using `secAND2-PD` is shown in Figure 9a. We already discussed how to compute a product of up to three inputs in Section III-B. Hence the mini S-box AND stage can be implemented with 10 `secAND2-PD` instances and with a reduced latency of one clock cycle. For the Mini S-box AND Stage and the MUX Stage 1, the path delays are applied to the inputs $(x_0, x_1, x_2, x_3, x_4, x_5)$. For the MUX Stage 2, the delays are applied to the refreshed outputs of MUX Stage 1 and the linear Mini S-box XOR Stage outputs. The result is an S-box architecture with a latency of 2 clock cycles.

Our focus for DES design using `secAND2-PD` is to reduce implementation costs, latency to be specific. Therefore, we decided not to increase latency by adding any additional register stages outside the S-box. For the previous design using `secAND2-FF`, the input register is crucial for security as it is used to control the order of the input sequence. Because of this, the state register (R), see Figure 8b, is updated first and then the state register (R) is used to update the input register over the next few clock cycles, to compute the S-Box safely. But for `secAND2-PD`, the inherent path delay takes care of the input sequence. And for that reason, the output of the S-box is directly connected to the input register, see Figure 9b, instead of going through the state register, which saves a clock cycle. The state register is updated in parallel, and the S-box output register is also removed. Hence, the latency of our design with path delay is reduced to 2 clock cycles per round.

V. Implementing Path Delays in Hardware

Glitches are a well-known characteristic of CMOS circuits that make it difficult to implement masking securely in hardware. Glitches are unintended transitions at the output of gates in a circuit due to the delayed arrival of some input signals. To solve this, in Section II-D, we proposed using path delay to control the order in which inputs arrive. This section explains the steps we took to realize path delay in hardware. We use

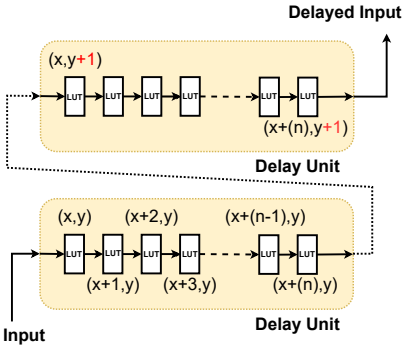


Fig. 10: An input is delayed by two `DelayUnits`. Each `DelayUnit` is of size n LUTs.

a SAKURA-G board with a Xilinx SPARTAN-6 FPGA to implement our protected DES implementation, using Xilinx ISE software. We use Look Up Tables (LUT) as a basic delay element, by connecting the input of the LUT directly to its output. The LUT acts as a buffer for the input and provides the path delay we need. Several LUTs are chained together to form a basic unit, which we refer to as a *`DelayUnit`*. For path delay to be an effective countermeasure, it needs to be replicable. If we were to let ISE design tools perform the placement and routing of our basic delay elements (LUT), the amount of delay would vary depending on where the LUTs are placed on the FPGA. This would result in an inconsistent outcome, and the amount of delay would be hard to quantify.

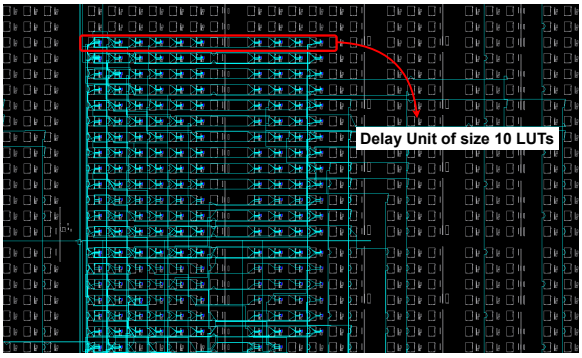


Fig. 11: `DelayUnits` for an S-box.

For that reason, we manually perform the placement of our delay elements by defining constraints. FPGA resources such as LUTs, flip-flops and multiplexers are grouped together in SLICEs to create configurable logic blocks. Every SLICE of a SPARTAN-6 FPGA contains four LUTs, and by manually choosing the location of the SLICE, we can control the placement of our delay elements. To create a `DelayUnit`, we place the delay elements horizontally by incrementing the x coordinate of the SLICE, as shown in Figure 10. If a signal has to be delayed by more than one `DelayUnit`, multiple `DelayUnits` are stacked vertically using location constraints on the FPGA. As a result, we obtain a deterministic layout with a delay amount that can be controlled and quantified.

Our next step is to investigate the optimal size for a `DelayUnit`, i.e. the number of LUTs chained together, to achieve practical security. As a general rule, the more the delay, the better it is for security. A large amount of delay would guarantee that the arrival of inputs is well separated in time. But it is also important to note that there is a trade-off between cost and security. Creating a large delay chain increases the utilization cost of our implementation and also limits the maximum operating frequency of our implementation. As the focus of our work is to find a trade-off between low-cost and practical security, we gradually increase the size of our delay chain until we find a good balance.

We tested several sizes of the `DelayUnit` through leakage assessment experiments and found a `DelayUnit` of size 10 LUTs to be optimal. Please refer to Section VII where we validate this claim with detailed leakage test results for different sizes of the `DelayUnit`. Figure 11 shows the placement and routing of the `DelayUnits` that correspond to one of the S-boxes of the protected DES design using `secAND2-PD` on a SPARTAN-6 FPGA. The placement and routing were performed by Xilinx ISE software with the location constraints we assigned to individual delay elements. Figure 12 shows a zoomed-out view covering all the S-boxes of DES. Each red rectangle represents one S-box, hence 8 in total.

VI. Implementation Results

The results of our full DES implementations are summarized in Table III. We include area reports from both ASIC and FPGA synthesis. The ASIC design results are gathered

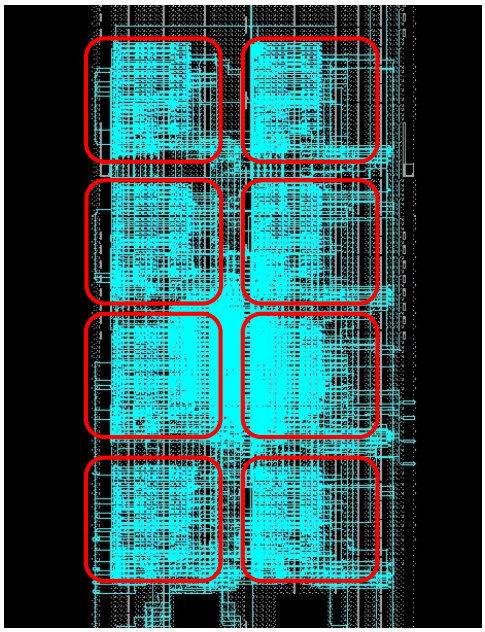


Fig. 12: All `DelayUnits` for DES using `secAND2-PD`.

using NanGate 45nm Open Cell Library [16] and synthesized with Synopsys DC Compiler 2017.09. We put elementary `secAND2` and XOR gates in modules and compile with `-exact_map` option to prevent the modules from being optimized. The FPGA design results are obtained from implementing the design on a Spartan-6 FPGA, the same model as used in the evaluation in Section VII, using the Xilinx ISE software. Note that to prevent optimization of modules and interaction between shares, we synthesize with "Keep Hierarchy" constraint set to on. Lastly, we provide in the table also the numbers from the work in [17], reporting a TDES implementation protected with DOM. Note that the number of cycles as reported in [17] is scaled down to compare with a DES implementation (the original cycle count for TDES is 5*48+4).

TABLE III: Utilization results of full DES implementations including the masked key schedule.

Version		ASIC [GEs]	FPGA [FF/ LUT]	Rand[§]	Cycle[§]	Max Freq.[‖] [MHz]
`secAND2-FF`		14488	819/ 2129	14	7	183
`secAND2-PD`		52273	678/ 7122	14	2	21
[17][#]	DOM-indep	13800[†]	-	176	5	-
	DOM-dep	22400[†]	-	528	5	-

[†] Calculated using a 28 nm RVT standard cell library from Global Foundries.
[#] Both versions from [17] do not mask the key schedule.
[§] per round
[‖] Obtained from Xilinx ISE timing report

A. Utilization for `secAND2-FF` version

From the area results, we can see that our design using `secAND2-FF` is more compact than the DOM-dep version in [17] but slightly larger than the DOM-indep version. Our design uses 30 `secAND2` gates per DES S-Box, compared to 22 DOM-indep gates in [17]. While `secAND2` gates are by design smaller than the DOM-indep gates, the additional modules

in our implementation (e.g. refreshing gadgets, input/output S-box registers, etc.) compensate for the gain to some extent. A non-negligible difference here is, however, that our GE results include the masked key schedule which adds around 900 GEs to the total area cost. This is not the case in [17], in which the numbers include the cost of an unmasked key schedule. The overhead caused by this module further explains the difference between both designs.

In terms of randomness usage, our design involves fewer bits, 14 bits per round, as we opt to reuse randomness across the eight different S-boxes. We note this decision does not have an impact on the first-order security (see Section VII), and hence we opt to use it in our reference implementation. In case our design would not recycle randomness, or in case the work in [17] did, then our numbers, 14*8 = 112 bits per round, would still be lower than both DOM-dep and DOM-indep. Additionally, we only expect a slight increase in area due to additional FFs required to store the 112 bits of randomness. We also note that in [17] there is no evaluation of the security of the DOM-indep version, but rather of the DOM-dep version which consumes 3 random bits per AND gate.

Lastly, our implementation takes 2 more cycles per round compared to [17]. While all designs feature a protected S-box with a latency of 5 cycles, the two additional input/output S-box registers in our design increase the overall round latency to 7 cycles. The input S-box register is a design choice, which facilitates control over the sequence of the input operands to the mini S-box stage and allows us flexible resetting of the flip-flops. Whether the output S-box register can be removed without affecting the security of the implementation is a question we leave for future work.

B. Utilization for `secAND2-PD` version

The latency for our design using `secAND2-PD` is two cycles per round, much less than compared to [17]. Our randomness usage remains the same, 14 bits per round, for both versions using `secAND2-FF` and `secAND2-PD`.

The usage of `DelayUnit` in our design using `secAND2-PD` complicates the area comparisons between designs. We use LUTs as delay elements in our FPGA implementation. In Section V, we explained that we find the optimal size for `DelayUnit` through multiple iterations and by using leakage results as feedback. However, this process of tweaking the implementation to find the optimal size would be very expensive and also impractical for ASIC implementations. Therefore, it is a challenging task to provide accurate ASIC implementation costs for our design using `secAND2-PD`. Instead, we follow a different approach to provide a reasonable area estimate. For our ASIC design, we use inverters (NOT gates) as our delay elements to buffer signals and measure the size of `DelayUnit` by the number of inverters they are composed of. We first synthesize a circuit without any delay elements, which is equivalent to a protected DES implementation using `secAND2-PD` but with `DelayUnit` of size 0 LUTs. And then estimate the path delay of the critical path of the circuit. We set the size of the `DelayUnit` as the number of inverters required to achieve a delay of the critical path multiplied by a factor of 3, to accommodate a safety margin. From further synthesis runs, we estimated the size of `DelayUnit` for the desired amount of delay to be 120 inverters. Table III shows the total area including all `DelayUnits`, each of size 120 inverters. If we exclude the `DelayUnits`, the area of the remaining circuit only adds up to *12592 GEs*. Having a generous amount of 120 inverters per `DelayUnit` contributes to the rest of the area utilization. Whether the size of `DelayUnit` can be reduced without affecting the security is hard to predict.

We would like to emphasize that the purpose of utilization results for our design using `secAND2-PD` is only to provide a realistic estimate and not a new state-of-the-art DES implementation. As it is difficult to provide an accurate number that would guarantee the security of an ASIC implementation without testing it.

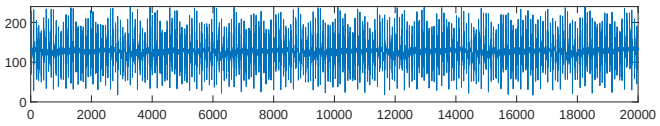


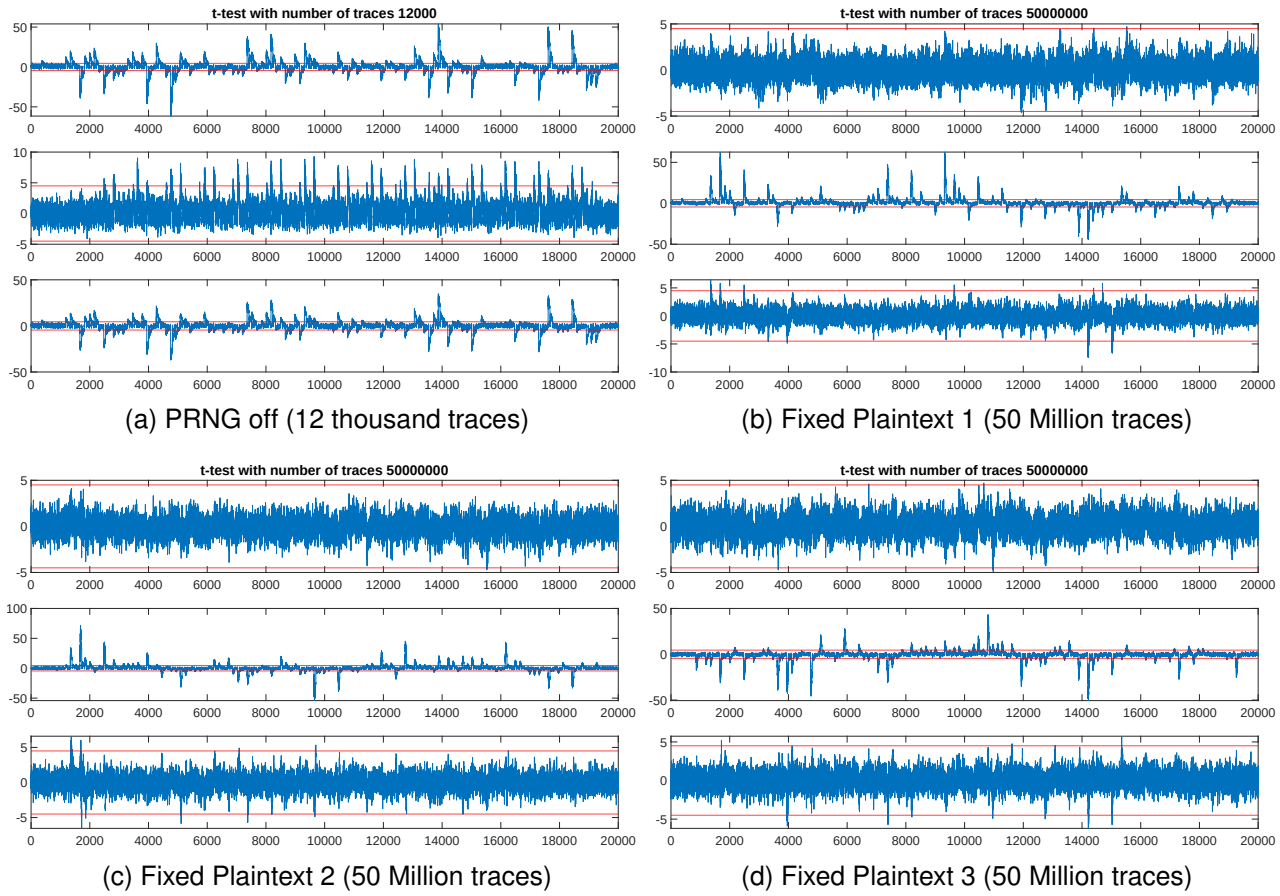
Fig. 13: Power trace for protected DES design using `secAND2-FF` (seven cycles per round).

VII. Security Evaluation

We use a SAKURA-G board to evaluate the side-channel security of our protected DES implementations on the target SPARTAN-6 FPGA. We measure the (onboard amplified) power consumption of the target FPGA on SMA connector J3 with a Tektronix DPO7254C oscilloscope sampling at 500MS/s and operating at 3 MHz. The power measurements cover the entire DES operation. We use the non-specific Test Vector Leakage Assessment (TVLA) methodology as explained in [18] and originally described in [15]. We focus on univariate analysis since our protected DES core processes both shares in parallel. We perform fixed vs. random tests where we provide either a fixed or a random plaintext to the masked DES core, in a random sequence. The DES key is fixed for all experiments but masked before every DES operation. We use only 14 random bits per round, thus recycling them for all 8 S-boxes in parallel.

A. Leakage assessment of protected DES design using `secAND2-FF`

Figure 13 shows the power trace (raw oscilloscope ADC output) covering the entire DES operation. Figures 14b, 14c, 14d display the results we obtained in three tests using three different fixed plaintexts for the design using `secAND2-FF`. We measured and analyzed 50 million traces for each of these tests. Each Figure shows from top to bottom: first-order univariate t-test result, second-order univariate t-test result and third-order univariate t-test result. Red lines indicate the commonly applied threshold at ± 4.5. As we can see, there is no evidence of first-order leakage. There are very few and minor crossings of the ± 4.5 threshold in the first-order t-test values, but they are not consistent across the three different



(a) PRNG off (12 thousand traces)

(b) Fixed Plaintext 1 (50 Million traces)

(c) Fixed Plaintext 2 (50 Million traces)

(d) Fixed Plaintext 3 (50 Million traces)

Fig. 14: Leakage assessment results for protected DES design using `secAND2-FF`.

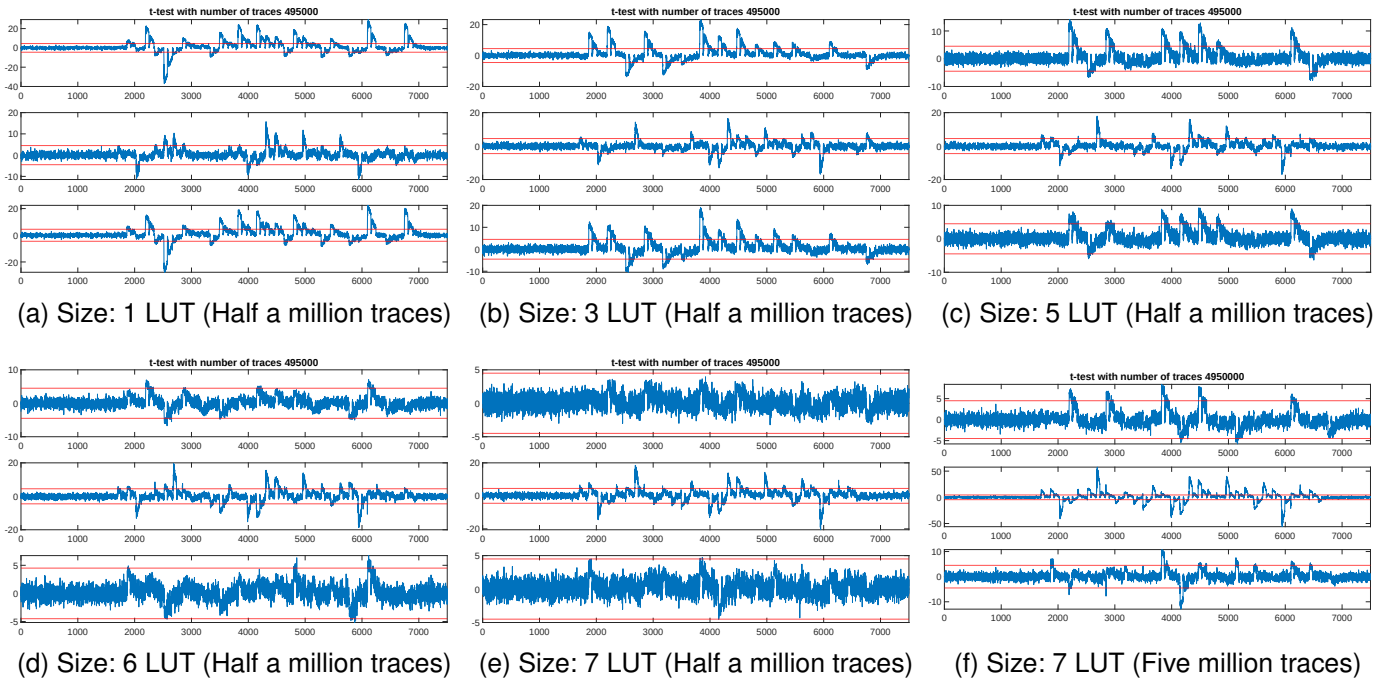
plaintexts, i.e. the threshold is not exceeded at the same time indexes, as is required by the TVLA methodology in order to deem an implementation leaking. More importantly, we have clear evidence of second-order leakage, with the second-order t-test reaching values as large as 60. Therefore an adversary would likely be better off using a second-order attack against our protected DES implementation, as similarly argued in [19], which require exponentially more traces by adding noise, as discussed earlier. Alternatively, one can simply prevent an attacker from acquiring the required number of traces with protocol-level countermeasures such as re-keying.

For completeness, we provide a result of the same leakage assessment performed with the PRNG switched off, which means all random bits used for the initial masking and used by the masked DES core are zero. Figure 14a shows that with as little as 12 000 traces we obtain very significant peaks in the first-order t-test, which confirms that our setup works well.

B. Finding the optimal `DelayUnit` size for `secAND2-PD`

In Section V, we explained that in practice, the size of a `DelayUnit` is important for security. In this subsection, we provide evidence to support our claim. We implement several different versions of our protected DES using `secAND2-PD`, which we explained in Section IV-C. The only difference between the versions is the size of `DelayUnit`. We can confidently say that any difference in leakage assessment results we might observe is directly attributed to the change in the `DelayUnit` size, as all other aspects of the implementation remain the same. For our first version, we start with the smallest possible size of a `DelayUnit` consisting of a single LUT. And for successive versions, we gradually increase the size of `DelayUnit`. We perform fixed vs. random tests for all versions. To make a fair comparison of the test results, we use the same fixed plaintext and collect the same number of traces for all versions.

Figure 16 shows the power trace covering the entire protected DES using `secAND2-PD`. Figure 15 shows the leakage assessment results for all versions. We collect half a million traces for all versions with the same fixed plaintext. Each subfigure shows from top to bottom: first-order univariate t-test result, second-order univariate t-test result and third-order univariate t-test result. We see pronounced leakage for the design using the smallest `DelayUnit` size, see Figure 15a. A simple explanation would be that a `DelayUnit` consisting of just a single LUT is not enough to have any significant impact on the path delay. Therefore it cannot guarantee the delayed sequence required for secure computation of `secAND2-PD`, hence the leakage. From the rest of the figures, we can notice that there is a pattern with the first-order leakage as we increase `DelayUnit` size. We can see a decrease in first-order leakage as we increase the size. And for a `DelayUnit` of size 7 LUTs, see Figure 15e, we can no longer see any first-



(a) Size: 1 LUT (Half a million traces) (b) Size: 3 LUT (Half a million traces) (c) Size: 5 LUT (Half a million traces)

(d) Size: 6 LUT (Half a million traces) (e) Size: 7 LUT (Half a million traces) (f) Size: 7 LUT (Five million traces)

Fig. 15: Leakage assessment results to find the optimal `DelayUnit` size for `secAND2-PD`.

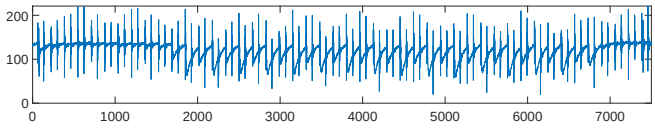


Fig. 16: Power trace for protected DES design using `secAND2-PD` (two cycles per round).

order leakage with half a million traces. But upon collecting more traces, we eventually started noticing first-order leakage, which indicates that a size of 7 LUTs is still not the optimal size. Figure 15f shows the leakage assessment results with 5 million traces. So we increased the size further and found the `DelayUnit` size of 10 LUTs to be optimal. And from our experiments, sizes of more than 10 LUTs do not contribute to any significant improvement in first-order leakage and only result in increased utilization costs. The critical path for our protected DES using `secAND2-PD` lies in the mini S-Box AND stage, see Figure 9a. The mini S-Box AND stage has product terms with a maximum degree of 3 and the critical path of `secAND2-PD` is determined by the input share which is delayed the most. To compute a product of three variables, the input share c_1 is delayed by 4 `DelayUnits`, see Table II. From the Xilinx ISE timing report, with a `DelayUnit` size of 10 LUTs, the FPGA implementation of our protected DES design with `secAND2-PD` has a maximum operating frequency of 21 MHz. In comparison, the FPGA implementation of our protected DES using `secAND2-FF` results in a maximum frequency of 183 MHz.

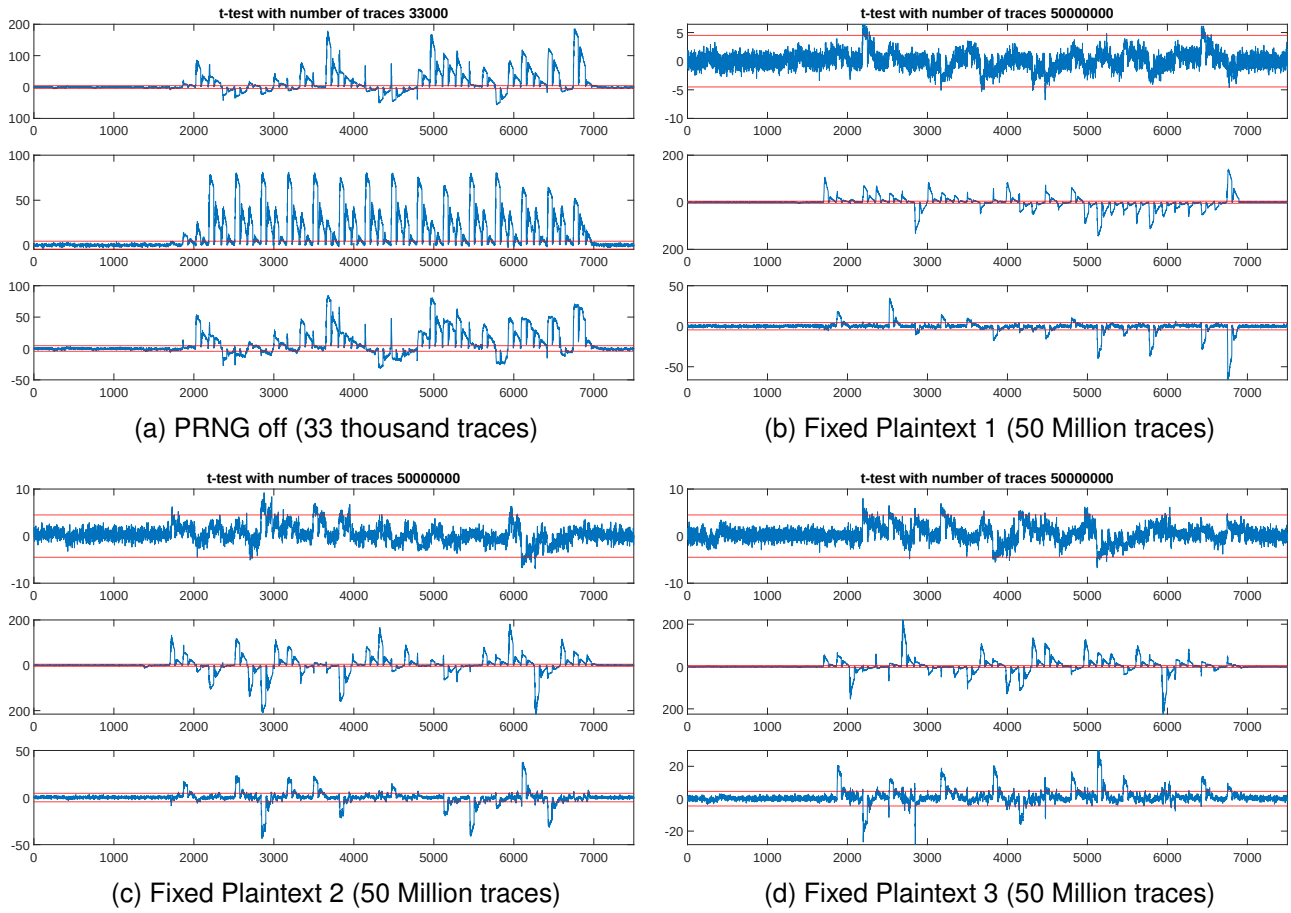
In the next subsection, we present the leakage assessment results of our final version of protected DES with `DelayUnit` size of 10 LUTs.

C. Leakage assessment of protected DES design using `secAND2-PD`

Figure 17 shows the leakage test results with the same three fixed plaintexts we used to evaluate the protected DES with `secAND2-FF`. The sanity check with PRNG off in Figure 17d shows that with 33,000 traces we obtain very significant peaks in the first-order t-test. Comparing the t-test results with PRNG on to our previous results using `secAND2-FF`, we can see that the first-order t-statistic does indeed exceed the threshold of ± 4.5. We think there are two possible explanations for this observation. The first explanation is that the `DelayUnit` size of 10 LUTs does not provide enough delay. The security of `secAND2-PD` relies on the input sequence. We demonstrated in Section VII-B that increasing the amount of delay will separate the arrival time of the inputs, which is necessary for a safe computation. Therefore, a `DelayUnit` size of more than 10 LUTs should be more secure. But we conducted further experiments, and we did not notice much improvement from increasing the `DelayUnit` size, which leads us to our second explanation. For *d+1*-implementations with *d=1*, i.e. 2-share implementations, multiple prior works, [20], have noted first-order leakage even if the implementations are first-order probing secure [21]. Provable secure implementations do not always translate to secure masked implementations due to the presence of physical effects such as *coupling* [22]. These effects are particularly relevant to our design because of the long path delays we create in our circuit for `secAND2-PD`, which in turn creates a lot of opportunity for coupling.

VIII. Conclusion

We describe two approaches to designing and implementing a low-cost first-order secure Boolean masked DES encryption



(a) PRNG off (33 thousand traces)

(b) Fixed Plaintext 1 (50 Million traces)

(c) Fixed Plaintext 2 (50 Million traces)

(d) Fixed Plaintext 3 (50 Million traces)

Fig. 17: Leakage assessment results for protected DES design using `secAND2-PD`.

engine in hardware in the presence of glitches. While provable security is certainly important and informative in this domain, it can but does not always lead to implementations that are secure in practice. This is typical because the model that was used to prove security does not entirely capture physical reality. We therefore do not aim for provable security but for practical security by requiring no first-order leakage up to a number of measurements where we see strong evidence of higher-order leakage. Admittedly our design and implementation ideas are strongly inspired by approaches like TI and DOM. But we add security measures only where practical leakage assessment pointed out an issue, and not to make the theory work. We construct two first-order Boolean masked DES cores with our two AND gates, `secAND2-FF` and `secAND2-PD`.

Using `secAND2-FF`, our design shows no evidence of first-order leakage with 50M traces. The only related work we found is by Sasdrich and Hutter [17]. Taking the difference in the key schedule into account, our area number is comparable to their DOM-indep result and we need less randomness per S-box than their cheapest variant, DOM-indep. They perform leakage assessment only for the significantly more costly (area and randomness) DOM-dep implementation. Our implementation is moderately slower, 115 clock cycles compared to 84 clock cycles, which is our trade-off choice. Using `secAND2-PD`, we substantially reduce the number of clock cycles compared to DOM-indep and DOM-dep implementations, while the randomness cost remains the same. Our `secAND2-PD` gadget, uses LookUpTables (LUTs), which is an FPGA component, to create path delays. Hence, a fair comparison of our area with the reported ASIC numbers of [17] is hardly possible. However, we emphasize that the purpose of our `secAND2-PD` design is to provide a hardware-oriented solution for masking. Recently, there has been an increasing interest in low-latency masking techniques. Some of the latest approaches focus on preventing glitch propagation by substituting registers with dual-rail encodings and asynchronous logic. However, while achieving low latency, these methods often face limitations in lower maximum frequencies. Our `secAND2-PD` gadget provides an alternative solution tailored for applications such as smart cards or RFID, which do not require fast clock frequencies. Considering there is not much research on practical solutions for masking on hardware, we demonstrate the effectiveness of using LUT-based path delay as an interesting new research direction.

Acknowledgments

This work was supported in part by the Flemish Government through the Cybersecurity Research Program with grant number VOEWICS02, by the European Commission through the

Horizon 2020 research and innovation program under grant agreement Belfort ERC Advanced Grant 101020005 695305, and through the Horizon Europe research and innovation program under grant agreement HORIZON-CL3-2021-CS-01-02 101070008 ORSHIN.

REFERENCES

[1] S. V. D. Kumar, J. Balasch, B. Gierlichs, and I. Verbauwhede, "Low-cost first-order secure boolean masking in glitchy hardware," in *2023 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2023, pp. 1–2.

[2] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Advances in Cryptology - CRYPTO '99*, ser. LNCS, M. J. Wiener, Ed., vol. 1666. Springer, 1999, pp. 388–397.

[3] S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi, "Towards sound approaches to counteract power-analysis attacks," in *Advances in Cryptology - CRYPTO '99*, ser. LNCS, M. J. Wiener, Ed., vol. 1666. Springer, 1999, pp. 398–412.

[4] L. Goubin and J. Patarin, "DES and differential power analysis (the "duplication" method)," in *Cryptographic Hardware and Embedded Systems - CHES'99*, ser. LNCS, Ç. K. Koç and C. Paar, Eds., vol. 1717. Springer, 1999, pp. 158–172.

[5] S. Mangard, T. Popp, and B. M. Gammel, "Side-channel leakage of masked CMOS gates," in *Topics in Cryptology - CT-RSA 2005*, ser. LNCS, A. Menezes, Ed., vol. 3376. Springer, 2005, pp. 351–365.

[6] S. Nikova, C. Rechberger, and V. Rijmen, "Threshold implementations against side-channel attacks and glitches," in *Information and Communications Security - ICICS 2006*, ser. LNCS, P. Ning, S. Qing, and N. Li, Eds., vol. 4307. Springer, 2006, pp. 529–545.

[7] H. Groß, S. Mangard, and T. Korak, "Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order," in *Theory of Implementation Security - TIS@CCS 2016*, B. Bilgin, S. Nikova, and V. Rijmen, Eds. ACM, 2016, p. 3.

[8] S. Dhooghe, S. Nikova, and V. Rijmen, "Threshold implementations in the robust probing model," in *Proceedings of ACM Workshop on Theory of Implementation Security, TIS@CCS 2019, London, UK, November 11, 2019*, B. Bilgin, S. Petkova-Nikova, and V. Rijmen, Eds. ACM, 2019, pp. 30–37. [Online]. Available: https://doi.org/10.1145/3338467.3358949

[9] S. Faust, V. Grosso, S. M. D. Pozo, C. Paglialonga, and F. Standaert, "Composable masking schemes in the presence of physical defaults & the robust probing model," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2018, no. 3, pp. 89–120, 2018. [Online]. Available: https://doi.org/10.13154/tches.v2018.i3.89-120

[10] E. Trichina, "Combinational Logic Design for AES SubByte Transformation on Masked Data," http://eprint.iacr.org/2003/236, 2003.

[11] A. Biryukov, D. Dinu, Y. L. Corre, and A. Udovenko, "Optimal first-order boolean masking for embedded iot devices," in *Smart Card Research and Advanced Applications - CARDIS 2017*, ser. LNCS, T. Eisenbarth and Y. Teglia, Eds., vol. 10728. Springer, 2017, pp. 22–41.

[12] H. Groß, K. Stoffelen, L. D. Meyer, M. Krenn, and S. Mangard, "First-order masking with only two random bits," in *Theory of Implementation Security - TIS@CCS 2019*, B. Bilgin, S. Petkova-Nikova, and V. Rijmen, Eds. ACM, 2019, pp. 10–23.

[13] Y. Ishai, A. Sahai, and D. A. Wagner, "Private circuits: Securing hardware against probing attacks," in *Advances in Cryptology - CRYPTO 2003*, ser. LNCS, D. Boneh, Ed., vol. 2729. Springer, 2003, pp. 463–481.

[14] A. Beckers, L. Wouters, B. Gierlichs, B. Preneel, and I. Verbauwhede, "Provable secure software masking in the real-world," in *Constructive Side-Channel Analysis and Secure Design - COSADE 2022*, ser. LNCS, J. Balasch and C. O'Flynn, Eds., vol. 13211. Springer, 2022, pp. 215–235.

[15] G. Goodwill, B. Jun, J. Jaffe, and P. Rohatgi, "A testing methodology for side channel resistance validation," http://csrc.nist.gov/newsevents/non-invasive-attack-testing-workshop/08Goodwill.pdf, 2011.

[16] NANGATE, "The nangate 45nm open cell library," https://www.nangate.com.

[17] P. Sasdrich and M. Hutter, "Protecting triple-des against DPA - A practical application of domain-oriented masking," in *Constructive Side-Channel Analysis and Secure Design - COSADE 2018*, ser. LNCS, J. Fan and B. Gierlichs, Eds., vol. 10815. Springer, 2018, pp. 207–226.

[18] B. Bilgin, B. Gierlichs, S. Nikova, V. Nikov, and V. Rijmen, "Higher-order threshold implementations," in *Advances in Cryptology - ASIACRYPT 2014*, ser. LNCS, P. Sarkar and T. Iwata, Eds., vol. 8874. Springer, 2014, pp. 326–343.

[19] ——, "A more efficient AES threshold implementation," in *Progress in Cryptology - AFRICACRYPT 2014*, ser. LNCS, D. Pointcheval and D. Vergnaud, Eds., vol. 8469. Springer, 2014, pp. 267–284.

[20] N. Müller, T. Moos, and A. Moradi, "Low-latency hardware masking of PRINCE," in *Constructive Side-Channel Analysis and Secure Design - 12th International Workshop, COSADE 2021, Lugano, Switzerland, October 25-27, 2021, Proceedings*, ser. Lecture Notes in Computer Science, S. Bhasin and F. D. Santis, Eds., vol. 12910. Springer, 2021, pp. 148–167. [Online]. Available: https://doi.org/10.1007/978-3-030-89915-8_7

[21] T. D. Cnudde, M. Ender, and A. Moradi, "Hardware masking, revisited," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2018, no. 2, pp. 123–148, 2018. [Online]. Available: https://doi.org/10.13154/tches.v2018.i2.123-148

[22] T. D. Cnudde, B. Bilgin, B. Gierlichs, V. Nikov, S. Nikova, and V. Rijmen, "Does coupling affect the security of masked implementations?" in *Constructive Side-Channel Analysis and Secure Design - 8th International Workshop, COSADE 2017, Paris, France, April 13-14, 2017, Revised Selected Papers*, ser. Lecture Notes in Computer Science, S. Guilley, Ed., vol. 10348. Springer, 2017, pp. 1–18. [Online]. Available: https://doi.org/10.1007/978-3-319-64647-3_1