Uniform Sharing in Multiple Stages: NullFresh for Arbitrary Functions

Artemii Ovchinnikov[1], Aein Rezaei Shahmirzadi[2] and Siemen Dhooghe[1]

[1] COSIC, ESAT, KU Leuven, Leuven, Belgium
firstname.lastname@esat.kuleuven.be
[2] PQShield, Oxford, UK
aein.shahmirzadi@pqshield.com

Abstract. In the field of hardware masking, threshold implementations are a well-known technique that provides glitch-resistant power analysis security. While they guarantee probing security, finding a uniform sharing without additional randomness is difficult, making it challenging to apply to certain functions and, consequently, making it impossible to develop a tool that can straightforwardly generate the masked circuit. Additionally, this approach forces designers to use at least three shares in the underlying masking, which can make the design more costly. Other schemes, like DOM, which can work with two shares, often require fresh randomness. To address these issues, Shahmirzadi and Moradi introduced the NullFresh masking technique at CHES 2021. This method allows for uniform sharing with no additional randomness, using the minimal number of shares. However, similar to original threshold implementations, it is not always straightforward to find a NullFresh masking for arbitrary functions.
In this work, we introduce an automated technique to provide masking for arbitrary functions, ensuring first-order security. This technique is applicable to functions where the number of output bits does not exceed the number of input bits. While this technique introduces additional register stages (resulting in higher latency and area) compared to existing methods, it addresses the automation challenges of threshold implementations, which have remained an open problem since their inception. We present the masking technique, along with proofs of glitch-extended probing security, and demonstrate its application to several ciphers, including Prince, Midori, Skinny, Keccak, and AES. The masked designs were verified using SILVER and PROLEAD, and tested on an FPGA through TVLA.

Keywords: Masking, Side-channel Analysis, Probing Security, Uniformity

1 Introduction

Over the last 25 years, both industry and academic communities have collaborated to improve masking countermeasures, a technique to protect against power-analysis attacks first introduced by Kocher *et al.* [KJJ99]. In masking, each intermediate variable $x \in \mathbb{F}_2$ is split into several shares $x_0, ..., x_s \in \mathbb{F}_2$ such that the sum is equal to the original variable $x = \sum_{i=0}^{s} x_i$. This countermeasure is highly effective in hardware implementations but incurs costs, including increased area overhead, additional register stages to prevent glitches, and the need for fresh randomness during computation. These costs have led to significant efforts to improve masking schemes. In particular, area costs were reduced by minimizing the number of shares required to maintain security, as in the well-known domain-oriented masking scheme [GMK16], register stages were reduced to improve the latency of the scheme such as the generic low-latency masking scheme [GIB18], and randomness was reduced using threshold implementation-like techniques [NRS11] such as NullFresh masking [SM21a] or by suggesting power-analysis-safe randomness generators such as the

Trivium cipher [Can06, CMM+23]. In this work, we focus on providing masking schemes that reduce the need for randomness.

In masking, we distinguish between techniques that provide arbitrary composable security and handcrafted designs. The former automatically ensures security by replacing each gate or part of a circuit with a secure masked variant, but at a higher cost compared to handcrafted designs. Composability involves conservative assumptions, leading to a more expensive final masked circuit. In particular, composable secure designs always require fresh randomness [Dho21, Theorem 3.1]. As a result, randomness-reduced masking schemes have relied on non-composable properties, such as the serial composable uniformity property from threshold implementations [NRS11]. This property ensures that a uniformly random masked vector of the input secret is mapped to a uniformly random masked vector of the output. While these designs typically have lower costs and do not require fresh randomness for first-order security, they have two main drawbacks.

First, handcrafted designs tend to be error-prone during the design stage. While modern tools like SILVER [KSM20] and PROLEAD [MM22] can help identify errors, their application to large designs remains challenging. Second, not every function has an easily obtainable uniform masking. The only existing solution that guarantees uniform sharing for an arbitrary function is the result by Piccione [PAB+23]. However, with this approach, the number of input shares equals $t + 2$, where t is the algebraic degree of the target function, making the resulting circuit quite large. Hence, there is a need to improve existing approaches to develop a generic method that maintains low or no randomness requirements with a minimal number of shares.

Contributions. In this work, we address the two downsides outlined above by proposing a novel technique to achieve a uniform masking for arbitrary functions that can be applied coordinate function by coordinate function, provided that the number of output bits does not exceed the number of input bits. While this technique does not offer arbitrary composable security, uniformity in addition to probing security still guarantees sequential composability [DNR19, Lemma 2]. As a result, when applying the automatic masking technique, implying correct, non-complete and uniform shares, on each S-box of a substitution-permutation cipher, the circuit is claimed to be first-order robust probing secure in case of the sequential composability. More specifically, in this work, we present *a technique that needs no fresh randomness and guarantees first-order glitch-extended probing security and uniformity over multiple register stages.*

The aforementioned masking technique can be seen as the automation of the NullFresh masking technique by Shahmirzadi *et al.* [SM21a]. This work represents a step forward in the generation of automated masked designs by avoiding trivially composable gadgets, thereby reducing implementation costs and minimizing the potential for errors typically associated with handcrafted designs. The technique has been proven secure in the glitch-extended probing model and has been applied to several ciphers such as Midori, Prince, Skinny, Keccak, and AES. Implementation costs for each of these applications have been calculated and compared with previous studies. The masking schemes have been evaluated using SILVER [KSM20] and the PROLEAD tool [MM22], and they have also been tested on FPGA using TVLA [GJJR11]. We upload the code of our tool, which implements the automatic masking generation algorithm, to a GitHub repository.[1]

2 Preliminaries

2.1 Notation

We denote stochastic variables by capital letters, share indices are denoted by a subscript, stages - by a superscript, and a share vector is denoted in bold. The binary negation of a

[1] https://github.com/KULeuven-COSIC/MultiStage_Uniformity

bit x is denoted by a bar $\overline{x}$. The operation + is used as an XOR operation. $\tilde{z}$ represents the shares of the value z, and $\tilde{g}(.)$ refers to the shared representation of the function g. We make use of vectors of bits, where a value $y \in \mathbb{F}_2^n$ is denoted in bits as $y = (y[0], ..., y[n-1])$. For the notation, we consider those bit vectors as lists in Python, meaning that $y[:k]$ denotes all bits until (and not including) the kth bit, $y[k:]$ denotes all bits from (including) k, $y[::2]$ denotes all even bits in y, and $y[1::2]$ denotes all uneven bits in y. Algorithms presented further in the paper similarly utilize Python-like pseudo code notation.

2.2 Boolean Masking and Threshold Implementations

To protect against side-channel analysis, we make use of the Boolean masking technique [CJRR99, GP99]. With s_x-shared masking, a secret bit $x \in \mathbb{F}_2$ is shared as a vector $\mathbf{x} = (x_0, ..., x_{s_x-1}) \in \mathbb{F}_2^{s_x}$ such that $\sum_{i=0}^{s_x-1} x_i = x$. In the case that $(X_0, ..., X_{s_x-2})$ is distributed uniformly random over $\mathbb{F}_2^{s_x-1}$, we call the sharing uniform.

When making calculations with shared variables, the computation should not undo the properties from the sharing. To that end, we make use of the properties by Nikova *et al.* [NRR06] called Threshold Implementation (TI).

Definition 1 (Properties of threshold implementations [NRR06])**.** Let $F : \mathbb{F}_2^n \to \mathbb{F}_2^m$ be a function and $\mathbf{F} : \mathbb{F}_2^{ns_x} \to \mathbb{F}_2^{ms_y}$ be a sharing of F. The sharing $\mathbf{F}$ is said to be

- *correct* if $\sum_{i=0}^{s_y-1} F_i(\mathbf{x}) = F(\sum_{i=0}^{s_x-1} x_i)$ for all $x_0, \ldots, x_{s_x-1} \in \mathbb{F}_2^n$,
- *non-complete* if F_i depends on at most $s_x - 1$ input shares,
- *uniform* if $\mathbf{F}$ maps a uniform random sharing of any $x \in \mathbb{F}_2^n$ to a uniform random sharing of $F(x) \in \mathbb{F}_2^m$.

2.3 Glitch-Extended Probing Security

We model side-channel security through the "glitch-extended probing model" [FGP+18], which is a security model based on the "probing model" introduced by Ishai *et al.* [ISW03].

Consider that a masked algorithm is represented as a circuit, a directed acyclic graph where each edge (a wire) represents a digital value (a bit) and each vertex is a binary operation. In the dth-order probing model, an adversary is able to observe d intermediate wires of the circuit during the execution of the algorithm. These d wires should be simulatable by a simulator that is not given any of the secret values (such as the key or plaintext of a block cipher). As a result, it suffices to show that the probed values are random values following some fixed distribution independent of the secret values.

In the context of the first-order security, the aforementioned concept comes down to placing a probe in a computational phase (i.e., the combinational logic), observing a set of registered values or initial inputs of the circuit (following the glitch-extended probing model, where a probe propagates back to the registered values or initial inputs), and proving that these sets of registered values are distributed in a way that the distribution itself is independent of any function of the secret input of the entire algorithm. In this work, we prove this kind of security by breaking the proof down into multiple parts:

1. we consider a circuit that consists of register layers and specific blocks (in the example of a cipher, the S-boxes and linear layers are these blocks);
2. we demonstrate that non-linear parts of these specific blocks are generated by the algorithm presented in this work;
3. the generated blocks of functions map a uniform masked input to a uniform masked output;

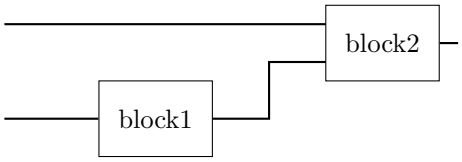


Figure 1: An example of a sequential composition.

4. we emphasize that if that block has a uniform input, then a probe in that block observes values whose distribution does not change if the input secret is changed (thus indicating probing security).

There is an important keyword in this explanation, namely, we show that the block maps a uniform input to a uniform output. This is not possible in the arbitrary composition of blocks, but is only possible with the sequential composition of blocks. We discuss more on sequential composability in Section 2.4, and the uniformity concerns have been previously discussed by Dhooghe *et al.* [DNR19].

In the *robust glitch-extended* probing model [FGP+18], a probe on a combinatorial circuit is extended to all signals (up to registers or primary inputs) involved in the probed wire's computation. The security of a gadget in this model follows from the previous paragraph's conclusion with the aforementioned extension of the probes. The introduction of such a simple model enabled the relevant scientific communities to develop formal verification tools to evaluate a small circuit [BBC+19, KSM20] and statistical tools to evaluate larger circuits [MM22]. Furthermore, it helped to reduce the implementation cost of several schemes while maintaining the same level of security.

2.4 Sequential Composability

The term *sequential composability* is less commonly used in the masking community; however, it is often sufficient for the masked realization of symmetric-key cryptographic algorithms. Following the work from Canetti [Can20], we can consider three types of composability: sequential, parallel, and concurrent. Concurrent composability is what is often referred to as arbitrary composability in masking. This is where an input variable is split and piped to two or more blocks. Sequential composition, which includes parallel composition following Canetti's stand-alone setting in the case of this work, consists of any combination of the following two scenarios: a second block uses the full output of a first block; a second block works with separate inputs compared to a first block. We give an example in Figure 1. We should mention specifically what is not considered sequential composability: when the number of outputs is greater than the number of inputs. In other words, we do not consider that a value is used twice or that a wire is split. When it comes to symmetric ciphers, or better said, permutations or compression functions, sequential composability is often sufficient to create a secure masking.

2.5 Techniques To Achieve Uniformity

Since the proposal of threshold implementations, the open question was posed to find probing secure sharings that are uniform. To that end, several solutions were put forward.

Brute-force techniques were developed by Bilgin *et al.* [BNN+12, BNN+15] to find threshold implementations for small functions. These techniques are combined with decomposition techniques to decompose the function into lower algebraic degree functions to lower the number of shares needed for the threshold implementation. More recently, the work by Shahmirzadi *et al.* [SM21a] relaxed the non-completeness property to probing security, where the brute-force search methods can be used to find sharings using the minimal number of shares using multiple clock cycles.

Fresh randomness is a popular method to achieve uniformity. By applying established methods such as TI [NRR06], one can construct a non-complete and probing-secure sharing, which is then refreshed with new randomness to ensure the output remains uniform. While this is a simple technique, it comes at the cost of additional (often expensive) randomness.

The *changing of the guards* technique was introduced by Daemen [Dae17], where the input of a neighboring S-box is XORed with the output of a non-complete S-box to achieve uniformity. While this technique requires some additional randomness to initiate the process and involves extra XORs, it is a straightforward method for achieving uniformity.

A universal construction was created by Piccione *et al.* [PAB+23] where $t+2$ shares are needed for a degree t function to provide a non-complete and uniform sharing. From all of the above techniques, only the *universal construction* is capable of providing an answer to the original question from threshold implementations which is to provide non-complete and uniform solutions for any function without using additional inputs. However, the technique potentially requires a lot of shares when the algebraic degree of the function is large while it only works for first-order security.

2.6 Masking with $d+1$ Shares

Threshold implementations can also be costly, due to the large number of input shares, which increase area requirements and the need for fresh randomness when combining functions or ensuring uniformity [CBR+15]. The number of shares depends on the algebraic degree of the target function, often leading to larger designs. To reduce costs, some studies decompose the function into multiple lower-degree functions, but ensuring uniformity in each function is challenging. Additionally, integrating non-linear functions requires register layers to maintain non-completeness, which can increase latency, especially with a large number of smaller functions.

Two separate studies have attempted to decouple the number of input shares from the algebraic degree [RBN+15, GMK16]. These studies introduced approaches that use $d+1$ input shares for d^{th}-order security while also ensuring glitch resistance in hardware with potentially lower implementation costs due to the reduced number of input shares. Typically, these methodologies require new randomness to ensure non-completeness, contrasting with threshold approaches that might necessitate fresh masks for uniformity. In the $d+1$ sharing technique, a masked Boolean function is segmented into two distinct sections by a register layer to prevent glitch propagation.

Drawing from the Domain Oriented Masking (DOM) method introduced by Groß *et al.* [GMK16], a two-share variant of a two-input AND gate $f(a,b) = ab = y$ can be realized as follows. The dashed vertical line separates the stages with a register, while the solid horizontal line highlights the domains.

$$\begin{array}{llll:l} f_0(a_0,b_0) & = a_0b_0 & \to x'_0 & \\ f_1(a_0,b_1,r) & = a_0b_1 + r & \to x'_1 & x'_0 + x'_1 = y_0 \\ \hline f_2(a_1,b_0,r) & = a_1b_0 + r & \to x'_2 & x'_2 + x'_3 = y_1 \\ f_3(a_1,b_1) & = a_1b_1 & \to x'_3 & \end{array}, \tag{1}$$

In the given equation, a_0, a_1, b_0, b_1 represent the uniformly distributed input shares, r is a single-bit of fresh randomness, and y_0, y_1 are the output shares. The functions f_l are referred to as *component functions*, and their outcomes are meant to be stored in registers x'_l. The product a_ib_j, where $i \neq j$, is referred to as a *cross-domain term*. As shown above, f_1 and f_2 are involved in the computation of these terms, since these functions operate on different share indices of their inputs. The segment that produces output shares via XOR operations on registered values is called the *compression layer*. A technique for realizing first-order $d+1$ hardware implementations without the need for new randomness has been presented in [SM21a], which is explained in Section 3.2 in more detail.

3 A Uniform Sharing in Multiple Register Stages

This section presents the main contribution of the paper, namely the masking approach to ensure uniformity for arbitrary functions. Our proposed technique guarantees first-order security and output uniformity without additional randomness across multiple register stages.

3.1 A Trivial Solution to Uniformity

Given an arbitrary function F, that maps to an output space smaller or equally sized to the input space, a trivial transformation to make a uniform sharing of F in two shares is to map (x_0, x_1) to $(f(x) + x_0, x_0)$, where x is shared uniformly, with x_0 and x_1 being independent shares. The following theorem serves as a starting point for this paper's masking techniques.

Theorem 1. *The masked function*

$$\tilde{F} : \mathbb{F}_2^{2n} \to \mathbb{F}_2^{2k} : \tilde{F}(x_0, x_1) = (f(x) + x_0[:k], x_0[:k])$$

of the function $f : \mathbb{F}_2^n \to \mathbb{F}_2^k$, with $k \leq n$, is uniform, where $x = x_0 + x_1$.

Proof. Each separate output share (x_0 or $f(x) + x_0$) is uniformly randomly distributed given that the input share x_0 was uniformly randomly distributed. Since $\tilde{F}$ maps a uniform input sharing to a uniform output sharing, the function is uniform. □

A gentle remark: the theorem states $k \leq n$. It is not possible to find uniform sharings that map to an output space that is larger in size than the input space.

3.2 The NullFresh Multiplication

While the sharing from Theorem 1 is evidently uniform, the question remains on how to securely calculate it since one of the output shares directly works on the input secret. To achieve the secure computation of the function in Theorem 1, we first recall the masked multiplier by Shahmirzadi *et al.* [SM21a]. Consider the uniform first-order two-shared AND gate operating on the uniformly masked independent input bits a, b, where f_i are stored in registers x'_i:

$$\begin{array}{lll} f_0(a_0, b_0) = a_0b_0 + b_0 & \to x'_0 & \\ f_1(a_0, b_1) = a_0b_1 & \to x'_1 & x'_0 + x'_1 = y_0 \\ \hline f_2(a_1, b_0) = a_1b_0 + b_0 & \to x'_2 & x'_2 + x'_3 = y_1 \\ f_3(a_1, b_1) = a_1b_1 & \to x'_3 & \end{array}, \tag{2}$$

If we look at Algebraic Normal Form (ANF) of each value, then we can write:

$$\begin{array}{lll} f_0(a_0, b_0) = a_0b_0 + b_0 & \to x'_0 & \\ f_1(a_0, b_1) = a_0b_1 & \to x'_1 & y_0 = x'_0 + x'_1 = a_0b_0 + a_0b_1 + b_0 \\ \hline f_2(a_1, b_0) = a_1b_0 + b_0 & \to x'_2 & y_1 = x'_2 + x'_3 = a_1b_0 + a_1b_1 + b_0 \\ f_3(a_1, b_1) = a_1b_1 & \to x'_3 & \end{array}. \tag{3}$$

The above can be represented as two multiplexers, where the multiplexers work over two cycles, with the shares of b as their inputs and the shares of a as their select signal since $y_0 = \overline{a_0}b_0 + a_0b_1$ and $y_1 = \overline{a_1}b_0 + a_1b_1$. The security of this paper's masking relies on that representation.

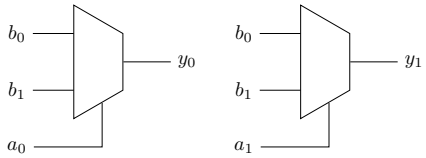


Figure 2: The masked AND gate from Eq. (3) is represented as multiplexers where the multiplexers work over two cycles (in parallel).

By examining Figure 2, a visual explanation of why the masked AND gate is probing secure is provided. Since the multiplexer is calculated over two cycles, essentially its select signal is pre-loaded, making the multiplexer act as a wire. Since each input wire only caries a single share of the secret b, it is evidently non-complete and thus probing secure. We generalize the above trick, which allows us to compute arbitrary functions by splitting them up as multiplexers.

3.3 Direct Sharing

In this work, we provide an algorithm to share arbitrary functions. This algorithm always starts by providing an expansion from a function to all non-complete products of its input's shares. This part of the algorithm is called "DirectShare" which is the naming used in the thesis by Bilgin [Bil15, Sec. 3.2.1]. In other words, the direct sharing algorithm is the one that provides a non-complete sharing in multiple output terms of a function for a single input variable. This algorithm works on a function f with n input bits and takes the index i which is the index it shares. The algorithm is outlined in Algorithm 1, which we refer to as $DirectShare_{FO}$ because it works with two shares and can be used to achieve first-order security. It works on the fact that for binary functions, each monomial term in its algebraic normal form either is multiplied by the term $x[i]$ or it is not, i.e., there are no higher degree terms in $x[i]$. Then, $x[i]$ is masked, and the split is done in the straightforward manner.

Algorithm 1: $DirectShare_{FO}$: Direct sharing with two shares. In this algorithm, the terms c and d are functions over the remaining variables $x[0], ..., x[i-1], x[i+1], ..., x[n-1]$.

Input: A function $f(x[0], ..., x[n-1]) \in \mathbb{F}_2$ and an index i
Output: Functions f_0 and f_1 such that $f_0 + f_1 = f$ and with f_0 working on $x_0[i]$ and f_1 on $x_1[i]$ with $x_0[i] + x_1[i] = x[i]$

1 Consider $f = c + x[i]d = c + (x_0[i] + x_1[i])d$
2 Set $f_0 = c + x_0[i]d$
3 Set $f_1 = x_1[i]d$
4 **return** $f_0 + x_0[i - 1 \ mod \ n]$, $f_1 + x_0[i - 1 \ mod \ n]$

3.4 A Solution in Multiple Stages without Randomness

In this section, we introduce our new masking technique, which guarantees first-order glitch-extended probing security and the uniformity of its output for arbitrary functions by sharing them over multiple register stages.

We start showcasing this masking technique via its application to an AND gate and compare it to the NullFresh solution from Section 3.2. Consider the function

$$f : (x[0], x[1]) \mapsto y = x[0]x[1]\,,$$

working on independent inputs. We wish to create the uniform output sharing $(y + x_0[0], x_0[0])$, with $x_0[0], x_1[0], x_0[1], x_1[1]$ being uniform input masks. We make use of

$DirectShare_{FO}$ which takes in a function $f(x[0], ..., x[n-1])$ and an index i and gives back two functions $f_0(x[0], ..., , x_0[i-1], x_0[i], x[i+1], ..., x[n-1])$ and $f_1(x[0], ..., , x_0[i-1], x_1[i], x[i+1], ..., x[n-1])$ where the ith variable is shared while $x_0[(i-1) \mod n]$ is added to both functions such that $f_0 + f_1 = f$. Note that n is the number of input bits. After the first application of $DirectShare_{FO}$ with $i = 0$, the following division is obtained using Algorithm 1 (monomial $x_0[0]$ is a part of $y + x_0[0]$):

$$f_0 = x_0[0]x[1] + x_0[0] + x_0[1]$$
$$f_1 = x_1[0]x[1] + x_0[1]$$

Here, the bit $x_0[1]$ serves as a mask for both equations. Subsequently, we still need to share the term $x[1]$, thus we apply another round of $DirectShare_{FO}$ to both functions, now $i = 1$:

$$f_{00} = x_0[0]x_0[1] + x_0[0] + x_0[1] + x_0[0]$$
$$f_{01} = x_0[0]x_1[1] + x_0[0]$$
$$f_{10} = x_1[0]x_0[1] + x_0[1] + x_0[0]$$
$$f_{11} = x_1[0]x_1[1] + x_0[0]$$

The resulting masking contains the full range of non-complete shares ready for the further compression. The blue terms $x_0[0]$ are added in accordance with Algorithm 1. Nevertheless, they are defined as redundant given that the sets were already non-complete. After removing these terms, the equations can be reformulated as a compression chain by essentially reversing the previously described steps. Thus, the following sharing in three register stages is acquired, where each vertical dashed line represents a register layer, the superscripts indicate the stage number, and the horizontal line separates the second share domain:

$$\begin{array}{l:l:l}
f^0_{00} = x_0[0]x_0[1] + x_0[0] + x_0[1] & & \\
f^0_{01} = x_0[0]x_1[1] & f^1_0 = f^0_{00} + f^0_{01} & \\
f^0_{10} = x_1[0]x_0[1] + x_0[1] & f^1_1 = f^0_{10} + f^0_{11} & f^2_0 = f^1_0 + f^1_1 \\
f^0_{11} = x_1[0]x_1[1] & & \\
\hline
f^0_{1^*} = x_0[0] & f^1_{1^*} = f^0_{1^*} & f^2_1 = f^1_{1^*}
\end{array} \tag{4}$$

Namely f^0_i, f^1_i, and f^2_i are stored in registers first, and then the registered values are processed in the next steps. In order to maintain the continuity of the flow, we revert back to the ANF of each value as follows:

$$\begin{array}{l:l:l}
f^0_{00} = x_0[0]x_0[1] + x_0[0] + x_0[1] & & \\
f^0_{01} = x_0[0]x_1[1] & f^1_0 = x_0[0]x[1] + x_0[0] + x_0[1] & \\
f^0_{10} = x_1[0]x_0[1] + x_0[1] & f^1_1 = x_1[0]x[1] + x_0[1] & f^2_0 = x[0]x[1] + x_0[0] \\
f^0_{11} = x_1[0]x_1[1] & & \\
\hline
f^0_{1^*} = x_0[0] & f^1_{1^*} = x_0[0] & f^2_1 = x_0[0]
\end{array}$$

The above looks a lot like the NullFresh solution in terms of multiplexers. Namely, we rewrite the above sharing so that the negation of certain bits is better shown.

$$\begin{array}{l:l:l}
f^0_{00} = \overline{x_0[0]}x_0[1] + x_0[0] & & \\
f^0_{01} = x_0[0]x_1[1] & f^1_0 = x_0[0]\overline{x[1]} + x_0[1] & \\
f^0_{10} = \overline{x_1[0]}x_0[1] & f^1_1 = x_1[0]x[1] + x_0[1] & f^2_0 = x[0]x[1] + x_0[0] \\
f^0_{11} = x_1[0]x_1[1] & & \\
\hline
f^0_{1^*} = x_0[0] & f^1_{1^*} = x_0[0] & f^2_1 = x_0[0]
\end{array}$$

Looking at the equations above, we observe that f^1_0 can be seen as the multiplexing of the inputs $x_0[1]$ and $x_1[1]$ depending on the $x_0[0]$ select signal. Similarly, f^2_0 can be seen as the multiplexing of $x_0[0]$ versus $x_1[0]$ using the secret value $x[1]$ as the select signal. To recall, due to this select signal the adversary's observed values from a probe are limited.

For example, probing f_0^2 either provides the share $x_0[0] + x_0[1], x_0[1]$ when $x[1] = 0$ or $x_0[1], x_1[0] + x_0[1]$ when $x[1] = 1$. An adversary can not distinguish between the two cases, thus successfully hiding $x[1]$, and the observed inputs of the multiplexer are non-complete. This security is formally proven in Theorem 2.

Compared to the NullFresh AND gate from Eq. (3), the above solution requires an additional register stage. However, the above equations already show a glimpse of how to automate the process for any Boolean function. Namely, working from the output to the input, each coordinate function is processed through a recursive iteration of Algorithm 1, thereby systematically exploring the full range of its input variables. The resulting list of functions is composed as a multiple stage compression chain with registers at the end of each step. The second share domain represents a pipeline for a single masked bit of input that is also applied as a mask for the according coordinate function from the first domain. Each coordinate function is masked with a different input bit, making the approach applicable to the functions with $k \leq n$, with n being a number of input bits and k - a number of output bits. We formalize the proposed method in Algorithm 2. Note that registered values are denoted by $||*||$ within the pseudocode. Furthermore, the registers in Lines 10 and 13 are optional; they are included to ensure the design is fully pipelined.

Algorithm 2: MultiStage *first-order* uniform sharing of g with **two** shares

Input: A function $g(x[0], ..., x[n-1]) \in \mathbb{F}_2$ and an arbitrary index $j \in \{0, \ldots, n-1\}$
Output: A masking $\tilde{g}(\tilde{x}) = (x_0[j] + g(x), x_0[j])$, where $(z^n[0], z^n[2^n])$ are the output shares

1 **Function** *Expand*(*f, i, j*)**:**
2 $f_0, f_1 = DirectShare_{FO}(f, i)$
3 $i = i - 1 \mod n$
4 **if** $i == j$ **then return** $[f_0 + x_0[i], f_1 + x_0[i]]$
5 **else return** $Expand(f_0, i, j) + Expand(f_1, i, j)$ `// list concat.`
6
7 **Algorithm 2:**
8 $z^0[:2^n + 1] = []$ `// empty list to update with the further expansion`
9 $||z^0[:2^n]|| += Expand(g + x_0[j], j, j)$ `// ||*|| - register the result`
10 $||z^0[2^n]|| = x_0[j]$ `// optional registers for full pipelining.`
11
12 **for** $c = 1$ **to** n **do**
13 $||z^c[:2^{n-c+1}:]|| = z^{c-1}[:2^{n-c+1}:2] + z^{c-1}[1:2^{n-c+1}:2]$
14 $||z^c[2^n]|| = z^{c-1}[2^n]$ `// optional registers for full pipelining.`

To demonstrate how to apply Algorithm 2, let's take function $g(x) = x[0]x[1]x[2] + x[0]x[1] + x[2]$ as a target and apply the algorithm with $j = 1$. Note that j can be chosen arbitrarily, as long as $j \in \{0, \ldots, n-1\}$. The output shares are $(g + x_0[j = 1], x_0[j = 1])$. In the first step, we apply the function $DirectShare_{FO}$ to $g + x_0[1]$, resulting in two functions described below with $i = 1$ (see Line 2 in Algorithm 2):

$$g_0 = x[0]x_0[1]x[2] + x[0]x_0[1] + x[2] + x_0[0] + x_0[1]$$
$$g_1 = x[0]x_1[1]x[2] + x[0]x_1[1] + x_0[0]$$

Then, we apply $DirectShare_{FO}$ to the both functions g_0 and g_1 with $i = 0$ resulting in the functions below:

$$g_{00} = x_0[0]x_0[1]x[2] + x_0[0]x_0[1] + x[2] + x_0[0] + x_0[2] + x_0[1]$$
$$g_{01} = x_1[0]x_0[1]x[2] + x_1[0]x_0[1] + x_0[2]$$
$$g_{10} = x_0[0]x_1[1]x[2] + x_0[0]x_1[1] + x_0[0] + x_0[2]$$
$$g_{11} = x_1[0]x_1[1]x[2] + x_1[0]x_1[1] + x_0[2]$$

Then, as the last step, the function $DirectShare_{FO}$ is applied to g_{00}, g_{01}, g_{10} and g_{11} with $i = 2$, leading to 8 functions as follow:

$$
\begin{aligned}
g_{000} &= x_0[0]x_0[1]x_0[2] + x_0[0]x_0[1] + x_0[2] + x_0[1] + x_0[0] + x_0[2] \\
g_{001} &= x_0[0]x_0[1]x_1[2] + x_1[2] \\
g_{010} &= x_1[0]x_0[1]x_0[2] + x_1[0]x_0[1] + x_0[2] \\
g_{011} &= x_1[0]x_0[1]x_1[2] \\
g_{100} &= x_0[0]x_1[1]x_0[2] + x_0[0]x_1[1] + x_0[0] + x_0[2] \\
g_{101} &= x_0[0]x_1[1]x_1[2] \\
g_{110} &= x_1[0]x_1[1]x_0[2] + x_1[0]x_1[1] + x_0[2] \\
g_{111} &= x_1[0]x_1[1]x_1[2]
\end{aligned}
$$

As one can see, in the last step, $x_0[1]$ will not be added to all component functions due to the addition of $x_0[i]$ in Line 4 of Algorithm 2. Now, we have all the component functions ready, and then we compress them pairwise in multiple stages to get the masked design. The final design is as follows, where vertical dashed lines are the register layers:

$$
\begin{array}{l:l}
g^0_{000} = x_0[0]x_0[1]x_0[2] + x_0[0]x_0[1] + x_0[2] + x_0[0] + x_0[2] + x_0[1] & \\
g^0_{001} = x_0[0]x_0[1]x_1[2] + x_1[2] & g^1_{00} = g^0_{000} + g^0_{001} \\
g^0_{010} = x_1[0]x_0[1]x_0[2] + x_1[0]x_0[1] + x_0[2] & g^1_{01} = g^0_{010} + g^0_{011} \\
g^0_{011} = x_1[0]x_0[1]x_1[2] & \\
g^0_{100} = x_0[0]x_1[1]x_0[2] + x_0[0]x_1[1] + x_0[0] + x_0[2] & \\
g^0_{101} = x_0[0]x_1[1]x_1[2] & g^1_{10} = g^0_{100} + g^0_{101} \\
g^0_{110} = x_1[0]x_1[1]x_0[2] + x_1[0]x_1[1] + x_0[2] & g^1_{11} = g^0_{110} + g^0_{111} \\
g^0_{111} = x_1[0]x_1[1]x_1[2] & \\
\hline
g^0_{1^*} = x_0[1] & g^1_{1^*} = g^0_{1^*}
\end{array}
$$

$$
\begin{array}{l:l}
g^2_0 = g^1_{00} + g^1_{01} & g^3_0 = g^2_0 + g^2_1 \\
g^2_1 = g^1_{10} + g^1_{11} & \\
\hline
g^2_{1^*} = g^1_{1^*} & g^3_1 = g^2_{1^*}
\end{array}
$$

Table 1: Complexity of the proposed masking technique with examples of S-boxes.

	General formulas	(1)	(2)	(3)	(4)
Shares	d	2	2	2	2
Input bits	n	4	4	4	8
Func. degree	t	3	3	3	7
Unmasked monoms.*	$m_i, (i \leq n)$	26	22	24	1009
Comp. func.	d^n	16	16	16	256
Stages	$n+1$	5	5	5	9
$\oplus$'s	$d\sum_{i=1}^{t} d^{i-1} - d^n + \sum_{i=1}^{n} d^i m_i$	144	123	97	24304
$\odot$'s	$\sum_{i=2}^{n}(i-1)d^i m_i$	188	164	88	100820

* by order i, we provide the total in the table;
(1) PRINCE; (2) MIDORI; (3) SKINNY; (4) AES

Latency and area complexity. The latency of the proposed algorithm depends on the number of input bits. The area depends on the number and structure of component functions and grows exponentially. The relations are described in Table 1, which includes some examples considered later in the paper.

Masking techniques comparison. Due to differences in the underlying approaches, comparing the complexity relations of the proposed method in Table 1 with those of other existing techniques appears challenging. For instance, while TI defines the number of input shares as $td + 1$, there's no guarantee of achieving uniform sharing without randomness. Universal construction [PAB+23] based on TI with $t + 2$ shares does not provide a general theoretical analysis of area overhead, focusing on AES example. Most papers that propose the use of composable gadgets like HPC1, HPC2 (both [CGLS21]), HPC3 [KM22], HPC3o [CGM+24], HPC4 [CSV24], AND-XOR [WFP+24] and others discuss the implementation complexity of the AND gate, rather than an entire function, focus on fresh randomness requirement and sometimes implement asynchronous logic. One of the competing algorithms that resembles our MultiStage approach in terms of its application to the entire function and its provision of a two shared output in form of $\tilde{g}(\tilde{x}) = (r + g(x), r)$ with r representing a fresh mask is the Generic Hardware Private Circuits (GHPC) [KSM22] algorithm. In contrast to our solution, it allows arbitrary composability and constant latency of 1 or 2 clock cycles. However, GHPC works only for first-order security and requires fresh randomness. The minimal number of fresh masks required is directly proportional to the number of output bits of a masked function m. In order to achieve reduced latency, this number should increase up to $m * 2^n$, where n stands for the number of input bits of the function, making our solution more appealing. Area benchmarks of S-boxes masked using our approach are comparable to those masked with GHPC. Another round of comparison based on the performance benchmarks provided in Section 4.7.

Glitch-extended probing security of Algorithm 2 We start with the lemma that summarizes the security provided by Algorithm 2. In short, the lemma states that two shares y and z can be safely added together given that the two shares are masked by the value $x_0[\ell - 1]$ (which does not appear anywhere else in their algebraic normal form) and either y is uniquely masked by $x_0[\ell]$ or z is. This final requirement is achieved by a "multiplexer" form which there is a term $\overline{a}x_0[\ell]$ in y and a term $ax_1[\ell]$ in z for some bit a ($f(\Omega')$ in the lemma).

Lemma 1. *Consider the variables $(x[0], ..., x[n-1]) \in \mathbb{F}_2^n$ and consider their uniform two-sharing $\Omega = ((x_0[0], x_1[0]), ..., (x_0[n-1], x_1[n-1])) \in \mathbb{F}_2^{2n}$, then any probe placed within the operation $y + z \in \mathbb{F}_2$ only views information independent of the secret variable, with*

$$y = g(\Omega') + \overline{f(\Omega')}x_0[\ell] + x_0[\ell - 1] \qquad z = h(\Omega') + f(\Omega')x_1[\ell] + x_0[\ell - 1]$$

for

$$\Omega' = \Omega \setminus \{x_0[\ell], x_1[\ell], x_0[\ell - 1], x_1[\ell - 1]\} \cup \{x[\ell], x[\ell - 1]\},$$

some $\ell \in \{0, ..., n-1\}$ and some functions $f, g, h : \mathbb{F}_2^{2n-2} \to \mathbb{F}_2$. Note that we follow Python's wrapping meaning that $x_0[-1] = x_0[n-1]$.

Proof. Recall that for probing security, it is sufficient that for any choice of secret values the probed values' distribution is the same (and thus can be simulated from scratch). First, it is clear that y and z are separately probing secure since both are masked by the variable $x_0[\ell - 1]$ (thus, their separate distributions follow the uniform distribution no matter the choice of secrets). We prove that the distributions of y and of z are independent which results in both values being jointly probing secure.

$$
\begin{aligned}
\Pr[Y,Z] &= \Pr[Y,Z|f(\Omega')=0]\Pr[f(\Omega')=0] + \Pr[Y,Z|f(\Omega')=1]\Pr[f(\Omega')=1] \\
&= \Pr[Y|f(\Omega')=0]\Pr[Z|f(\Omega')=0]\Pr[f(\Omega')=0] + \\
&\quad \Pr[Y|f(\Omega')=1]\Pr[Z|f(\Omega')=1]\Pr[f(\Omega')=1] \\
&= \Pr[f(\Omega')=0]/4 + \Pr[f(\Omega')=1]/4 \\
&= 1/4 \\
&= 1/4(\Pr[f(\Omega')=0] + \Pr[f(\Omega')=1])(\Pr[f(\Omega')=0] + \Pr[f(\Omega')=1]) \\
&= \Pr[Y]\Pr[Z]\,,
\end{aligned}
$$

where the second equality comes from the distributions being independent when either $f(\Omega')=0$ or $f(\Omega')=1$. For example, when $f(\Omega')=0$, we have the following sharing

$$
\begin{aligned}
y &= g(\Omega') + x_0[\ell] + x_0[\ell-1] \\
z &= h(\Omega') + x_0[\ell-1],
\end{aligned}
$$

where y and z are clearly independent since y is refreshed by $x_0[\ell]$, but z does not use $x_0[\ell]$ or $x_1[\ell]$. The third equality holds since Y and Z are uniform when $f(\Omega')=0$ or when $f(\Omega')=1$.

□

Theorem 2. *Given a uniform input sharing, Algorithm 2 is 1st-order glitch-extended probing secure.*

Proof. Probing any $x_i[j]$ is possible, but it leads to observing only one share of each input bit, which means an incomplete set of shares is observed, which is insufficient to reveal any information about the secret. As an alternative in the algorithm, a probe can be placed on the values in z^c for $c \in \{0,\ldots,n\}$. We start by noting that $z^n[1]$ and any value in z^0 are clearly glitch-extended probing secure since they are non-complete in the uniform input. For the remaining values $z^c[i]$, each variable is itself uniformly random distributed no matter the input secret since it can be written as

$$
z^c[i] = h(\{x_0, x_1\} \setminus \{x_0[c], x_1[c]\}) + x_0[c]\,,
$$

meaning that it can be written as a function that does not depend on the shares of the variable $x[c]$ while being remasked by the uniform share $x_0[c]$. What is left to show is that the addition $z^c[i] = z^{c-1}[2i] + z^{c-1}[2i+1]$ is glitch-extended probing secure.

From Algorithm 2, we see that $z^{c-1}[2i], z^{c-1}[2i+1]$ is the direct sharing of $z^c[i]$ in the variable $j+n-c$ and that $z^{c-1}[2i]$ is added by $x_0[j+n-c]$. As a result, the addition follows the form of Lemma 1, proving the security of the addition. □

As a closing remark for this section, we should highlight that we have proven the security of our algorithm under the glitch-extending probing model, where transitional leakage and crosstalk-like effects were not taken into account. Hence, we make no claim regarding transitional and crosstalk leakage. To account for these effects, the masking approach must be refined accordingly. In any masked design, it is a recommended practice to avoid supplying the same input for two consecutive cycles; this is particularly critical for implementations where transitional leakage was not explicitly addressed.

4 Case Studies

In this section, we apply the algorithms from Section 3 to mask several cryptographic primitives including Midori, Prince, Skinny, Keccak, and AES. We remind that all of the introduced algorithms are aimed to provide a glitch-extended probing secure and uniform

sharing for an arbitrary non-linear function. Since our algorithms provide a first-order secure masked variant of any given function, we developed a tool that receives the function as a lookup table and generates the masked variant as HDL files for further use. Affine functions are straightforward to mask in Boolean masking, as the masked variant can be realized by applying the linear part of the function to each share, with the constant added to the first share.

The performance benchmarks are aggregated in Table 2 for further comparison with related works. It includes automatically generated versions of chosen ciphers together with several handcrafted examples. Some of those examples rely on the uniformity of the masking by design, while others use fresh randomness, which is noticeable from the *Fresh* sub-column of Table 2. Randomness used for masking is split into two parts for the reader's convenience. Initial randomness is applied once to mask the plaintext or both the plaintext and the key. The values in *Init.* column of Table 2 accounts for the total number of initial bits required, whereas *Fresh* randomness is applied at every clock cycle. We denote that numbers for the area, which are measured in gate equivalences (GEs), and time delay for the critical path of the circuit (nanoseconds) are calculated based on different cell libraries implying that any single gate, *e.g.* NAND-gate, has its own area, power or signal delay propagation parameters for a chosen library. We refer the reader to those libraries for a more thorough comparison. Randomness and latency requirements are independent of the later conditions. To highlight how fresh randomness requirements may affect the total area of a design combined with a PRNG (pseudo-random number generator) and compare the resulting overhead, we provide additional area numbers as if the design was using Bivium or Trivium for randomness generation with respect to the amount of random bits required per clock cycle. We synthesize those PRNGs as explained in [CMM+23], using NANGATE 45nm cell library. Given that this is the only library at our disposal and that there is no possibility to re-synthesize all other designs using it, combined area numbers for some of the works are provided by incorporating results of different libraries. Table 2 also illustrates a masking type that was primarily used for each referenced implementation.

4.1 PRINCE

PRINCE [BCG+12] is a lightweight cipher with a 16-nibble state arranged in a 4x4 square and a 128-bit key extended to 192 bits. Two 64-bit parts are used as whitening keys, and the rest as the round key. The state function includes 4-bit S-boxes, `ShiftRows`, and `MC` on each column. PRINCE has 12 rounds, with the first half in a "forward" direction and the last half in "backward," using inverse operations, so both the S-box $F(a, b, c, d)$: `BF32AC916780E5D4` and its inverse `B732FD89A6405EC1` are employed.

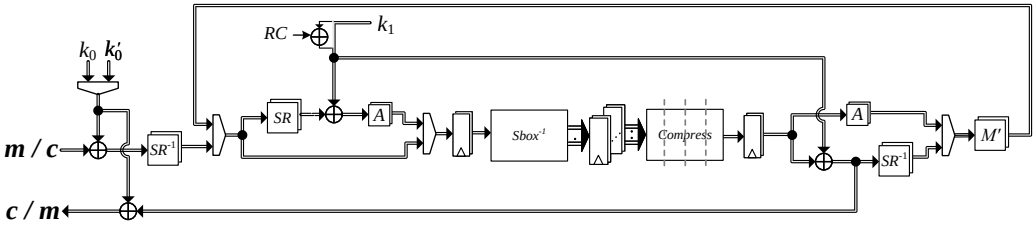


Figure 3: First-order secure round-based design of PRINCE based on Section 3.4.

First-order secure design In this paper, we use the fact that the PRINCE S-box and its inverse are affine equivalent as $S = A \circ S^{-1} \circ A$ with
A:`B8A93021EDFC6574`: $\langle x = a + b + d + 1, \quad y = a + 1 \quad z = d, \quad t = c + 1 \rangle$,
where (d, c, b, a) are the input bits of the function A and (t, z, y, x) are the output bits (with d, t - most significant bits; a, x - least significant bits). As a result, we used our tool, which

Table 2: Performance figures of different implementations, using Synopsis Design Compiler. We indicate with $*$ that the design was automatically generated.

Cipher	Design	Latency (cycles)	Rand. (bits) *Init.*[5]	Rand. (bits) *Fresh*[6]	Delay (ns)	Area (kGE) |	Area (kGE) †	Area (kGE) ‡	Mask. Type
Prince	[MS16][7]	40	128	0	4.00	9.29[1]	9.29	9.29	[TI]
	[BKN19]	24	192	112	8.37	13.19[1]	15.44	16.63	[DOM]
	[SM21a]	24	192	0	5.74	11.46[1]	11.46	11.46	[NullFresh]
	[VDB+24]	12	192	288	2.06	13.69[3]	20.39	24.11	[TSM]
	this work	72	192	0	1.45	24.9[3]	24.9	24.9	[MultStage]
Midori-64	[MS16][7]	32	128	0	4.00	7.30[1]	7.30	7.30	[TI]
	[SM21a][7]	32	64	0	4.99	7.56[1]	7.56	7.56	[NullFresh]
	[KM22]	85	192	512	1.18	14.91[3]	24.02	29.05	[HPC3]
	[KMMS22]*	51	192	64	0.85	30.54[3]	31.98	32.7	[GHPC]
	[KMMS22]*	153	192	256	1.10	17.8[3]	22.52	25.08	[HPC2]
	[KMMS22]*	153	192	256	0.95	54.3[3]	59.02	61.58	[HPC2]
	this work	96	192	0	1.11	21.5[3]	21.5	21.5	[MultiStage]
Skinny-64-64	[BJK+16]	704[4]	256	0	0.95	2.0[1]	2.0	2.0	[TI]
	[KM22]	99	128	128	1.06	6.47[3]	8.99	10.34	[HPC3]
	[KMMS22]*	99	128	64	0.80	28.85[3]	30.29	31.01	[GHPC]
	[KMMS22]*	165	128	64	0.45	20.08[3]	21.52	22.24	[GHPC]
	[KMMS22]*	165	128	64	0.53	20.2[3]	21.64	22.36	[HPC2]
	[WFP+24]*	132	128	128	0.45	13.7[3]	16.22	17.57	[AND-XOR1]
	[WFP+24]*	132	128	64	0.50	15.8[3]	17.24	17.97	[AND-XOR2]
	this work	192	128	0	0.52	18.7[3]	18.7	18.7	[MultiStage]
Keccak-f [1600]	[BDN+14]	25	3200	4	1.69	116.6[3]	117.06	117.25	[TI]
	[BDN+14]	24	4800	0	1.70	139.4[3]	139.4	139.4	[TI]
	[GSM17]	72	1600	0	0.84	111.8[2]	111.8	111.8	[DOM]
	[SM21b]	72	1600	0	1.29	129.3[2]	129.3	129.3	[NullFresh]
	[ZZL+24]	24	1600	2	1.31	83.5[3]	83.93	84.09	[custm.]
	this work	96	1600	0	0.56	162.2[3]	162.2	162.2	[MultiStage]

[1] UMC 180 library, [2] UMC 130 library, [3] NANGATE 45 library, [4] Nibble-serial impl.,
[5] Plaintext and key (if shared) mask bits, [6] Fresh masks per every cycle, [7] With no key masking
| w/o PRNG, † with Bivium PRNG, ‡ with Trivium PRNG.

applies Algorithm 2, to generate a first-order secure masked variant of the S-box inverse with no randomness. We used the same architecture for the rest of PRINCE as [SM21b], which is an architecture where the S-box is implemented using affine transformations in composition with the S-box inverse. The masked realization of the linear part is pretty straightforward, as mentioned earlier where the same function is applied to each share individually. The general structure of our fully-pipeline round-based first-order secure design is depicted in Figure 3. A register is placed at the input of the S-box inverse to limit the propagation of probes within the S-box inverse. There is a register layer at the output of the coordinate functions (the input of the compression layer), as usual. The entire compression layer has a latency of 5 cycles since PRINCE uses 4-bit S-boxes. For other details on the S-box complexity, we refer the reader to Table 1. Note, that a register must be placed at the output of the S-box inverse to ensure glitch-extended probing security, as the affine function A mixes multiple output bits from the S-box inverse. Hence, each round of PRINCE is performed in 6 cycles, completing the encryption in a total of 72 cycles. We confirmed the first-order security of our masked S-box inverse using SILVER and evaluated the entire design with PROLEAD and TVLA, indicating the first-order security of our designs.

Among the designs listed in Table 2, the only first-order secure design using two shares without fresh randomness has been proposed in [SM21a]. However, the non-linear layer in the design is non-uniform, making it potentially vulnerable to SCA analysis. We should highlight that this uniformity issue was addressed in [WFW+25], where the linear layer is replaced by a non-linear layer to achieve a uniform and probing-secure solution for both the S-box and its inverse. Since the area overhead of their PRINCE implementation is not reported, we cannot perform an exact comparison with our designs; however, Section 4.2 of that work indicates that the entire design incurs a larger area overhead compared to [SM21a]. The proposed design in [BKN19] uses the DOM methodology and, therefore, requires 112

bits of randomness per S-box, making it heavily dependent on fresh randomness. Due to the use of randomness, placing some registers is unnecessary, resulting in better latency compared to our work. The work in [MS16] employs classical TI to achieve first-order security, where the S-box is decomposed into quadratic functions, thus requiring three shares. While the area overhead of our design is larger than the state of the art, it requires no fresh randomness. To make Algorithm 2 generalized for any function, we made more conservative assumptions, which led to the introduction of extra registers in the design and consequently increased latency compared to handcrafted designs that are highly optimized for PRINCE.

Key schedule masking In the realm of cryptographic hardware security, protecting the main encryption and decryption rounds of a block cipher from side-channel attacks is a well-understood necessity. However, a comprehensive security strategy must extend this protection to a critical, and sometimes overlooked, component: the key schedule. In almost all realistic scenarios, an attacker can control the plaintext and observe the ciphertext while the encryption key remains fixed. Consequently, traditional dynamic power analysis methods like Differential Power Analysis (DPA) [KJJ99] or Correlation Power Analysis (CPA) [BCO04] become inapplicable on the key schedule. However, a plain implementation of the key schedule opens the door for template attacks [CRR03]. At its core, a template attack is a profiled attack. This means the attacker first dedicates significant effort to characterizing the physical behavior of a device before ever attacking the actual target. The goal of the profiling stage is to create a highly accurate statistical model – a "template" – of the clone device's power consumption for every possible secret key fragment the attacker wants to target. Then, the attacker takes the captured trace on the target device and compares it against their library of pre-built templates. The template that best fits the victim's trace reveals the value of the secret subkey byte. To avoid such an attack, the key schedule of the block cipher needs to be masked as well. Since the key schedule of PRINCE is a linear function, we apply the key schedule function to each share individually. While this would lead to slightly higher area overhead, it would result in higher security guarantees.

Glitch-extended probing security We discuss the glitch-extended probing security of the masked PRINCE. The security of the masked S-box has already been addressed in Section 3.4. However, we discuss the security of the entire masked cipher.

Lemma 2. *Consider a round of a masked cipher which operates on a uniformly masked two-shared input. The linear functions are masked share-wise, and the non-linear layer is shared following Algorithm 2 and separated from the linear layers by registers. Then, this masked round is first-order glitch-extended probing secure.*

Proof. We consider an adversary that can place a single probe. A probe can only be placed in either a single masked S-box or a linear/affine layer (where both functions are walled-off by registers). In other words, a probe placed in any function will have its propagation, under the glitch-extended probing model, confined to that same function due to the registers placed at the input and output of the S-box. Thus, a probe placed within the S-box propagates only through the S-box, and a probe placed in the affine layer remains within that function, propagating at most to all its inputs. Given that the S-box is shared with Algorithm 2, which is a uniform function, and given that the share-wise masking of an invertible linear layer is again uniform, all masked functions are supplied with uniform inputs. Placing a probe in a share-wise function with a uniform input provides a distribution independent of the round's secret input. The security stemming from placing a probe in the non-linear layer is guaranteed by Theorem 2. □

The entire masking of PRINCE is then probing secure since it is a sequential composition (see Section 2.4) of a uniform masking of its round function. Our masked design of PRINCE follows the description from Lemma 2, thus we consider its masking secure.

4.2 Midori

In this section, we mask the MIDORI-64 block cipher as introduced by Banik *et al.* [BBI+15]. The MIDORI-64 (named MIDORI from here on out) consists of a 64-bit state, split into 16 nibbles, together with a 128-bit key. The key schedule is linear consisting of choosing either the most significant 64-bits, the least significant ones, or the sum of the two together with the sum of a round-dependent constant. The state function consists of 4-bits S-boxes (`CAD3EBF789150246`) on the 16 nibbles, a permutation between the nibbles, and the application of a matrix to four nibbles each. It is important to mention that there are some security issues for this cipher as discussed in [TLS16]. We have selected this cipher to showcase our masking technique as its S-box has been used in other block ciphers like CRAFT [BLMR19] and MANTIS [BJK+16].

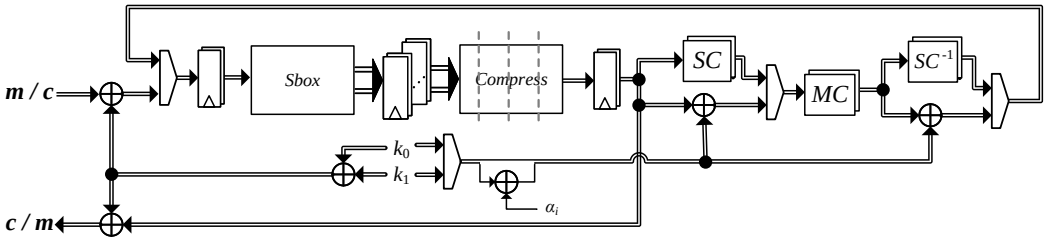


Figure 4: First-order secure round-based design of MIDORI-64.

First-order secure design Similar to PRINCE, we used our tool to generate the first-order secure masked version of MIDORI S-box. For the entire cipher, we opted for the architecture presented by Shahmirzadi and Moradi [SM21a]. We added the automatically generated first-order secure S-box in this architecture, where the schematic of the design is depicted in Figure 4. The only other part that had to be adjusted manually is the control logic for the key schedule and round constants. Note that since the linear layer does not mix different output bits of the same S-box, placing a register layer at the output of the S-box is unnecessary for first-order design. Nevertheless, we keep the register at the output to make the paper consistent. Each round of MIDORI is performed in 6 clock cycles.

The performance results of our implementation are shown in Table 2. As one can see, the manually crafted designs in [MS16] and [SM21a] outperform our design in terms of latency and area overhead while requiring no randomness at all. However, our design outperforms the gadget-based implementations that are generated automatically with the use of AGEMA [KMMS22] in terms of overall randomness requirements and area, when a lightweight PRNG is considered alongside the masked MIDORI.

Key schedule masking Since the key schedule of MIDORI is linear, the masking function is applied to each share individually. We report the implementation costs of designs with the masked key schedule in Table 2.

Glitch-extended probing security The security argument follows a similar pattern to the one of Lemma 2 for PRINCE. We omit a detailed discussion here for conciseness, as the security proofs are similar to PRINCE.

4.3 SKINNY

SKINNY is a tweakable block cipher [BJK+16], where the state consists of 16 nibbles, with a simple affine tweakey schedule and a state function using nibble-wise cubic S-boxes and lightweight matrix multiplication. We implement the 64-bit state and 64-bit tweakey version.

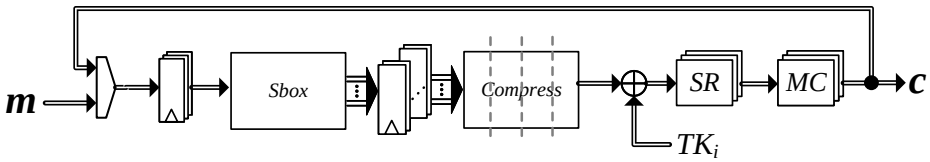


Figure 5: First-order secure round-based design of SKINNY.

First-order secure design SKINNY uses a 4-bit S-box, where two coordinate functions have an algebraic degree of two and two other coordinate functions have an algebraic degree of three. We used our tool to generate the first-order masked realization of the S-box with no randomness and added it into the architecture shown in Figure 5. It is necessary to place a register at the input of the S-box, performing each round in 6 clock cycles.

In the original publication [BJK+16], the authors provided a three-share first-order secure design of SKINNY in a byte-serial architecture. Hence, the latency is high while the area overhead is quite low (see Table 2). We are not aware of any other work presenting first-order secure design apart from automatically generated designs using AGEMA [KMMS22] and AGMNC [WFP+24]. As reported in Table 2, our design significantly outperforms those implementations in terms of randomness requirements, slightly surpasses HPC-based designs in terms of area overhead, and underperforms improved AND-XOR combinations with respect to the same metric. Our design requires one fewer input share than the original publication [BJK+16].

Key schedule masking The key schedule of SKINNY is a straightforward linear function, which allows its operations to be applied directly to each share.

Glitch-extended probing security Due to the properties of the linear layer combining only a single bit of a masked S-box, the final register layer of the masked S-box can be removed for the first-order variant, similar to the masking of MIDORI. However, to maintain consistency, that register is retained. The security argument for the masked SKINNY is then the same as for the masked MIDORI and PRINCE, based on Lemma 2.

4.4 Keccak

KECCAK by Bertoni *et al.* [BDPA] is a family of hash functions (standardized as SHA-3) based on the KECCAK-$f[b]$ permutation where $b = r + c$ is the state size of the permutation with r as its rate and c as its capacity for the sponge construction. KECCAK-$f[b]$ consists of several permutations between the bits of its state, xors between the state bits, the application of round constants, and the application of a five-bit S-box. The different sizes of the state b affect only the number of rounds and round constants used in KECCAK function. Thus, our automatically generated S-box will suit to any of those implementations with the minimal number of shares $d + 1$. The schematic layout of KECCAK-$f[b]$ is shown in Figure 6.

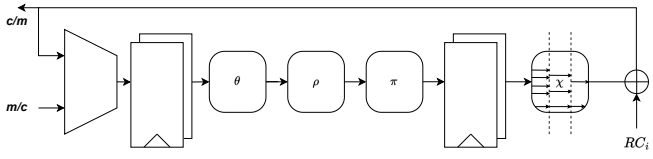


Figure 6: Masked KECCAK–$f[b]$ architecture using the masking from Section 3.4.

First-order secure design KECCAK S-box has interesting characteristics compared to other case studies we have investigated so far. As mentioned before, it has a five-bit S-box while

the degree of its function is equal to two. Additionally, each output bit is affected by only three bits of input and the ANF of all coordinate functions is the same with different input bits. Hence, we just applied Algorithm 2 to one coordinate function using our tool, and we get five instances of it to make the KECCAK S-box (χ function). The component functions as a result of applying the algorithm are given below for one coordinate function, where $(a_0, a_1, d_0, d_1, e_0, e_1)$ are the input bits and the ANF is $de + d + a$.

$$x_0(a_0, d_0, e_0) = d_0e_0 \oplus a_0 \oplus d_0 \oplus a_0 \oplus d_0 \oplus e_0 \qquad x_1(a_0, d_0, e_1) = d_0e_1$$
$$x_2(a_0, d_1, e_0) = d_1e_0 \oplus d_1 \oplus e_0 \qquad x_3(a_0, d_1, e_1) = d_1e_1$$
$$x_4(a_1, d_0, e_0) = a_1 \oplus d_0 \oplus e_0 \qquad x_5(a_1, d_0, e_1) = 0$$
$$x_6(a_1, d_1, e_0) = e_0 \qquad x_7(a_1, d_1, e_1) = 0$$

As one can see, some component functions are either zero or equal to a correction term used for further compression (see above in green). Thus, unnecessary area and latency are introduced. We can optimize the masked version by starting with only four shares instead of eight and compressing first on the e variable, followed by both a and d. Hence, we can reduce the implementation to just four component functions, $x_0(.)$ to $x_3(.)$, and thereby realize the first-order secure KECCAK χ function with a latency of three clock cycles. This example demonstrates that our algorithm is not necessarily the optimal solution, and further optimization is possible depending on the function. To make the approach universally applicable to any function, we made conservative assumptions that may sacrifice efficiency.

We adopted the same structure as in [SM21b] to implement a first-order secure KECCAK permutation. One register layer is placed at the input and another at the output of the S-box to ensure glitch-extended probing security. Note that the KECCAK linear layer mixes the output bits of the same S-box, and a register layer at the S-box output is essential. Performance results can be found in Table 2.

Glitch-extended probing security Due to the linear layer combining multiple bits per S-box, the output of the masked S-box has to be registered. Thus, the security argument for the masked KECCAK follows the same lines as for the masked PRINCE.

4.5 AES with Tower-Field Decomposition

AES-128 is a NIST-standardized cipher [DR01] with a 128-bit state and key, each divided into 16 bytes in a square structure. It consists of 10 rounds, with each round applying an S-box to each byte, `ShiftRows`, and matrix multiplication on the state's columns.

It has become a common practice for the past couple of decades to implement a masked AES S-box by utilizing Canright's S-box [Can05] based on the tower field decomposition. The aforementioned approach gave birth to several implementations pursuing dissimilar goals: latency reduction, degree reduction for non-linear functions, and randomness reduction. The reason for this decomposition is an overhead in complexity given by sharing the degree-seven function. We quickly recall the tower-field decomposition of the AES S-box as described by Canright [Can05]. In the tower-field decomposition, the S-box is computed using $GF(2^4)$ multiplications and $GF(2^4)$ inversion together with linear operations such as the input and output isomorphism and square-scale operation. The decomposition is shown in Figure 7. In the following, we explain how we use our method to create a first-order secure masked S-box and how we achieve a first-order secure byte-serial implementation using it.

4.5.1 Input and Output Isomorphisms

We have used the same functions that are employed in Canright's design [Can05]. Since these functions are linear, we instantiate them twice and apply them to each share individually. Note that we should place a register layer right after the input affine layer as well as before

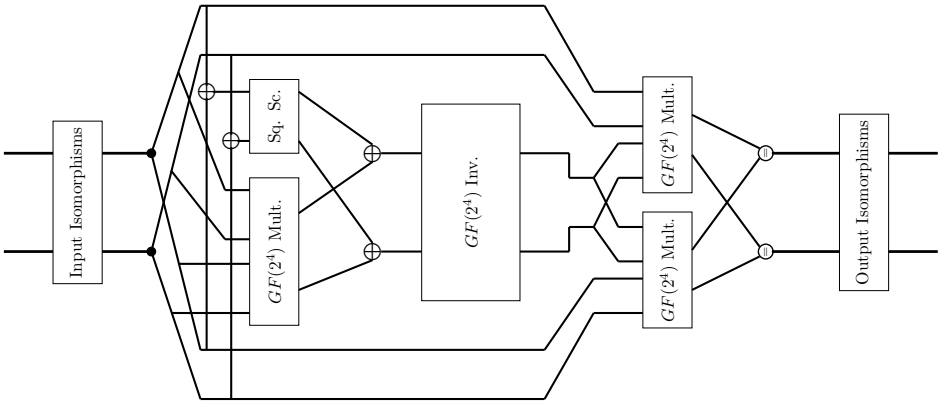


Figure 7: Representation of the AES S-box with the tower-field decomposition.

the output affine layer to prevent the propagation of probes under the glitch-extended probing model and achieve security under this model.

4.5.2 Multiplier

The first non-linear operation in the tower-field decomposition is the $GF(2^4)$ multiplier. There are three $GF(2^4)$ multipliers in total, one after the input isomorphisms and two multipliers at the end before the output isomorphism.

To achieve a first-order realization of $GF(2^4)$, we can utilize Equation 4 in Section 3.4. In that section, we presented a first-order realization of a two-input AND operation in three stages. However, this approach can be extended to any field multiplication, with the security proofs remaining valid. For the sake of completeness, in this section, we present a uniform sharing of the $GF(2^n)$ multiplier in three stages similar to the AND operation. While our design of a first-order secure AES S-box specifically requires a $GF(2^4)$ multiplier, we provide a generalized discussion here.

Consider the function

$$f : \mathbb{F}_{2^n}^2 \to \mathbb{F}_{2^n} : (a, b) \mapsto ab\,.$$

Consider the algebraic normal form of the kth bit of f

$$f^k = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} a[i]b[j] = \sum_{j=0}^{n-1} M_j^k(a)b[j]\,, \tag{5}$$

for some linear Boolean functions M_j^k.

There are several secure solutions for the three stages, we provide one example. Denoting $ab[k]$ the kth output bit of the $GF(2^n)$ multiplication, the following sharing for f^k is first-order probing secure.

$z_0^0 = a_0b_0[k] + M_0^k(a_0) + b_0[0]$		
$z_1^0 = a_0b_1[k]$	$z_0^1 = a_0b[k] + M_0^k(a_0) + b_0[0]$	
$z_2^0 = a_1b_0[k] + b_0[0]$	$z_1^1 = a_1b[k] + b_0^{sum}$	$z_0^2 = ab[k] + M_0^k(a_0)$
$z_3^0 = a_1b_1[k]$		
$z_4^0 = M_0^k(a_0)$	$z_2^1 = M_0^k(a_0)$	$z_1^2 = M_0^k(a_0)$

Looking to Eq. (5), we can rewrite the above equations as follows.

$$
\begin{array}{lll}
z_0^0 = \overline{M_0^k(a_0)}b_0[0] + \sum_{j=1}^{n-1} M_j^k(a_0)b_0[j] + M_0^k(a_0) & & \\
z_1^0 = M_0^k(a_0)b_0[0] + \sum_{j=1}^{n-1} M_j^k(a_0)b_1[j] & z_0^1 = M_0^k(a_0)\overline{b[0]} + \sum_{j=1}^{n-1} M_j^k(a_0)b[j] + b_0[0] & \\
z_2^0 = \overline{M_0^k(a_0)}b_0[0] + \sum_{j=1}^{n-1} M_j^k(a_1)b_0[j] & z_1^1 = M_0^k(a_0)b[0] + \sum_{j=1}^{n-1} M_j^k(a_1)b[j] + b_0[0] & z_0^2 = \sum_{j=1}^{n-1} M_j^k(a)b[j] + M_0^k(a_0) \\
z_3^0 = M_0^k(a_1)b_0[0] + \sum_{j=1}^{n-1} M_j^k(a_1)b_1[j] & & \\
\hline
z_4^0 = M_0^k(a_0) & z_2^1 = M_0^k(a_0) & z_1^2 = M_0^k(a_0)
\end{array}
$$

Following Lemma 1, it is clear that the above sharing is robust probing secure since it follows the typical multiplexer structure. The above sharing also results in a uniform sharing when applied to all bits in the multiplication. This is due to the Boolean function $M_0 : \mathbb{F}_{2^n} \to \mathbb{F}_{2^n} : a \mapsto (M_0^0(a), ..., M_0^{n-1}(a))$ being equal to the multiplication $a \cdot v$, with $v = (10...0)$ the zero-vector with a 1 on the first position, it is a permutation of a. As a result, the first share of the full multiplication outcome is equal to the secret ab plus $M_0(a_0)$ and since $M_0(a_0)$ is a permutation of a_0 where (a_0, a_1) was a uniform sharing, the output sharing is uniform.

The above sharing can be adapted freely by changing M_0 to M_j or by adding a different $b_0[j]$ for any $j \in \{0, ..., n-1\}$ to have different output shares. It thus follows that $a_0^{M^k} = (a_0 \cdot (1...1))[k]$ the kth bit of the multiplication with the constant $(1...1)$. Similarly, the mapping from a_0 to $(a_0^{M^0}, ..., a_0^{M^{n-1}})$ is invertible. Thus applying the above sharing to each bit results again in a uniform mapping, making the sharing both robust-probing secure and uniform.

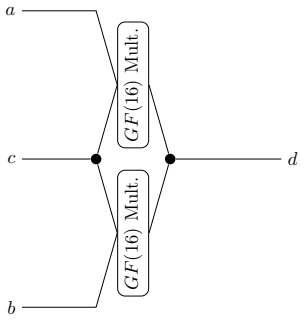


Figure 8: View of the final multipliers AES S-box.

The above strategy is based on the rationale of creating a uniform $GF(2^4)$ multiplier. Focusing on the final multipliers of the tower-field decomposition, as shown in Figure 8, let's assume that (a, b, c) are uniform (we will discuss in Section 4.5.5 how to achieve this). For the two parallel multipliers, following the notation from Figure 8, the first multiplier outputs the shares $(M_0(a_0), ac + M_0(a_0))$, and the second multiplier outputs $(M_0(b_0), bc + M_0(b_0))$. Since (a_0, a_1, b_0, b_1) are jointly uniform and M_0 is a permutation, the output remains uniform.

4.5.3 $GF(2^4)$ Squre-Scale Function

Since this function is linear, its sharing can be done share-wise. In other words, the identical function is directly applied to each individual share of the masked input.

4.5.4 $GF(2^4)$ Inversion

We have utilized the same $GF(2^4)$ inversion function as in Canright's design [Can05]. This function can be viewed as a 4-bit S-box, similar to those S-boxes in PRINCE, MIDORI, and SKINNY, as discussed in previous sections. We then used our tool to generate its first-order secure masked realization with two shares and no fresh masks.

Table 3: Performance figures of first-order AES S-box, using Synopsis Design Compiler excluding RNGs. The automatically generated designs are marked with $*$.

Cipher	Design	No. of Shares	Latency (cycles)	Rand. (bits/S-box)	Delay (ns)	Area (kGE) |	Area (kGE) †	Area (kGE) ‡	Mask. Type
AES S-box	[MPL+11]	3	4	48	-	4.2[1]	6.83	8.16	[TI]
	[BGN+14]	4	2	44	-	3.70[1]	6.27	7.55	[TI]
	[BGN+15]	3	2	32	-	2.84[1]	5.22	6.39	[TI]
	[BGN+15]	3	2	44	-	3.65[1]	6.22	7.5	[TI]
	[CRB+16]	2	5	54	-	2.0[2]	4.73	6.12	[CMS]
	[GMK16]	2	8	18	-	2.6[1]	4.74	5.79	[DOM]
	[GMK16]	2	5	28	-	2.8[1]	5.11	6.25	[DOM]
	[UHA17]	2	5	64	-	1.42[3]	4.31	5.81	[TI]
	[WM18]	4	16	0	-	4.20[1]	4.20	4.20	[TI]
	[Sug19]	3	4	0	-	3.50[2]	3.50	3.50	[TI]
	[MRB18]	2	5	19	-	1.69[2]	3.85	4.91	[Mult.Mask]
	[SBHM20]	2	1	36	-	3.48[2]	5.43	7.13	[LMDPL]
	[CGLS21]	2	8	34	-	2.19[2]	4.6	5.79	[HPC3]
	[KM22]	2	4	68	-	1.85[2]	4.81	6.34	[HPC2]
	[PAB+23][4]	9	1	0	-	166.37[1]	166.37	166.37	$[t+2]$
	[PAB+23][5]	5	2	0	-	22.05[1]	22.05	22.05	$[t+2]$
	[VDB+24]	2	1	262	2.7	20.5[2]	26.77	30.17	[TSM]
	[HB25]	2	3	32	-	1.88[2]	4.26	5.43	[HPC3.1]
	[HB25]	2	4	34	-	1.91[2]	4.32	5.51	[HPC3.1]
	[HB25]	2	4	29	-	1.81[2]	4.14	5.29	[HPC1]
	[KMMS22]*	2	8	68	0.40	4.26[2]	7.22	8.75	[HPC1]
	[KMMS22]*	2	8	34	0.51	5.34[2]	7.75	8.94	[HPC2]
	[WFP+24]*	2	6	66	0.49	2.89[2]	5.81	7.34	[AND-XOR1]
	[WFP+24]*	2	6	33	0.55	3.96[2]	6.35	7.53	[AND-XOR2]
	[CGM+24]*[6]	2	4	46	-	2.78[2]	5.38	6.68	[HPC3o]
	[CGM+24]*[6]	2	6	34	-	3.34[2]	5.75	6.94	[HPC3o]
	[CGM+24]*[7]	2	4	36	-	1.95[2]	3.9	5.6	[HPC3o]
	[KSM22]	2	2	8	-	77.15[1]	77.68	77.9	[GHPC]
	[KSM22]	2	1	2048	-	64.11[1]	99.74	119.47	[GHPC$_{LL}$]
	this work^C	2	13	4	0.77	4.17[2]	4.63	4.82	[MultiStage]
	this work^O	2	9	0	0.17	71.6[2]	71.6	71.6	[MultiStage]

[1] UMC 180 lib, [2] NANGATE 45 lib, [3] TSMC 65 lib; [4] without decomp., [5] with decomp.;
[6] Boyar-Peralta repr. [BP12], [7] Canright repr. [Can05] (best rand. and area options);
| w/o PRNG, † with Bivium PRNG, ‡ with Trivium PRNG;
^{O} With no decomposition., ^{C} Tower-field approach.

4.5.5 First-Order Secure AES S-box

Having uniform and glitch-extended probing secure constructions for all the components—namely, $GF(2^4)$ inversion, $GF(2^4)$ multipliers, input and output isomorphisms, and square-scale—we can now build a first-order secure AES S-box. In the composition of these components, we ensure that the inputs of each component originate from registered values. More specifically, the output of the input isomorphism is stored in registers, which are then fed into the $GF(2^4)$ multiplier and square-scale. Furthermore, the outputs of the $GF(2^4)$ multiplier and the square-scale function are also stored in registers. Thus, the inputs of the subsequent XOR operations (see Figure 7) are registered values, and we also store their outputs in registers, which then serve as inputs for the $GF(2^4)$ inversion.

Based on Figure 8, we denote the output of the $GF(2^4)$ inversion by c, and the upper and lower branches by a and b, respectively. A closer look at Figure 7 reveals that the 8-bit input is mapped to 12-bit (a, b, c), which are then fed into the last two parallel multipliers. Since the output size is larger than the input, we need to introduce 4-bit randomness at the output of the $GF(2^4)$ inversion solely to ensure (a, b, c) are jointly uniform. We should highlight that the values refreshed by new randomness are also stored in registers. The uniformity of the last two multipliers is discussed in Section 4.5.2. We store the results of the last two multipliers in registers, and the final operation performed is the output isomorphism. In summary, the design incorporates a register layer after the input isomorphism, three register layers for the $GF(2^4)$ multiplier, one register layer for the XOR result of the square-scale function with the $GF(2^4)$ multiplier, four register layers for the $GF(2^4)$ inversion, a register layer after refreshing the output of the $GF(2^4)$ inversion, and three register layers for the

final two multipliers. This brings the total latency of the S-box to 13 cycles.

Each individual component is uniform and first-order secure, and we ensure that registers are placed at all interconnections. We also confirm that each component receives uniform input shares. A probe in any part of the design does not propagate beyond the component itself. Since the security of each component has been discussed in previous subsections, the entire S-box design is therefore first-order secure. We further confirmed the security and uniformity of our complete AES S-box design with SILVER [KSM20].

We refer to Table 3 for a comparison with the state of the art. As one can see, even with the area overhead of a PRNG, our tower-field design outperforms most of the other works in terms of area overhead. The main reasons are the use of the minimal number of input shares, i.e., two shares, and the minimal randomness requirement. The latency, on the other hand, is on the higher end due to the fact that we require minimal randomness and must place registers to ensure security under the glitch-extended probing model.

4.5.6 First-Order Secure AES Encryption

To ensure a fair comparison with the state of the art, we implemented the AES encryption function using a byte-serial architecture. The complete AES encryption process requires 290 clock cycles. An overview of the data path for this design is presented in Figure 9.

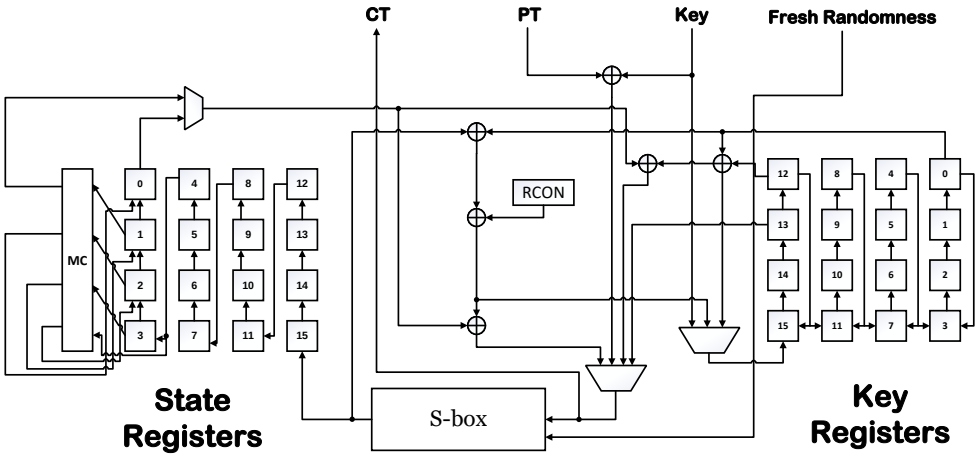


Figure 9: Architecture of Our Serialized AES Encryption Implementation.

Both the state and key are represented as 4×4 byte arrays, where the byte at row i and column j is indexed as $4j + i$. The initial 16 clock cycles are dedicated to byte-wise loading of the plaintext and key into the module. Concurrently, the `AddRoundKey` operation is performed, and its output immediately feeds into the S-box. Subsequently, four bytes of the key are processed by the S-box over the next four cycles. During this period, the key registers are temporarily disabled, awaiting the output of the first key-byte S-box computation. Once this S-box result for the key schedule is available, the next round's key byte is generated and added to its corresponding state byte. This newly generated key byte is then stored in the key registers for use in the subsequent round.

The `ShiftRows` operation is applied immediately after the last state byte exits the S-box, which is not explicitly depicted in Figure 9 for diagram simplicity. The MixColumns operation is executed in the subsequent clock cycle. Notably, this design allows for continuous operation: a new key and plaintext can be loaded simultaneously with the byte-wise output of the ciphertext.

Unlike case studies so far, the key schedule in AES is a non-linear operation. The key schedule operates iteratively, generating new 32-bit words that constitute the round keys.

Key transformations involve specific sub-components: `RotWord` cyclically shifts the bytes of a 32-bit word, `SubWord` applies the non-linear AES S-box to each byte of a word for substitution, and `Rcon` (Round Constant) provides an XORed value that introduces variety and prevents attacks based on symmetry. In our implementation, we use the same AES S-box for `SubWord` opearion and apply the linear operations like `RotWord` to each share indvisually.

In terms of randomness complexity, the S-box requires 4 bits of randomness per cycle. Consequently, the design requires 128 bits for sharing the plaintext, 128 bits for sharing the key, and $4 \times 290 = 1160$ bits of randomness for the S-box operations. As a result, our design necessitates a total of 1416 random bits per encryption, including the sharing of the plaintext and key.

Table 4: Performance figures of first-order byte-serial AES implementation.

(using Synopsis Design Compiler, and UMC 180 standard cell library, no compile_ultra)

Cipher	Design	Latency (cycles)	Rand. (bits) *Init.*[v]	Rand. (bits) *Fresh*[w]	Delay (ns)	Area(GE) |	Area(GE) †	Area(GE) ‡	Mask. Type
AES	[WM18]	2804	384	0[b]	-	7600	7600	7600	[TI]
	[MPL+11]	266	512	48	-	11114	12293	12869	[TI]
	[BGN+14]	246	512	44	-	9102	10217	10758	[TI]
	[BGN+15]	246	512	44	-	11221	12336	12877	[TI]
	[BGN+15]	246	512	32	-	8119	9041	9479	[TI]
	[Sug19]	266	512	0[b]	-	17100[c]	17100	17100	[TI]
	[GMK16]	216	256	28	-	7600	8457	8859	[DOM]
	[GMK16]	246	256	18	-	7100	7793	8105	[DOM]
	[CRB+16]	276	256	54	-	6681[c]	7956	8588	[CMS]
	[UHA17]	219	128	64	-	6321[d]	7756	8479	[TI]
	[SM21a]	246	256	1	6.25	7136	7564	7730	[NullFresh]
	[SM21a]	246	256	0	6.25	7707	7707	7707	[NullFresh]
	[SM21a]	246	128	1	5.74	6247	6675	6841	[NullFresh]
	[SM21a]	246	128	0	5.74	6818	6818	6818	[NullFresh]
	[KMMS22]*	2043	256	34	0.91	42078[c]	43034	43489	[GHPC]
	[KMMS22]*	1135	256	136	0.91	25656[c]	28316	29737	[GHPC$_{LL}$]
	[KMMS22]*	3859	256	414	2.56	33124[cn]	40558	44657	[HPC2]
	[KMMS22]*	2043	256	34	0.98	42146[c]	43192	43557	[HPC2]
	this work	290	256	4	1.12	8805[c]	9266	9451	[MultiStage]

[v] Plaintext and key (if shared) mask bits; [w] Fresh masks per cycle
[b] Using changing of the guards; [c] Using NANGATE 45; [d] Using TSMC 65
[n] no pipelining; * Automatically generated design
| w/o PRNG; † with Bivium PRNG; ‡ with Trivium PRNG

We refer to Table 4 for a comparison of our design with the state of the art. Our design was synthesized using the NANGATE 45nm Library with the Synopsys Design Compiler tool, similar to other case studies. For security reasons, we also used a compile option that avoided any optimization across modules. Our design significantly outperforms masked implementations based on PINI gadgets, such as GHPC, HPC2, and GHPC$_{LL}$. The randomness requirement is at least one order of magnitude lower, and the area overhead is 3–4 times smaller. The latency is also considerably better. Compared to non-PINI designs like DOM and TI, our design is quite comparable, and in most cases, exhibits lower randomness requirements. Note that the S-box in [SM21a] is non-uniform, which could potentially lead to security vulnerabilities.

4.6 AES S-box without Decomposition

While using the tower-field approach makes the implementation costs of masked design lower, our approach allows us to generate the S-box straightforwardly in nine stages, avoiding solving complex mathematical tasks for secure decomposition. The other benefit is that

the technique allows to use the minimal number of shares for the linear layer with no fresh randomness. The main drawback, on the other hand, is a substantial increase in area and latency compared to most competitors.

Table 3 shows the performance figures of our design. Our masked AES S-box outperforms most state-of-the-art designs in terms of randomness requirements, as no fresh masks are used. However, other automatically generated design techniques allow for significantly better area cost and latency with the example from [CGM+24] leading the way.

An approach introduced in [PAB+23] achieves first-order secure and uniform sharing of any given function based on TI, enabling the development of a tool. Hence it is fair to compare our design with the outcome of this paper. The technique in [PAB+23] defines the number of input shares as $t + 2$ for first-order security without randomness, where t is the algebraic degree of the target function. As seen in Table 3, the design without decomposition is significantly larger than ours due to the use of 9 input shares. The authors also decomposed the AES S-box into two cubic functions, applying the technique to reduce the area footprint with 5 input shares, though this comes at the cost of doubling the latency. In contrast, our design uses the minimum number of shares, i.e., two shares, making the AES design potentially smaller, as the linear layer needs to be instantiated only twice, rather than 9 or 5 times.

4.7 A Comparative Overview

In this section, we discuss the proposed method in general and highlight its key differences from other techniques.

Our method, which we refer to as MultiStage, can work with *arbitrary functions*. It requires no fresh masks for first-order security. Before this study, most methodologies that work with arbitrary functions were based on composability notions and focused primarily on secure two-input AND gadgets trivially composable under the PINI notion, such as HPC1 [CGLS21], HPC2 [CGLS21], HPC3 [KM22], and HPC4 [CSV24]. The key difference of our approach is that we reduce the randomness complexity of the designs, but the masked version of the target function is non-PINI. Therefore, it should be carefully checked when composing these gadgets.

For small S-boxes like Prince, Skinny, and Midori, our approach appears to offer advantages in terms of area overhead, particularly when considering the area overhead of the PRNG. Since we use no fresh masks for first-order security, our designs exhibit improved area efficiency. The latency is on par with or better than methods using composability notions, depending on the specific technique employed.

The area overhead of our algorithm grows exponentially with the number of input bits. Consequently, the direct application of our method to larger S-boxes, such as the AES S-box, would lead to a substantial area overhead. However, breaking it down into smaller sub-functions, as in a tower-field approach, could yield promising results, making it smaller than almost all other S-boxes using composability notions, albeit requiring more latency in terms of clock cycles.

TI [NRR06] is often considered the first method when it comes to comparison with non-PINI techniques. It defines the number of input shares as $td+1$, where t is the algebraic degree of the target function and d is the desired security order. Thus, the minimum number of input shares is three. However, the main drawback is that there is no guarantee that the masked version is uniform with no fresh masks, which is crucial for the composability of the masked realization. For example, no first-order secure AND operation has been reported that is both probing-secure and uniform with three shares. This represents the main difference between our method and TI: we guarantee both probing security and uniformity.

A more equitable comparison would be with the technique introduced in [PAB+23], which achieves first-order secure and uniform sharing of arbitrary functions based on TI, where the number of input shares is defined as $t+2$. Looking at Table 3, we can observe that the direct application of our method to AES S-box would lead to a twofold improvement

in area overhead, though it incurs higher latency. With decomposition of the S-box, our design is approximately five times smaller, again with higher latency.

DOM [GMK16] is another non-PINI method that uses $d+1$ inputs and adds fresh masks into the design, ensuring it is always probing secure and uniform. However, its randomness complexity grows exponentially with the number of inputs, meaning that the area overhead also grows exponentially. The methodology to achieve first-order security with no fresh randomness has been introduced in [SM21a], using $d+1$ input shares. However, the methodology is limited to cubic functions, and there is no guarantee it leads to a solution. For example, no uniform sharing has been reported there for the PRINCE S-box. As one can observe, our method works with arbitrary functions, which is a clear distinction from this technique. Furthermore, unlike DOM, our method does not require fresh randomness for first-order security.

Overall, manually-crafted designs using non-PINI methods outperform our designs in terms of area overhead in this paper, as they are highly optimized for each specific case. The latency of our method is $n+1$ cycles for an n-bit S-box. The latency in other papers varies depending on the technique used. For example, when using TI, the S-box is usually decomposed into quadratic functions, so the latency depends on how many quadratic functions are needed in a row to construct the target S-box. A general remark can be made, however, that manually-crafted designs exhibit latency on par with or better than our designs.

As a final note, we should highlight that "changing of the guards", introduced in [Dae17], is just a technique to achieve uniformity and hence is not a masking scheme. It cannot be used to solve any security issues with the probing model; it can only transform non-uniform but probing-secure designs into a uniform sharing. This technique differs from our method by nature, as our method provides both probing security and uniformity.

5 Verification

5.1 PROLEAD Experiments

PROLEAD [MM22] is a tool for statistically verifying the glitch- and transition-extended probing security of a given netlist. The tool was recently enhanced [MM24] by inclusion of "robust but relaxed" probing model, which is out of the scope of this work. We use the modified version for better performance with the robust probing model. It performs hypothesis tests based on the G-test of independence [SR81] to verify whether the probed values' distribution is different between taking a fixed input or a randomly chosen one. This makes the test similar to TVLA [CDG+13] which is commonly performed on a device to test the security of masking. In contrast to hardware tests, PROLEAD verifies the security of the given design against probing adversaries. While several other verification tools are available, PROLEAD can handle the security of larger designs, detect flaws related to glitches, and work directly on the generated netlist. We note that while PROLEAD is a useful tool for sanity check implementations, a leakage-free result by the tool does not guarantee security on hardware as it is not a formal verification tool. For this guarantee, we checked the first-order security of our S-box constructions using SILVER [KSM20] and provided security proofs. We also conducted TVLA on FPGA in Section 5.2 to provide an extra level of confidence.

The verification results of PROLEAD are accumulated in Table 5, demonstrating how many clock cycles of each implementation were tested, the memory and time required for one test, and the number of traces taken per experiment. The overall assessment result is added in the last column, showing *OKAY* if the p-value of the G-test is below 5 and **does not grow gradually** or *LEAKAGE* otherwise. According to the works [MM22] and [ZZL+24], the absence of continuous growth is of crucial importance in the verification of the absence of leakage. For the experiments, we used an 8-core Intel® Xeon® Gold 6244 Processor allowing 16 simultaneous threads with 256 GB of RAM.

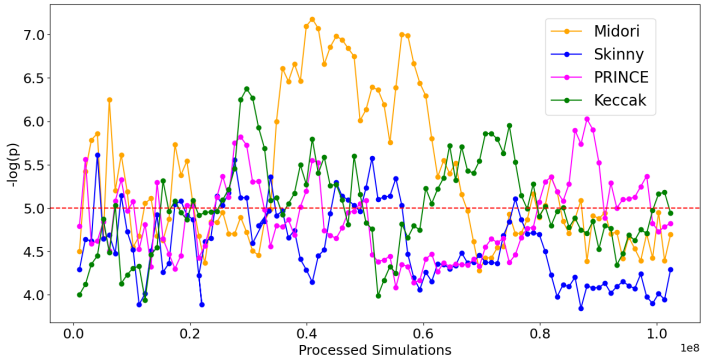


Figure 10: PROLEAD evaluation results.

The choice of our test scenarios is based on the available resources to us, the reliability of different modes of PROLEAD, and design structures. We evaluated our designs under the glitch-extended probing model and verified the absence of leakage for all rounds of PRINCE, MIDORI and SKINNY (see Figure 10), as well as all cycles for the AES S-box and AES byte-serial implementation (see Figure 11). The resulting size of the masked KECCAK-f[1600] function makes it feasible to run tests only for 2.5 rounds of the permutation. We observe that p-value slightly exceeds the threshold sometimes, but goes down again during 100 million experiments. It is possible due to the large number of probing sets (around 10^5) or some particular fixed set values, and it is important not to observe a gradually growing leakage, which is absent in our case. In the next section, we verify the absence of leakage in proposed designs by means of TVLA.

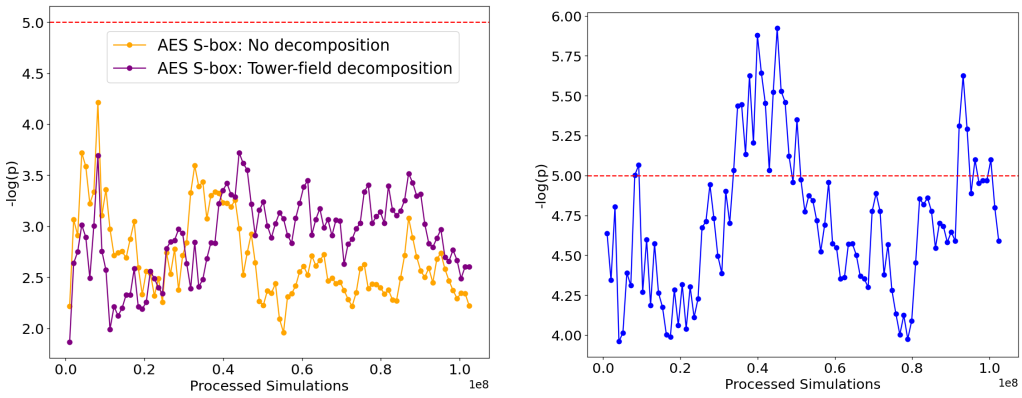
5.2 Experimental Analysis

For the experimental security evaluation, we rely on a widely used approach called TVLA [CDG+13], which has been a standard in the field for the past decade. The test aims to determine if a device's power consumption during encryption depends on secret values. While it doesn't reveal the secret or suggest how to exploit the vulnerability, it shows potential leakage points. The experiments exploited a Sakura-G board [SAK] designed for side-channel evaluation and a DPO7254C oscilloscope. The control FPGA handles randomness generation (AES-128) and input sharing, while the main FPGA runs

Table 5: Results of PROLEAD's tests (compact mode) in the robust-probing model for the different masked designs.

Cipher	Cycles tested	RAM (GB)	Time (hours.)	#Traces (millions)	Result
Prince	72	23.1	28.3	100	OKAY
Midori-64	80	23.7	28.9	100	OKAY
Skinny-64-64	160	41.5	59.3	100	OKAY
Keccak-f[1600]	10	239.9	23.4	100	OKAY
AES S-box[1]	9	46.5	2.9	100	OKAY
AES S-box[2]	13	3.2	1.8	100	OKAY
AES (byte-serial)[2]	290	21	28.3	100	OKAY

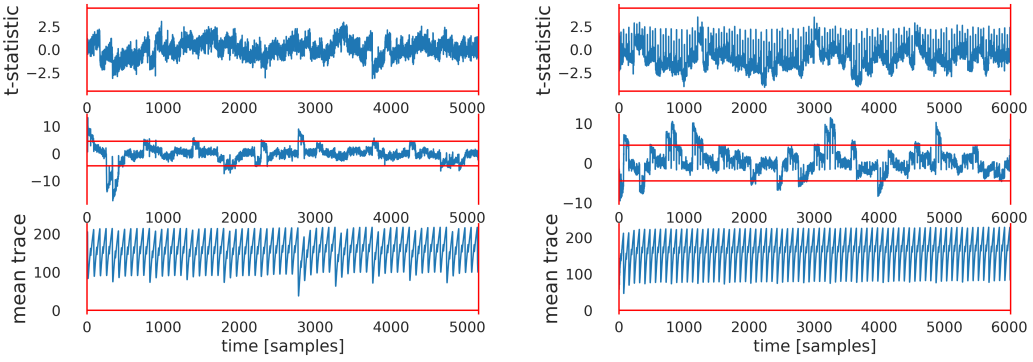
[1] Without decomposition, [2] With S-box decomposition



(a) AES S-box, 1st-order security

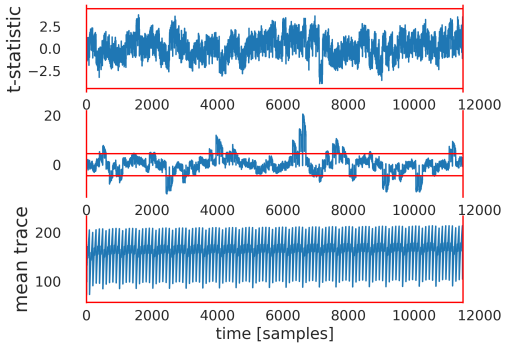
(b) AES byte-serial, 1st-order security

Figure 11: PROLEAD evaluation results for the first-order secure AES S-box, with and without decomposition, and for the byte-serial AES design.



(a) PRINCE, 1st-order security

(b) MIDORI-64, 1st-order security



(c) SKINNY-64-64, 1st-order security

Figure 12: TVLA results, first-order security and the mean traces.

the tested ciphers. Both FPGAs use a 6.144 MHz clock. FastFrames were collected from the oscilloscope at 500 Ms/sec, with an amplifier connected to the main board output via a 1Ω shunt resistor.

We collect 100 million traces to demonstrate the security of the proposed countermeasure. To explicitly demonstrate that we tested first-order secure designs, we provide the second-order TVLAs as well that are supposed to show leakage. The results are depicted in

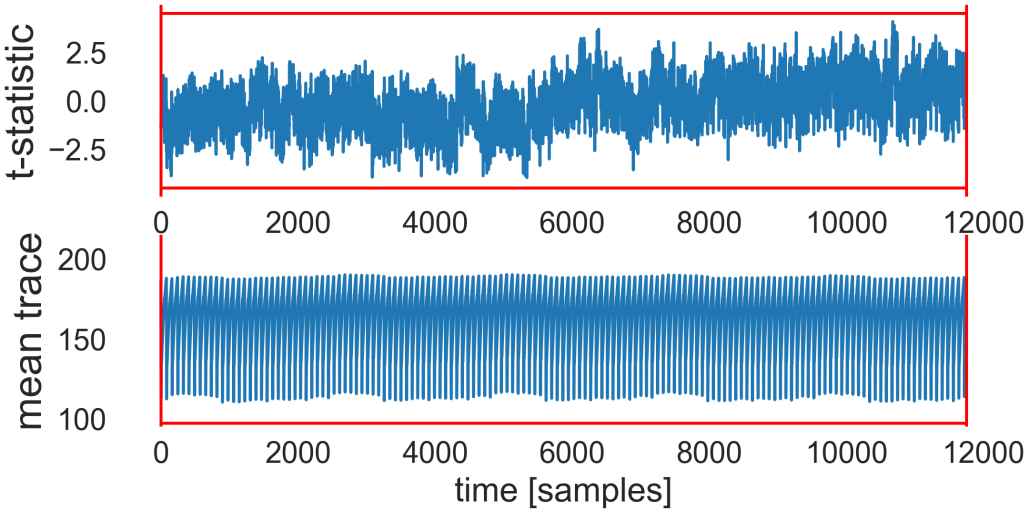


Figure 13: TVLA results (100 million traces), first-order security and the mean trace of five rounds of the byte-serial AES.

Figures 12(b), 12(a), 12(c) for Midori, Prince and Skinny, respectively. As expected, the first-order TVLA comparing means difference of fixed and random input sets shows no leakage for three designs, while the second-order comparison of variances highlights its presence. We do not provide TVLA data for Keccak-f[1600] and the AES without decomposition, since the first cipher is infeasible to process on Sakura-G FPGA, and the masking of the AES S-box is not possible to synthesize with Xilinx ISE software. Therefore, we stick to the formal results of PROLEAD and SILVER for these designs. TVLA results for five rounds of the byte-serial AES implementation are provided in Figure 13.

6 Conclusion and Future Works

In this work, we proposed a masking technique for the first-order sharing of arbitrary functions over multiple stages such that the sharings are uniform and require no additional randomness. The masking technique is an automatization of the NullFresh masking technique by Shahmirzadi *et al.*, namely while NullFresh was not evident to apply to arbitrary functions and required a brute force searching technique, our masking technique works for any function. The masking technique was then applied to popular cryptographic primitives and implemented on FPGA where both a tool-based verification via PROLEAD was performed as well as test vector leakage assessment.

Considering future work, we note that there are tools that automatically generate masking schemes from an insecure netlist or a template such as AGEMA [KMMS22] and EasiMask [BSG23]. It is the current belief that arbitrary composable security notions are necessary in the building of such tools, however, the techniques from this work can allow for such automatization using only sequential composable notions. Namely, the work from this paper can directly be used in the generation of masked S-boxes for the EasiMask tool and future research for the automatization of threshold implementations using the AGEMA tool could prove to be very interesting[2].

[2]This stems from the observation that a Moor (or Mealy) machine consist of the sequential application of the same combinatorial logic, meaning that the masking of this combinatorial logic using a serial composable notion is sufficient for security.

Acknowledgments. This work was supported by the Flemish Government through the Cybersecurity Research Program with grant number: VOEWICS02.

References

[BBC+19] Gilles Barthe, Sonia Belaïd, Gaëtan Cassiers, Pierre-Alain Fouque, Benjamin Grégoire, and François-Xavier Standaert. maskVerif: Automated Verification of Higher-Order Masking in Presence of Physical Defaults. In *ESORICS 2019*, volume 11735 of *Lecture Notes in Computer Science*, pages 300–318. Springer, 2019.

[BBI+15] Subhadeep Banik, Andrey Bogdanov, Takanori Isobe, Kyoji Shibutani, Harunaga Hiwatari, Toru Akishita, and Francesco Regazzoni. Midori: A Block Cipher for Low Energy. In *ASIACRYPT 2015*, volume 9453 of *Lecture Notes in Computer Science*, pages 411–436. Springer, 2015.

[BCG+12] Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knezevic, Lars R. Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, Peter Rombouts, Søren S. Thomsen, and Tolga Yalçin. PRINCE - A low-latency block cipher for pervasive computing applications - extended abstract. In *ASIACRYPT 2012*, volume 7658 of *Lecture Notes in Computer Science*, pages 208–225. Springer, 2012.

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation Power Analysis with a Leakage Model. In *CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 16–29. Springer, 2004.

[BDN+14] Begül Bilgin, Joan Daemen, Ventzislav Nikov, Svetla Nikova, Vincent Rijmen, and Gilles Van Assche. Efficient and First-Order DPA Resistant Implementations of Keccak. In *CARDIS 2013*, volume 8419 of *Lecture Notes in Computer Science*, pages 187–199. Springer, 2014.

[BDPA] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The keccak reference. http://keccak.noekeon.org.

[BGN+14] Begül Bilgin, Benedikt Gierlichs, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. A More Efficient AES Threshold Implementation. In *AFRICACRYPT 2014*, volume 8469 of *Lecture Notes in Computer Science*, pages 267–284. Springer, 2014.

[BGN+15] Begül Bilgin, Benedikt Gierlichs, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. Trade-offs for threshold implementations illustrated on AES. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 34(7):1188–1200, 2015.

[Bil15] Begül Bilgin. *Threshold implementations : as countermeasure against higher-order differential power analysis*. PhD thesis, University of Twente, Enschede, Netherlands, 2015.

[BJK+16] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. The SKINNY family of block ciphers and its low-latency variant MANTIS. In *CRYPTO 2016*, volume 9815 of *Lecture Notes in Computer Science*, pages 123–153. Springer, 2016.

[BKN19] Dusan Bozilov, Miroslav Knezevic, and Ventzislav Nikov. Optimized Threshold Implementations: Minimizing the Latency of Secure Cryptographic Accelerators. In *CARDIS 2019*, volume 11833 of *Lecture Notes in Computer Science*, pages 20–39. Springer, 2019.

[BLMR19] Christof Beierle, Gregor Leander, Amir Moradi, and Shahram Rasoolzadeh. CRAFT: lightweight tweakable block cipher with efficient protection against DFA attacks. *IACR Trans. Symmetric Cryptol.*, 2019(1):5–45, 2019.

[BNN+12] Begül Bilgin, Svetla Nikova, Ventzislav Nikov, Vincent Rijmen, and Georg Stütz. Threshold implementations of all 3 ×3 and 4 ×4 s-boxes. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*, volume 7428 of *Lecture Notes in Computer Science*, pages 76–91. Springer, 2012.

[BNN+15] Begül Bilgin, Svetla Nikova, Ventzislav Nikov, Vincent Rijmen, Natalia N. Tokareva, and Valeriya Vitkup. Threshold implementations of small S-boxes. *Cryptogr. Commun.*, 7(1):3–33, 2015.

[BP12] Joan Boyar and René Peralta. A small depth-16 circuit for the AES s-box. In Dimitris Gritzalis, Steven Furnell, and Marianthi Theoharidou, editors, *Information Security and Privacy Research - 27th IFIP TC 11 Information Security and Privacy Conference, SEC 2012, Heraklion, Crete, Greece, June 4-6, 2012. Proceedings*, volume 376 of *IFIP Advances in Information and Communication Technology*, pages 287–298. Springer, 2012.

[BSG23] Fabian Buschkowski, Pascal Sasdrich, and Tim Güneysu. Easimask-towards efficient, automated, and secure implementation of masking in hardware. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2023, Antwerp, Belgium, April 17-19, 2023*, pages 1–6. IEEE, 2023.

[Can05] David Canright. A Very Compact S-Box for AES. In Josyula R. Rao and Berk Sunar, editors, *CHES 2005*, volume 3659 of *Lecture Notes in Computer Science*, pages 441–455. Springer, 2005.

[Can06] Christophe De Cannière. Trivium: A stream cipher construction inspired by block cipher design principles. In Sokratis K. Katsikas, Javier López, Michael Backes, Stefanos Gritzalis, and Bart Preneel, editors, *Information Security, 9th International Conference, ISC 2006, Samos Island, Greece, August 30 - September 2, 2006, Proceedings*, volume 4176 of *Lecture Notes in Computer Science*, pages 171–186. Springer, 2006.

[Can20] Ran Canetti. Universally composable security. *J. ACM*, 67(5):28:1–28:94, 2020.

[CBR+15] Thomas De Cnudde, Begül Bilgin, Oscar Reparaz, Ventzislav Nikov, and Svetla Nikova. Higher-order threshold implementation of the AES s-box. In *CARDIS 2015*, volume 9514 of *Lecture Notes in Computer Science*, pages 259–272. Springer, 2015.

[CDG+13] Jeremy Cooper, Elke DeMulder, Gilbert Goodwill, Joshua Jaffe, Gary Kenworthy, Pankaj Rohatgi, et al. Test vector leakage assessment (TVLA) methodology in practice. In *International Cryptographic Module Conference*, volume 20, 2013.

[CGLS21] Gaëtan Cassiers, Benjamin Grégoire, Itamar Levi, and François-Xavier Standaert. Hardware private circuits: From trivial composition to full verification. *IEEE Trans. Computers*, 70(10):1677–1690, 2021.

[CGM+24] Gaëtan Cassiers, Barbara Gigerl, Stefan Mangard, Charles Momin, and Rishub Nagpal. Compress: Generate small and fast masked pipelined circuits. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(3):500–529, 2024.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 1999.

[CMM+23] Gaëtan Cassiers, Loïc Masure, Charles Momin, Thorben Moos, Amir Moradi, and François-Xavier Standaert. Randomness generation for secure hardware masking - unrolled trivium to the rescue. *IACR Cryptol. ePrint Arch.*, page 1134, 2023.

[CRB+16] Thomas De Cnudde, Oscar Reparaz, Begül Bilgin, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. Masking AES with d+1 Shares in Hardware. In *CHES 2016*, volume 9813 of *Lecture Notes in Computer Science*, pages 194–212. Springer, 2016.

[CRR03] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In *Cryptographic Hardware and Embedded Systems - CHES 2002*, pages 13–28, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[CSV24] Gaëtan Cassiers, François-Xavier Standaert, and Corentin Verhamme. Low-latency masked gadgets robust against physical defaults with application to ascon. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(3):603–633, 2024.

[Dae17] Joan Daemen. Changing of the Guards: A Simple and Efficient Method for Achieving Uniformity in Threshold Sharing. In *CHES 2017*, volume 10529 of *Lecture Notes in Computer Science*, pages 137–153. Springer, 2017.

[Dho21] Siemen Dhooghe. *New Security Models for Passive and Active Physical Attacks*. PhD thesis, KU Leuven, 2021.

[DNR19] Siemen Dhooghe, Svetla Nikova, and Vincent Rijmen. Threshold implementations in the robust probing model. In Begül Bilgin, Svetla Petkova-Nikova, and Vincent Rijmen, editors, *Proceedings of ACM Workshop on Theory of Implementation Security, TIS@CCS 2019, London, UK, November 11, 2019*, pages 30–37. ACM, 2019.

[DR01] Joan Daemen and Vincent Rijmen. Advanced Encryption Standard (AES). National Institute of Standards and Technology (NIST), FIPS PUB 197, U.S. Department of Commerce, Nov. 2001.

[FGP+18] Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, and François-Xavier Standaert. Composable Masking Schemes in the Presence of Physical Defaults & the Robust Probing Model. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):89–120, 2018.

[GIB18] Hannes Groß, Rinat Iusupov, and Roderick Bloem. Generic Low-Latency Masking in Hardware. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(2):1–21, 2018.

[GJJR11] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. A testing methodology for sidechannel resistance validation. In *NIST non-invasive attack testing workshop*, 2011. https://csrc.nist.gov/csrc/media/events/non-invasive-attack-testing-workshop/documents/08_goodwill.pdf.

[GMK16] Hannes Groß, Stefan Mangard, and Thomas Korak. Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order. In *Theory of Implementation Security - TIS@CCS 2016*, page 3. ACM, 2016.

[GP99] Louis Goubin and Jacques Patarin. DES and differential power analysis (the "duplication" method). In Çetin Kaya Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*, volume 1717 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 1999.

[GSM17] Hannes Groß, David Schaffenrath, and Stefan Mangard. Higher-Order Side-Channel Protected Implementations of Keccak. In *DSD 2017*, pages 205–212. IEEE Computer Society, 2017.

[HB25] Vedad Hadzic and Roderick Bloem. Efficient and composable masked AES s-box designs using optimized inverters. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2025(1):656–683, 2025.

[ISW03] Yuval Ishai, Amit Sahai, and David A. Wagner. Private Circuits: Securing Hardware against Probing Attacks. In *CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 463–481. Springer, 2003.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *CRYPTO 1999*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[KM22] David Knichel and Amir Moradi. Low-latency hardware private circuits. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 1799–1812. ACM, 2022.

[KMMS22] David Knichel, Amir Moradi, Nicolai Müller, and Pascal Sasdrich. Automated generation of masked hardware. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):589–629, 2022.

[KSM20] David Knichel, Pascal Sasdrich, and Amir Moradi. SILVER - Statistical Independence and Leakage Verification. In *ASIACRYPT 2020*, volume 12491 of *Lecture Notes in Computer Science*, pages 787–816. Springer, 2020.

[KSM22] David Knichel, Pascal Sasdrich, and Amir Moradi. Generic hardware private circuits towards automated generation of composable secure gadgets. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):323–344, 2022.

[MM22] Nicolai Müller and Amir Moradi. PROLEAD A probing-based hardware leakage detection tool. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(4):311–348, 2022.

[MM24] Nicolai Müller and Amir Moradi. Robust but relaxed probing model. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(4):451–482, 2024.

[MPL+11] Amir Moradi, Axel Poschmann, San Ling, Christof Paar, and Huaxiong Wang. Pushing the Limits: A Very Compact and a Threshold Implementation of AES. In *EUROCRYPT 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 69–88. Springer, 2011.

[MRB18] Lauren De Meyer, Oscar Reparaz, and Begül Bilgin. Multiplicative Masking for AES in Hardware. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):431–468, 2018.

[MS16] Amir Moradi and Tobias Schneider. Side-Channel Analysis Protection and Low-Latency in Action - - Case Study of PRINCE and Midori -. In *ASIACRYPT 2016*, volume 10031 of *Lecture Notes in Computer Science*, pages 517–547, 2016.

[NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold Implementations Against Side-Channel Attacks and Glitches. In *ICICS 2006*, volume 4307 of *Lecture Notes in Computer Science*, pages 529–545. Springer, 2006.

[NRS11] Svetla Nikova, Vincent Rijmen, and Martin Schläffer. Secure Hardware Implementation of Nonlinear Functions in the Presence of Glitches. *J. Cryptology*, 24(2):292–321, 2011.

[PAB+23] Enrico Piccione, Samuele Andreoli, Lilya Budaghyan, Claude Carlet, Siemen Dhooghe, Svetla Nikova, George Petrides, and Vincent Rijmen. An optimal universal construction for the threshold implementation of bijective s-boxes. *IEEE Trans. Inf. Theory*, 69(10):6700–6710, 2023.

[RBN+15] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. Consolidating masking schemes. In *CRYPTO 2015*, volume 9215 of *Lecture Notes in Computer Science*, pages 764–783. Springer, 2015.

[SAK] SAKURA. Side-channel Attack User Reference Architecture. http://satoh.cs.uec.ac.jp/SAKURA/index.html.

[SBHM20] Pascal Sasdrich, Begül Bilgin, Michael Hutter, and Mark E. Marson. Low-latency hardware masking with application to AES. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(2):300–326, 2020.

[SM21a] Aein Rezaei Shahmirzadi and Amir Moradi. Re-consolidating first-order masking schemes nullifying fresh randomness. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(1):305–342, 2021.

[SM21b] Aein Rezaei Shahmirzadi and Amir Moradi. Second-order SCA security with almost no fresh randomness. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(3):708–755, 2021.

[SR81] R.R. Sokal and F.J. Rohlf. *Biometry: The Principles and Practice of Statistics in Biological Research*. W. H. Freeman, 1981.

[Sug19] Takeshi Sugawara. 3-Share Threshold Implementation of AES S-box without Fresh Randomness. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(1):123–145, 2019.

[TLS16] Yosuke Todo, Gregor Leander, and Yu Sasaki. Nonlinear invariant attack - practical attack on full scream, iscream, and midori64. In *ASIACRYPT 2016*, volume 10032 of *Lecture Notes in Computer Science*, pages 3–33, 2016.

[UHA17] Rei Ueno, Naofumi Homma, and Takafumi Aoki. Toward More Efficient DPA-Resistant AES Hardware Architecture Based on Threshold Implementation. In *COSADE 2017*, Lecture Notes in Computer Science, pages 50–64. Springer, 2017.

[VDB+24] Dilip Kumar S. V., Siemen Dhooghe, Josep Balasch, Benedikt Gierlichs, and Ingrid Verbauwhede. Time sharing - A novel approach to low-latency masking. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(3):249–272, 2024.

[WFP+24] Lixuan Wu, Yanhong Fan, Bart Preneel, Weijia Wang, and Meiqin Wang. Automated generation of masked nonlinear components: - from lookup tables to private circuits. In Martin Andreoni, editor, *Applied Cryptography and Network Security Workshops - ACNS 2024 Satellite Workshops, AIBlock, AIHWS, AIoTS, SCI, AAC, SiMLA, LLE, and CIMSS, Abu Dhabi, United Arab Emirates, March 5-8, 2024, Proceedings, Part I*, volume 14586 of *Lecture Notes in Computer Science*, pages 319–339. Springer, 2024.

[WFW+25] Lixuan Wu, Yanhong Fan, Weijia Wang, Bart Preneel, and Meiqin Wang. Extending randomness-free first-order masking schemes and applications to masking-friendly s-boxes. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2025(1):340–366, 2025.

[WM18] Felix Wegener and Amir Moradi. A First-Order SCA Resistant AES Without Fresh Randomness. In *COSADE 2018*, volume 10815 of *Lecture Notes in Computer Science*, pages 245–262. Springer, 2018.

[ZZL+24] Cankun Zhao, Hang Zhao, Jiangxue Liu, Bohan Yang, Wenping Zhu, Shuying Yin, Min Zhu, Shaojun Wei, and Leibo Liu. Breaking ground: A new area record for low-latency first-order masked SHA-3 advancing from the 4x area era to the 3x area era. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(4):231–257, 2024.