Journal of Cryptographic Engineering (2022) 12:15–51
https://doi.org/10.1007/s13389-021-00276-5

REGULAR PAPER

Optimized threshold implementations: securing cryptographic accelerators for low-energy and low-latency applications

Dušan Božilov[1,2] · **Miroslav Knežević**[1] · **Ventzislav Nikov**[1]

Received: 2 July 2020 / Accepted: 3 October 2021 / Published online: 25 November 2021

Abstract
Threshold implementations have emerged as one of the most popular masking countermeasures for hardware implementations of cryptographic primitives. In this work, we provide three TI optimization techniques: First, a generic construction for $d+1$ TI sharing achieves the minimal number of output shares for any n-input Boolean function of degree $t = n-1$ and for any d. Next, we present a methodology for finding minimal number of output shares in $d+1$ TI when $t < n-1$. Third, a heuristic for minimizing the number of output shares for higher-order $td+1$ TI for any n, any t and $d \leq 2$ is proposed. In addition, we describe an optimization for the secure AES schedule which achieves maximum throughput for a serial implementation. Then, we demonstrate the applicability of our results on $d+1$ and $td+1$ TI versions, for first- and second-order secure, low-latency and low-energy implementations of the PRINCE block cipher. We show the fastest and the most energy efficient known TI-protected implementations of PRINCE.

Keywords Threshold Implementations · PRINCE · AES · SCA · Masking

1 Introduction

Historically, the field of lightweight cryptography has been focused on designing algorithms that leave an as small footprint as possible when manufactured in silicon. Small area implicitly results in low power consumption, which is another, equally important, optimization target.

Hitting these two targets comes at a price since the performance and energy consumption of lightweight cryptographic primitives are far from being competitive, and for most online applications, they are often not meeting the requirements. There are only a handful of designs that consider latency and energy consumption among their main design goals. PRINCE [3] and Midori [2] are two prominent examples.

The inherent vulnerability to physical attacks is a serious threat that the field of lightweight cryptography has been exposed to since its creation. Side-channel analysis is one of the most powerful examples of such an attack. To resist an adversary that has access up to d wires inside the circuit [27], the secret value has to be shared into at least $d+1$ random shares using a masking technique. An example of such a scheme is Boolean masking where the secret is shared into shares using Boolean addition and the shares are then independently processed in a way that prevents revealing the secret information.

In order to circumvent a masked implementation, attackers need to extract and combine the secret information from several shares, i.e., they need to employ a higher-order attack of degree d at least. These attacks are harder to mount because they are susceptible to the amount of noise collected during the trace acquisition. Higher-order SCA protection incurs penalties in the silicon area, execution time, power consumption and the number of random bits required for secure execution. Increased cost comes from the number of shares that are required. When protecting a nonlinear function, the number of output shares grows significantly and depends on the number of input shares, the algebraic degree of the function, the number of nonlinear terms the function has and the security order that needs to be achieved. For example, the number of output shares in consolidated masking scheme [41] exponentially grows with the algebraic degree of the function with the number of shares being lower bound by $(d+1)^t$, with t being the algebraic degree.

The challenge of designing secure cryptographic circuits becomes significantly harder once the strict requirements

✉ Dušan Božilov
dusan.bozilov@esat.kuleuven.be

[1] NXP Semiconductors, Leuven, Belgium

[2] imec-COSIC, KU Leuven, Leuven, Belgium

have to be met for at least one of the following metrics: silicon area, latency, power or energy consumption. In the context of this paper, and as stated in [29], we consider latency as the total time needed to execute a single cryptographic operation. Minimizing latency can be achieved by increasing the frequency the circuit can operate on or by reducing the clock cycle count of the operation. Hence, one design outperforms another with regard to latency if the product of the number of clock cycles and the minimal clock period is smaller in that design.

Recently, multiple examples targeting both low-latency and side-channel protection emerged. The resulting effort produced a variety of AES [22,44,46], KECCAK [1] and PRINCE [33]. Their results indicate that low-latency side-channel design is a significantly more difficult problem than designing a countermeasure by optimizing area or the amount of randomness, which are the typical design criteria addressed by the scientific community. Therefore, designing side-channel countermeasures for low-latency or low-energy implementations is considered to be an important open problem.

Threshold implementation (TI) [36] is a provably secure masking scheme specifically designed to counter side-channel leakage caused by the presence of glitches in hardware. The countermeasure removes the dependency between the number of nonlinear terms and the number of input shares, which is a big advantage over classical masking schemes. In [6], the authors extended the approach of TI to counter higher-order (uni-variate) attacks. The theory suggests the usage of at least $td + 1$ number of input shares in order to make a Boolean function with algebraic degree t secure against a dth-order side-channel attack. That is the reason why the TI scheme introduced in [6] is often referred to as a $td + 1$ TI. In 2015, the authors of [41] proposed a consolidated masking scheme (CMS) and reduced the required number of input shares needed to resist a dth-order attack to $d + 1$, regardless of the algebraic degree of the shared function. Recall that this is theoretically the lowest bound on the number of input shares with respect to the order of security d. Recently, more schemes using $d + 1$ shares such as domain-oriented masking (DOM) and unified masking approach (UMA) emerged [23,24], where the essential difference with CMS is in the way the refreshing of the output shares is performed. Since the security of CMS, DOM, UMA, HPC [14], and other descendant schemes relies on the TI principles [19], in this paper we refer to all these schemes as $d + 1$ TI.

The goal of generic low-latency masking (GLM) [22] is to be a generalized concept for low-latency masking that is supposed to be applicable to any implementation and protection order. However, the authors have applied their concept to designs that are not low-latency, and therefore, it is difficult to compare their approach. We have to stress that the goal to achieve minimal latency is not equivalent to get only execution within fewer cycles since, at the same time, the complexity of the circuit grows, resulting in a longer critical path. In other words, one gets a design that can be executed in fewer cycles but also with a lower max frequency. It has been pointed out in [31] that all these generalized concepts have to use another re-sharing technique since the original ones have flaws for $d > 2$.

While the established theory of TI guarantees that the number of input shares linearly grows with the order of protection d, it does not provide efficient means to keep the exponential explosion of the number of output shares under control. The state of the art is a lower bound of $(d + 1)^t$ given in [41], while in [6] the authors described a method to obtain a TI sharing with $\binom{td+1}{t}$ output shares. The latter work also notes that the number of output shares can sometimes be reduced by using more than $td + 1$ input shares. Aside from a formula for the lower bound in [41], there was not much other work of applying $d + 1$ TI to functions with a higher degree than 2. The only exception is the AES implementations by [46,47] where $d + 1$ TI is applied to the inversion of $GF(2^4)$, which is a function of algebraic degree 3. However, even for this particular case, the first attempt [47] resulted in sharing with the minimal number of output shares, but it did not satisfy the non-completeness property of TI. Only in the follow-up publication [46] was the sharing correct and minimal. Also, for the particular case of a cubic function, it is fairly easy to find the minimal first-order sharing of eight output shares by exhaustive trial-and-error approach.

1.1 Our contribution

In this paper, we introduce three optimization techniques for optimizing threshold implementations, making the low-latency implementations of side-channel secure designs practical. In particular, we first provide a generic construction for $d + 1$ TI that achieves the optimal number of output shares for any n-input Boolean function of degree $t = n - 1$ for any security order d. Second, when $t < n-1$, we present a methodology based on discrete optimization techniques that provides optimal or near-optimal sharings of several classes of Boolean functions of any degree up to eight variables, for first- and second-order TI. Third, for $td + 1$ TI, we present a heuristic for higher-order protection for any n-input Boolean function of any degree t and up to second-order protection, which yields a number of output shares significantly lower than $\binom{td+1}{t}$. Last but not least is the optimization for secure serial AES implementation schedule which achieves maximum throughput.

Next, we investigate the energy consumption of different approaches, an important design factor yet often overlooked in the literature. To demonstrate the feasibility of the above methods, we have applied the optimized $d + 1$ TI and $td + 1$

TI on the hardware implementation of PRINCE. Then, we demonstrate how to reduce the latency to achieve the fastest known TI-protected implementation of PRINCE, and we also show the most energy-efficient round-based secure implementation of PRINCE using $d+1$ TI sharing.

Finally, we would like to point out that the method of minimizing the number of output shares is of general interest since it can equally well be applied to any cryptographic implementation and any design optimization criteria.

2 Preliminaries

We use small letters to represent elements of the finite field $\mathcal{F}_2^n$. Subscripts are used to specify each bit of an element or each coordinate function of a vectorial Boolean function, i.e., $x = (x_1, \ldots, x_n)$, where $x_i \in \mathcal{F}_2$ and $S(x) = (S_1(x), \ldots, S_m(x))$, where S is defined from $\mathcal{F}_2^n$ to $\mathcal{F}_2^m$ and S_i's are defined from $\mathcal{F}_2^n$ to $\mathcal{F}_2$. We omit subscripts if $n = 1$ or $m = 1$. We use subscripts also to represent shares of one-bit variables. The reader should be able to distinguish from the context if we are referring to specific bits of unshared variable or specific shares of a variable. We denote Hamming weight, concatenation, cyclic right shift, right shift, composition, multiplication and addition with $wt(.)$, $||$, $\ggg$, $\gg$, $\circ$, . and $+$, respectively.

Every Boolean function S can be represented uniquely by its *Algebraic Normal Form* (ANF): $S(x) = \sum_{i=(i_1,\ldots,i_n)\in\mathcal{F}_2^n} a_i x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n}$. Then, the *algebraic degree* of a Boolean function S is $\deg(S) = \max\{wt(i) : i \in \mathcal{F}_2^n, a_i \neq 0\}$. The algebraic degree of a vectorial Boolean function S is equal to the highest algebraic degree of its coordinate functions S_j.

Two permutations S and S' are affine equivalent if and only if there exist affine permutations C and D satisfying $S' = C \circ S \circ D$. We refer to C as the output and D as the input transformation.

2.1 Threshold implementations

The TI sharing designed to protect against the dth-order attack we will simply refer to as the dth-order TI. The most important property that ensures the security of TI even in the presence of glitches is *non-completeness*. The dth-order non-completeness property requires any combination of up to d component functions to be independent of at least one input share. As shown in [19], non-completeness is a necessary condition for the security of masking in the presence of glitches. When cascading multiple nonlinear functions, the first-order sharing must also satisfy the *uniformity*; namely, a sharing is uniform if and only if the sharing of the output preserves the distribution of the unshared output. In other words, for a given unmasked value, all possible combinations of output shares representing that value are equally likely to happen. Finding a uniform sharing for any vectorial Boolean function is still an open problem, although several heuristics exist, such as sharing with correction terms or adding virtual shares [8,11]. Fortunately, uniformity can still be achieved by refreshing the output shares when no uniform sharing is available.

Given the shares $x_1, \ldots, x_n$ a (first and second order) refreshing can be realized by mapping $(x_1, \ldots, x_n)$ to $(y_1, \ldots, y_n)$ using n random values $r_1, \ldots, r_n$ as follows:

$$\begin{aligned} y_1 &= x_1 + r_1 + r_n \\ y_i &= x_i + r_{i-1} + r_i, \qquad i \in \{2, \ldots, n\}. \end{aligned} \tag{1}$$

This refreshing scheme is called *ring re-masking*. A simpler refreshing using $n-1$ random values exists especially for the first-order secure implementations as we can re-mask the shares $x_1, \ldots, x_n$ in the following way:

$$\begin{aligned} y_i &= x_i + r_i, \qquad i \in \{1, \ldots, n-1\}, \\ y_n &= x_n + r_1 + \cdots + r_{n-1}. \end{aligned} \tag{2}$$

An improvement regarding the number of random bits used when multiplication gate is shared has been achieved in [24] where the amount of randomness required is halved compared to CMS. In [23], the authors have shown that the amount of randomness for sharing a multiplication gate can be further reduced to one-third, although this comes at a significant performance cost. Since our goal is to build low-latency side-channel secure implementations, we do not take the approach of UMA. Instead, we choose CMS/DOM for $d+1$ TI designs. In this paper, we will interchangeably use the terms mask refreshing and re-masking. We would also like to mention an alternative mask refreshing technique presented by Daemen [17], which achieves a uniform $td+1$ sharing by reusing shares of adjacent S-box inputs to create a statewide permutation, which automatically creates the entire S-box layer stage uniform without needing any external randomness.

In order to prevent glitch propagation when cascading nonlinear functions, TI requires register (s) to be placed between the nonlinear operations. Otherwise, the non-completeness property may be violated, and the leakage of the secret internal state is likely to be manifested.

When sharing a nonlinear function using TI in a single-cycle implementation, the number of output shares is typically larger than the number of input shares. This is likely to occur when applying $td+1$ TI, in which constructions like one provided in [6] produce sharings of Boolean functions of degree t with $\binom{td+1}{t}$ output shares and with $td+1$ inputs shares. The number of output shares in $d+1$ TI for nonlinear functions is always larger than the fixed $d+1$ input shares and

is lower bound by $(d+1)^t$, in which t is the algebraic degree of the Boolean function. In order to minimize the number of output shares, we need to refresh and recombine (compress) some shares by adding several of them together. To prevent glitches from revealing unmasked values, decreasing the number of shares can only be done after storing these output shares into a register. The output shares that are going to be recombined together still need to be carefully chosen such that they do not reveal any unmasked value.

A sharing of a nonlinear function can also be realized without output share explosion, albeit at the cost of making the evaluation take several cycles, like already mentioned decomposition approach or techniques such as Toffoli gate [20].

While using $d+1$ TI the relation between the input shares needs to obey a stronger requirement, namely shared input variables need to be *independent* [41]. This can be achieved in various ways—for example, by refreshing some of the inputs or by using a technique proposed in [24].

2.2 Minimizing implementation overheads using S-box decomposition

Similar to other side-channel countermeasures, the area overhead of applying TI increases polynomially with respect to the security order and exponentially with respect to the algebraic degree of the function we are trying to protect. To keep the large overheads caused by exponential dependency under control, designers often use decomposition of the higher-degree functions into several lower-degree functions. This approach has originally been demonstrated in [40] where the authors implemented a TI-protected PRESENT block cipher [10] by decomposing its cubic S-box into two simpler quadratic S-boxes. Finally, decomposition of the cubic 4-bit S-boxes into chains of smaller quadratic S-boxes was given in [11], which eventually enables compact, side-channel secure implementations of all 4-bit S-boxes.

Although a decomposition of nonlinear functions into several simpler functions of lower algebraic degree is the proper approach to use for area reduction of the TI-protected implementations, its side effect is the increased latency of the S-box evaluation and hence the entire implementation. Recall that the TI requires registers to be placed between the nonlinear operations in order to prevent glitch propagation, which in turn increases the latency. We will not use this approach since our goal is to achieve low latency.

2.3 A note on latency and energy efficiency

As already mentioned, most of the effort the scientific community has spent on designing secure implementations has been focused on reducing area overheads. Another important metric that had been given lots of attention is the amount of randomness used in protected implementations. While both of these metrics are important, the performance and energy consumption of secure implementations have been unjustly treated as less significant. It has been widely accepted that performance is the metric to sacrifice in order to achieve the lowest possible gate count. Contrary to this view, most of the practical applications nowadays require (very) fast execution, and it is often latency of the actual implementation that matters rather than the throughput. Energy consumption is another equally important metric, and, unlike power consumption, it cannot be well controlled by keeping the area low while sacrificing performance. Optimizing for energy consumption is, in fact, one of the most difficult optimization problems in (secure) circuit design since the perfect balance between the circuit power consumption and its execution speed needs to be hit.

The absolute latency is directly proportional to the number of clock cycles a certain operation takes to execute. At the same time, the absolute latency is inversely proportional to the clock frequency the system is running at. While the clock frequency is determined by taking into account multiple factors from the whole system, the most important of which is the overall power/energy consumption, the number of clock cycles a certain algorithm takes to execute is under the complete control of the designer. Especially when considering embedded devices, the tendency is to keep the clock frequency as low as we can while still meeting the performance requirements. That is the reason why minimizing the number of clock cycles of a certain algorithm is the most important strategy when it comes to minimizing the overall latency of that algorithm.

Although the majority of results available in public literature deal with area-efficient hardware architectures, there are still a few notable examples in which latency reduction has been the main target. In [33], the authors particularly explore the extreme case of a single clock cycle side-channel secure implementations of PRINCE and Midori. Moreover, they conclude that designing a low-latency side-channel secure implementation of cryptographic primitives remains an open problem.

2.4 PRINCE

As a proof of concept, we apply different flavors TI to PRINCE [3], a block cipher designed for low-latency hardware implementations. PRINCE block size is 64 bits, with a 128-bit key, used to derived three 64-bit internally used keys k_0, k_0' and k_1. Its α-reflection property allows reuse of the same circuitry for both encryption and decryption.

Although not designed to be efficient in software, a bit-sliced software implementation of PRINCE is also fast and can even be executed in fewer clock cycles than other

lightweight block ciphers such as PRESENT and KATAN, for example [38].

Here, we give a brief overview of PRINCE, and for a more detailed explanation of the cipher, we refer the reader to the original paper [3]. The block size is 64 bits, and a key has a length of 128 bits. The key is split into two 64-bit parts $k_0||k_1$ and expanded to $k_0||k_0'||k_1$ shown as follows:

$$(k_0||k_0'||k_1) = k_0||((k_0 \ggg 1) + (k_0 \gg 63))||k_1.$$

As depicted in Fig. 1, k_0 and k_0' are used as whitening keys at the start and at the end of the cipher and k_1 is used as round key in PRINCE$_{core}$ which consists of 12 rounds. More precisely, six rounds followed by the middle involution layer which is then followed by the six inverse rounds.

S-box layer The S-box is a 4-bit permutation of algebraic degree 3, and its look-up table is $S(x) = [B, F, 3, 2, A, C, 9, 1, 6, 7, 8, 0, E, 5, D, 4]$. The S-box inverse is in the same affine equivalence class as the S-box itself. Moreover, input and output transformations are the same:

$$S^{-1} = A_{io} \circ S \circ A_{io}. \tag{3}$$

The affine transformation A_{io} is given as $A_{io}(x) = [5, 7, 6, 4, F, D, C, E, 1, 3, 2, 0, B, 9, 8, A]$.

Linear layer Matrices M and M' define the diffusion layer of PRINCE. M' is an involution and M matrix can be obtained from M' by adding a shift-rows operation SR so that $M = \text{SR} \circ M'$. Recall that SR is a linear operation that permutes the nibbles of the PRINCE state.

RC$_i$ **addition** is a 64-bit round constant addition. The round constants $\text{RC}_0 \ldots \text{RC}_{11}$ are chosen such that $\text{RC}_i + \text{RC}_{11-i} = \alpha$ where α is a 64-bit constant. This property, called α-reflection property, together with the construction of PRINCE$_{core}$ rounds makes the decryption of PRINCE$_{core}$ same as the encryption with $k_1 + \alpha$ key.

2.5 PRINCE S-box decomposition

PRINCE S-box has an algebraic degree three and belongs to class $\mathcal{C}_{231}$ [11]. According to [11] and the tables given in [34], there are several hundreds of decompositions into three quadratic S-boxes and four affine transformations.

We choose a decomposition where all three quadratic S-boxes are the same, belonging to class Q_{294}, for its small area footprint. The look-up table of Q_{294} class is given in Eq. (4), while its ANF form is given with Eq. (5). Decomposition of PRINCE S-box to three quadratic S-boxes using other classes of 4-bit quadratic S-boxes exists, but there is no decomposition in which the same quadratic S-box is used in all three stages. This allows for an implementation that reuses the same quadratic implementation, with only the extra cost of a multiplexer to ensure correct evaluation during three stages. Additionally, using a decomposition containing a class that has no known uniform sharing (Q_{300} as stated in [11]), would mandate mask refreshing. Increased mask refreshing puts more burden on the PRNG which is used, making the entire system more complex, power inefficient and large [8].

$$Q_{294}(x) = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, B, A, E, F, D, C] \tag{4}$$

$$\begin{aligned} o_1 &= x \\ o_2 &= y \\ o_3 &= xy + z \\ o_4 &= xz + w. \end{aligned} \tag{5}$$

Decomposition leads to lower area and randomness requirement as they depend on the algebraic degree of the function when applying TI. On the other hand, performance is penalized. PRINCE S-box ANF $(o_1, o_2, o_3, o_4) = F(x, y, z, w)$ is given by:

$$\begin{aligned} o_1 &= 1 + wz + y + zy + wzy + x + wx + yx \\ o_2 &= 1 + wy + zy + wzy + zx + zyx \\ o_3 &= w + wz + x + wx + zx + wzx + zyx \\ o_4 &= 1 + z + zy + wzy + x + wzx + yx + wyx. \end{aligned} \tag{6}$$

S-box and its inverse decompositions used in our implementation are given in Eq. (7).

$$\begin{aligned} S &= A_1 \circ Q_{294} \circ A_2 \circ Q_{294} \circ A_3 \circ Q_{294} \circ A_4 \\ S^{-1} &= A_5 \circ Q_{294} \circ A_2 \circ Q_{294} \circ A_3 \circ Q_{294} \circ A_6. \end{aligned} \tag{7}$$

Here, A_1 to A_6 are affine transformations and their respective look-up tables are: $A_1(x) = [C, E, 7, 5, 8, A, 3, 1, 4, 6, F, D, 0, 2, B, 9]$, $A_2(x) = [6, D, 9, 2, 5, E, A, 1, B, 0, 4, F, 8, 3, 7, C]$, $A_3(x) = [0, 8, 4, C, 2, A, 6, E, 1, 9, 5, D, 3, B, 7, F]$, $A_4(x) = [A, 1, 0, B, 2, 9, 8, 3, 4, F, E, 5, C, 7, 6, D]$, $A_5(x) = [B, 8, E, D, 1, 2, 4, 7, F, C, A, 9, 5, 6, 0, 3]$, $A_6(x) = [9, 3, 8, 2, D, 7, C, 6, 1, B, 0, A, 5, F, 4, E]$.

The ANF of the Q_{294} function, $(x, y, z, w) = F(a, b, c, d)$, is given in Eq. (8)

$$\begin{aligned} x &= a \\ y &= b \\ z &= ab + c \\ w &= ac + d. \end{aligned} \tag{8}$$

We recall that, for a secure implementation with this decomposition method, nonlinear operations need to be separated by registers, making the evaluation of a single S-box take 3 clock cycles.

As discussed in Sect. 2.2, we explore the implementation of the decomposed S-box in order to address the issue of

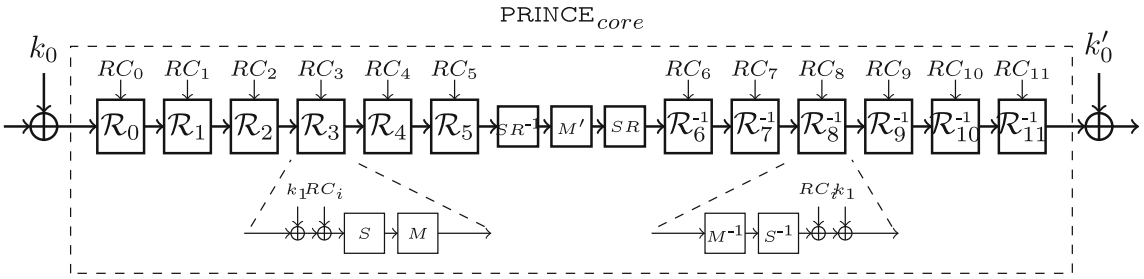


Fig. 1 PRINCE cipher

low-area and low-power applications but, in what follows, we also explore the sharing of the non-decomposed S-box to address the issue of low latency and low energy.

3 Efficiency techniques

To find a $td + 1$ or $d + 1$ sharing for a quadratic vectorial Boolean function is fairly straightforward in general and especially easy for the functions that have a simple ANF, e.g., a quadratic function with a single higher degree term. However, to find an efficient sharing for a vectorial Boolean function of algebraic degree three or higher and with several higher-degree terms may not be evident and the work required to find the minimal number of the output shares becomes increasingly difficult. In this section, we propose methods to deal with this complexity and we explicitly describe an optimal solution for the $d+1$ sharing for any security order d.

3.1 Efficient first- and second-order $td + 1$ sharing: ($td + 1$, $\forall n$, $\forall t$, $d \leq 2$)

To obtain a $td+1$ TI implementation, where t is the algebraic degree of the function and d is the security order, one needs to go through the following two computational phases:

(a) The expansion phase in which the shared function f uses $s_{\text{in}} \geq td+1$ input shares and results in s_{out} output shares. Output share functions f_i are referred to as component functions.
(b) The compression phase in which re-masked s_{out} output shares stored in a register are combined again to s_{in} shares.

This process takes two clock cycles except when $s_{\text{in}} = s_{\text{out}}$. For example, for the first-order TI only the first phase suffices. Additionally, if the TI sharing is uniform, the refreshing step can be removed.

The dth-order TI (more specifically, its *non-completeness* property) requires that any combination of up to d component functions f_i is missing at least one share for each of the input variables. The method presented in [6] demonstrates how to find a sharing with the minimum number of input shares, i.e., $s_{\text{in}} = td + 1$, which results in $s_{\text{out}} = \binom{s_{\text{in}}}{t}$ output shares. However, this approach does not guarantee that s_{out} is indeed the theoretical minimum. Even more, there are examples which show that by increasing s_{in} it is possible to decrease s_{out}.

We use s_{out} as a figure of merit since the amount of registers required to store the output shares and the amount of random bits required for refreshing increases with the number of output shares. Further on, we will describe a way to find a $td+1$ sharing with small s_{out}. It should be noted that the number of input shares s_{in} also contributes to the total number of registers used, so it would also be interesting to investigate $s_{\text{out}} + s_{\text{in}}$ as a figure of merit while investigating $td + 1$ TI sharings. However, increasing the number of input shares s_{in} significantly increases the combinatorial logic needed in a hardware implementation, which can negatively impact the area and power consumption.

For every output share, there is a subset of input shares of all variables that are permitted to be part of that share's equation. Definition 1 introduces the notions of output sets and output sharing sets, which will be used to argue TI properties of correctness and non-completeness for $td + 1$ TI.

Definition 1 Given a $td + 1$ TI sharing of a function f with s_{in} input and s_{out} output shares, we can enumerate the input shares as elements of a set $\mathcal{I} = \{0, \ldots s_{\text{in}}-1\}$. For each output share f_o, we can associate a set $\mathcal{O}$, which is a subset of I, containing indices of allowed input shares that can appear in the ANF of f_o. We refer to O as the **output set** of the oth output share of f. The set $\mathcal{S}$ containing all output sets of a sharing as elements is **output sharing set**.

Set representation determines the maximum degree of the output share component function and is equal to the number of elements in it. Further on, we will refer to a set with k elements as k-set. It should be noted that output sets impose no special restrictions for any particular input variable. Each input variable can have any of the allowed input shares present in the output share.

Equation (9) shows an example of how second-order secure sharing of function $xy + z$ can be obtained using six input shares and seven output shares. An illustration of this sharing is additionally represented by their output share sets on the left.

$$
\begin{aligned}
\{0,1,2\} \quad & o_1 = x_0y_1 + x_1y_0 + x_0y_2 + x_1y_2 + x_2y_1 \\
\{0,3,4\} \quad & o_2 = z_4 + x_4y_4 + x_0y_3 + x_0y_4 + x_3y_4 + x_4y_3 \\
\{1,3,5\} \quad & o_3 = z_3 + x_3y_3 + x_1y_3 + x_3y_1 + x_1y_5 \\
& \quad + x_3y_5 + x_5y_3 \\
\{2,4,5\} \quad & o_4 = z_5 + x_5y_5 + x_2y_4 + x_4y_2 + x_2y_5 \\
& \quad + x_5y_2 + x_4y_5 + x_5y_4 \\
\{0,1,4\} \quad & o_5 = z_0 + x_0y_0 + x_4y_0 + x_1y_4 + x_4y_1 \\
\{0,1,5\} \quad & o_6 = z_1 + x_1y_1 + x_0y_5 + x_5y_0 + x_5y_1 \\
\{0,2,3\} \quad & o_7 = z_2 + x_2y_2 + x_2y_0 + x_3y_0 + x_2y_3 + x_3y_2.
\end{aligned} \tag{9}
$$

As it can be seen, the output share sets dictate which indexes of variables are allowed in the corresponding output share. For example, for o_1 only input shares (i.e., indexes) 0, 1 and 2 are allowed. That requirement is indeed fulfilled by the formula describing o_1. Note that these sets do not uniquely define the sharing. We could, for example, remove the term x_0y_1 from o_1 and assign it to o_5 or o_6, as $\{0, 1\}$ is a subset of both $\{0, 1, 4\}$ and $\{0, 1, 5\}$. Output share sets would still be the same, and the sharing would still be correct second-order $td + 1$ sharing. A good heuristic for assigning monomials to output shares is that all output shares should have about the same amount of terms. That way the implementation becomes balanced and the critical path is roughly equal among all output shares.

We can reason about the properties of a given $td + 1$ sharing by inspecting its output sets. Consider a sharing of a function of degree t and the set S_t that contains all $\binom{s_{\text{in}}}{t}$ different t-sets of input shares. *Correctness* is satisfied if and only if the set that contains all different t-sets that are subsets of at least one output set is equal to the S_t. Indeed, if that is not the case, there would exist a cross-product of degree t that could not be part of any output share, making correctness of the sharing violated. For Eq. (9), given that the unshared function is of degree 2, we can check and confirm that all of the $\binom{6}{2}$ 2-sets are contained in output sets of the sharing. For *non-completeness* property, which ensures dth-order security, no union of d output sets is equal to a set covering all input shares $\{0, \ldots, s_{\text{in}} - 1\}$. Equivalently, any combination of d output shares does not contain all input shares. In Eq. (9), this is true for $d = 2$ as no union of two output sets gives the set of all input shares $\{0, 1, 2, 3, 4, 5\}$.

An interesting question when searching for the minimal number of shares is the size of output sets, as definedin Definition 1. Since any subset of output shares that contains all possible t-sets also contains all possible k-sets with $k < t$, and these smaller output sets do not contribute in the generation of a correct sharing of degree t function, we can conclude that k-sets with k smaller than t are redundant and can be omitted. On the other hand, the size of each output share set cannot be larger than $s_{\text{in}} - (t(d-1)+1)$. Let us assume otherwise, i.e., there is an output set of size at least $s_{\text{in}} - t(d-1)$. To cover the set of all input shares, we need to add missing $t(d-1)$ input shares to it from other output sets. Since the cardinality of an output set is at least t, we can do it with at most $(d-1)$ other output sets. But now the *non-completeness* would be violated since there are d output shares which cover all input shares.

Now, we are ready to describe the greedy algorithm (Fig. 2) which finds the sharing with small number of output shares.

Step *2b* in algorithm described in Fig. 2 involves randomly choosing k-set o among all possible k-set with the desired property. This nondeterministic behavior leads to different output sharings with different cardinalities. Hence, we iterate the greedy algorithm multiple times and choose the output sharing S_o with the smallest cardinality among all executions. After 100 iterations which complete on a regular PC within seconds, we have observed that the number of output shares Algorithm 2 provides remains the same, even if we increase the number of iterations to 10,000, while the execution time increases. Thus, for the results we present here we have run the greedy algorithm 100 times and chosen the smallest sharing among all executions.

The example of a single pass of the greedy algorithm is given in Table 1, for $S_{\text{in}} = 6, d = 2$ and $t = 2$, making set O containing all sets of size $k = s_{\text{in}} - (t(d-1)+1) = 3$. In each step of the greedy algorithm, these sets are scored according to the step *2b*. In bold, we mark the chosen set to be added to the output sharing, and in light gray, we highlight the sets that must be removed, because if they remain in the next steps of the algorithm, they could violate the non-completeness of the constructed sharing. The left column in each table is the score of a given k-set, or the amount of t-sets it contains, that are not present in S_o. The right column is all remaining k-sets that do not violate non-completeness if added to S_o. In the same column, above the horizontal line is partially constructed S_o represented by k-sets that are added to it. If we take the fourth table, k-set $\{0, 1, 4\}$ has a score of 1 as only $\{1, 4\}$ is the new t-set it would add, given $\{0, 1\}$ and $\{0, 4\}$ are already subsets of output shares $\{0, 1, 2\}$ and $\{0, 3, 4\}$. On the other hand, k-set $\{2, 4, 5\}$ has a score of 3 since none of the t-sets $\{2, 4\}$, $\{2, 5\}$ and $\{4, 5\}$ are present in any of the output shares. For this particular order of the k-sets, we end up with output sharing that contains seven shares.

Fig. 2 Greedy algorithm for efficient $td + 1$ sharing

1. For a given number of input shares s_{in}, security order d and algebraic degree t, we work with two sets:
   (a) A set O containing initially all k-sets with $k = s_{in} - (t(d-1)+1)$, with elements from $\{0, \ldots, s_{in} - 1\}$.
   (b) An empty solution set S_o, i.e. initially there are no output shares.
2. Then we iterate as follows until the combination of t-sets covered by its output shares S_o contains all different t-sets:
   (a) Remove all k-sets from O that would cause violation of non-completeness when combined with existing k-sets in S_o.
   (b) From set O we pick set o that covers the highest number of t-sets that are not subsets of any output set in S_o. If there are multiple k-sets with this property we randomly choose one of them as o.
   (c) We add o to S_o and remove it from O.
3. The set S_o is the found output sharing s_{out}.

Table 1 Example execution of the greedy algorithm for the case $s_{\text{in}} = 6, d = 2, t = 2$

	{0, 1, 2}		{0, 1, 2} **{0, 3, 4}**		{0, 1, 2} {0, 3, 4} **{1, 3, 5}**		{0, 1, 2} {0, 3, 4} {1, 3, 5} **{2, 4, 5}**		{0, 1, 2} {0, 3, 4} {1, 3, 5} {2, 4, 5} **{0, 1, 4}**		{0, 1, 2} {0, 3, 4} {1, 3, 5} {2, 4, 5} {0, 1, 4} **{0, 1, 5}**		{0, 1, 2} {0, 3, 4} {1, 3, 5} {2, 4, 5} {0, 1, 4} {0, 1, 5} **{0, 2, 3}**
3	**{0, 1, 2}**	2	{0, 1, 3}	1	{0, 1, 3}	0	{0, 1, 3}	1	**{0, 1, 4}**	1	**{0, 1, 5}**	1	**{0, 2, 3}**
3	{0, 1, 3}	2	{0, 1, 4}	1	{0, 1, 4}	1	{0, 1, 4}	1	{0, 1, 5}	1	{0, 2, 3}	0	{0, 2, 5}
3	{0, 1, 4}	2	{0, 1, 5}	2	{0, 1, 5}	1	{0, 1, 5}	1	{0, 2, 3}	1	{0, 2, 5}	0	{0, 3, 5}
3	{0, 1, 5}	2	{0, 2, 3}	1	{0, 2, 3}	1	{0, 2, 3}	1	{0, 2, 5}	1	{0, 3, 5}	0	{0, 4, 5}
3	{0, 2, 3}	2	{0, 2, 4}	1	{0, 2, 4}	2	{0, 2, 5}	1	{0, 3, 5}	1	{0, 4, 5}	1	{1, 2, 3}
3	{0, 2, 4}	2	{0, 2, 5}	2	{0, 2, 5}	1	{0, 3, 5}	1	{0, 4, 5}	1	{1, 2, 3}	0	{1, 2, 4}
3	{0, 2, 5}	3	**{0, 3, 4}**	2	{0, 3, 5}	2	{0, 4, 5}	1	{1, 2, 3}	0	{1, 2, 4}	0	{1, 3, 4}
3	{0, 3, 4}	3	{0, 3, 5}	2	{0, 4, 5}	1	{1, 2, 3}	1	{1, 2, 4}	0	{1, 3, 4}	0	{1, 4, 5}
3	{0, 3, 5}	3	{0, 4, 5}	2	{1, 2, 3}	2	{1, 2, 4}	1	{1, 3, 4}	0	{1, 4, 5}		
3	{0, 4, 5}	2	{1, 2, 3}	2	{1, 2, 4}	1	{1, 3, 4}	1	{1, 4, 5}	1	{2, 3, 4}		
3	{1, 2, 3}	2	{1, 2, 4}	2	{1, 3, 4}	2	{1, 4, 5}	1	{2, 3, 4}				
3	{1, 2, 4}	2	{1, 2, 5}	3	**{1, 3, 5}**	2	{2, 3, 4}	1	{2, 3, 5}				
3	{1, 2, 5}	3	{1, 3, 4}	3	{1, 4, 5}	2	{2, 3, 5}						
3	{1, 3, 4}	3	{1, 3, 5}	2	{2, 3, 4}	3	**{2, 4, 5}**						
3	{1, 3, 5}	3	{1, 4, 5}	3	{2, 3, 5}								
3	{1, 4, 5}	3	{2, 3, 4}	3	{2, 4, 5}								
3	{2, 3, 4}	3	{2, 3, 5}										
3	{2, 3, 5}	3	{2, 4, 5}										
3	{2, 4, 5}												
3	{3, 4, 5}												

Each table shows a single step of algorithm execution. Left column is the amount of t-sets not covered in S_o by the set on the right. Sets above the horizontal line are partially constructed sharing S_o

3.2 Optimal $d+1$ sharing for functions with degree $n-1$: ($d+1$, $\forall n$, $t=n-1$, $\forall d$)

To achieve dth-order security using $d+1$ sharing for a single term of degree t, i.e., a product of t variables, one gets exactly $(d+1)^t$ shares for the product [41]. Alternatively, for $s_{\text{in}} = d+1$ input shares and a product of t variables one gets $s_{\text{out}} = (d+1)^t$ output shares.

Main difference with $td+1$ sharing is how the *non-completeness* property is interpreted. Unlike with $td+1$ TI sharing in the $d+1$ TI sharing each output share should contain only one share per input variable, in other words if in an output share there are two shares of an input variable, then the dth-order non-completeness will be violated. Recall that for the $td+1$ TI using more shares per input variable is possible since the number of input shares is bigger. We observe this in Eqs. (9) and (11). In the first output share of Eq. (9), we see three input shares of x: x_0, x_1 and x_2. In $d+1$ sharing of Eq. (11), the first output share only has one input share of x: x_0.

Therefore, for $d+1$ case to ensure *non-completeness* it is enough to have only one share of each input variable present in any given output share. We will assume that the independence of input shares is always satisfied for the $d+1$ case.

Correctness of the sharing in $d+1$ case is achieved by verifying that each monomial of a shared term (product) in the unshared function f must be present in one of the output shares.

Consider again the function $xy+z$. One possible first-order $d+1$ sharing of it is given in Eq. (10).

$$\begin{array}{ll} (x, y, z) & \\ (0,0,0) & o_1 = x_0y_0 + z_0 \\ (0,1,*) & o_2 = x_0y_1 \\ (1,0,*) & o_3 = x_1y_0 \\ (1,1,1) & o_4 = x_1y_1 + z_1. \end{array} \tag{10}$$

The sharing can also be represented with a table as shown in the left side of Eq. (10). Each output share is a row of a table, and each column represents the shares of a different input variable. Entry in row i and column j is the allowed input share of jth input variable for ith output share.

Columns are representing the variables x, y and z, respectively. Compared to $td+1$ set representation, table representation restricts input shares per each variable separately, while output sets impose the restriction that was the same for all input variables.

The asterisk values indicate that we do not care about what input share of z is there, because the sharing of linear term z is ensured by combining rows 1 and 4 of the table. This also shows that the table representation of the sharing does not uniquely determine the exact formula for each output share, and there is a certain freedom in determining where we can insert the input shares.

For example, we can use the table of Eq. (10) to share function $x+y+xy+z$. There are two options for terms x_0 and x_1, rows 1 and 2 and rows 3 and 4, respectively. The same holds for terms y_0 and y_1, y_0 can be either in output share 1 or 3, and y_1 can be in output share 2 or 4.

Properties of *non-completeness* and *correctness* can be easily argued from the table representation. Since for every table row, each column entry in the table can represent only one input share of that column's variable, first-order *non-completeness* is automatically satisfied. For row 3 of the table in Eq. (10) by fixing the entries representing x to 1 and y to 0, we ensure that only x_1 and y_0 can occur in that output sharing. Hence, there is no way that x_0 or y_1 can be a part of that particular output share, which is the only way to violate *non-completeness* in $d+1$ sharing. *Correctness* of the table can be verified by checking correctness for every monomial in unshared function f individually. If the combined columns representing variables of the monomial contain all possible combinations of share indexes, sharing is correct. Indeed, if this is the case, all terms of shared product for each monomial can be present in the output sharing. Following example from Eq. (10), for monomial xy we see that all four combinations $\{(0,0),(0,1),(1,0),(1,1)\}$ are present in two columns representing variables x and y. Hence, all of the terms of shared product $xy = (x_0+x_1)(y_0+y_1) = x_0y_0+x_0y_1+x_1y_0+x_1y_1$ can be present in at least one output share. The same holds for $z = z_0+z_1$ as both combination $\{(0),(1)\}$ are present in output table of Eq. (10). Also, it is easy to see that the number of rows in correct sharing table is lower-bounded by the $(d+1)^t$, when the degree of the function is t.

Now, consider a function $xy+xz+yz$. One possible first-order $d+1$ sharing and its table are given in Eq. (11) with table entries on the left. Columns represent x, y and z, respectively.

$$\begin{array}{ll} (0,0,0) & o_1 = x_0y_0 + x_0z_0 + y_0z_0 \\ (0,1,1) & o_2 = x_0y_1 + x_0z_1 + y_1z_1 \\ (1,0,0) & o_3 = x_1y_0 + x_1z_0 \\ (1,1,1) & o_4 = x_1y_1 + x_1z_1 \\ (*,0,1) & o_5 = y_0z_1 \\ (*,1,0) & o_6 = y_1z_0. \end{array} \tag{11}$$

The table represented with Eq. (11) now has 6 rows representing different output shares, which is larger than theoretically minimal four shares, and is also easily obtained when we try to derive it by hand. The naive approach is to start by sharing xy into four shares. Next, we try to incorporate xz into these four shares by setting all indexes of z to be equal to y. The problem arises when we now try to add sharing of yz. In the existing four output shares, we have z

and y have the same indexes; thus, we are required to add two more shares for terms y_0z_1 and y_1z_0.

Further on, we will show that for any function with n input variables of degree $t = n - 1$ it is possible to have a $d + 1$ sharing with minimal $(d + 1)^t$ shares.

Definition 2 Table with n columns representing output sharing of a function of degree t with n input variables is referred to as a D^n-table. The number of rows of the table is the number of output shares for a given sharing. If the output sharing is correct, then D^n-table is t-degree correct D^n-table. t-degree correct D^n-table with minimal numbers of rows is called an *optimal* D^n-table. Optimal D^n-table that has $(d + 1)^t$ rows is called *ideal* D^n-table, denoted D^n_t-table

Obviously, for $t = n$ ideal D^n_n-table is just a table that contains all different $(d + 1)^t$ indexes of input variables in the terms of the shared product that occur when sharing a function of degree t. We can also consider each row of a D^n-table as an ordered tuple of size n. ith value in a such tuple represents the ith input variable, and its value is the allowed input share of that variable in the output share represented by the tuple. All tuple entries can have values from the set $\{0, \ldots, d\}$.

Definition 3 D^t-table D_1 is t-subtable of D^n-table D_2 for given t columns if D_2 reduced to these t columns is equal to D_1.

We have shown with the sharing in Eq. (10) how one can check the correctness of the table. Now, we generalize this by showing how to check if a given D^n-table can be used for sharing of any function of degree t. It turns out that it is sufficient to check correctness only for the terms of degree t, since if we are able to share a product of t variables with a number of output shares, we can also always share any product of a subset of these t variables using the same output shares.

It is easy to see that a D^n-table D can be used to share any function of degree t if and only if for any combination of t columns, D^t-table formed by chosen t columns contains all possible $(d + 1)^t$ ordered tuples of size t. In, other words, t-subtable of D for any t columns is t-degree correct D^t-table.

This comes from the fact that D^t-table that contains all possible $(d + 1)^t$ ordered t-tuples represents a correct sharing for functions of degree t. If this is true for any combination of t columns of D, we can correctly share any combination of products of size t from n input variables.

An example is given in Table 2 where D^3-table on the left can be used for first-order sharing of any function of degree 2 since all 3 D^2-tables obtained from it have all 4 possible ordered 2-tuples (0, 0), (0, 1), (1, 0) and (1, 1) as at least one of its rows.

Table 2 D^3-table and its 3 2-subtables

xyz	xy	xz	yz
000	00	00	00
011	01	01	11
100	10	10	00
111	11	11	11
001	00	01	01
110	11	10	10.

Next, we show how one can construct ideal D^n-table for any function for given n, d and $t = n - 1$. To recap, we first build a $(d + 1)^t \times n$ table D, where every row is a tuple of indexes (in a single row no variable index is allowed to be missing and, naturally, no variable index is duplicated) and t-subtable of D for any t columns is a t-degree correct D^t-table. Since $t = n - 1$, we can consider t-subtable generation as column removal from D. Such a D^n-table D is then equivalent to a sharing which fulfills the *correctness* and the *non-completeness* properties of TI. Constructing an ideal D^n_n-table is trivial by enumerating all ordered index n-tuples. The number of rows in it is $(d + 1)^n$.

Showing that a particular D^n-table with $(d + 1)^{n-1}$ rows is a D^n_{n-1}-table becomes equivalent to proving that removal of any single column (restriction to $n - 1$ columns or, equivalently, variables) from the D^n-table yields a D^{n-1}_{n-1}-table. Alternatively, any $(n - 1)$-subtable of D^n_{n-1}-table is a D^{n-1}_{n-1}-table.

Here, we will show how to build the D^n_t-table for the case when $t = n - 1$. For any given D^n_{n-1}-table and security order d, we will prove the existence of other d D^n_{n-1}-tables such that no n-tuple exists in more than one table. In other words, no two tables contain rows that are equal. We call such $d + 1$ D^n_{n-1}-tables *conjugate tables*, and the sharings produced from them *conjugate sharings*. Having all rows different implies that these $d + 1$ D^n_{n-1}-tables cover $(d + 1)(d + 1)^{n-1} = (d + 1)^n$ index n-tuples, i.e., all possible index n-tuples. Therefore, these $d + 1$ D^n_{n-1}-tables together form a D^n_n-table.

We build the $d + 1$ conjugate D^n_{n-1}-tables inductively. For a given d, we build $d + 1$ conjugate D^2_1-tables; then, assuming $d + 1$ conjugate D^n_{n-1}-tables exist, we construct $d + 1$ conjugate D^{n+1}_n-tables.

The *initial step* is simple: D^2_1 has two columns (for the variables x and y) and in each row i (enumerated from 0 to d) of each conjugate table j (enumerated from 0 to d) we set the value in the first column to be i, and the value of the second column to be $(i + j) \mod (d + 1)$, hence obtaining the $(d + 1)$ conjugate D-tables with $d + 1$ rows. Indeed, both columns of any of the constructed D^2_1-tables contain all values between 0 and d; so, by removing either column we always obtain a correct D^1_1-table. Also, this construction ensures that the second column never has the same index value in one row for different tables; therefore, no two rows

Fig. 3 Algorithm for optimal $d+1$ sharing

For $0 \le i \le d$ we construct the i-th D_n^{n+1}-table as follows:

1. Start with empty D-table P of $n+1$ variables.
2. For $0 \le j \le d$:
   (a) Take the j-th D_{n-1}^n-table and append one more column as last column.
   (b) Fill up the last column with the value $(i+j) \mod (d+1)$.
   (c) The obtained extended table is added to the D-table P.

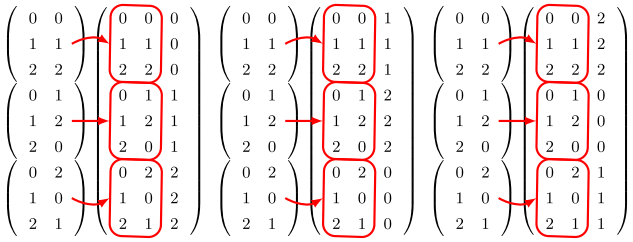


Fig. 4 Generating conjugate D_2^3-tables from D_1^2-tables

for different tables are the same, ensuring that formed tables are indeed conjugate.

Induction step Assume we have $d+1$ conjugate D_{n-1}^n-tables. Using them, we are now going to build $d+1$ conjugate D_n^{n+1}-tables in a following manner:

The example of the iterative step from Algorithm 3 is given in Fig. 4.

Lemma 1 *Given $d+1$ conjugate D_{n-1}^n-tables the algorithm described in Fig. 3 constructs $d+1$ conjugate D_n^{n+1}-tables.*

Proof First, let us show that the constructed $d+1$ D_n^{n+1}-tables are conjugate, i.e., there is no $(n+1)$-tuple which belongs to more than one of them. Let us assume there exists an $(n+1)$-tuple which belongs to two D_n^{n+1}-tables. This implies the existence of an n-tuple which belongs to two of the initial $d+1$ D_{n-1}^n-tables, contradicting the fact that these initial tables are conjugate.

Finally, any restriction to a particular set of columns has to have all the combinations of index n-tuples, i.e., the *correctness* property. In fact, it is sufficient to prove that any set of n columns in any of the new conjugate tables contains all possible n-tuples. Indeed, if we remove the last column in any of the so constructed tables, we get the union of the original $d+1$ D_{n-1}^n-tables forming one D_n^n-table. By definition, D_n^n-table satisfies this property. Lastly, we are left with the other case of removing one of the first n columns, which results in a table of dimensions $(d+1)^n \times n$. If we prove there are no duplicates among the $(d+1)^n$ tuples within this table, all combinations will be the table, making it again a D_n^n-table. Consider two n-tuples. If they are equal, their last indexes are also equal. By Algorithm 3 design, equality of the last indexes (these are in the $(n+1)$-st column) implies that the two $(n-1)$-tuples belong to one of the starting conjugate D_{n-1}^n-tables, i.e., they cannot be in different conjugate D_{n-1}^n-tables. However, for the $(n-1)$-tuples which belong to one of the starting D_{n-1}^n-tables by assumption it is known that there are no duplications and hence the considered two $(n-1)$-tuples cannot be equal. □

Theorem 1 *Any of the constructed conjugate D_{n-1}^n-tables by algorithm in Fig. 3 provides optimal sharing for given n, d and $t = n-1$.*

Proof The algorithm is applied inductively for the number of variables from 2 till n. Since one D_{n-1}^n-table contains exactly $(d+1)^{n-1}$ rows, we conclude it is optimal because this is the theoretical lower bound for the number of output shares for the case $t = n-1$. □

Recall that aside from a formula for the lower bound in [41], there was not much other work of applying $d+1$ TI to functions with higher degree than 2 with the only exception: the AES implementations by [46,47] where $d+1$ TI was applied to the inversion of $GF(2^4)$, which function has algebraic degree 3. When we tried to obtain by hand $d+1$ TI for PRINCE S-box of algebraic degree 3, we only managed to find output sharing for the most significant bit of the S-box with 12 and 44 output shares, for the first-order and the second-order $d+1$ TI prior to the discovery of Algorithm 3. The optimal solution is 8 and 27 output shares for these two cases, respectively, which is easily found using the approach described here.

Another benefit of using an algorithmic solution is it can easily be automated using a computer, removing the possibility of human error that is likely to occur, the more complex the ANF becomes.

It is well known that a balanced Boolean function of n variables has a degree at most $n-1$. Therefore, all $n \times n$ S-boxes which are permutations have a degree of at most $n-1$. Indeed, nearly all bijective S-boxes used in symmetric ciphers are chosen to have a maximum degree of $n-1$. In particular, inversion in the field always has the maximum degree of $n-1$, most notable example of its usage being the AES S-box. In the particular case of AES inversion, applying the algorithm shown here will produce the minimal number of shares, which is 128. This is, however, too large for any practical application.

The most notable exception where a low-degree function is used is KECCAK's [4] χ-function which is a 5×5 S-box of degree 2. A sharing with eight shares can be easily found for χ by hand, while a conjugate D^5-table will have 16 entries which corresponds to the optimal sharing for degree 4.

3.3 Optimal $d+1$ sharing for functions of up to 8 bits: $(d+1, n \leq 8, t < n-1, \forall d)$

In case when $t < n-1$ as was already shown from the KECCAK example, we see that the sharing obtained using Algorithm 3 does not give a solution with minimal number of output shares. Or in other words the method presented in the previous section is not optimal when the degree of the function is lower than $n-1$. Therefore, finding the optimal sharing for functions with a degree lower than $n-1$ needs to be performed in a different manner. In order to try to find a solution for this particular case, we must first reformulate our problem.

Recall that using sharing tables correctness of a given output sharing of a function F is equivalent to all of the terms in the ANF representation of F being correctly shared. In other words, for each term l of t variables, columns representing output shares should contain all different $(d+1)^t$ combinations with repetitions. We evaluate the shares of l from 1 to $(d+1)^t$ in lexicographic order and say that output share S covers ith share of term l if columns of S representing variables of l form the ith share of t variables.

The problem of finding a correct sharing can further be reduced to the set covering problem (SCP). Each different output share among different $(d+1)^n$ shares will be considered as a different set, and the family of these sets will be referred to as $\mathcal{D}$. The universe $\mathcal{U}$ of all elements to be covered is created by going through ANF, and for each term l, we add $(d+1)^t$ elements, where t is the degree of l, representing different shares of l. In other words, each share of each term is a separate element to be covered. Set S from $\mathcal{D}$ will contain element e from $\mathcal{U}$ if output share represented by S covers term share from F represented by e. Now given $\mathcal{D}$ and $\mathcal{U}$ we need to find subfamily $\mathcal{C} \subseteq \mathcal{D}$ with the minimal cardinality such that union of sets from $\mathcal{C}$ is $\mathcal{U}$. With respect to elements e from $\mathcal{U}$, there exists at least one set S from $\mathcal{C}$ that contains e.

We can further represent SCP in terms of decision variables. For all possible output shares from $\mathcal{D}$ $1 \ldots (d+1)^n$, we associate a $\{0, 1\}$ variable x_S denoting if share S is chosen. Now, our goal of finding correct and non-complete minimal sharing can be formulated as:

$$\text{minimize} \sum_{S \in \mathcal{D}} x_S \tag{12}$$

$$\text{subject to} \sum_{S: e \in S} x_S \geq 1, \ (\forall e, e \in \mathcal{U}). \tag{13}$$

Expression (12) is referred to as objective function, while inequalities given by (13) are called constraints.

Set covering problem is a well-known discrete optimization problem, which pops up in various applications, for example, in logical minimizer design, mobile network base station placement, etc. Hence, since we have reduced the problem of finding minimal sharing to set covering, we can try and use discrete optimization methods to find good solutions to it.

3.3.1 Constraint programming optimization techniques

We have used four different techniques to solve the underlying set covering problem: constraint programming, mixed integer programming, randomized greedy with restarts and simulated annealing with a greedy heuristic.

Constraint programming (CP) [42] focuses on finding a feasible solution given a number of constraints. Its original use is to determine if a problem is satisfiable. But it can also be used in minimization optimization, by finding a feasible solution with objective cost N, then adding new constraint such that objective functions has to be $\leq N$ and restarting the process. The cycle is repeated until the problem becomes infeasible, and the final objective value where a feasible solution is found is the minimal one. We have used freely available MiniZinc [37] software to solve our set covering problem.

Mixed integer linear programming (MILP) [45] relaxes the problem such that decision variables become non-binary, but continuous real values, $x_S \in [0, 1]$, and then tries to solve the underlying linear programming problem [18] to establish a lower bound of the objective function. Afterward, it tries to find the smallest solution such that all decision variables are integer, satisfying the original problem constraints. Like CP, MILP is able to prove the optimality of the solution. We have used Gurobi 9.0 [26] solver for this part.

The randomized greedy heuristic with restarts or iterated greedy (IG) is a technique where a solution is constructed in a greedy manner, and in each step, we take the set S that covers most uncovered elements so far. We stop when all elements are covered. All ties are broken randomly, i.e., if multiple sets cover an equal number of still uncovered elements, the algorithm randomly chooses one of them to add to the solution being built. We loop this approach multiple times and in the end take the solution from the iteration that has the smallest number of sets. Since ties are broken randomly, solutions will differ from run to run. The optimality of this technique cannot be proven, since the greedy algorithm finds local minima, not a global one for SCP. However, in practice, the solutions it finds are often close to optimal.

Simulated annealing [28] is a meta-heuristic where a neighbor solution with a higher objective function cost is accepted with a probability that gradually decreases during execution time. Intuitively, accepting worse solutions allows us to explore more of the search space and escape local minima. Over time lowering of the acceptance probability guides the search more and more toward good solutions, while the earlier rounds are used to explore large search space in a more indiscriminate fashion. The probability parameter is called temperature. We utilized the implementation approach given in [9,30] where we separate execution into multiple rounds. After each round, the temperature is decreased by a constant factor *cool*. In each round, a number I of neighbors are explored. Neighbor is constructed by removing some of the sets from the solution, then constructing a new solution using remaining sets as a starting point and adding new ones until all elements are covered again. We accept the new solution if it is better than the previous one, or if not we still accept it with probability $\exp(-\delta/\text{temp})$ where δ is the difference in objective functions of the new and the current solution.

The implementation details from [30] are given in the following table, while the simulated annealing algorithm for SCP is given in Algorithm 1.

Parameter	Description
A	Data structure providing relation information of which sets cover which element, typically given as a {0, 1} matrix.
cool	Temperature reduction ratio between rounds $temp_{r+1} = temp_r \times cool$
ρ	Used to determine the percentage of shares to be dropped from current solution, during neighbor construction.
R	Number of rounds to run.
I	Number of neighbors examined in each round.
rand()	Function that returns uniformly randomly value in range [0, 1].
$temp$	Temperature parameter used to determine the probability of accepting bad neighbor.
Construct()	Greedy heuristic method used to construct new covering after a percentage of shares has been removed from it.
Perturb()	Neighbor defining function. ρ factor of shares is stripped from the current solution. Shares removed have the least amount of uniquely covered elements.
RemoveRedundant()	Executed after a new solution has been constructed. It can happen that some of the chosen shares are redundant since all of the elements covered by it are already fully covered by other shares in the candidate solution.

Algorithm 1 Simulated Annealing algorithm for set cover problem.

Input: $A, R, I, temp, cool, \rho$
Result: Smallest found sharing $\mathcal{C}_{best}$
$\mathcal{C}^* := \{\}$
$\mathcal{C}^* := Construct(A, \mathcal{C}^*)$
$r := 0$
repeat
  $i := 0$
  repeat
    $\mathcal{C}' := Perturb(A, \mathcal{C}^*, \rho)$
    $\mathcal{C}' := Construct(A, \mathcal{C}')$
    $\mathcal{C}' := RemoveRedundant(A, \mathcal{C}')$
    $\delta := ObjectiveCost(\mathcal{C}') - ObjectiveCost(\mathcal{C}^*)$
    $rnd = rand()$
    if $(\delta \leq 0)$ *or* $(rnd \leq e^{-\delta/temp})$ **then**
      $\mathcal{C}^* := \mathcal{C}'$
      if $ObjectiveCost(\mathcal{C}' < ObjectiveCost(\mathcal{C}_{best})$ **then**
        $\mathcal{C}_{best} := \mathcal{C}'$
      end
    end
    $i := i + 1$
  until $i == I$
  $temp := temp \times cool$
  $r := r + 1$
until $r == R$
return $\mathcal{C}_{best}$

3.3.2 Sharing solutions

We have focused on applying four discrete optimization techniques on a set of different Boolean functions of up to eight bits, whose sharings can be used on any Boolean functions with the same number of input bits and algebraic degree. Since optimal sharing has already been explored for the case where degree $t \geq n - 1$ with n number of bits, we have investigated Boolean functions where $t < n - 1$. In order for the solution to be generic enough, we further assume the most extreme case. That is, if we have n variables and degree t, we search for a sharing of a function that has all $\binom{n}{t}$ t-degree terms present in the ANF. Sharing of such a function can be used for any n-bit function of degree t. However, a more efficient sharing for a given n-bit function F of degree t might exists, depending on the number of t-degree terms that are present in the ANF and their structure, and the same discrete optimization methodology can be applied on F to find possibly better sharing.

First, we have applied CP using Minizinc [37] with Chuffed 0.10.4 [16] and Google OR-tools [39] solvers. The resulting number of shares using CP solvers is given in Table 3. For $d = 1$, both options are able to find optimal sharing for all cases, except for 8-bit functions of degree 5, where a solution with 52 shares is found after few hours but without proof of optimality, even after running for the CP solver for several days on a regular PC. Optimality for other cases is proven within several tens of minutes. For the second-order case $d = 2$, CP solver is only able to prove

Table 3 Number of shares found using CP solver

t	$d=1$					$d=2$				
	2	3	4	5	6	2	3	4	5	6
$n=4$	**5**					**9**				
$n=5$	**6**	**10**				15	44			
$n=6$	**6**	**12**	**21**			–	–	–		
$n=7$	**6**	**12**	**24**	**42**		–	–	–	–	
$n=8$	**6**	**12**	**24**	52	**85**	–	–	–	–	–

Values in bold mean that solver proved optimality. Dashes mean that solver found no solution after running for an hour

Table 4 Number of shares found using MILP solver

t	$d=1$					$d=2$				
	2	3	4	5	6	2	3	4	5	6
$n=4$	**5**					**9**				
$n=5$	**6**	**10**				**11**	**33**			
$n=6$	**6**	**12**	**21**			**12**	**33**	119		
$n=7$	**6**	**12**	**24**	**42**		**12**	45	153	440	
$n=8$	**6**	**12**	**24**	52	**85**	14	63	–	–	–

Values in bold mean that solver proved optimality. Dashes mean that solver found no solution after running for an hour

optimality for the simplest of cases of 4 bits and degree 2. It is able to find some solutions when $n = 5$, but without proof of optimality. For $n > 5$, the solver is unable to provide any solutions within few minutes, so we deem it not suitable for those cases, due to the size of the search space. We did not see much difference in solution times between Chuffed and OR-tools solvers, although Chuffed seems to be able to finish the search slightly faster.

Next, we have tried to use MILP solver, in particular Gurobi 9.0 solver [26]. The resulting number of shares using MILP solver is given in Table 4. For $d = 1$, the solver was able to find the same solutions and prove optimality in less time. However, for the case of 8 bits and degree 5, a solution of 52 was found again, but without proof of optimality. When $d = 2$, MILP solver was more successful than CP one, finding optimal solutions for degree 2 functions for all $n = 4, 5, 6, 7$, and degree 3 functions of 5 and 6 bits. However, more complex cases seem to quickly become difficult for the solver.

From Tables 3 and 4, it becomes apparent that CP and MILP solvers struggle with $d = 2$, particularly for 8-bit functions of degree 4 or more. The CP solver struggles much more, as it is unable to find solutions for second-order sharings of functions with six or more input bits. This is not surprising due to the exponential increase in the number of decision variables. Hence, for the more difficult cases heuristics are the only possible way to find good solutions. Iterated greedy approach was made by using 100,000 runs and choosing the best solution among these runs. It finishes is just a matter of seconds even for harder cases. The program is sometimes able to find optimal solutions of CP and MILP approaches, but even when it does not the solutions it finds are within 30% of minimal, where a minimal solution is known. IG run results are given in Table 5.

Table 5 Number of shares found using IG heuristic

t	$d=1$					$d=2$				
	2	3	4	5	6	2	3	4	5	6
$n=4$	5					9				
$n=5$	6	12				11	36			
$n=6$	6	12	22			13	40	128		
$n=7$	6	12	24	47		14	45	138	409	
$n=8$	6	12	30	56	96	15	45	135	405	1387

Table 6 Number of shares found using SA heuristic

t	$d=1$					$d=2$				
	2	3	4	5	6	2	3	4	5	6
$n=4$	5					9				
$n=5$	6	10				11	33			
$n=6$	6	12	21			12	33	115		
$n=7$	6	12	24	42		12	40	130	379	
$n=8$	6	12	24	52	85	14	45	135	405	1234

As a way to improve on the results of the IG heuristic, we have also tested the simulated annealing technique. SA run results are given in Table 6. First, the IG technique was used with 20,000 runs to provide an initial solution, and then, SA was run with $R = 150$ rounds and $I = 100$ iterations in each round. The program ran extremely fast on a regular PC, and it was the slowest for 8-bit functions of degree six where it finished in about 5 min. SA program was able to find the same results for $d = 1$ as CP and MILP solver, finding optimal solutions in much faster times. For $d = 2$, it was able to improve on some of the instances compared to the IG approach, but solution quality was at most 10% better in instances where CP and MILP were unable to provide solutions in a reasonable amount of time. In some instances with 8 bits and degrees of 3, 4 and 5, it was unable to improve upon the solution provided by the IG approach. Modifying SA annealing parameters had a limited impact on the quality of the solution. We determine that the cooling factor of 0.91, starting temperature of 100 and removal coefficient ρ of 0.2 to be working well in almost all cases. ρ had the most impact on the improvements, and we notice that smaller values are more beneficial for larger t values, about 0.1, while values of 0.3 work better on smaller t values.

Finally, we can put together the solutions of all four approaches to collect the best ones. Aggregate results are

Table 7 Best sharing using all four approaches

t	$d=1$					$d=2$				
	2	3	4	5	6	2	3	4	5	6
$n=4$	**5**					**9**				
$n=5$	**6**	**10**				**11**	**33**			
$n=6$	**6**	**12**	**21**			**12**	**33**	115		
$n=7$	**6**	**12**	**24**	**42**		**12**	40	130	379	
$n=8$	**6**	**12**	**24**	52	**85**	14	45	135	405	1234

Values in bold mean that solver proved optimality

presented in Table 7. Examining the best found solutions, we can determine that the optimal sharing does not give a large increase in the number of output shares compared to the trivial bound of $(d+1)^t$. For $d=1$, increase is up to 50%, except in the hardest case with 8 variables and degree 5 functions where, we have 52 shares compared to the bound 32 shares, an increase of a little over 60%. A similar situation happens with $d=2$ where found solutions are within 70% increase. Due to the particular case of SCP for sharing being somewhat pathological, where all shares cover an equal amount of elements, it is difficult for the solver to find optimal solutions with the increasing number of total shares, while good solutions are still relatively easily discovered using greedy heuristic.

Comparison of our result with the recent ones presented in Sect. 3.2 and [48] is presented in Table 8. Obviously, the method presented in Sect. 3.2, although achieving optimality when $t=n-1$ is ineffective in cases where $t<n-1$. Greedy heuristic given in [48] finds solutions that are multiples of $(d+1)^t$. However, the authors only presented the solution for a specific case of 8-bit degree 3 function, and for $(n=4,t=2,d=2)$, $(n=4,t=2,d=3)$, $(n=5,t=2,d=4)$, $(n=5,t=3,d=3)$, $(n=6,t=3,d=3)$ cases, optimal solution is found while not providing more information. Hence, in Table 8 we indicate the smallest number of output shares algorithm presented in [48] could potentially find. Our method provides better results in all cases for first and second security order.

Concrete sharings are given in Tables 11, 12, 13 and 14 in "Appendix A." If we examine the found solutions for $d=1$, we can see that many of the solutions have symmetric order. One would assume that optimal solutions will always have sharings with such structure, but apparently this is not the case. For example, if we provide this additional constraint to the CP solver, it will no longer find optimal sharing of 7-bit functions of degree 2 to be 12, but 14. The symmetric structure is obtained from the CP solver, probably based on the heuristic it was using to parse the search space.

3.3.3 Sharing for AES cubic power functions

As an example of an application of the generic method presented here, we demonstrate the improved sharing of power functions x^{26} and x^{49} that are used to create AES inversion in work given by [35,48]. Both of them are cubic functions, and in [48], a sharing using 16 output shares is used to produce a first-order secure $d+1$ implementation. The sharing is only valid for the lowest bit of the power functions, but using the rotational symmetry of power functions in a normal basis, we can use the same sharing for other output bits as well. The normal basis generator chosen in [48] is $\beta=205$. Table 7 gives a sharing with 12 shares which is 4 shares or 25% less for a generic cubic function. However, if we apply the smallest sharing on the exact ANF of the cubic power functions, we can improve upon this result. Using the MILP set covering program, we were able to find a first-order sharing for both power functions x^{26} and x^{49} with ten shares each, six shares or 37.5 percent less than the original sharing. Furthermore, we have applied the same method to a second-order $d+1$ of the same power function and normal basis generator. Surprisingly, both functions have sharing with $(d+1)^t=27$ shares, the theoretical minimum. In order to make sure this is the best sharing, we have investigated seven other pairs of cubic functions that yield inversion when composed together: (13, 98), (19, 161), (38, 208), (52, 152), (67, 137), (76, 104), (134, 196). In addition, we have expanded the search over all 128 normal basis generators. However, the exhaustive search showed that ten shares is the smallest number of shares in every case. Other generators that can be used to produce the sharing with ten shares of the cubic power functions are {36, 96, 117, 124, 140, 199, 202}. Interestingly enough, in all the pairs of cubic power functions that produce inversion, the generators that have ten shares in output sharing are always the same. To make the notation as succinct as possible, we will only enumerate the chosen shares of power functions x^{26} and x^{49} with normal basis generator 205 in their lexicographical order, first share having index 0 and the last share having the index $(d+1)^n-1$. In other words, we look at all possible shares as n digit words in base $d+1$. For example, if we had a sharing with $d=2, n=4, t=2$ given as [2, 12, 25, 31, 44, 45, 60, 64, 77], it means that the actual nine shares are

(0, 0, 0, 2) (0, 1, 1, 0) (0, 2, 2, 1) (1, 0, 1, 1) (1, 1, 2, 2)
(1, 2, 0, 0)(2, 0, 2, 0) (2, 1, 0, 1) (2, 2, 1, 2).

A shortcut to getting actual output shares for first-order designs using this notation where allowed shares are 0 and 1 is to observe the binary representation of the index value, and each bit represents allowed share of that input variable. It should be noted that this representation of the sharing, while succinct, is not unique with respect to the actual distribu-

Table 8 Comparison to previously known results

t	$d=1$					$d=2$				
	2	3	4	5	6	2	3	4	5	6
Proposed methodology										
$n=4$	**5**					**9**				
$n=5$	**6**	**10**				**11**	**33**			
$n=6$	**6**	**12**	**21**			**12**	**33**	115		
$n=7$	**6**	**12**	**24**	**42**		**12**	40	130	379	
$n=8$	**6**	**12**	**24**	52	**85**	14	45	135	405	1234
Construction from Table 3										
$n=4$	8					27				
$n=5$	16	16				81	81			
$n=6$	32	32	32			243	243	243		
$n=7$	64	64	64	64		729	729	729	729	
$n=8$	128	128	128	128	128	2187	2187	2187	2187	2187
Construction from [48]										
$n=4$	≥ 8					**9**				
$n=5$	≥ 8	≥ 16				≥ 18	≥ 54			
$n=6$	≥ 8	≥ 16	≥ 32			≥ 18	≥ 54	≥ 162		
$n=7$	≥ 8	≥ 16	≥ 32	≥ 64		≥ 18	≥ 54	≥ 162	≥ 486	
$n=8$	≥ 8	≥ 16	≥ 32	≥ 64	≥ 128	≥ 18	≥ 54	≥ 162	≥ 486	≥ 1458

Values in bold indicate that the found solution is optimal with respect to the number of output shares, meaning no better solution can be found

tion of ANF terms, as low-degree monomials have multiple possible output shares they can be a part of. For the first-order sharing of an 8-bit function, there are $2^8 = 256$ possible shares, while for the second-order sharing there are $3^8 = 6561$ total shares. The chosen shares for the first-order sharing are given as follows:

$$\text{shares}(x^{26}) = [0, 30, 43, 93, 114, 154, 181, 195, 232, 246]$$
$$\text{shares}(x^{49}) = [0, 30, 79, 115, 124, 165, 187, 214, 217, 234].$$

The chosen shares for the second-order sharing are given as follows:

$$\begin{aligned}\text{shares}(x^{26}) = [&0, 439, 626, 796, 1145, 1338, 1511, 1938,\\ &2044, 2388, 2575, 2690, 3103, 3290, 3474,\\ &3863, 3975, 4162, 4533, 4639, 5069, 5212,\\ &5408, 5754, 5927, 6111, 6550]\end{aligned}$$

$$\begin{aligned}\text{shares}(x^{49}) = [&0, 338, 673, 930, 1025, 1324, 1617,\\ &1919, 2011, 2218, 2553, 2882, 3148, 3231,\\ &3542, 3745, 4053, 4148, 4436, 4762, 5097,\\ &5276, 5368, 5676, 5963, 6271, 6354].\end{aligned}$$

3.4 Optimizing secure AES schedule for maximum throughput

The design of side-channel secure AES schemes has been traditionally optimized for very low area, most commonly using a single S-box instance to compute both the SubBytes and round key update. While this approach is justified, as there are multiple applications such as RFID devices, where area and power are quite constrained, it still fares rather poorly with regards to latency and throughput. In the case of AES-128, the AES variant that is most prominent in the academic literature, taking this approach limits the execution time asymptotically to 200 cycles per execution with each round performing 20 S-box operations. To ease the notation from now on, we will refer to AES-128 simply as AES.

Adding the required ShiftRows, MixColumns, key schedule operations and the several clock cycle latency of the S-box in side-channel protected implementations increases the total latency even further. Several side-channel AES implementations [7,15,25,32] require at least 246 cycles to complete one AES encryption. Recently, several approaches have been proposed that achieve the throughput of one encryption per 200 cycles, while achieving latency of 216 cycles [25,47]. But they are only feasible for S-boxes that compute the output in five or less cycles, which is a restrictive requirement, with most existing side-channel secure AES S-box implementations take six or more cycles, going as far as nine cycles in the work presented in [21]. Here, we propose a new method of scheduling that can achieve a throughput of 20 cycles per round for S-box latency up to 11 cycles, allowing for serialized AES implementation with the highest possible latency/throughput. First, we declare a set of data dependencies that need to be followed in order to have a correct AES round implementation using a serialized S-box:

- All state bytes that are to be overwritten by the S-box output need either to be in the S-box pipeline or to have finished S-box operation.
- All MixColumn input operation column bytes need to have finished S-box operation. The last byte can be collected straight from the S-box output and not from the state registers, however.
- State byte in the next round can only be used as S-box input if the MixColumn operation for that byte's column has been performed.
- Key byte must not be updated before it is used in the current round.
- First four bytes of the key can only be updated if the corresponding S-box output has been completed.
- Except for the first column of the key, each byte in subsequent columns can only be updated if the corresponding byte from the previous column has been updated.

We can program these rules in the MiniZinc constraint modeling language, providing latency of the S-box as a parameter. MiniZinc will then try to satisfy all constraints and output a solution if it exists or if it cannot find a solution after exploring the entire search space it will state that the problem is unsatisfiable. We have run our constraint model for different latency, and it has always found a scheduling for latency up to 11 clock cycles. The model is unsatisfiable for S-box latency of 12, proving that we cannot find a solution for any S-box latency of 12 or more cycles.

We illustrate our solution on the nine-cycle latency S-box, which can, for example, be used to improve the scheduling of M&M implementation [21].

All other solutions are presented in "Appendix B." The state and key update schedule is shown in Fig. 5, while the corresponding timing diagram is shown in Fig. 6. In our design, ShiftRows operation is performed together with the S-box output. That is, the output of the S-box is written to the state byte where it would be after performing the shift row operation. Another way to look at it is that each column is written back diagonally with rotational wrapping around to the state matrix, not back to itself. This is clearly shown in Figs. 5 and 6. MixColumn operation is performed as soon as all of the column bytes are ready, i.e., in 28th, 24th, 29th and 30th cycle, respectively. The result of MixColumn is ready in the following cycle. The key addition is only performed after the MixColumn operation, except in the last round where it is performed before ShiftRows. The initial key addition is performed during the loading stage. From the timing diagram in Fig. 6, we see that the actual output for all the state bytes is ready 10 cycles after the beginning of the following round.

It should be noted that the trade-off in using our approach is an additional multiplexer as we are required to read from and write to arbitrary bytes of the state and the key matrix, which slightly increases the area occupied by the implementation. However, we can achieve speed-ups of roughly 20–35% for existing implementations with S-box latency greater than 6.

4 Implementations, results and evaluation

4.1 Implementations

In this section, we provide eight implementations of PRINCE: first- and second-order protection, using $td + 1$ and $d + 1$ sharing, with and without S-box decomposition.

4.1.1 PRINCE unprotected implementation

Figure 7 represents the architecture of unprotected round-based PRINCE. Here, we evaluate the S-box in one cycle. One encryption is performed in 12 clock cycles. We utilize the fact that the S-box and its inverse are in the same equivalence class to reuse the same circuitry for both first and last rounds, with the exception that affine transformation circuits A_{io} are used during the evaluation of S^{-1}. By adding an extra multiplexer, we can perform decryption as well. To minimize the overhead of adding decryption, we use the round counter design as explained in [33].

Following Fig. 7, when evaluating the S-box, the data travel through multiplexers $\alpha_1 - \beta_2 - \delta_1$, except in the first round where the path is $\alpha_1 - \beta_1 - \delta_1$. Similarly, when evaluating the inverse S-box, active path is $\alpha_2 - \gamma_1 - \delta_2$, except in the last round where we use $\alpha_1 - \gamma_2 - \delta_2$.

Next, in Sects. 4.1.2–4.1.6 we first present the TIs of the S-box and its building block Q_{294}, and then, in Sect. 4.1.7 we present the actual secure implementations of PRINCE.

4.1.2 TIs of Q_{294}

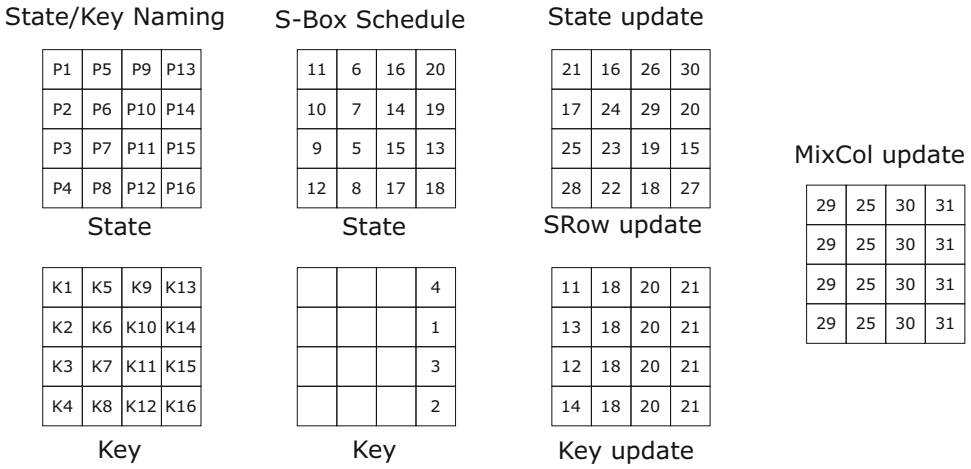
We have implemented $td + 1$ and $d + 1$ variants of TI for both the first- and the second-order Q_{294} implementations. We use the first-order $td + 1$ direct TI sharing [11] and the second-order $td + 1$ sharing with five input shares and ten output shares, as explained in [6]. For the $d + 1$ first- and second-order implementations, we used sharing given in [41]. Compression is applied to the second-order $td + 1$ and both $d + 1$ versions. Detailed description of the hardware architecture is given in "Appendix B."

4.1.3 First-order secure $td + 1$ TI of PRINCE S-box

For the first-order $td + 1$ design, we generated sharing with four input and output shares following [11].

Output bits are refreshed using Eq. (2), requiring 12 random bits per S-box. The exact sharing does not fit within the

Fig. 5 S-box pipeline schedule



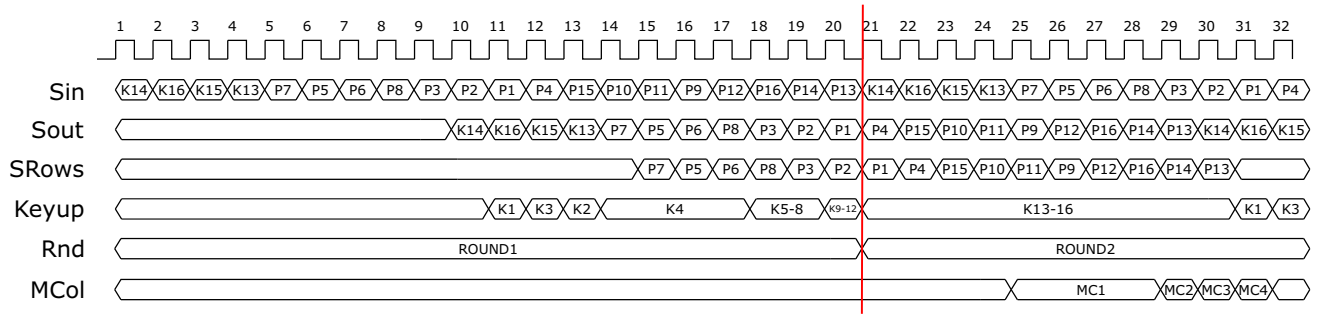


Fig. 6 Timing diagram of S-box usage in the pipeline schedule

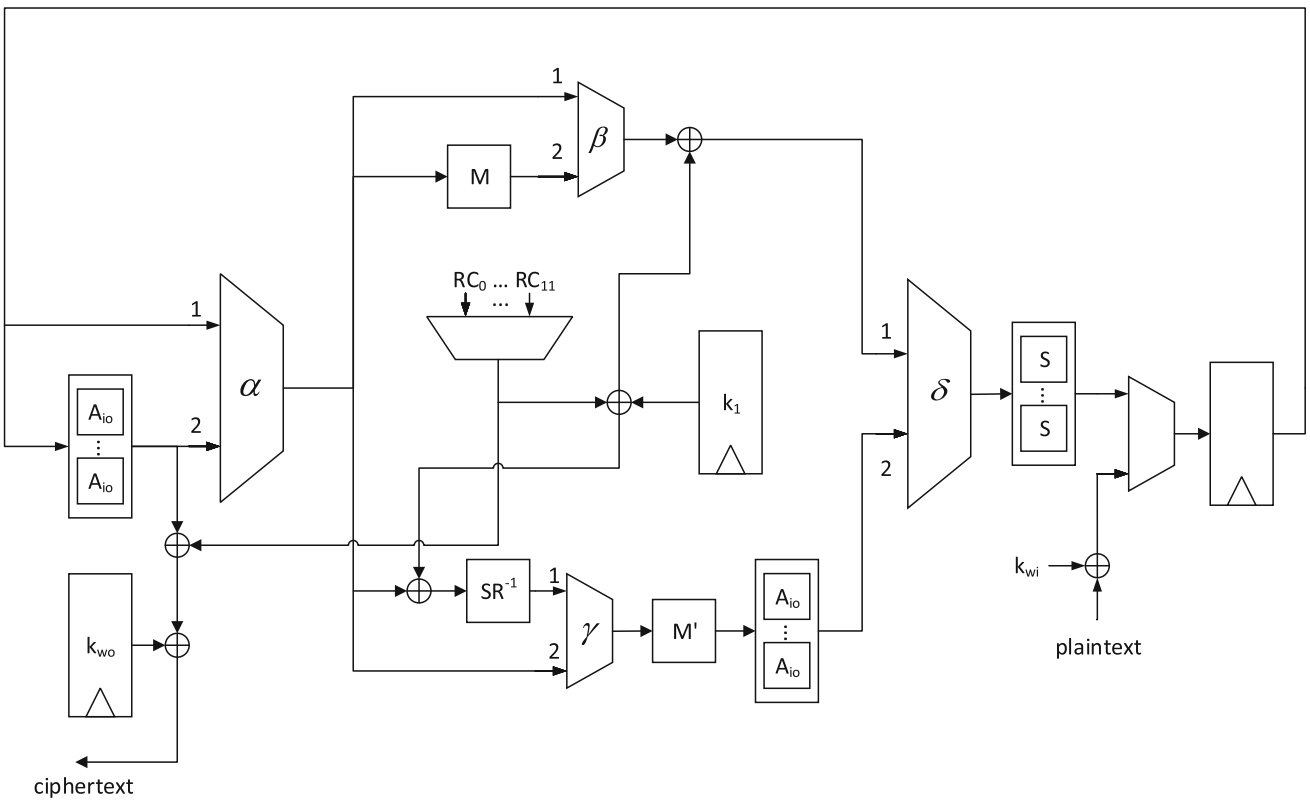


Fig. 7 Unprotected PRINCE round-based architecture

margins of this paper and will be provided as a Verilog code online [12].

4.1.4 Second-order secure *td* + 1 TI of PRINCE S-box

To create a second-order secure masking for the PRINCE S-box with $d = 2$, we have used the iterated greedy algorithm described in Sect. 3.1. This algorithm provides a solution that has 17 output shares and 8 input shares. Compared to the solution given in [6], which had 35 output shares and 7 input shares, we have reduced the total number of shares by almost a half. All output bits are refreshed using the ring re-masking from Eq. (1), requiring 68 random bits per S-box. Since the rest of the PRINCE core is using three shares (see Sect. 4.1.7), we generate five extra shares before the S-box input which consumes 20 random bits extra. Therefore, the whole S-box evaluation uses 88 random bits. Again, the exact sharing will be provided as an online Verilog code [12].

4.1.5 First-order secure *d* + 1 TI of PRINCE S-box

To implement the first-order secure masking of PRINCE S-box, with $d = 1$, we use the algorithm described in Sect. 3.2 to obtain a conjugate D_3^4-table. This table represents an optimal solution for two input shares with eight output shares for each input/output bit of the S-box. Recall that the PRINCE S-box is a 4×4-bit S-box and that it has a degree 3.

All output bits are refreshed using the mask refreshing as given in Eq. (2), requiring 7 bits of randomness per output bit, or 28 bits per S-box in total. The optimal sharing is given in Eq. (14) as conjugate D_3^4-table. The exact sharing will again be provided as an online Verilog code [12].

$$
\begin{aligned}
&(x, y, z, w)\\
&(0, 0, 0, 0)\\
&(1, 1, 0, 0)\\
&(0, 1, 1, 0)\\
&(1, 0, 1, 0)\\
&(0, 0, 1, 1)\\
&(1, 1, 1, 1)\\
&(0, 1, 0, 1)\\
&(1, 0, 0, 1).
\end{aligned} \tag{14}
$$

As an example consider the first coordinate function of PRINCE as given in Eq. (6): $o^1 = 1 \oplus wz \oplus y \oplus zy \oplus wzy \oplus x \oplus wx \oplus yx$. Then, the optimal sharing is obtained as follows:

$$
\begin{aligned}
o_1^1 &= 1 + w_0z_0 + y_0 + z_0y_0 + w_0z_0y_0 + x_0 + w_0x_0 + y_0x_0\\
o_2^1 &= y_1 + z_0y_1 + w_0z_0y_1 + x_1 + w_0x_1 + y_1x_1\\
o_3^1 &= w_0z_1 + z_1y_1 + w_0z_1y_1 + y_1x_0\\
o_4^1 &= z_1y_0 + w_0z_1y_0 + y_0x_1\\
o_5^1 &= w_1z_1 + w_1z_1y_0 + w_1x_0\\
o_6^1 &= w_1z_1y_1 + w_1x_1\\
o_7^1 &= w_1z_0 + w_1z_0y_1\\
o_8^1 &= w_1z_0y_0.
\end{aligned} \tag{15}
$$

Continuing for the second bit's algebraic function $o^2 = 1 + yw + yz + xz + yzw + xyz$ optimal sharing is:

$$
\begin{aligned}
o_1^2 &= 1 + y_0w_0 + y_0z_0 + x_0z_0 + y_0z_0w_0 + x_0y_0z_0\\
o_2^2 &= y_1z_0w_0 + x_1y_1z_0\\
o_3^2 &= y_1w_0 + y_1z_1 + y_1z_1w_0 + x_0y_1z_1\\
o_4^2 &= x_1z_1 + y_0z_1w_0 + x_1y_0z_1\\
o_5^2 &= y_0w_1 + y_0z_1 + x_0z_1 + y_0z_1w_1 + x_0y_0z_1\\
o_6^2 &= y_1z_1w_1 + x_1y_1z_1\\
o_7^2 &= y_1w_1 + y_1z_0 + y_1z_0w_1 + x_0y_1z_0\\
o_8^2 &= x_1z_0 + y_0z_0w_1 + x_1y_0z_0.
\end{aligned} \tag{16}
$$

Optimal sharing for the third bit with algebraic function $o^3 = w + x + zw + xw + xz + xzw + xyz$ is:

$$
\begin{aligned}
o_1^3 &= w_0 + x_0 + z_0w_0 + x_0w_0 + x_0z_0 + x_0z_0w_0 + x_0y_0z_0\\
o_2^3 &= x_1z_0w_0 + x_1y_1z_0\\
o_3^3 &= z_1w_0 + x_0z_1w_0 + x_0y_1z_1\\
o_4^3 &= x_1w_0 + x_1z_1 + x_1z_1w_0 + x_1y_0z_1\\
o_5^3 &= w_1 + z_1w_1 + x_0w_1 + x_0z_1 + x_0z_1w_1 + x_0y_0z_1\\
o_6^3 &= x_1z_1w_1 + x_1y_1z_1\\
o_7^3 &= z_0w_1 + x_0z_0w_1 + x_0y_1z_0\\
o_8^3 &= x_1 + x_1w_1 + x_1z_1 + x_1z_0w_1 + x_1y_0z_0.
\end{aligned} \tag{17}
$$

Finally, for the fourth bit of PRINCE S-box and its function $o^4 = 1 + z + x + yz + xy + yzw + xzw + xyw$ optimal sharing is given with:

$$
\begin{aligned}
o_1^4 &= 1 + z_0 + x_0 + y_0z_0 + x_0y_0 + y_0z_0w_0 + x_0z_0w_0 + x_0y_0w_0\\
o_2^4 &= x_1y_1 + y_1z_0w_0 + x_1z_0w_0 + x_1y_1w_0\\
o_3^4 &= y_1z_1 + y_1z_1w_0 + x_0z_1w_0 + x_0y_1w_0\\
o_4^4 &= y_0z_1w_0 + x_1z_1w_0 + x_1y_0w_0\\
o_5^4 &= z_1 + y_0z_1 + y_0z_1w_1 + x_0z_1w_1 + x_0y_0w_1\\
o_6^4 &= y_1z_1w_1 + x_1z_1w_1 + x_1y_1w_1\\
o_7^4 &= y_1z_0 + x_0y_1 + y_1z_0w_1 + x_0z_0w_1 + x_0y_1w_1\\
o_8^4 &= x_1 + x_1y_0 + y_0z_0w_1 + x_1z_0w_1 + x_1y_0w_1.
\end{aligned} \tag{18}
$$

Note that the sharing of the cubic terms is unique, while there are more options for the sharings of the lower-degree terms and that is why one needs to avoid repetitions.

4.1.6 Second-order secure *d* + 1 TI of PRINCE S-box

For the second-order secure masking of PRINCE S-box, with $d = 2$, we again use the algorithm described in Sect. 3.2 to obtain the conjugate D_3^4-table. This table represents an optimal solution with 3 input shares and 27 output shares for each

input/output bit of the S-box. All output bits are refreshed using the ring re-masking as given in Eq. (1), requiring 108 random bits for the whole S-box. The optimal sharing is given in Eq. (19) as a conjugate D_3^4-table, where the same rules are used as in the previous section for $d = 1$ case. The exact sharing will be provided as an online Verilog code [12].

$$
\begin{array}{lll}
(x, y, z, w) & & \\
(0,0,0,0) & (0,0,1,1) & (0,0,2,2) \\
(1,1,0,0) & (1,1,1,1) & (1,1,2,2) \\
(2,2,0,0) & (2,2,1,1) & (2,2,2,2) \\
(0,1,1,0) & (0,1,2,1) & (0,1,0,2) \\
(1,2,1,0) & (1,2,2,1) & (1,2,0,2) \\
(2,0,1,0) & (2,0,2,1) & (2,0,0,2) \\
(0,2,2,0) & (0,2,0,1) & (0,2,1,2) \\
(1,0,2,0) & (1,0,0,1) & (1,0,1,2) \\
(2,1,2,0) & (2,1,0,1) & (2,1,1,2). \qquad (19)
\end{array}
$$

4.1.7 Protected implementations of the PRINCE cipher

Figure 8 depicts the data path of the hardware implementation for the four protected round-based implementations of PRINCE which use the S-box decomposition. All the data lines have the width of $64 \times s$ bits, where s is the number of input shares. The only exception is the RC constant output, which has a width of 64 bits. The sharing of the non-linear layer, followed by the linear layer, re-masking and compression layer is denoted with the NLRC block in Fig. 8. Hardware implementations of NLRC layers of Q_{294} are discussed in "Appendix B."

In order to support both encryption and decryption, input and output whitening keys, k_{wi} and k_{wo} are either k_0 or k'_0, depending on what is being executed. We only require one extra multiplexer to implement this feature. When evaluating the S-box, the data path of the multiplexers is $\alpha_1 - \beta_2 - \delta_1$ in the first, $\alpha_2 - \delta_2$ in the second and $\alpha_2 - \delta_3$ in the third clock cycle, except in the first round where the third cycle path is $\alpha_1 - \beta_1 - \delta_1$. Similarly, when evaluating the inverse S-box, the active inputs of multiplexers are $\alpha_3 - \gamma_1 - \delta_4$ in the first, $\alpha_2 - \delta_2$ in the second, and $\alpha_2 - \delta_3$ in the third clock cycle, except in the last round where the path during the third cycle is $\alpha_2 - \gamma_2 - \delta_4$.

For $td + 1$ implementations, we use three and five shares, respectively, for the affine operations in order to reduce the amount of randomness required for the execution. This incurs an additional penalty in the area occupied by the implementation. Recall that the output of the S-box component functions for $td + 1$ TI is shared with three and ten shares, respectively, for the first- and the second-order secure implementations. Re-masking and compression are done only for the second-order $td + 1$ TI. The $d + 1$ implementations use two and three shares for the first- and the second-order secure implementation, respectively. The output of the S-box component functions is shared with four and nine shares, respectively, for the first- and the second-order secure implementations. Re-masking and compression are required in both cases.

The round constant is added to only one of the shares. The key is shared with the same number of shares as the plaintext. Unlike most of the secure implementations available in the literature, in this paper we focus on the round based implementation instead of the serialized one. This greatly reduces the execution time, at the expense of increased area and the required amount of randomness per clock cycle. In order to decrease area, we employ multiplexers to avoid instantiating additional registers for the three stages of the S-box evaluation. Since PRINCE has 12 rounds and each with S-box evaluations has 3 (for $d = 1$ and $td + 1$ TI) or $2 * 3$ stages (for $d + 1$ TI and $td + 1$ TI but $d > 1$), the total execution takes 36 or 72 clock cycles.

Figure 9 represents the architecture for the four protected round-based implementations of PRINCE without S-box decomposition. The architecture is almost the same as for the unprotected design. The implementations of NLRC layers of PRINCE S-box are discussed in Sects. 4.1.3–4.1.6.

When evaluating the S-box, the data path of the multiplexers is $\alpha_1 - \beta_2 - \delta_1$ except in the first round where the path in the first clock cycle is $\alpha_1 - \beta_1 - \delta_1$. Similarly, when evaluating the inverse S-box, the active inputs of multiplexers are $\alpha_2 - \gamma_1 - \delta_2$, except in the first cycle of the last round where the path is $\alpha_1 - \gamma_2 - \delta_2$. Unlike in the unprotected version, the S-box evaluation takes two cycles (S-box layer and linear layer are separated into two cycles); hence, it takes 24 cycles for one encryption/decryption operation. The exception is the first-order $td + 1$ implementation where S-box evaluation takes one cycle, making encryption/decryption latency 12 cycles.

For $td + 1$ implementations, we use four and three shares, respectively, for the affine operations. Recall that the output of the S-box component functions for $td + 1$ TI is shared with 4 and 17 shares, respectively, for the first- and the second-order secure implementations. Compression is required only for the second-order $td + 1$ implementation, while re-masking is applied for both of them. The $d + 1$ implementations use two and three shares for the first- and the second-order secure implementation, respectively. Recall that the output of the S-box component functions is shared with 8 and 27 shares, respectively. Re-masking and compression are required in both cases.

4.1.8 Randomness reduction

The resharing of the first-order secure implementation is performed according to the DOM [24] rules, in which

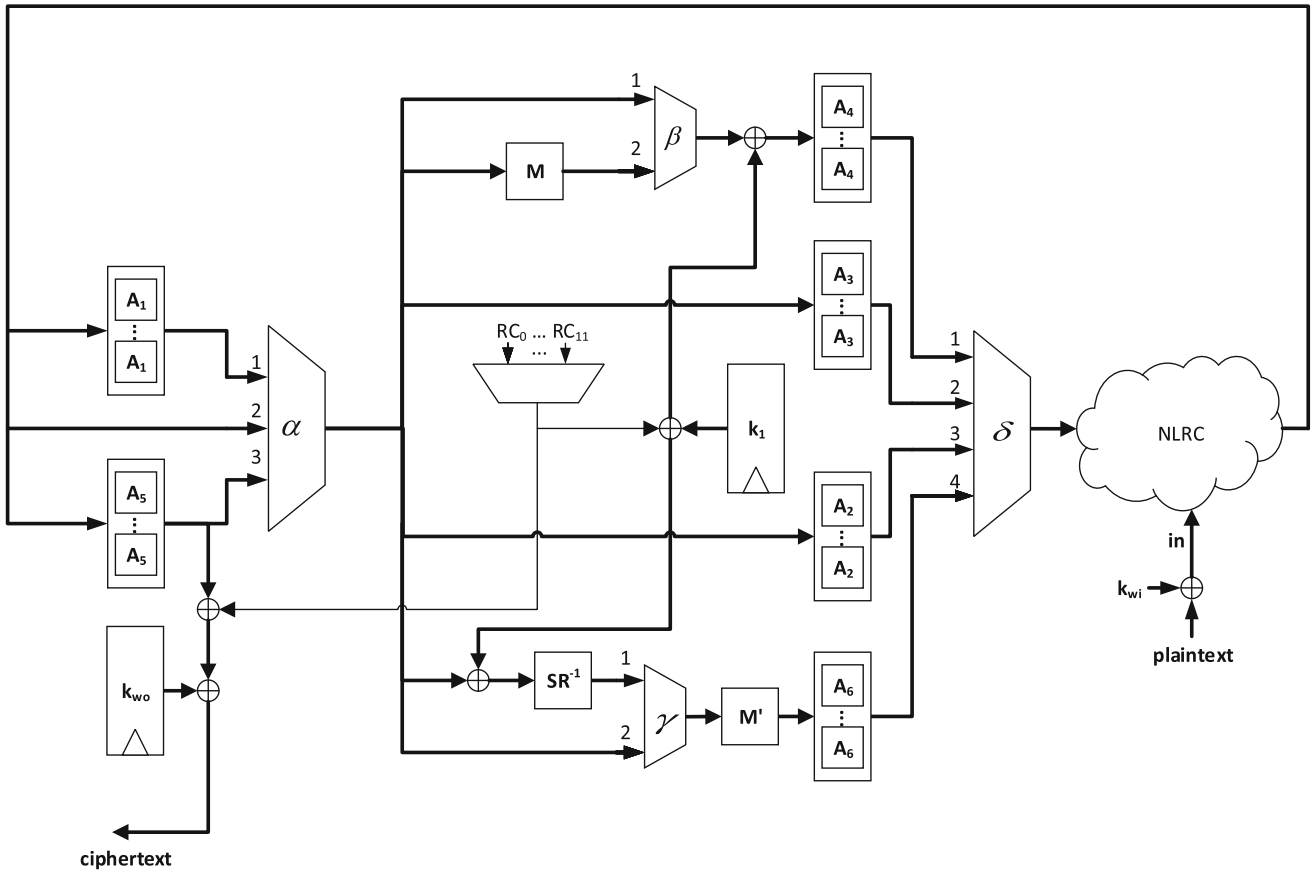


Fig. 8 TI PRINCE round-based architecture with decomposition

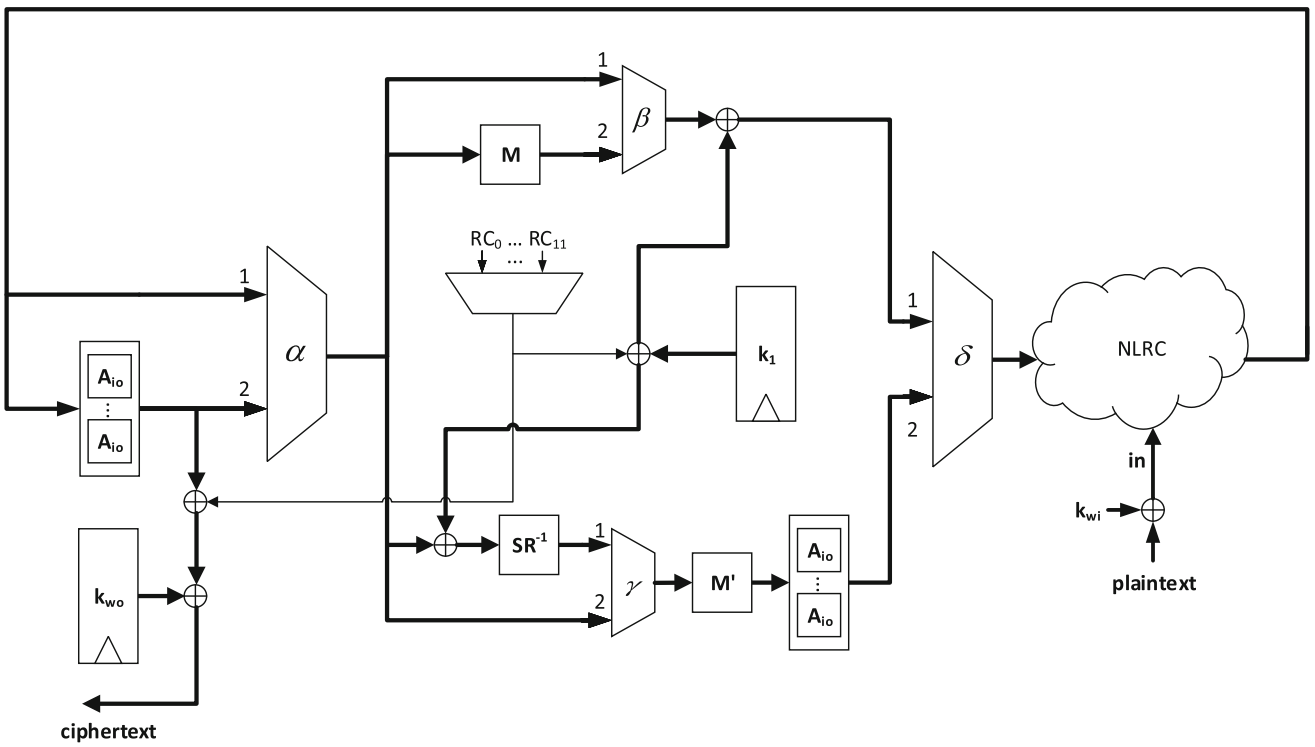


Fig. 9 TI PRINCE round-based architecture without decomposition

complementary domains are re-masked using the same randomness, with no re-masking for output shares containing only one domain. It can be noticed from Eq. (14) that output shares o_1, o_2, o_3, o_4 have complementary domains of shares o_6, o_5, o_8, o_7, respectively. If we consider eight output shares of 4-bit length, the re-masking is given with Eq. (20), where o_i and ro_i are S-box outputs before and after re-masking and r_i are random 4-bit values, requiring 12 random bits. Recombination is achieved by adding shares ro_1, ro_2, ro_3, ro_4 into one and ro_5, ro_6, ro_7, ro_8 into another recombined share.

$$\begin{aligned} ro_1 &= o_1 \quad ro_2 = o_2 + r_1 \quad ro_3 = o_3 + r_2 \quad ro_4 = o_4 + r_3 \\ ro_5 &= o_5 + r_1 \quad ro_6 = o_6 \quad ro_7 = o_7 + r_3 \quad ro_8 = o_8 + r_2. \end{aligned} \tag{20}$$

Let us take a closer look at the PRINCE round structure. As explained in Sect. 2.4, the mixing layer consists of matrices M, M' or M^{-1}. Recall that M can be obtained from M' using nibble shuffling operation SR, i.e., $M = \text{SR} \circ M'$. The involution 64×64 matrix M' is constructed as block diagonal matrix with entries (M_0, M_1, M_1, M_0), where M_0 and M_1 are 16×16 matrices.

This structure implies that 16-bit chunks of the state are processed independently. Therefore, we can use the same randomness for all four 16-bit blocks for the attacker case of $d = 1$ and $d = 2$.

Namely, assuming the PRINCE state is composed of 16 nibbles enumerated from 0 to 15 then the following four groups can be formed (0, 1, 2, 3), (4, 5, 6, 7), (8, 9, 10, 11) and (12, 13, 14, 15). When evaluating the S-boxes in a given group, the randomness required can be reused for the evaluation of the S-boxes in the other groups. It can also be observed that the nibble shuffling

$$\begin{aligned} &\text{SR} : (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15) \\ &\quad \rightarrow (0, 5, 10, 15, 4, 9, 14, 3, 8, 13, 2, 7, 12, 1, 6, 11) \\ &\text{SR}^{-1} : (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15) \\ &\quad \rightarrow (0, 13, 10, 7, 4, 1, 14, 11, 8, 5, 2, 15, 12, 9, 6, 3) \end{aligned}$$

does not cause mixing of the S-boxes outputs obtained with the same randomness. Hence, using this structure in round-based implementation reduces the amount randomness by a factor of four. Although the probing model does not accurately reflect the HW specifics like glitches and the parallelism of the implementations, note the first increases the attacker capabilities, while the second diminishes it, below we will still use this attacker model in the argumentations.

When we consider the first-order attacker, he can probe one share out of two at a given cycle; thus, the reuse of randomness is not exploitable. For the case of second-order attacker, he is able to get either (a) two shares out of three of one nibble or (b) one share of the two nibbles using the same randomness at a given cycle. In case of (a), again the attacker cannot exploit the reuse of randomness since he does not know anything about this second value which can be combined with his own knowledge for the first value. In the case of (b), the attacker is unable to mount a bivariate attack using points from different rounds (and hence cycles) due to the re-masking after each operation and the key addition (all of them done in the same cycle), and since the nibble shuffling does not cause mixing of the S-boxes outputs in the same round.

4.2 Results

To demonstrate our results, we first use the 90-nm CMOS library provided by TSMC and consider the worst power voltage temperature (PVT) corner case (the temperature of +125° C and the supply voltage of 1.0 V). Please note that the worst corner case is used in almost all industrial applications. In most of the scientific publications, however, a typical corner case is usually reported, giving an optimistic estimate of what will eventually be manufactured in silicon. However, in order to have a fair comparison, we have also synthesized our designs as well as the previously existing TI PRINCE implementation [33] using TSMC 90-nm library using the typical case of +25° C. The authors of [33] provided us with their implementations, allowing for an apple to apple comparison of the best designs using the same compiler and library, as the synthesis results for design presented in [33] differ from the original paper.

For synthesis, we use Cadence Encounter RTL Compiler version 14.20-s034 to evaluate the proposed architectures. The designs are synthesized using the operating frequency of 10 MHz and the power consumption is estimated by simulating a back-annotated post-synthesis netlist with 100 random test vectors, using Cadence Incisive Enterprise Simulator version 15.10.006. Energy is calculated for one complete encryption/decryption operation. Table 9 shows area, power and energy consumption, the number of random bits required per clock cycle and the maximum frequency for all the hardware implementations measured at the worst PVT corner case. All the designs have their unconstrained critical paths well below 100 ns and, hence, collecting area figures and power/energy consumption at the frequency of 10 MHz guarantees fair comparison.

It should be noted that the area, power and energy consumption of PRNG is not included in Tables 9 and 10, thus making the obtained results favoring solutions with more randomness. In practice, one must take the impact of PRNG into account and it is expected that higher-throughput PRNGs consume more area, power and energy. However, in most security applications, PRNG is a component shared between multiple resources, making its impact on the overall area, power and energy consumption limited.

Table 9 Area/power/energy/randomness/latency/max frequency comparison at worst-case PVT

PRINCE	Area[a] (GE)	Power[a] (uW)	Energy[a] (pJ)	Rand/Cycle (bits)	Clock # (cycle)	f_{max} (MHz)	Latency @ f_{max} (ns)
Unprotected	3597	57	69	0	12	285	42.2
[33] 1st $(td+1)^2$	9484	66	264	0	40	328	122
1st $(d+1)$[b]	8701	97	698	24	72	260	277
1st $(td+1)$[b]	14,153	75	270	0	36	268	134
1st $(d+1)$[3,4]	11,613	99	238	112	24	285	84.0
1st $(td+1)$[3,4]	31,116	576	691	48	12	204	58.8
2nd $(d+1)$[b]	13,421	161	1159	72	72	250	288
2nd $(td+1)$[b]	18,767	232	1670	40	72	243	296
2nd $(d+1)$[3,4]	32,444	374	898	432	24	292	82.2
2nd $(td+1)$[3,4]	177,647	1533	3679	352	24	282	85.1

[a]Area, power and energy figures given at 10 MHz operating frequency
[b]With S-box decomposition
[c]Without S-box decomposition
[d]Designs introduced in this paper

Table 10 Area/power/energy/randomness/latency/max frequency comparison at normal case PVT

PRINCE	Area (GE)	Power (uW)	Energy (pJ)	Rand/Cycle (bits)	Clock # (cycle)	f_{max} (MHz)	Latency @ f_{max} (ns)
Unprotected	3596	57	68	0	12	381	31.5
[33] 1st $(td+1)$[a]	9502	66	264	0	40	421	95.1
1st $(d+1)$[b]	11,634	100	241	48	24	379	63.3
2nd $(d+1)$[b]	32,477	364	874	1728	24	375	64.0

[a]With S-box decomposition
[b]Without S-box decomposition

As expected, the first-order $d+1$ TI design with S-box decomposition occupies the smallest area, compared to other secure implementations. Compared to the first-order $td+1$ TI architecture with S-box decomposition, this comes at the cost of extra randomness required.

We report an interesting observation when comparing the energy consumption of different architectures. The smallest energy consumption of 238 pJ has been achieved for the first-order secure $d+1$ TI architecture without S-box decomposition presented here. This is closely followed by design from [33] with the S-box decomposition 264 pJ. We attribute the absence of randomness needed for resharing in this specific design to its low power consumption, despite the area of both versions of first-order $td+1$ TI architectures with the S-box decomposition being larger compared to several other designs in Table 9. The absence of randomness greatly reduces the switching activity of the circuit lowering the power consumption considerably. Still, we observe that the first-order $d+1$ TI architecture without the S-box decomposition consumes only 5% more energy, while being 33% faster. Another interesting observation is that the first-order secure designs consume considerably less energy compared to second-order designs.

For second-order designs, those without decomposition lead to large area overheads (particularly in $td+1$ scenario) and a high number of random bits compared to simpler designs. We conclude that the $d+1$ designs are still interesting implementation choices if enough randomness can be provided. Second-order $td+1$, on the other hand, is quite unpractical due to its large area overheads and large power and energy consumption.

One can see that all protected designs except first-order $td+1$ without the S-box decomposition have their maximal frequency within 20% of each other. The reason for the first-order $td+1$ without the S-box decomposition smaller maximum frequency is the absence of register prior to the S-box operation. Also, the implementation from [33] has a smaller critical path compared to our designs. The critical path for all implementations goes from the round counter to the S-box input register. In the first-order $td+1$ without the S-box decomposition case, we do not have the S-box input register, thus making the critical part longer. Still, even with this limitation, the $td+1$ first-order version has smaller total latency compared to other designs.

Compared to our designs, the design described in [33] stores the key in plain, requiring less area for key storage, we

leave out the question of whether this is a good approach from the security point of view, but it has certainly impact on the area, power and energy consumption. In addition, the authors of [33] proposed different affine transformations of decomposed S-boxes and their architecture has simplified interface and control logic. That is the reason why their design is considerably smaller and has lower power consumption than the first-order $td + 1$ version of our proposal with the S-box decomposition. When the energy consumption is compared, however, the two designs perform similarly with the design of [33] being 2.3% more efficient. This is obviously due to the fact that our first-order $td + 1$ design with the S-box decomposition is 10% faster in terms of the required number of clock cycles.

Another interesting observation is that our first-order $td + 1$ design with the S-box decomposition has lower power consumption than four other designs from Table 9, while having larger area. As discussed previously, this is due to no additional randomness being required during the encryption/decryption process. We have investigated the impact of mask refreshing further and came to the conclusion that adding (or removing) the mask refreshing might change the power consumption up to a factor of two. An example of this is the first-order $d + 1$ TI without the S-box decomposition, where the power/energy consumption drops by 40% if the random inputs are set to zero. Hence, when achieving lower power/energy is the main requirement using uniform sharing is the best approach.

Table 9 also clearly shows the difference between power and energy consumption. The most extreme example is comparison between first-order $td + 1$ design without the S-box decomposition and $d + 1$ design with the S-box decomposition. Although $td + 1$ design without the S-box decomposition has almost six times the power consumption, it has slightly smaller energy consumption, as it takes six times less clock cycles to complete.

As can be seen by the reported figures, adding side-channel countermeasures increases the size of the unprotected PRINCE by at least a factor of 2.5. It should be noted that one has the penalty of extra clock cycles as well in all the cases except the first-order $td + 1$ without the S-box decomposition version.

The fastest unprotected PRINCE with the worst-case PVT synthesis takes 42.2 ns, followed by first-order $td + 1$ TI without decomposition which takes 58.8 ns, i.e., 39% latency increase; next is the second-order $d + 1$ TI without decomposition which takes 82.2 ns, i.e., additional 41% latency increase. Also, all designs without the S-box decomposition have significantly smaller latency compared to the implementation presented in [33], ranging from 1.4 to 2 times performance increase.

Table 10 shows area, power and energy consumption, the number of random bits required per clock cycle and the maximum frequency for three hardware implementations, one given by Moradi [33], and two that newly proposed ones all measured at the typical PVT case.

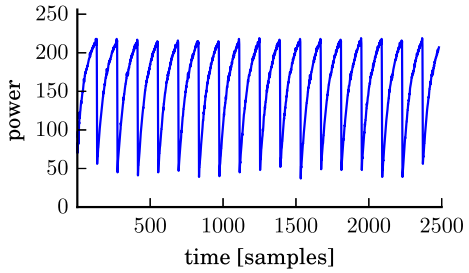


Fig. 10 Example power trace waveform used to perform the t-test on first-order PRINCE

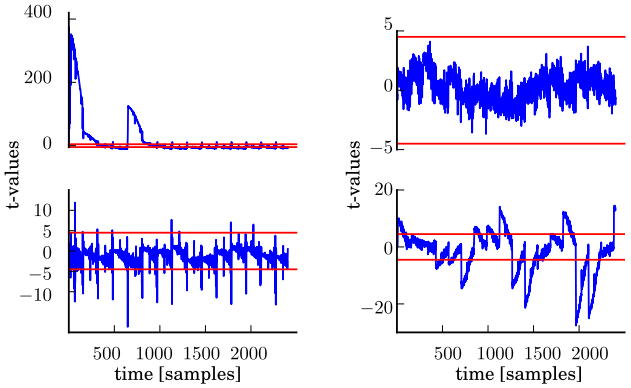


Fig. 11 Leakage detection test results on first-order PRINCE. PRNG off (left) and PRNG on (right). First-order (top) and second-order (bottom) t-test results

At the maximum frequency, our first-order design surpasses the previous state of the art by reducing latency by almost a third. The energy consumption of our first-order at the frequency of 10 MHz is lower by almost 10%. On the other hand, the implementation from [33] beats our version with respect to area, power consumption, maximal running frequency and randomness required. Potentially, it also can achieve higher throughput, with small modifications to the finite state machine, so it processes three messages at once. Given that our goal was to minimize implementation latency and energy, these results are not surprising.

4.3 Side-channel evaluation

We first provide an evaluation of the first-order PRINCE without S-box decomposition using optimal $d + 1$ sharing which design was programmed onto a Xilinx Spartan-6 FPGA. The platform used is a Sakura-G board. The design is separated into two FPGAs to minimize the noise: One performs the PRINCE encryption and the second FPGA handles the I/O and the start signal. Our core runs at 3.072 MHz, while the sampling rate is 500 million samples per second. Since

Table 11 Sharing indices of best shares for security order $d = 1$

	t	Sharing indices
$n = 4$		
	2	(1, 6, 8, 11, 13)
$n = 5$		
	2	(3, 12, 20, 24, 29, 30)
	3	(2, 5, 8, 11, 14, 17, 20, 23, 26, 29)
$n = 6$		
	2	(2, 21, 30, 35, 45, 56)
	3	(3, 9, 10, 15, 20, 27, 36, 43, 48, 53, 54, 60)
	4	(1, 4, 7, 10, 13, 16, 19, 22, 25, 28, 31, 34, 37, 40, 43, 46, 49, 52, 55, 58, 61)
$n = 7$		
	2	(24, 43, 54, 66, 85, 109)
	3	(6, 25, 37, 43, 50, 60, 67, 76, 85, 90, 96, 127)
	4	(0, 9, 14, 21, 23, 26, 35, 36, 47, 50, 57, 60, 67, 70, 77, 80, 91, 92, 101, 104, 106, 113, 118, 127)
	5	(1, 6, 10, 12, 15, 16, 19, 21, 25, 30, 32, 35, 37, 41, 46, 50, 52, 55, 56, 59, 61, 66, 68, 71, 72, 75, 77, 81, 86, 90, 92, 95, 97, 102, 106, 108, 111, 112, 115, 117, 121, 126)
$n = 8$		
	2	(15, 64, 119, 154, 177, 236)
	3	(12, 27, 33, 54, 85, 106, 130, 189, 207, 216, 228, 243)
	4	(9, 20, 31, 36, 42, 51, 66, 71, 88, 109, 113, 126, 129, 142, 146, 167, 184, 189, 204, 213, 219, 224, 235, 246)
	5	(1, 6, 8, 15, 19, 28, 36, 43, 50, 53, 57, 62, 69, 74, 80, 87, 89, 94, 96, 99, 109, 110, 118, 122, 124, 127, 128, 131, 133, 137, 145, 148, 154, 159, 161, 162, 173, 174, 183, 184, 193, 198, 203, 204, 210, 221, 231, 232, 241, 244, 251, 254)
	6	(3, 5, 6, 9, 10, 12, 17, 18, 20, 24, 31, 33, 34, 36, 40, 47, 48, 55, 59, 61, 62, 65, 66, 68, 72, 79, 80, 87, 91, 93, 94, 96, 103, 107, 109, 110, 115, 117, 118, 121, 122, 124, 129, 130, 132, 136, 143, 144, 151, 155, 157, 158, 160, 167, 171, 173, 174, 179, 181, 182, 185, 186, 188, 192, 199, 203, 205, 206, 211, 213, 214, 217, 218, 220, 227, 229, 230, 233, 234, 236, 241, 242, 244, 248, 255)

one trace consists of 2500 points, we are able to cover the first seven rounds of the execution. The power waveform is given in Fig. 10.

We apply a non-specific leakage detection test [13] on the input plaintext following the standard methodology [43], and the resulting t-test graphs are shown in Fig. 11. First, we turn PRNG off to verify the validity of the setup and leakage is detected with 1 million traces. The left-hand side in Fig. 11 demonstrates a strong first-order leakage during the loading of the plaintext and the key. This can be attributed to one share of both the key and the plaintext being equal to the unshared value, while the other share is zero. Another strong peak is during the first S-box execution as there is still a high correlation to the input. Leakage is present in later rounds as well due to a lack of additional randomness, although it becomes smaller, potentially due to jitter and misalignment in later rounds. Second-order leakage can also be observed when the masks are off. When PRNG is on, no first-order leakage is detected after 100 million traces, while second-order leakage is observed as expected.

Due to the size and randomness needed, the second-order design did not fit onto the same FPGA board. Instead, the design is tested against simulated power traces. We measured the estimated power consumption by running a post-synthesis simulation with back-annotated netlist. Input-to-output timing delays and current consumption of every gate in the netlist were taken into account and modeled as specified by the technology liberty timing file. In our simulations, one clock cycle is represented with 50 sample points and we cover the first seven rounds of the execution. One million traces have been obtained with PRNG switched on, and two thousand traces with PRNG off. Simulated traces are perfectly aligned, they do not contain any measurement noise, and the numerical noise of the samples is minimized by having a precision of 32-bit floating-point representation compared to 8-bit obtained from the FPGA setup.

Table 12 Sharing indices of best shares for security order $d = 2$, part 1

	t	Sharing indices
$n = 4$		
	2	(2, 12, 25, 31, 44, 45, 60, 64, 77)
$n = 5$		
	2	(0, 49, 71, 93, 106, 110, 139, 185, 195, 199, 234)
	3	(0, 5, 17, 19, 31, 38, 51, 61, 64, 66, 77, 87, 94, 101, 109, 116, 117, 131, 138, 152, 153, 160, 169, 171, 183, 188, 192, 203, 205, 208, 218, 231, 238)
$n = 6$		
	2	(0, 41, 143, 238, 295, 369, 408, 470, 555, 580, 674, 676)
	3	(0, 52, 68, 104, 112, 145, 159, 178, 201, 209, 224, 269, 275, 280, 312, 331, 336, 369, 380, 416, 438, 481, 499, 534, 547, 560, 586, 611, 624, 653, 672, 676, 711)
	4	(7, 11, 23, 24, 32, 34, 39, 44, 46, 54, 67, 75, 80, 82, 86, 96, 103, 108, 121, 128, 132, 143, 146, 147, 158, 160, 165, 176, 178, 180, 188, 191, 199, 204, 210, 220, 222, 225, 233, 235, 246, 251, 256, 261, 268, 271, 276, 281, 292, 302, 307, 312, 317, 324, 331, 338, 344, 359, 360, 367, 372, 382, 395, 397, 402, 407, 415, 420, 427, 436, 446, 450, 458, 459, 466, 471, 482, 488, 490, 498, 503, 505, 513, 529, 536, 537, 547, 549, 554, 562, 573, 577, 588, 593, 598, 608, 609, 613, 623, 624, 637, 639, 655, 657, 671, 678, 683, 686, 700, 703, 707, 715, 719, 722, 726)
$n = 7$		
	2	(0, 483, 632, 679, 872, 995, 1144, 1257, 1525, 1667, 1812, 2050)
	3	(1, 131, 173, 222, 288, 349, 380, 445, 521, 552, 570, 611, 721, 753, 873, 877, 920, 977, 1009, 1043, 1086, 1209, 1264, 1316, 1393, 1419, 1435, 1488, 1501, 1532, 1547, 1642, 1762, 1794, 1863, 1916, 1953, 2053, 2103, 2174)
	4	(0, 23, 52, 65, 66, 97, 111, 118, 149, 157, 166, 186, 198, 209, 224, 255, 268, 287, 302, 315, 330, 365, 371, 379, 389, 404, 425, 439, 453, 472, 496, 515, 517, 519, 565, 572, 585, 601, 636, 665, 687, 694, 702, 725, 737, 748, 769, 807, 822, 852, 857, 871, 873, 887, 905, 919, 944, 948, 976, 999, 1022, 1043, 1064, 1069, 1088, 1099, 1110, 1152, 1176, 1190, 1200, 1213, 1224, 1258, 1272, 1285, 1289, 1297, 1322, 1334, 1344, 1363, 1383, 1399, 1409, 1441, 1473, 1490, 1503, 1513, 1541, 1564, 1583, 1614, 1629, 1633, 1653, 1669, 1690, 1694, 1703, 1738, 1740, 1761, 1777, 1805, 1813, 1833, 1845, 1866, 1870, 1880, 1882, 1901, 1931, 1951, 1955, 1965, 1997, 2012, 2038, 2040, 2052, 2087, 2098, 2108, 2145, 2149, 2164, 2184)

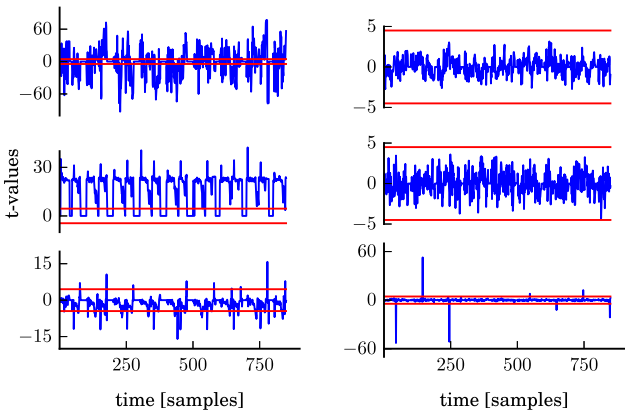


Fig. 12 Leakage detection test results on second-order PRINCE. PRNG off (left) and PRNG on (right). First-, second- and third-order (top–middle–down) t-test results

The second-order implementation t-test results are shown in Fig. 12. We notice that with PRNG off, leakage occurs in all orders with only two thousand traces. With PRNG on, the design exhibits no leakage in the first and second-order t-test, while several points leak in the third order. More precisely, third-order leakage occurs during the writing of the S-box output to the register every other cycles.

5 Conclusion and outlook

As discussed in [33], designing a low-latency side-channel protection in general, and for PRINCE block cipher in particular, has been identified as an open problem. In this work, we have shown the fastest round-based first- and second-order secure implementations of PRINCE using $td+1$ and $d+1$ TI

Table 13 Sharing indices of best shares for security order $d = 2$, part 2

	t	Sharing indices
$n = 7$		
	5	(0, 8, 14, 19, 24, 30, 36, 43, 47, 49, 56, 58, 69, 77, 83, 90, 95, 97, 103, 109, 113, 123, 134, 135, 143, 148, 154, 156, 163, 165, 179, 184, 185, 196, 199, 201, 206, 207, 221, 222, 227, 241, 245, 250, 256, 258, 266, 276, 282, 289, 296, 300, 307, 314, 315, 325, 327, 341, 342, 349, 353, 365, 367, 372, 382, 385, 389, 393, 401, 410, 414, 421, 425, 429, 432, 436, 443, 457, 460, 467, 472, 478, 480, 481, 485, 490, 497, 502, 507, 514, 521, 527, 531, 545, 546, 552, 560, 565, 572, 573, 577, 579, 593, 597, 601, 603, 614, 616, 622, 635, 637, 645, 648, 655, 662, 667, 677, 688, 690, 698, 699, 705, 711, 719, 724, 731, 736, 741, 746, 751, 764, 767, 769, 779, 780, 784, 786, 799, 801, 808, 809, 814, 816, 820, 830, 837, 841, 849, 853, 856, 860, 871, 872, 873, 878, 888, 896, 900, 910, 915, 920, 925, 933, 940, 945, 958, 962, 965, 966, 972, 977, 982, 992, 997, 1000, 1016, 1020, 1032, 1035, 1040, 1048, 1061, 1064, 1066, 1068, 1074, 1087, 1089, 1100, 1102, 1104, 1107, 1112, 1123, 1126, 1128, 1133, 1135, 1146, 1160, 1164, 1169, 1171, 1175, 1179, 1190, 1192, 1203, 1211, 1213, 1216, 1224, 1238, 1239, 1242, 1246, 1257, 1262, 1267, 1277, 1280, 1281, 1282, 1288, 1297, 1310, 1314, 1321, 1328, 1333, 1340, 1344, 1352, 1353, 1358, 1365, 1372, 1376, 1380, 1385, 1388, 1393, 1399, 1405, 1410, 1416, 1430, 1438, 1441, 1445, 1449, 1461, 1468, 1484, 1487, 1492, 1500, 1504, 1506, 1517, 1518, 1521, 1526, 1534, 1540, 1544, 1554, 1560, 1565, 1572, 1577, 1579, 1584, 1591, 1596, 1603, 1610, 1613, 1628, 1631, 1633, 1638, 1645, 1647, 1651, 1661, 1673, 1675, 1686, 1690, 1697, 1698, 1705, 1708, 1713, 1718, 1719, 1722, 1728, 1733, 1739, 1741, 1753, 1756, 1763, 1769, 1771, 1775, 1779, 1789, 1791, 1801, 1805, 1806, 1812, 1819, 1826, 1832, 1838, 1842, 1848, 1850, 1858, 1865, 1869, 1873, 1877, 1885, 1889, 1897, 1905, 1910, 1911, 1920, 1926, 1934, 1936, 1946, 1950, 1958, 1961, 1963, 1974, 1981, 1993, 1997, 1998, 2005, 2013, 2019, 2021, 2025, 2036, 2038, 2041, 2048, 2053, 2060, 2064, 2076, 2083, 2090, 2097, 2098, 2104, 2110, 2121, 2126, 2127, 2131, 2138, 2142, 2144, 2149, 2152, 2162, 2166, 2173, 2186)
$n = 8$		
	2	(0, 1255, 1923, 2045, 2347, 3100, 3210, 3599, 3844, 4729, 4796, 4926, 5490, 5909)
	3	(33, 424, 512, 635, 740, 919, 1191, 1348, 1534, 1608, 1706, 1830, 1907, 2025, 2170, 2261, 2277, 2354, 2589, 2626, 2710, 2784, 3148, 3227, 3269, 3423, 3852, 3982, 4193, 4318, 4510, 4667, 4846, 5037, 5223, 5287, 5353, 5445, 5633, 5730, 5825, 5861, 5939, 6138, 6511)
	4	(5, 33, 92, 151, 188, 207, 238, 268, 318, 378, 446, 495, 613, 644, 682, 807, 839, 904, 954, 984, 1015, 1037, 1102, 1133, 1140, 1162, 1273, 1310, 1338, 1397, 1495, 1560, 1651, 1682, 1703, 1731, 1799, 1936, 1997, 2016, 2026, 2057, 2085, 2122, 2172, 2261, 2268, 2299, 2364, 2497, 2562, 2594, 2653, 2674, 2705, 2733, 2792, 2820, 2857, 2888, 2955, 2986, 3023, 3042, 3073, 3083, 3133, 3194, 3250, 3368, 3396, 3426, 3448, 3486, 3635, 3652, 3770, 3789, 3857, 3906, 3937, 3975, 4028, 4062, 4093, 4115, 4145, 4195, 4291, 4320, 4387, 4418, 4474, 4593, 4645, 4676, 4710, 4741, 4763, 4797, 4859, 4908, 4949, 5064, 5086, 5103, 5165, 5252, 5317, 5369, 5457, 5488, 5544, 5605, 5627, 5692, 5742, 5752, 5783, 5811, 5858, 5877, 5908, 5946, 5968, 6005, 6033, 6144, 6181, 6203, 6241, 6374, 6439, 6504, 6557)
	5	(6, 9, 21, 35, 38, 61, 76, 100, 123, 140, 152, 188, 202, 214, 216, 245, 271, 295, 309, 330, 333, 374, 388, 409, 424, 447, 450, 464, 476, 479, 502, 516, 545, 560, 569, 581, 595, 607, 619, 633, 645, 657, 669, 683, 724, 734, 775, 798, 801, 810, 834, 839, 851, 863, 865, 877, 898, 901, 939, 956, 979, 982, 994, 1005, 1020, 1037, 1049, 1070, 1084, 1110, 1125, 1146, 1163, 1175, 1213, 1239, 1256, 1268, 1270, 1303, 1318, 1332, 1358, 1394, 1397, 1408, 1420, 1434, 1446, 1449, 1471, 1485, 1497, 1538, 1550, 1562, 1573, 1614, 1623, 1635, 1652, 1667, 1678, 1690, 1693, 1719, 1745, 1759, 1771, 1795, 1807, 1809, 1821, 1833, 1838, 1862, 1871, 1886, 1900, 1923,

Table 13 continued

t	Sharing indices
	1952, 1955, 1967, 1981, 1993, 2004, 2007, 2028, 2072, 2095, 2098, 2107, 2133, 2157, 2174, 2204, 2207, 2218, 2259, 2280, 2297, 2323, 2347, 2356, 2359, 2382, 2385, 2397, 2414, 2426, 2452, 2466, 2478, 2492, 2516, 2528, 2531, 2542, 2554, 2568, 2583, 2616, 2633, 2647, 2659, 2673, 2697, 2714, 2726, 2728, 2740, 2752, 2761, 2787, 2802, 2819, 2840, 2852, 2881, 2904, 2929, 2941, 2943, 2955, 2972, 2984, 2996, 3020, 3034, 3046, 3057, 3081, 3096, 3122, 3136, 3162, 3174, 3206, 3229, 3241, 3291, 3308, 3320, 3329, 3344, 3355, 3358, 3370, 3381, 3384, 3413, 3436, 3439, 3477, 3486, 3498, 3501, 3515, 3530, 3553, 3556, 3577, 3591, 3620, 3644, 3651, 3680, 3695, 3709, 3730, 3742, 3745, 3768, 3771, 3785, 3797, 3809, 3859, 3873, 3885, 3890, 3902, 3916, 3940, 3942, 3954, 3966, 3978, 4004, 4007, 4045, 4066, 4069, 4080, 4124, 4135, 4150, 4173, 4190, 4238, 4252, 4266, 4278, 4302, 4314, 4328, 4331, 4343, 4354, 4369, 4375, 4387, 4425, 4442, 4463, 4478, 4492, 4504, 4515, 4518, 4530, 4539, 4568, 4583, 4606, 4620, 4632, 4649, 4661, 4664, 4687, 4690, 4711, 4723, 4725, 4778, 4790, 4813, 4839, 4854, 4883, 4897, 4920, 4923, 4959, 4985, 4988, 4999, 5023, 5035, 5047, 5049, 5061, 5078, 5102, 5112, 5138, 5164, 5179, 5200, 5214, 5226, 5258, 5267, 5279, 5291, 5293, 5317, 5331, 5343, 5348, 5372, 5386, 5400, 5436, 5448, 5462, 5465, 5488, 5491, 5503, 5527, 5550, 5553, 5567, 5593, 5608, 5619, 5631, 5634, 5648, 5660, 5672, 5684, 5722, 5736, 5757, 5801, 5815, 5827, 5849, 5852, 5863, 5875, 5878, 5889, 5904, 5913, 5937, 5954, 5968, 6016, 6030, 6056, 6059, 6082, 6123, 6140, 6152, 6161, 6187, 6202, 6225, 6237, 6249, 6261, 6278, 6290, 6292, 6304, 6330, 6342, 6347, 6397, 6406, 6409, 6432, 6447, 6461, 6464, 6476, 6497, 6511, 6523, 6552)

sharing correspondingly, as well as the most energy-efficient round-based first-order secure implementation of PRINCE using $d + 1$ TI sharing. We have introduced several methods for optimizing threshold implementations, which make the low latency, low energy and higher throughput of side-channel secure designs practical. Then, we have investigated several different trade-offs that occur in side-channel secure designs. Particularly, we discuss the energy consumption of given implementations, an important factor in several applications, such as battery-powered devices.

On the methods for optimized TI sharings: First, we provided an algorithm that produces a $d + 1$ TI sharing with the optimal (minimum) number of output shares for any n-input Boolean function of degree $t = n - 1$ and for any security order d. Second, when $t < n - 1$, we presented discrete optimization-based methodology, which can be used to find good, and in many cases optimal, sharings of Boolean functions up to 8 bits. Third, we presented a heuristic for minimizing the number of output shares for higher-order $td + 1$ TI. We highlight that these contributions are of general interest since the method of minimizing the number of output shares can be applied to any cryptographic design. The last optimization is on the secure AES scheduling which achieves maximum throughput for a serial implementation. The proposed algorithms should be of use during the design of low-latency side-channel hardware implementations of many symmetric designs, since almost all in-use symmetric algorithms incorporate S-boxes that have no more than eight input bits. We would like to add that improving the heuristics to provide sharing solutions for security orders higher than two would be beneficial, especially for software implementations which are typically realized in higher security orders [5].

Next, we reported, evaluated and compared hardware figures for eight different TI-protected round-based versions of PRINCE cipher, namely $d+1$ and $td+1$ TI versions, first- and second-order secure, with or without the S-box decomposition. The $td+1$ TI versions tend to consume less randomness. The $d + 1$ TI versions with decomposition achieve lower area and power consumption. The first-order designs without decomposition have favorable energy consumption. The comparison with state of the art showed that our designs have more than 30% lower latency compared to the architecture presented in [33], while the energy consumption is lower by about 10%. It should, however, be noted that the design presented in [33] still has the highest power efficiency reported in the literature.

We would like to summarize that the generic algorithm for achieving the minimal number of output shares is a necessary, but not sufficient condition when designing for low-latency and low-energy applications. Applying TI on higher-degree functions reduces the total clock count, in effect reducing latency and energy consumed during one operation. However, due to increased circuit complexity it increases the area and the critical path of the design, which have a negative impact on energy consumption and latency, respectively. A

Table 14 Sharing indices of best shares for security order $d = 2, n = 8, t = 6$

Shares used
(1, 12, 20, 24, 33, 38, 41, 43, 50, 54, 58, 62, 68, 75, 79, 84, 87, 90, 97, 103, 107, 108, 113, 115, 116, 119, 125,
127, 129, 137, 138, 145, 148, 159, 162, 170, 173, 175, 185, 193, 201, 207, 209, 214, 223, 226, 230, 231, 236,
245, 246, 260, 265, 277, 279, 283, 288, 289, 296, 302, 307, 321, 328, 332, 335, 336, 344, 353, 357, 361, 374,
379, 392, 393, 396, 403, 410, 414, 421, 424, 429, 433, 437, 449, 450, 453, 459, 462, 465, 470, 471, 481, 485,
494, 496, 500, 504, 511, 515, 516, 528, 535, 544, 549, 552, 556, 560, 568, 572, 582, 583, 588, 598, 605, 606,
618, 621, 628, 638, 640, 644, 651, 655, 657, 665, 668, 670, 681, 684, 689, 691, 694, 701, 704, 715, 726, 732,
738, 743, 745, 751, 758, 760, 768, 774, 782, 790, 794, 798, 802, 806, 815, 821, 825, 831, 835, 840, 847, 850,
860, 861, 865, 872, 876, 882, 892, 897, 903, 909, 917, 925, 927, 935, 938, 940, 945, 950, 958, 969, 973, 980,
987, 990, 995, 1004, 1005, 1010, 1024, 1028, 1029, 1035, 1039, 1043, 1044, 1053, 1060, 1063, 1070, 1074,
1078, 1081, 1089, 1096, 1100, 1101, 1112, 1113, 1118, 1119, 1129, 1133, 1136, 1138, 1140, 1148, 1159, 1162,
1164, 1169, 1173, 1183, 1185, 1195, 1197, 1204, 1207, 1211, 1215, 1222, 1227, 1235, 1239, 1243, 1245, 1255,
1259, 1261, 1263, 1274, 1275, 1279, 1287, 1295, 1302, 1309, 1310, 1315, 1322, 1331, 1337, 1341, 1348, 1352,
1354, 1359, 1366, 1371, 1382, 1387, 1392, 1399, 1404, 1415, 1417, 1419, 1427, 1432, 1439, 1445, 1451, 1452,
1456, 1458, 1466, 1471, 1476, 1481, 1483, 1490, 1495, 1500, 1505, 1507, 1513, 1515, 1525, 1529, 1536, 1541,
1543, 1549, 1553, 1554, 1558, 1574, 1578, 1584, 1591, 1599, 1602, 1607, 1609, 1613, 1615, 1623, 1627, 1629,
1637, 1642, 1648, 1652, 1653, 1658, 1660, 1663, 1666, 1676, 1686, 1693, 1695, 1700, 1706, 1707, 1712, 1713,
1717, 1720, 1727, 1728, 1732, 1745, 1749, 1762, 1764, 1769, 1775, 1777, 1780, 1783, 1791, 1805, 1806, 1812,
1816, 1822, 1823, 1826, 1828, 1835, 1838, 1846, 1851, 1857, 1871, 1876, 1883, 1884, 1892, 1900, 1905, 1913,
1915, 1917, 1921, 1925, 1934, 1935, 1941, 1945, 1948, 1959, 1965, 1967, 1979, 1980, 1985, 1987, 1990, 1994,
1995, 2000, 2004, 2009, 2016, 2020, 2028, 2033, 2036, 2043, 2050, 2052, 2062, 2067, 2072, 2074, 2080, 2084,
2092, 2094, 2105, 2108, 2110, 2116, 2120, 2128, 2130, 2138, 2140, 2145, 2150, 2151, 2163, 2169, 2176, 2183,
2187, 2192, 2194, 2200, 2204, 2206, 2208, 2217, 2223, 2230, 2234, 2243, 2251, 2264, 2265, 2270, 2276, 2278,
2286, 2291, 2293, 2296, 2304, 2309, 2310, 2321, 2326, 2328, 2334, 2339, 2341, 2353, 2363, 2364, 2368, 2375,
2378, 2383, 2387, 2398, 2400, 2406, 2411, 2412, 2419, 2426, 2434, 2436, 2439, 2442, 2450, 2455, 2458, 2465,
2471, 2478, 2484, 2495, 2499, 2500, 2506, 2510, 2512, 2524, 2528, 2535, 2541, 2549, 2554, 2556, 2560, 2570,
2572, 2574, 2585, 2586, 2594, 2595, 2599, 2602, 2615, 2619, 2632, 2634, 2638, 2645, 2647, 2660, 2666, 2670,
2678, 2684, 2688, 2689, 2695, 2699, 2700, 2704, 2712, 2717, 2724, 2725, 2728, 2730, 2733, 2741, 2745, 2757,
2761, 2763, 2768, 2779, 2781, 2789, 2794, 2800, 2804, 2810, 2818, 2821, 2823, 2829, 2834, 2836, 2843, 2847,
2853, 2867, 2868, 2872, 2882, 2883, 2889, 2893, 2906, 2914, 2924, 2926, 2934, 2939, 2941, 2944, 2957, 2965,
2967, 2974, 2976, 2982, 2987, 2990, 3001, 3003, 3014, 3017, 3024, 3026, 3031, 3037, 3043, 3045, 3056, 3060,
3067, 3073, 3077, 3080, 3090, 3094, 3100, 3108, 3113, 3115, 3120, 3123, 3128, 3133, 3143, 3153, 3157, 3160,
3162, 3170, 3185, 3186, 3191, 3196, 3201, 3208, 3220, 3227, 3232, 3237, 3248, 3252, 3256, 3258, 3262, 3266,
3269, 3271, 3273, 3284, 3292, 3294, 3304, 3309, 3313, 3317, 3318, 3326, 3330, 3340, 3345, 3355, 3362, 3368,
3369, 3377, 3378, 3388, 3392, 3393, 3400, 3406, 3416, 3417, 3421, 3423, 3432, 3437, 3441, 3445, 3449, 3458,
3465, 3469, 3478, 3481, 3483, 3485, 3493, 3506, 3507, 3511, 3515, 3521, 3525, 3527, 3532, 3536, 3540, 3544,
3545, 3551, 3555, 3557, 3564, 3567, 3571, 3575, 3577, 3590, 3593, 3595, 3600, 3615, 3616, 3624, 3630, 3634,
3637, 3641, 3648, 3656, 3660, 3667, 3674, 3679, 3689, 3693, 3698, 3699, 3707, 3713, 3715, 3718, 3733, 3735,
3739, 3747, 3752, 3753, 3758, 3759, 3763, 3770, 3773, 3775, 3781, 3783, 3791, 3804, 3808, 3812, 3813, 3816,
3827, 3831, 3834, 3838, 3846, 3851, 3859, 3868, 3874, 3876, 3879, 3884, 3890, 3895, 3898, 3905, 3906, 3911,
3921, 3924, 3928, 3935, 3940, 3943, 3947, 3957, 3963, 3974, 3975, 3980, 3984, 3988, 3997, 4008, 4019, 4020,
4027, 4031, 4037, 4039, 4041, 4050, 4054, 4064, 4067, 4071, 4075, 4080, 4085, 4088, 4093, 4095, 4106, 4110,
4114, 4116, 4126, 4130, 4131, 4139, 4144, 4155, 4161, 4165, 4169, 4176, 4177, 4181, 4189, 4195, 4197, 4202,
4205, 4209, 4213, 4216, 4224, 4229, 4232, 4241, 4255, 4260, 4272, 4275, 4289, 4291, 4299, 4302, 4309, 4312,
4316, 4321, 4334, 4335, 4342, 4346, 4348, 4352, 4358, 4368, 4373, 4377, 4383, 4396, 4400, 4402, 4409, 4414,

Table 14 continued

Shares used
4419, 4426, 4435, 4442, 4443, 4448, 4460, 4462, 4467, 4472, 4474, 4486, 4492, 4502, 4506, 4509, 4520, 4525, 4532, 4534, 4538, 4542, 4546, 4557, 4561, 4566, 4573, 4578, 4580, 4586, 4591, 4595, 4602, 4603, 4608, 4612, 4616, 4618, 4622, 4633, 4638, 4647, 4655, 4660, 4666, 4668, 4675, 4677, 4680, 4683, 4688, 4690, 4694, 4698, 4709, 4713, 4721, 4723, 4727, 4732, 4737, 4739, 4744, 4751, 4755, 4760, 4762, 4765, 4776, 4780, 4787, 4793, 4797, 4801, 4805, 4810, 4812, 4815, 4826, 4831, 4835, 4849, 4854, 4862, 4864, 4866, 4871, 4872, 4879, 4883, 4892, 4894, 4897, 4902, 4907, 4914, 4922, 4927, 4935, 4939, 4941, 4954, 4955, 4959, 4961, 4967, 4968, 4985, 4989, 4993, 4996, 5000, 5001, 5004, 5015, 5017, 5027, 5031, 5038, 5046, 5049, 5050, 5057, 5062, 5067, 5079, 5083, 5087, 5091, 5095, 5099, 5105, 5110, 5116, 5118, 5124, 5136, 5141, 5146, 5153, 5157, 5162, 5167, 5169, 5175, 5179, 5183, 5185, 5193, 5197, 5210, 5216, 5223, 5228, 5229, 5236, 5240, 5244, 5245, 5252, 5259, 5265, 5269, 5276, 5279, 5284, 5289, 5293, 5305, 5313, 5318, 5322, 5327, 5328, 5335, 5339, 5352, 5355, 5359, 5366, 5371, 5377, 5385, 5390, 5391, 5392, 5399, 5401, 5408, 5416, 5421, 5423, 5429, 5430, 5441, 5446, 5448, 5451, 5454, 5462, 5467, 5469, 5477, 5485, 5490, 5501, 5506, 5515, 5518, 5520, 5525, 5531, 5540, 5546, 5551, 5553, 5562, 5576, 5577, 5584, 5588, 5594, 5599, 5606, 5607, 5618, 5623, 5625, 5630, 5638, 5640, 5646, 5650, 5654, 5658, 5666, 5671, 5675, 5676, 5682, 5686, 5692, 5700, 5704, 5707, 5717, 5724, 5735, 5737, 5741, 5743, 5748, 5754, 5759, 5760, 5771, 5772, 5776, 5784, 5790, 5795, 5797, 5801, 5807, 5809, 5815, 5823, 5831, 5836, 5838, 5842, 5846, 5852, 5859, 5863, 5871, 5876, 5878, 5888, 5893, 5895, 5909, 5910, 5913, 5924, 5935, 5937, 5941, 5943, 5948, 5956, 5963, 5972, 5980, 5982, 5986, 5988, 5993, 6002, 6006, 6010, 6012, 6017, 6023, 6028, 6030, 6035, 6041, 6043, 6045, 6051, 6052, 6054, 6058, 6065, 6073, 6075, 6083, 6089, 6097, 6099, 6104, 6109, 6112, 6117, 6125, 6132, 6140, 6142, 6147, 6155, 6160, 6164, 6168, 6172, 6176, 6188, 6189, 6194, 6201, 6205, 6210, 6211, 6227, 6233, 6235, 6240, 6248, 6252, 6256, 6265, 6277, 6285, 6290, 6296, 6298, 6300, 6310, 6315, 6321, 6327, 6335, 6338, 6343, 6346, 6351, 6358, 6366, 6371, 6377, 6388, 6391, 6396, 6401, 6406, 6409, 6414, 6422, 6430, 6432, 6434, 6435, 6440, 6445, 6450, 6460, 6464, 6465, 6471, 6481, 6485, 6491, 6493, 6501, 6506, 6510, 6518, 6523, 6532, 6534, 6542, 6548, 6549, 6554, 6556)

circuit designer should take all these parameters into consideration since the optimal design choice heavily depends on the algorithm in question, alongside the constraints imposed upon the design. In the case of PRINCE block cipher, our work shows that for achieving low latency it is more efficient not to perform S-box decomposition.

Acknowledgements We would like to thank Amir Moradi for providing us with HDL code of PRINCE TI presented in [33].

Appendix A

A.1. First-order secure $td + 1$ TI of Q_{294}

We use first-order $td + 1$ direct TI sharing [11] with three shares. Here, we recall that $d = 1$ and $t = 2$. The actual sharing is given in Eq. (21).

$$\begin{aligned} x_1 &= a_1 & z_1 &= a_1b_1 \oplus a_1b_2 \oplus a_2b_1 \oplus c_1 \\ x_2 &= a_2 & z_2 &= a_2b_2 \oplus a_2b_3 \oplus a_3b_2 \oplus c_2 \\ x_3 &= a_3 & z_3 &= a_3b_3 \oplus a_3b_1 \oplus a_1b_3 \oplus c_3 \\ y_1 &= b_1 & w_1 &= a_1c_1 \oplus a_1c_2 \oplus a_2c_1 \oplus d_1 \\ y_2 &= b_2 & w_2 &= a_2c_2 \oplus a_2c_3 \oplus a_3c_2 \oplus d_2 \\ y_3 &= b_3 & w_3 &= a_3c_3 \oplus a_3c_1 \oplus a_1c_3 \oplus d_3. \end{aligned} \tag{21}$$

Figure 13 depicts the hardware implementation of the $td+1$ version of Q_{294}.

A.2. Second-order secure $td + 1$ TI of Q_{294}

We use the second-order $td + 1$ TI sharing of Q_{294} with five input shares and ten output shares as shown in Eq. (22). In this case, we have $d = 2$ and $t = 2$. The shares are first processed and thus expanded, then refreshed and stored into a register. Next, they are compressed into five shares using the method explained in [6]. Values in Eq. (22) denoted with the overline represent the output after the compression step.

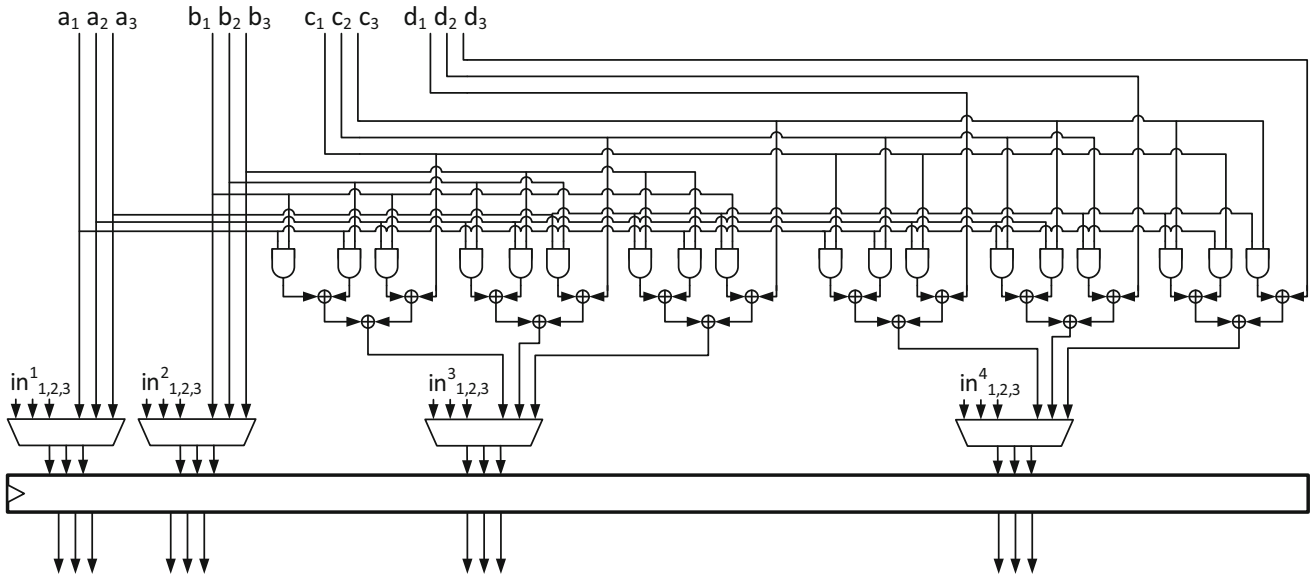


Fig. 13 First-order secure sharing of Q_{294} with $td + 1$ TI

$$
\begin{aligned}
x_1 &= a_1 & & & & & y_1 &= b_1 \\
x_2 &= a_2 & & & & & y_2 &= b_2 \\
x_3 &= a_3 & & & & & y_3 &= b_3 \\
x_4 &= a_4 & & & & & y_4 &= b_4 \\
x_5 &= a_5 & & & & & y_5 &= b_5 \\
z_1 &= a_1b_3 \oplus a_3b_1 & z_6 &= a_1b_1 \oplus a_1b_2 \oplus a_2b_1 \oplus c_1 & & & \bar{z}_1 &= z_1 \oplus z_6 \\
z_2 &= a_2b_4 \oplus a_4b_2 & z_7 &= a_2b_2 \oplus a_2b_3 \oplus a_3b_2 \oplus c_2 & & & \bar{z}_2 &= z_2 \oplus z_7 \\
z_3 &= a_3b_5 \oplus a_5b_3 & z_8 &= a_3b_3 \oplus a_3b_4 \oplus a_4b_3 \oplus c_3 & & & \bar{z}_3 &= z_3 \oplus z_8 \\
z_4 &= a_4b_1 \oplus a_1b_4 & z_9 &= a_4b_4 \oplus a_4b_5 \oplus a_5b_4 \oplus c_4 & & & \bar{z}_4 &= z_4 \oplus z_9 \\
z_5 &= a_5b_2 \oplus a_2b_5 & z_{10} &= a_5b_5 \oplus a_5b_1 \oplus a_1b_5 \oplus c_5 & & & \bar{z}_5 &= z_5 \oplus z_{10} \\
w_1 &= a_1c_3 \oplus a_3c_1 & w_6 &= a_1c_1 \oplus a_1c_2 \oplus a_2c_1 \oplus d_1 & & & \bar{w}_1 &= w_1 \oplus w_6 \\
w_2 &= a_2c_4 \oplus a_4c_2 & w_7 &= a_2c_2 \oplus a_2c_3 \oplus a_3c_2 \oplus d_2 & & & \bar{w}_2 &= w_2 \oplus w_7 \\
w_3 &= a_3c_5 \oplus a_5c_3 & w_8 &= a_3c_3 \oplus a_3c_4 \oplus a_4c_3 \oplus d_3 & & & \bar{w}_3 &= w_3 \oplus w_8 \\
w_4 &= a_4c_1 \oplus a_1c_4 & w_9 &= a_4c_4 \oplus a_4c_5 \oplus a_5c_4 \oplus d_4 & & & \bar{w}_4 &= w_4 \oplus w_9 \\
w_5 &= a_5c_2 \oplus a_2c_5 & w_{10} &= a_5c_5 \oplus a_5c_1 \oplus a_1c_5 \oplus d_5 & & & \bar{w}_5 &= w_5 \oplus w_{10}. \qquad (22)
\end{aligned}
$$

Please note that in order to avoid multivariate attacks, where the attacker probes values from different time samples, only nonlinear parts need to be refreshed, namely $z_1, \ldots, z_5$ and $w_1, \ldots, w_5$. Therefore, we need ten random bits for each shared Q_{294} function.

The sub-circuit used to generate two output bits of a partial evaluation of shared nonlinear function $xy + z$ is shown in Fig. 14. Figure 15 showcases the hardware implementation of the $td + 1$ Version of Q_{294}.

A.3. First-order secure $d + 1$ TI of Q_{294}

We use the first-order sharing given in [41] and shown in Eq. (23). In this case, it holds $d = 1$. Unlike $td + 1$ TI, the

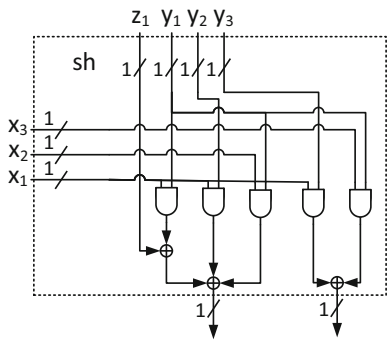


Fig. 14 Generating two outputs bits for partial evaluation of $xy + z$

first-order secure sharing here has four output shares for the nonlinear component functions. For the linear parts, however, we need only two shares instead of three. Compression and mask refreshing are needed to reduce the number of output shares and make the output uniform, respectively.

$$
\begin{aligned}
x_1 &= a_1 & y_1 &= b_1\\
x_2 &= a_2 & y_2 &= b_2\\
z_1 &= a_1b_1 \oplus c_1 & w_1 &= a_1c_1 \oplus d_1\\
z_2 &= a_1b_2 & w_2 &= a_1c_2\\
z_3 &= a_2b_2 \oplus c_2 & w_3 &= a_2c_2 \oplus d_2\\
z_4 &= a_2b_1 & w_4 &= a_2c_1\\
\bar{z}_1 &= z_1 \oplus z_2 & \bar{w}_1 &= w_1 \oplus w_2\\
\bar{z}_2 &= z_3 \oplus z_4 & \bar{w}_2 &= w_3 \oplus w_4. \qquad (23)
\end{aligned}
$$

Shares that contain quadratic terms are refreshed as given in Eq. (2) before storing into a register. We have two shared output component functions with four shares, for which we need six random bits. As in the second-order secure $td + 1$ version we set appropriate register bits to 0 during initial loading to ensure correctness of the execution. A detailed hardware implementation of the $d + 1$ TI sharing of Q_{294} is depicted in Fig. 16.

A.4. Second-order secure $d + 1$ TI of Q_{294}

Next, we create a second-order secure masking of Q_{294} following the work of [41]. In this case, $d = 2$. Three input shares are needed for all the operations. However, sharing a nonlinear operation $xy + z$ produces nine output shares that need to be first refreshed, then stored into a register and finally compressed. We give the formula for $d + 1$ second-order secure sharing in Eq. (24).

$$
\begin{aligned}
x_1 &= a_1 & y_1 &= b_1\\
x_2 &= a_2 & y_2 &= b_2\\
x_3 &= a_3 & y_3 &= b_3\\
z_1 &= a_1b_1 \oplus c_1 & w_1 &= a_1c_1 \oplus d_1\\
z_2 &= a_1b_2 & w_2 &= a_1c_2\\
z_3 &= a_1b_3 & w_3 &= a_1b_3
\end{aligned}
$$

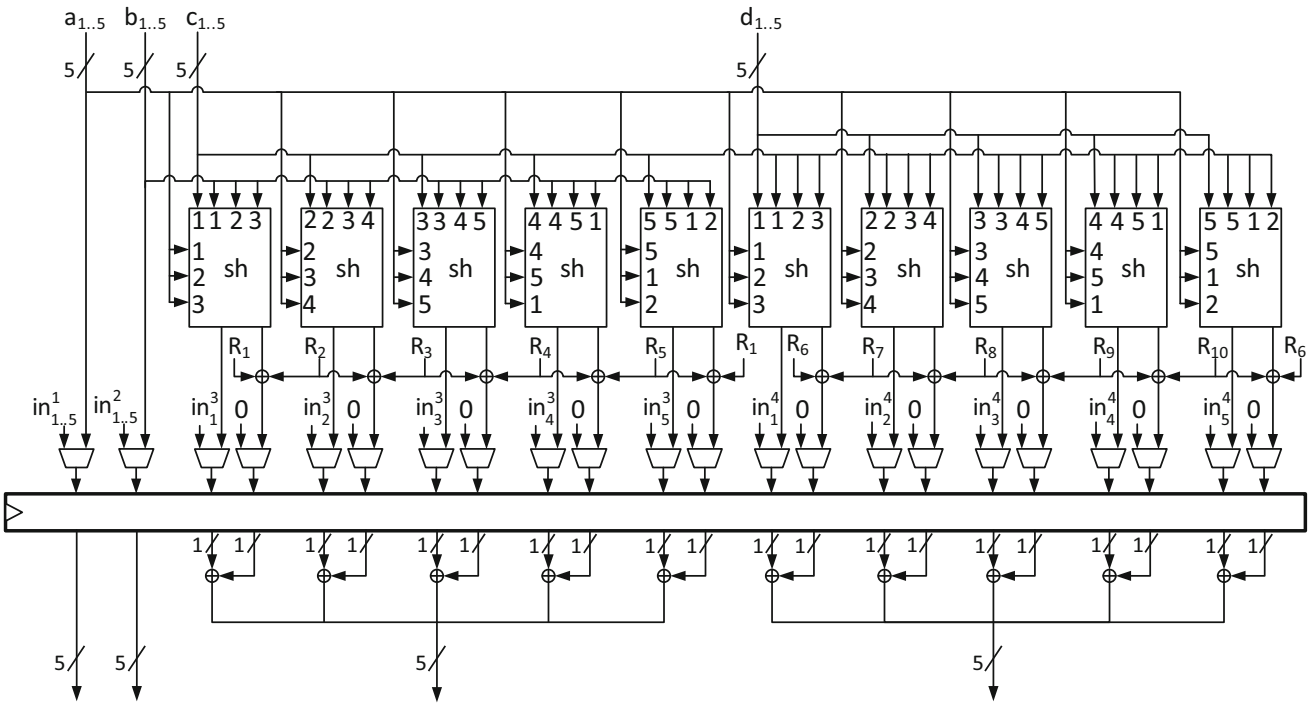


Fig. 15 Second-order secure sharing of Q_{294} with $td + 1$ TI

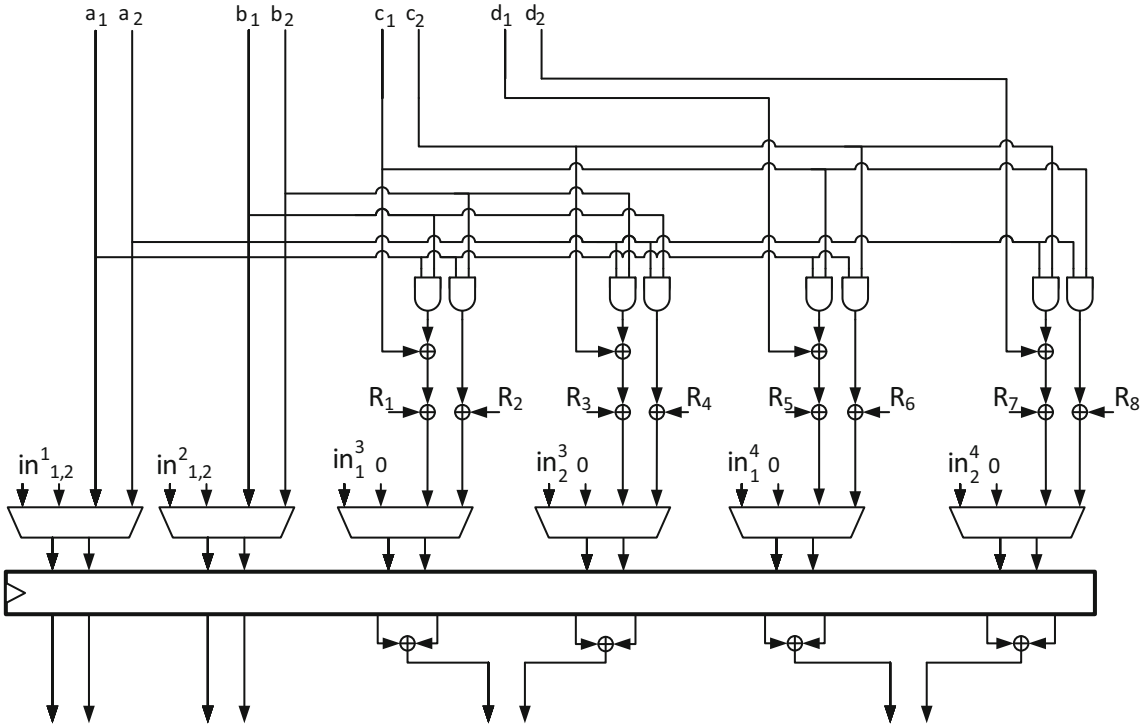


Fig. 16 First-order secure sharing of Q_{294} with $d+1$ TI

$$
\begin{aligned}
z_4 &= a_2b_1 & w_4 &= a_2c_1\\
z_5 &= a_2b_2 \oplus c_2 & w_5 &= a_2c_2 \oplus d_2\\
z_6 &= a_2b_3 & w_6 &= a_2c_3\\
z_7 &= a_3b_1 & w_7 &= a_3c_1\\
z_8 &= a_3b_2 & w_8 &= a_3c_2\\
z_9 &= a_3b_3 \oplus c_3 & w_9 &= a_3c_3 \oplus d_3\\
\bar{z}_1 &= z_1 \oplus z_2 \oplus z_3 & \bar{w}_1 &= w_1 \oplus w_2 \oplus w_3\\
\bar{w}_2 &= w_4 \oplus w_5 \oplus w_6 & \bar{w}_2 &= w_4 \oplus w_5 \oplus w_6\\
\bar{w}_3 &= w_7 \oplus w_8 \oplus w_9 & \bar{w}_3 &= w_7 \oplus w_8 \oplus w_9. \qquad (24)
\end{aligned}
$$

A hardware diagram of this sharing is depicted in Fig. 17.

Appendix B

Scheduling for the AES control for single S-box implementation where S-box latency is 6, 7, 8, 10 or 11 cycles is given with Figs. 18, 19, 20, 21, 22.

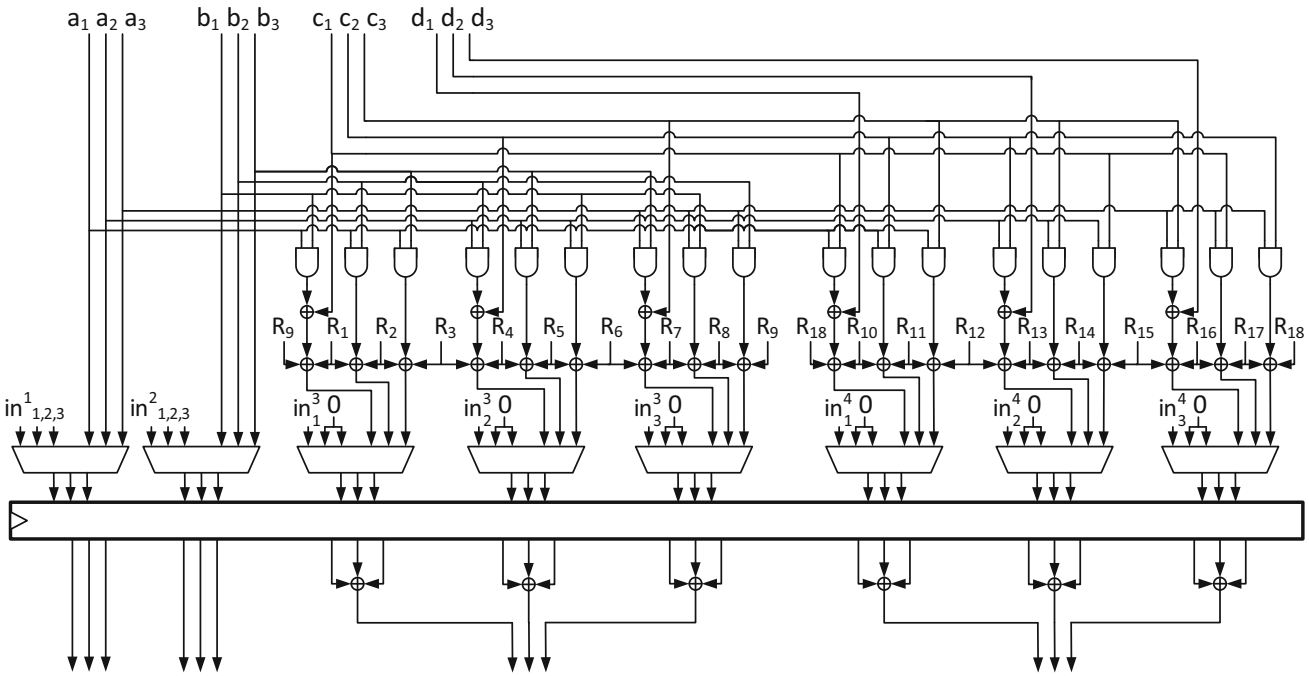


Fig. 17 Second-order secure sharing of Q_{294} with $d + 1$ TI

Fig. 18 S-box pipeline schedule with six-cycle latency

S-Box Schedule

15	2	20	9
16	18	12	7
14	1	19	6
13	17	10	8

State update

22	9	27	16
25	19	14	23
26	13	21	8
15	20	24	17

State

			3
			5
			11
			4

SRow update

12	21	22	23
18	21	22	23
11	21	22	23
10	21	22	23

MixCol update

27	21	28	24
27	21	28	24
27	21	28	24
27	21	28	24

Key

Key update

For 11-cycle S-box latency schedule, MixColumn input of the last byte is obtained directly from the S-box output and is not being written being read from the state, unlike in other cases presented here.

Appendix C

Here, we give a quick reference for the found sharings for the cases examined in Sect. 3.3. Again, we use the succinct notation, where we only given chosen shares in their lexicographical order.

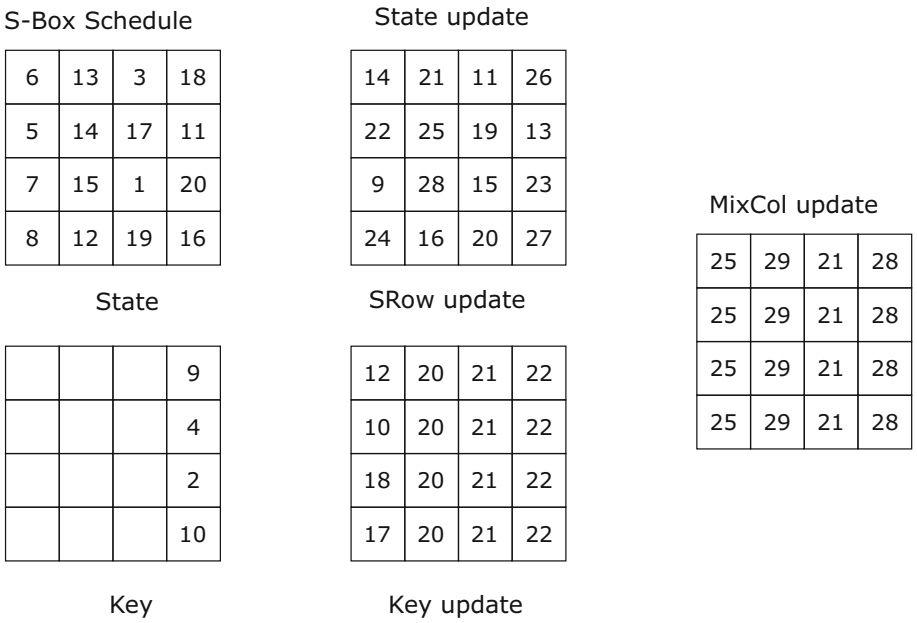


Fig. 19 S-box pipeline schedule with seven-cycle latency

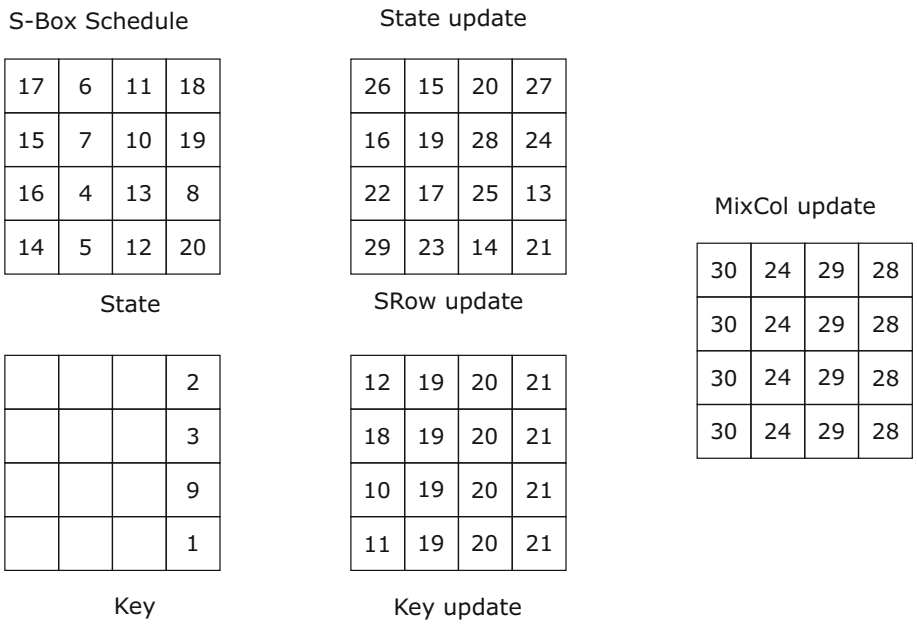


Fig. 20 S-box pipeline schedule with eight-cycle latency

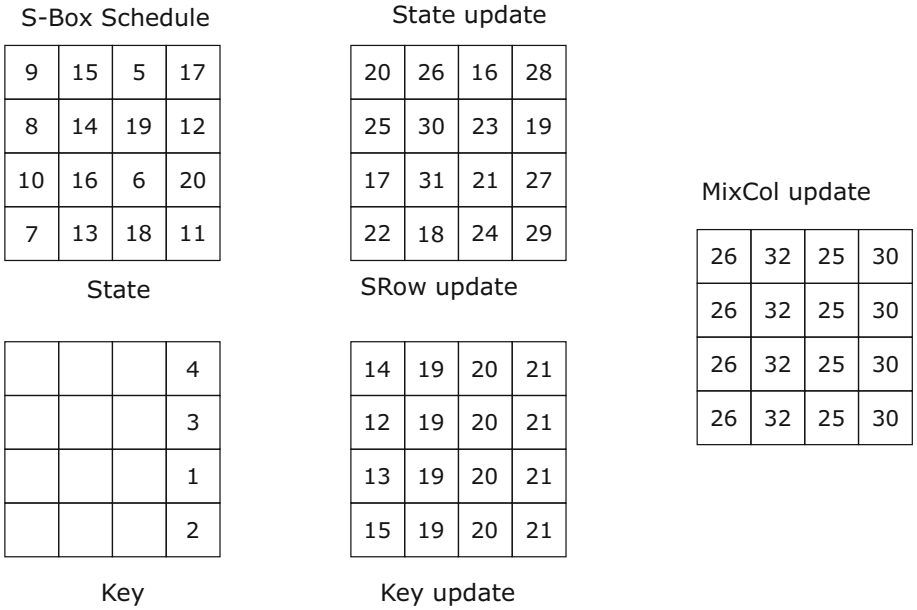


Fig. 21 S-box pipeline schedule with ten-cycle latency

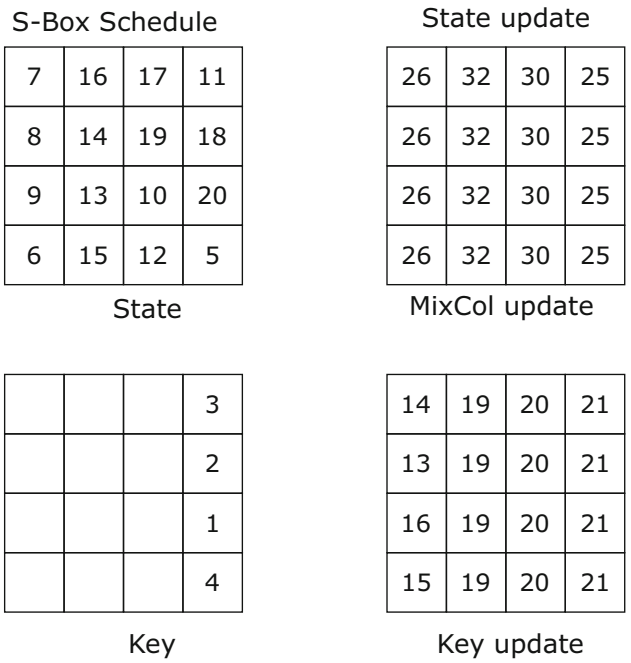


Fig. 22 S-box pipeline schedule with 11-cycle latency

References

1. Arribas, V., Bilgin, B., Petrides, G., Nikova, S., Rijmen, V.: Rhythmic Keccak: SCA security and low latency in HW. IACR Trans. Cryptogr. Hardw. Embed. Syst. 2018(1), 269–290 (2018)
2. Banik, S., Bogdanov, A., Isobe, T., Shibutani, K., Hiwatari, H., Akishita, T., Regazzoni, F.: Midori: a block cipher for low energy. In: ASIACRYPT 2015, pp. 411–436. Springer, New York (2015)
3. Borghoff, J., Canteaut, A., Güneysu, T., Kavun, E.B., Knezevic, M., Knudsen, L.R., Leander, G., Nikov, V., Paar, C., Rechberger, C., Rombouts, P.: PRINCE: a low-latency block cipher for pervasive computing applications. In: ASIACRYPT 2012, LNCS, pp. 208–225. Springer, Berlin (2012)
4. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: The KECCAK reference. http://keccak.noekeon.org/ (2011)
5. Balasch, J., Gierlichs, B., Grosso, V., Reparaz, O., Standaert, F.X.: On the cost of lazy engineering for masked software implementations. In: Joye, M., Moradi, A. (eds) 13th International Conference on Smart Card Research and Advanced Applications, CARDIS 2014, Paris, France, November 5–7, 2014. Revised Selected Papers, Volume 8968 of Lecture Notes in Computer Science, pp. 64–81. Springer (2014)
6. Bilgin, B., Gierlichs, B., Nikova, S., Nikov, V., Rijmen, V.: Higher-order threshold implementations. In: ASIACRYPT 2014, LNCS. pp. 326–343. Springer (2014)
7. Bilgin, B., Gierlichs, B., Nikova, S., Nikov, V., Rijmen, V.: A more efficient AES threshold implementation. In: Pointcheval, D., Vergnaud, D. (eds) Progress in Cryptology–AFRICACRYPT 2014, pp. 267–284. Springer, Cham (2014)
8. Bilgin, B.: Threshold implementations: as countermeasure against higher-order differential power analysis. PhD thesis, University of Twente, Enschede, Netherlands (2015)
9. Brusco, M.J., Jacobs, L.W., Thompson, G.M.: A morphing procedure to supplement a simulated annealing heuristic for cost-andcoverage-correlated set-covering problems. Ann. Oper. Res. 86, 611–627 (1999)
10. Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J., Seurin, Y., Vikkelsoe, C.: PRESENT: An ultra-lightweight block cipher. In: Paillier, P., Verbauwhede, I. (eds.) Cryptographic Hardware and Embedded Systems–CHES 2007, pp. 450–466. Springer, Berlin (2007)
11. Bilgin, B., Nikova, S., Nikov, V., Rijmen, V., Stütz, G.: Threshold implementations of all 3 ×3 and 4 ×4 s-boxes. In: CHES 2012, LNCS, pp. 76–91. Springer (2012)
12. Božilov, D.: PRINCE s-boxes verilog implementation (2021). https://github.com/dusanbozilov/PRINCETI
13. Cooper, J., DeMulder, E., Goodwill, G., Jaffe, J., Kenworthy, G., Rohatgi, P.: Test vector leakage assessment (TVLA) methodology in practice. In: International Cryptographic Module Conference (2013)
14. Cassiers, G., Grégoire, B., Levi, I., Standaert, F.-X.: Hardware private circuits: from trivial composition to full verification. Cryp-

tology ePrint Archive, Report 2020/185. https://eprint.iacr.org/2020/185 (2020)
15. De Cnudde, T., Reparaz, O., Bilgin, B., Nikova, S., Nikov, V., Rijmen, V.: Masking AES with d+1 shares in hardware. In: Cryptographic Hardware and Embedded Systems—CHES **2016**, pp. 194–212 (2016)
16. Chu, G., Stuckey, P.J.: Chuffed solver description. https://github.com/chuffed/chuffed (2014)
17. Daemen, J.: Changing of the guards: a simple and efficient method for achieving uniformity in threshold sharing. In: Fischer, W., Homma, N. (eds) Proceedings of 19th International Conference on Cryptographic Hardware and Embedded Systems—CHES 2017, Taipei, Taiwan, September 25–28, 2017, Volume 10529 of Lecture Notes in Computer Science, pp. 137–153. Springer (2017)
18. Dantzig, G.: Linear Programming and Extensions. Rand Corporation Research Study. Princeton University Press, Princeton (1963)
19. De Meyer, L., Bilgin, B., Reparaz, O.: Consolidating security notions in hardware masking. IACR Trans. Cryptogr. Hardw. Embed. Syst. 2019(3), 119–147 (2019)
20. Daemen, J., Dobraunig, C.E., Eichlseder, M., Gross, H., Mendel, F., Primas, R.: Protecting against statistical ineffective fault attacks. IACR Trans. Cryptogr. Hardw. Embed. Syst. 2020(3), 508–543 (2020)
21. De Meyer, L., Arribas Abril, V., Nikova, S., Nikov, V., Rijmen, V.: M&M: masks and macs against physical attacks. IACR Trans. Cryptogr. Hardwa. Embed. Syst. 2019, 25–50 (2018)
22. Gross, H., Iusupov, R., Bloem, R.: Generic low-latency masking in hardware. IACR Trans. Cryptogr. Hardw. Embed. Syst. 2018(2), 1–21 (2018)
23. Gross, H., Mangard, S.: Reconciling $d+1$ masking in hardware and software. In: Cryptographic Hardware and Embedded Systems—CHES, Springer (2017)
24. Gross, H., Mangard, S., Korak, T.: Domain-oriented masking: compact masked hardware implementations with arbitrary protection order. In: Proceedings of the ACM Workshop on Theory of Implementation Security, TIS@CCS 2016 Vienna, Austria, p. 3 (2016)
25. Gross, H., Mangard, S., Korak, T.: An efficient side-channel protected AES implementation with arbitrary protection order. In: Handschuh, H. (ed.) Topics in Cryptology—CT-RSA, vol. 2017, pp. 95–112 (2017)
26. LLC Gurobi Optimization. Gurobi optimizer reference manual. http://www.gurobi.com (2020)
27. Ishai, Y., Sahai, A., Wagner, D.: Private circuits: securing hardware against probing attacks. In: CRYPTO 2003, pp. 463–481. Springer, Berlin (2003)
28. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. Science 220(4598), 671–680 (1983)
29. Knezević, M., Nikov, V., Rombouts, P.: Low-latency encryption—Is "Lightweight = Light + Wait"? In: CHES 2012, LNCS, pp. 426–446. Springer (2012)
30. Minotra, D.: A study of heuristic-algorithms for set-covering problems (2008)
31. Moos, T., Moradi, A., Schneider, T., Standaert, F.X.: Glitch-resistant masking revisited- or why proofs in the robust probing model are needed. Cryptogr. Hardw. Embed. Syst. TCHES 2: 256–292 (2019)
32. Moradi, A., Poschmann, A., Ling, S., Paar, C., Wang, H.: Pushing the limits: a very compact and a threshold implementation of AES. In: Paterson, K.G. (ed.) Advances in Cryptology—EUROCRYPT 2011, pp. 69–88 (2011)
33. Moradi, A., Schneider, T.: Side-channel analysis protection and low-latency in action—case study of PRINCE and Midori. In: ASIACRYPT 2016, LNCS. Springer (2016)
34. Nikova, S.: TI tools for the 3 x 3 and 4 x 4 S-boxes. http://homes.esat.kuleuven.be/~snikova/ti_tools.html (2012)
35. Nikova, S., Nikov, V., Rijmen, V.: Decomposition of permutations in a finite field. Cryptogr. Commun. 11, 379–384 (2019)
36. Nikova, S., Rechberger, C., Rijmen, V.: Threshold implementations against side-channel attacks and glitches. In: ICICS 2006, LNCS, pp. 529–545. Springer (2006)
37. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: towards a standard CP modelling language. In: Christian Bessière, (ed.) Principles and Practice of Constraint Programming–CP 2007, pp. 529–543. Springer, Berlin (2007)
38. Papapagiannopoulos, K.: High throughput in slices: the case of PRESENT, PRINCE and KATAN64 ciphers. In: RFIDSec 2014, LNCS, pp. 137–155. Springer (2014)
39. Perron, L., Furnon, V.: OR-Tools. https://developers.google.com/optimization/ (2020)
40. Poschmann, A., Moradi, A., Khoo, K., Lim, C.W., Wang, H., Ling, S.: Side-channel resistant crypto for less than 2,300 GE. J. Cryptol. **24**(2), 322–345 (2011)
41. Reparaz, O., Bilgin, B., Nikova, S., Gierlichs, B., Verbauwhede, I.: Consolidating masking schemes. In: CRYPTO 2015, LNCS, pp. 764–783. Springer (2015)
42. Rossi, F., Van Beek, P., Walsh, T.: Handbook of Constraint Programming (Foundations of Artificial Intelligence). Elsevier, Amsterdam (2006)
43. Reparaz, O., Gierlichs, B., Verbauwhede, I.: Fast leakage assessment. In: Cryptographic Hardware and Embedded Systems—CHES, vol. 2017, pp. 387–399 (2017)
44. Sasdrich, P., Bilgin, B., Hutter, M., Marson, M.E.: Low-latency hardware masking with application to AES. IACR Trans. Cryptogr. Hardw. Embed. Syst. 2020(2), 300–326 (2020)
45. Schrijver, A.: Theory of Linear and Integer Programming. Wiley, Hoboken (1986)
46. Ueno, R., Homma, N., Aoki, T.: A systematic design of tamper-resistant galois-field arithmetic circuits based on threshold implementation with (d+1) input shares. In: IEEE 47th International Symposium on Multiple-Valued Logic (ISMVL), pp. 136–141 (2017)
47. Ueno, R., Homma, N., Aoki, T.: Toward more efficient DPA-resistant AES hardware architecture based on threshold implementation. In: Constructive Side-Channel Analysis and Secure Design—COSADE, vol. 2017, pp. 50–64 (2017)
48. Wegener, F., De Meyer, L., Moradi, A.: Spin me right round rotational symmetry for FPGA-specific AES: extended version. J. Cryptol. 33:1114 (2020)