Combined Masking and Shuffling for Side-Channel Secure Ascon on RISC-V

Linus Mainka[1] and Kostas Papagiannopoulos[1]

Informatics Institute, University of Amsterdam, Amsterdam, The Netherlands.
initial.lastname@uva.nl
https://ivi.uva.nl/

Abstract. Both masking and shuffling are very common software countermeasures against side-channel attacks. However, exploring possible combinations of the two countermeasures to increase and fine-tune side-channel resilience is less investigated. With this work, we aim to bridge that gap by both concretising the security guarantees of several masking and shuffling combinations presented in earlier work and additionally investigating their randomness cost. We subsequently implement these approaches to also analyse their performance. In this context, we present five different protected implementations of the new standard for lightweight cryptography, Ascon, on a 32-bit RISC-V architecture: A 3rd-order masked, unshuffled implementation and three combined 3rd-order masked and shuffled implementations. Additionally, we present a levelled implementation where only the particularly vulnerable keyed initialisation and finalisation of the permutation are masked and shuffled, while the rest is only shuffled. To further improve the security and performance of our implementations we make use of the Probe Isolating Non-Interference (PINI) masked AND gadget, coupled with techniques like bit-slicing and bit-interleaving. Utilising benchmarking and an MI-shortcut security analysis, we pinpoint the best masking-shuffling combinations that maximize security at reasonable overheads.

Keywords: Side-Channel Countermeasures · Cryptographic Implementations · RISC-V.

1 Introduction

Side-Channel attacks such as Differential Power Analysis [22] pose one of the most significant threats to cryptographic implementations nowadays. How to prevent these kinds of attacks has thus become a major focus of research. Designers of cryptographic primitives have begun to consider side-channel resistance as one of the design concerns when devising new cryptographic algorithms. For example, the new standard for lightweight cryptography, Ascon [14], already incorporates features such as bitslicing [6] to prevent cache-timing attacks [4] and specifically tailored aspects of the encryption process to allow for easy integration of side-channel countermeasures such as masking [10] or shuffling [20]. Both of these techniques are able to provide standalone protection and can, in

theory, be extended for an arbitrary amount of protection, given sufficient noise to amplify and sufficient independent operations to shuffle.

Related work. The Ascon algorithm suite and the Ascon-128 variant in particular have undergone significant scrutiny and found to be cryptographically sound [15]. At the same time, it has been demonstrated that side-channel attacks on Ascon are feasible and can lead to partial or even full key recovery. Correlation power analysis (CPA) has been used to recover the complete encryption key from an unprotected Ascon implementation using $8,000$ traces [24]. When augmenting the CPA with deep learning techniques, the number of traces can be reduced further to $1,000$ and partial keys can be recovered even from 1st-order secure implementations [33], highlighting the need for strong higher-order secure implementations.

Several implementations of Ascon that make use of these side-channel countermeasures have been presented. A levelled 1st-order masked software implementation was presented in [1], yet it might not suffice for a thorough defense against SCAs. Two 2nd-order masked implementations of Ascon have been proposed in [19] and [27] for software and hardware, respectively. To the best of our knowledge, no implementations of Ascon with higher masking orders have been explored.

To achieve strong hardening against side-channel attacks, it is rarely the best choice to spend all effort on a single countermeasure. As suggested by Mangard et al. [23], it is more prudent to combine several cheap countermeasures and such combinations have been analysed in the past. In [29], Prouff et al. presented a probability-theoretic argument for the effectiveness of combining masking and shuffling. This was subsequently expanded by Azouaoui et al. in [2], where several different combinations of these countermeasures were discussed and evaluated.

Contribution. In this work, we expand and concretise the work of Azouaoui et al. [2]. We apply two of the combinations (Shuffle Tuples and Shuffle Shares) from that paper to a specific platform and algorithm and present an additional new combination that leverages bit-interleaving for a further increased security benefit. Additionally, we give a new way to shuffle a particular masked AND gadget, PINI-AND [9] and finally quantify the security increases the presented techniques provide according to the Mutual Information metric [30] by adapting the shortcut formulas put forward in [2].

Furthermore, we apply the discussed techniques to Ascon on a RISC-V 32-bit architecture, resulting in five side-channel protected implementations: One 3rd-order masked implementation, three versions combining 3rd-order masking with shuffling approaches and one levelled version, which exploits the fact that only the initialisation and finalisation of the Ascon scheme need to be heavily secured against side-channel attacks, whereas the associated data and message processing parts only need much lighter protection [3].

2 Background

2.1 Ascon

In 2023, the U.S. National Institute of Standards and Technology (NIST) concluded the lightweight cryptography (LWC) standardisation process by selecting the Ascon [14] family of algorithms as the winner [31]. For this work, we are concerned with the main recommendation of the Ascon suite, Ascon-128.

The encryption algorithm itself is permutation-based using a 320-bit state made up of 5 64-bit registers. An execution of the permutation consists of a certain number of rounds. During every round r, three steps are performed:

1. A one byte round constant c_r is added to the third state register.
2. A five-bit substitution layer (S-Box) is applied to the state. This S-Box iteratively processes a vertical slice of the five state registers, changing the values according to a look-up table.
3. A linear diffusion layer is applied, rotating every state register twice by different amounts and then XORing the two, as well as the unrotated state register, back together.

A visualisation of these operations can be found in Appendix B. The lookup table of the S-Box poses a significant hurdle in multiple regards. For one, lookup tables are often prone to side-channel attacks such as cache-timing attacks [4]. More significantly, since we want to mask the permutation, we need to have a way of performing the lookup without unmasking the current value. While it is technically possible to implement a masked lookup table, this quickly becomes infeasible with higher masking orders as the memory requirements to store these tables quickly grow larger than most embedded devices have at their disposal [12].

To circumvent these issues, bitslicing [6] can be used, which is often less computationally demanding and requires less memory. In fact, the design of the Ascon S-Box was created with bitslicing explicitly in mind. In the Ascon specification, the authors already provide a sequence of XOR, NOT, and AND operations to perform on the state that yields the same output as performing the "normal" lookup-table-based S-Box.

$$
\begin{array}{lllll}
x_0 = x_0 \oplus x_4; & x_4 = x_4 \oplus x_3; & x_2 = x_2 \oplus x_1; & & \\
t_0 = x_0; & t_1 = x_1; & t_2 = x_2; & t_3 = x_3; & t_4 = x_4; \\
t_0 = \neg t_0; & t_1 = \neg t_1; & t_2 = \neg t_2; & t_3 = \neg t_3; & t_4 = \neg t_4; \\
t_0 = t_0 \wedge x_1; & t_1 = t_1 \wedge x_2; & t_2 = t_2 \wedge x_3; & t_3 = t_3 \wedge x_4; & t_4 = t_4 \wedge x_0; \\
x_0 = x_0 \oplus t_1; & x_1 = x_1 \oplus t_2; & x_2 = x_2 \oplus t_3; & x_3 = x_3 \oplus t_4; & x_4 = x_4 \oplus t_0; \\
x_1 = x_1 \oplus x_0; & x_0 = x_0 \oplus x_4; & x_3 = x_3 \oplus x_2; & x_2 = \neg x_2; &
\end{array}
$$

2.2 Bit Interleaving

Considering that we are trying to implement an encryption based on a 64-bit state on a 32-bit system, we need to split each state register into two 32-bit registers without hampering performance too heavily. The naïve separation into two

registers by simply splitting the register in the middle causes difficulties in the linear layer: With this separation, rotations of full state registers become cumbersome: A single rotation of a state register would require 7 instructions and three temporary registers in this scenario. Instead, we use a solution proposed by Bertoni et al. [5] called "Bit Interleaving": Rather than splitting the register in half, we separate the bits into even and uneven bits so that the first 32-bit register contains all the bits at even positions $b_0, b_2, \ldots, b_{62}$ of the original register and the second 32-bit register contains all the bits at odd positions $b_1, b_3, \ldots, b_{63}$. This configuration does not affect any operations except rotations. For a register x interleaved into x_e and x_o containing the even and odd bits of x, respectively, rotations by an even amount $2r$ can now be implemented by simply rotating both registers by r. Rotations by an uneven amount $2r + 1$ are implemented by rotating x_e by r, rotating x_o by $r + 1$ and then swapping the two registers. A visualisation of bit interleaving and these rotations is given in Appendix B. Although the RV32IM ISA we have chosen does not provide any rotation instructions[1], utilising this technique allows us to implement rotation with four shifts, two XORs, and one temporary register, meaning one less operation and two fewer temporary registers than the naïve version.

2.3 Masking

One of the most common countermeasures against SCAs is masking, which tries to prevent the leakage of intermediate values through the power consumption or electromagnetic emissions by "masking" relevant intermediate values with random values. More precisely, to mask a value x, it is split into $d + 1$ "shares" $x_0, \ldots, x_d$ using an involutory operation $\circ$ so that $x_0, \ldots, x_{d-1}$ contain random values $r_0, \ldots, r_{d-1}$ and $x_d = x \circ r_0 \circ \cdots \circ r_{d-1}$. One can then only obtain the original value by combining all d shares: $x_0 \circ \cdots \circ x_d = x$. This implies that an adversary trying to obtain x must now obtain all shares of the value to be able to unmask it. Consequently, there is no set of d shares an adversary can obtain that reveals any information about the original value. This is commonly referred to as d-th order security.

Commonly, $\circ$ denotes the XOR operation. Performing linear operations on masked values is straightforward. Difficulties arise when one tries to also perform non-linear operations such as AND on masked values. Several approaches have been presented to solve this, usually by incorporating additional randomness. A frequently used approach is that of Ishai, Sahai, and Wagner [21]. While this approach originally only promised $d/2$-th order security, adaptations of it have been presented that promise d-th order security, albeit with special requirements for the individual shares that sometimes require mask refreshing when composing operations [28]. However, these mask refreshing procedures have also been shown to not be unconditionally d-th order secure [13].

[1] At the time of writing, no ratified extension to RISC-V implementing rotations exists yet.

Instead, we use a relatively novel scheme proposed by Cassiers and Standaert [9] for performing masked AND operations. In contrast to the ISW scheme, this approach has been proven to be d-th order secure as is. Moreover, it is arbitrarily composable, both with itself and any other masked operation, as long as the other operation also fulfils the notion of Probe Isolating Non-Interference (PINI) introduced in the paper. This includes any linear operation masked with the same number of shares as the PINI-AND gadget. The computation and randomness cost of a PINI-AND operation is the same as with the ISW scheme (both $\mathcal{O}(d^2)$). The original algorithm also requires $\mathcal{O}(d^2)$ amounts of memory. However, in the paper the authors hint at an adaptation of the gadget that gives the same security guarantees while only requiring linear memory. Our implementation makes use of this more efficient gadget. We give an explicit description of this adaptation in Section 3.4.

2.4 Shuffling

The idea of the shuffling countermeasure is to randomise the order of execution of a set of operations. Varying the point in execution at which a certain value is calculated between different executions of the encryption increases the difficulty for an adversary to correlate values from a given power trace or similar side-channel measurement [20].

Formally, shuffling takes an input vector y containing all inputs to process with a given operation $\mathtt{op}(\cdot)$. Additionally, a permutation $\theta \in \Theta_{|y|}$ is supplied where $\Theta_{|y|}$ is the set of all permutations of the sequence $[0, |y| - 1]$. During the execution of the cipher, the program iterates over θ so that at iteration i, y_{θ_i} is accessed, and $z_{\theta_i} = \mathtt{op}(y_{\theta_i})$ computed.

Two things need to be observed in particular when shuffling: First, the set of objects to shuffle must be sufficiently similar that an adversary cannot easily tell them apart in a power analysis. In practice, this usually means that all operations have to be the same and inputs to the operations need to be either all fixed or all random. For example, AND and XOR operations can already have distinct enough "leakage signatures" that an adversary can be able to tell them apart in a power trace. Secondly, one must be aware that if there is enough leakage of the permutation used to shuffle, an adversary can simply obtain this permutation and the benefit of shuffling vanishes.

2.5 Mutual Information

The Mutual Information (MI) framework [30] is a system of information-theoretic metrics created to measure the amount of information an implementation leaks while performing encryptions. Based on information theory, it employs metrics such as Shannon's Conditional Entropy to correlate leakages measured by an adversary to specific keys used in an encryption. It can thus give an estimation of the information leakage of a certain value (given sufficiently noisy conditions) and further be used to estimate the probability of success of an adversary recovering that value given a certain number of leakage measurements [16,11].

Concretely, let us assume we have a vector of secret keys K and a corresponding matrix of leakages L where every element of the matrix is a vector of leakages measured while using key k_i. We further assume that keys are uniformly randomly distributed, in which case we can calculate the conditional probability of a key given a leakage. Using this, we can estimate the mutual information $\mathrm{MI}(K;L)$ via the key entropy by sampling from this key distribution [7]. Generally, $\mathrm{MI}(K;L)$ is bound from above by the key entropy and is zero if leakages convey no information at all about the key used. Realistically, neither of these cases is likely. Usually, it is a value between 0 and 1 and should ideally be as small as possible. In that case, one can estimate the number of traces N needed to perform a statistical attack like correlation power analysis (CPA) very roughly using the mutual information: $N \geq \frac{c}{\mathrm{MI}(K;L)}$. Here, c is a constant depending on the key entropy and the desired success rate of the attack. We will use this formula to estimate the security improvements of our devised schemes in Section 5.

3 Combining Masking and Shuffling for Bit-Interleaved Schemes

As mentioned, the schemes presented in [2] serve as the baseline for this work. In this section, we discuss the adaptations of these schemes we devised to fit the specific structure of Ascon and the design choices made for this implementation. Although our focus here was Ascon, we want to highlight that the approaches presented can also be applied to other algorithms or platforms where the state register size of the algorithm is a multiple of two of the platform register size. For simplicity, the approaches here are shown with an interleaving factor of two. They could, however, be trivially adapted to other interleaving factors, e.g. using an interleaving factor of four if one were to implement Ascon on a 16-bit system.

Before we can discuss combinations of masking and shuffling, we need to establish how we mask values. A general algorithm for this masking and interleaving approach, independent of Ascon, is given in Algorithm 1.

To present our shuffling countermeasures in the following Sections (3.1. 3.2, 3.3 and 3.4), we use the following notation: Let $A = [a_0, \ldots, a_{n-1}]$ and $B = [b_0, \ldots, b_{n-1}]$ be two lists each consisting of n d-th order masked, interleaved values as previously described, so e.g. $a_i = [[a^0_{i_e}, \ldots, a^d_{i_e}], [a^0_{i_o}, \ldots, a^d_{i_o}]]$. Furthermore, we define $C = [c_0, \ldots, c_{n-1}]$, where $c_i = \mathtt{op}(a_i, b_i)$ is the result of combining a_i and b_i with a binary linear operation $\mathtt{op}(\cdot)$ adapted to work on these masked, interleaved shares. A visualisation of all three approaches is given in Appendix B.

3.1 Shuffling Tuples

The first and simplest approach to combining masking and shuffling countermeasures, as presented in [2], is to mask first and then shuffle in the same way as in an unmasked implementation. More precisely: given a permutation θ of $[0, n-1]$,

Algorithm 1 Masking and interleaving values

Inputs: $[a_0, \ldots, a_{n-1}]$, Masking order d
for $i = 0$ to n **do**
  $a_{i_e}, a_{i_o} \leftarrow \texttt{interleave}(a_i)$
  for $j = 0$ to $d-1$ **do**
    $a_{i_e}^j \xleftarrow{\$} \mathbb{F}_{2^{32}}$
    $a_{i_o}^j \xleftarrow{\$} \mathbb{F}_{2^{32}}$
  end for
  $a_{i_e}^d \leftarrow a_{i_e} \oplus a_{i_e}^0 \oplus \cdots \oplus a_{i_e}^{d-1}$
  $a_{i_o}^d \leftarrow a_{i_o} \oplus a_{i_o}^0 \oplus \cdots \oplus a_{i_o}^{d-1}$
end for
Outputs:
$[[[a_{0_e}^0, \ldots, a_{0_e}^d], [a_{0_o}^0, \ldots, a_{0_o}^d]], \ldots, [[a_{n_e}^0, \ldots, a_{n_e}^d], [a_{n_o}^0, \ldots, a_{n_o}^d]]]$

we can shuffle the masked operations as we would if they were unmasked by performing them in the order $\texttt{op}(a_{\theta_0}, b_{\theta_0}), \ldots, \texttt{op}(a_{\theta_{n-1}}, b_{\theta_{n-1}})$. A precise algorithm for this procedure is given in Algorithm 2 and our interleaved implementation utilises it as is.

Algorithm 2 "Shuffling Tuples" masking and shuffling combination

Inputs: $a = [a_0, \ldots, a_{n-1}],\ b = [b_0, \ldots, b_{n-1}],\ \texttt{op}(\cdot),\ \theta$
for $i = 0$ to $n-1$ **do**
  $k \leftarrow \theta_i$ ▷ Index to calculate next
  $c_k \leftarrow \texttt{op}(a_k, b_k)$ ▷ Perform the masked operation
end for
Outputs: $c = [c_0, \ldots, c_{n-1}]$ so that $\forall i :\ \texttt{op}(a_i, b_i) = c_i$

3.2 Shuffling Shares

For the following two shuffling approaches (Sections 3.2, 3.3), the methods for shuffling linear and non-linear operations differ. Here, we will only discuss approaches for shuffling masked linear operations and extend our approach to non-linear operations in Section 3.4.

In this second approach, we do not shuffle the entire masked operations with each other but instead shuffle at the share-level. Note that there is no benefit to shuffling the order of processing of shares of the same value. This is because to successfully recover an unmasked value, an adversary needs to obtain all shares of that value, irrespective of the order in which they obtain them. Instead, we shuffle over the shares of different values in the following way: For two linear operations $a_0 \oplus b_0$ and $a_1 \oplus b_1$ we shuffle first the processing of all $a_{i_e}^k \oplus b_{i_e}^k$, where k is the current share index as dictated by the permutation used. In a

second step, we process all $a^k_{i_o} \oplus b^k_{i_o}$. The sequences we shuffle over now are the sets of i-th shares of first the even, then the odd half of every interleaved register. Overall, this means we shuffle $d+1$ sets of n operations. See Algorithm 3 for a precise description of this procedure.

Algorithm 3 "Shuffling Shares" masking and shuffling combination

Inputs: $a = [a_0, \ldots, a_{n-1}],\ b = [b_0, \ldots, b_{n-1}],\ \texttt{op}(\cdot),\ \theta$
for $i \in \{e, o\}$ **do** ▷ First process the shares of the even register, then the odd
  for $j = 0$ to d **do** ▷ Iterate over the shares
    for $l = 0$ to $n-1$ **do**
      $k \leftarrow \theta_l$ ▷ Index to calculate next
      $c^j_{k_i} \leftarrow \texttt{op}(a^j_{k_i}, b^j_{k_i})$ ▷ Perform the masked operation
    end for
  end for
end for
Outputs: $c = [c_0, \ldots, c_{n-1}]$ so that $\forall i:\ \texttt{op}(a_i, b_i) = c_i$

3.3 Shuffling Everything "Light"

The third described countermeasure combination in [2] is called "Shuffling Everything", where the authors suggest combining all shares of all registers used into one permutation. Albeit technically possible, this approach grows significantly more complex when incorporating non-linear operations. Both due to suspected diminishing returns with respect to security and due to the scope of this work we elected not to investigate this, but instead to find more straightforward ways of shuffling more things with each other.

For shuffling shares, we only shuffle between registers at exactly the same position. This means that we differentiate between the "even" and "odd" registers of which one state register is comprised. However, this is not strictly necessary: in a linear operation (that is not a shift or rotation), the even and odd registers making up one state register never interact with each other. Thus, we can shuffle the i-th shares of an even and odd register with each other as well. We refer to this as "Shuffling Everything Light" (Shuffle EL). The permutation θ thus consists not only of the indices $[0, n-1]$ of the complete operations to shuffle, but also includes for every index an even and an odd variant. Consequently, the permutation is of the following form.

$$\theta = \{0_e, 0_o, 1_e, 1_o, \ldots, n-1_e, n-1_o\}, \quad |\theta| = 2n$$

This doubles the number of operations we can shuffle with one permutation. A more precise description is given in Algorithm 4.

Algorithm 4 "Shuffling Everything Light" masking and shuffling combination

Inputs: $a = [a_0, \ldots, a_{n-1}],\ b = [b_0, \ldots, b_{n-1}],\ \mathtt{op}(\cdot),\ \theta = \{0_e, 0_o, \ldots, n-1_e, n-1_o\}$
for $j = 0$ to d **do** ▷ Iterate over the shares
  for $l = 0$ to $2n-1$ **do** ▷ We now combine the even and odd part
    $k \leftarrow \theta_l$ ▷ Index to calculate next
    $c_k^j \leftarrow \mathtt{op}(a_k^j, b_k^j)$ ▷ Perform the masked operation
  end for
end for
Outputs: $c = [c_0, \ldots, c_{n-1}]$ so that $\forall i:\ \mathtt{op}(a_i, b_i) = c_i$

3.4 Shuffling PINI-AND Operations

When shuffling linear operations, there is a clear separation of when each share is processed. In a non-linear operation, this separation becomes much less clear, requiring us to devise specially crafted shuffling schemes, different from those for linear operations.

An algorithmic description of the gadget we use is given in Algorithm 5. It differs slightly from the original algorithm description given in [9], as we have implemented the hinted-at adaptated version which requires only linear memory by directly calculating every z_{ij} and z_{ji} from each r_{ij}, removing the necessity to store each r_{ij}.

Algorithm 5 PINI AND gadget with linear memory requirements

Inputs: $a = [a_0, \ldots, a_d]$, $b = [b_0, \ldots, b_d]$
for $i = 0$ to d **do**
  $c_i \leftarrow a_i b_i$
end for
for $i = 0$ to d **do**
  for $j = i + 1$ to d **do**
    $r_{ij} \xleftarrow{\$} \mathbb{F}_{2^{32}};\ r_{ji} \leftarrow r_{ij}$
    $z_{ij} = (a_i + 1) \cdot r_{ij} + a_i \cdot (b_j + r_{ij})$
    $z_{ji} = (a_j + 1) \cdot r_{ji} + a_j \cdot (b_i + r_{ji})$
    $c_i \leftarrow c_i + z_{ij}$
    $c_j \leftarrow c_j + z_{ji}$
  end for
end for
Outputs: $c = [c_0, \ldots, c_d]$ so that $c = a \wedge b$

To determine a shuffling approach for this gadget, we need to consider when each share is processed and which computed values have a dependence relation. When calculating the initial $c_i = a_i b_i$ in Algorithm 5 we can still clearly separate the share accesses but during the calculation of the values z_{ij}, different shares are accessed, muddying the distinction. This makes shuffling across shares infeasible for this gadget. On the other hand, note that the computation of every z_{ij} is

independent from all other intermediate values computed in the gadget, meaning the order in which they are computed does not matter. This yields a first approach for shuffling. However, shuffling the computation of intermediate values inside one PINI gadget does not enhance security, similarly to how shuffling the processing of shares inside one linear gadget does not enhance security.

Consequently, we settled on shuffling the computation of $a_i b_i$ and z_{ij} for different a and b, thus shuffling across gadgets, instead of inside a gadget. This means we, for example, shuffle the computation of all $a_0 b_0$ for a given set of PINI AND operations. The same applies for the computation of all z_{01}, z_{10}, and so forth. The logic of *how* we shuffle across gadgets (i.e. whether we shuffle over each interleaved register separately or whether we combine these) is exactly the same as with linear gadgets, allowing for a seamless combination of linear and non-linear gadgets in our countermeasure combinations. For completeness, the algorithms for shuffling shares and shuffling everything light have also been written out in Algorithms 6 and 7, respectively. In these algorithms, $x \overset{+}{\leftarrow} y$ is a shorthand for $x \leftarrow x + y$.

Algorithm 6 "Shuffling Shares" masking and shuffling combination for PINI-AND

```
Inputs: $a = [a_0, \ldots, a_{n-1}],\ b = [b_0, \ldots, b_{n-1}]$,
        Set $\Theta_n$ of permutations of length $n$ to sample from
for $l \in \{e, o\}$ do        ▷ First process the shares of the even register, then the odd
    for $k = 0$ to $d$ do        ▷ Iterate over the shares
        $\theta \overset{\$}{\leftarrow} \Theta_n$        ▷ Use a random permutation
        Shuffle($[a^k_{0_l} \wedge b^k_{0_l}, \ldots, a^k_{n-1_l} \wedge b^k_{n-1_l}], \theta$)        ▷ Shuffle the calculation of $a_i b_i$
    end for
    for $i = 0$ to $d$ do
        for $j = i + 1$ to $d$ do
            $\theta \overset{\$}{\leftarrow} \Theta_n$
            Shuffle($[c_{i0_l} \overset{+}{\leftarrow} z_{ij0_l}, \ldots, c_{i0_l} \overset{+}{\leftarrow} z_{ij0_l}], \theta$)        ▷ Shuffle the calculation of $z_{ij}$
            $\theta \overset{\$}{\leftarrow} \Theta_n$
            Shuffle($[c_{j0_l} \overset{+}{\leftarrow} z_{ji0_l}, \ldots, c_{j0_l} \overset{+}{\leftarrow} z_{ji0_l}], \theta$)        ▷ Shuffle the calculation of $z_{ji}$
        end for
    end for
end for
Outputs: $c = [c_0, \ldots, c_{n-1}]$ so that $\forall i :\ \mathtt{op}(a_i, b_i) = c_i$
```

3.5 Randomness Cost

Generally, to mask a set of n r-bit values in our configuration with an interleaving factor of l, we need to generate $n \cdot r \cdot d$ bits of randomness. Note that the required randomness does not depend on the interleaving factor, as the overall size of the values to mask does not change. In the case of Ascon, for example, this results

Algorithm 7 "Shuffling Everything Light" masking and shuffling combination for PINI-AND

Inputs: $a = [a_0, \ldots, a_{n-1}],\ b = [b_0, \ldots, b_{n-1}]$,
  Set Θ_{2n} of permutations of length $2n$ to sample from
for $k = 0$ to d **do** ▷ Iterate over the shares
  $\theta \xleftarrow{\$} \Theta_{2n}$ ▷ Use a random permutation
  Shuffle($[a^k_{0_e} \wedge b^k_{0_e}, a^k_{0_o} \wedge b^k_{0_o} \ldots, a^k_{n-1_e} \wedge b^k_{n-1_e}, a^k_{n-1_o} \wedge b^k_{n-1_o}], \theta$)
end for
for $i = 0$ to d **do**
  for $j = i + 1$ to d **do**
    $\theta \xleftarrow{\$} \Theta_{2n}$
    Shuffle($[c_{i0_e} \xleftarrow{+} z_{ij0_e}, c_{i0_o} \xleftarrow{+} z_{ij0_o}, \ldots, c_{i0_e} \xleftarrow{+} z_{ij0_e}, c_{i0_o} \xleftarrow{+} z_{ij0_o}], \theta$)
    $\theta \xleftarrow{\$} \Theta_{2n}$
    Shuffle($[c_{j0_e} \xleftarrow{+} z_{ji0_e}, c_{j0_o} \xleftarrow{+} z_{ji0_o}, \ldots, c_{j0_e} \xleftarrow{+} z_{ji0_e}, c_{j0_o} \xleftarrow{+} z_{ji0_o}], \theta$)
  end for
end for
Outputs: $c = [c_0, \ldots, c_{n-1}]$ so that $\forall i :\ \mathtt{op}(a_i, b_i) = c_i$

in a randomness requirement of $5 \cdot 64 \cdot d = 320d$ bits. The randomness cost of shuffling stems from generating the shuffling permutation alone. Generally, a permutation of length n can be generated with $\Theta(n \log_2 n)$ bits of randomness [17]. For shuffling tuples, a single permutation of length n is needed to shuffle the n operations. For shuffling shares, one permutation is needed per share index. Thus, for d-th order security, $\Theta((d+1)n \log_2(n))$ bits of randomness are required. For shuffling everything light, the situation is the same as for shuffling shares, with the only difference being the larger size of $2n$ for the permutation. Following, the randomness required is $\Theta((d + 1)2n(\log_2(n) + 1))$ bits.

On top of this, we need to account for the extra randomness required by all schemes for the computation of each PINI-AND gadget. Since every PINI-AND computation on an r-bit device requires $r\frac{(d+1)d}{2}$ bits of randomness, computing a set of n such operations requires an additional $n \cdot r\frac{(d+1)d}{2}$ bits of randomness. The randomness cost for shuffling PINI-AND operations with these schemes is not directly bound by the security order, but by the number of independent operations performed inside the non-linear gadget. Thus, in our case, shuffling shares requires $\Theta(d^2 n \log_2(n))$ bits of randomness and shuffling everything light requires $\Theta(d^2 2n(\log_2(n) + 1))$ bits. Shuffling tuples does not require any extra randomness to shuffle the PINI-AND operations.

For clarity, the randomness values of all schemes, both in the linear and non-linear case, are also listed in Table 1. Refer to Appendix B for a graph showing the randomness requirement versus the masking order d. In Section 5, we will provide specific numbers for the randomness needed when applying these schemes to a third-order masked and a levelled version of Ascon.

Table 1: The randomness requirements of the different shuffling approaches, measured in bits. Values in the non-linear column do not include the randomness required in the computation of the non-linear gadgets.

	Linear	Non-linear
Shuffle tuples	$n\log_2(n)$	$n\log_2(n)$
Shuffle shares	$(d+1)n\log_2(n)$	$d^2n\log_2(n)$
Shuffle EL	$(d+1)2n(\log_2(n)+1)$	$d^2 2n(\log_2(n)+1)$

4 Implementation

To benchmark the schemes proposed in a practical setting, we implemented five protected variants of the Ascon-128 encryption scheme. One implementation uses only 3rd-order masking for protection. Three implementations combine 3rd-order masking with Shuffle Tuples, Shuffle Shares and Shuffle EL respectively. Finally, we also implemented a levelled version, where only the initialisation and finalisation are masked and shuffled (with 3rd-order masking and Shuffle EL), while the rest of the cipher is shuffled (with Shuffle EL) but is not masked. All five variants were implemented in RV32IM assembly code and the source code can be found here: `https://uva-hva.gitlab.host/l.mainka/side-channel-secure-ascon`.

4.1 Optimising the S-Box

The bitsliced S-Box presented in the Ascon specification (shown again in Section 2.1) provides a good foundation for an initial implementation. There are, however, means to optimise it in our case. Concretely, the second and third row of the given bitsliced S-Box consist only of copy and NOT operations, respectively and can be combined into a single operation. Going further, we can also combine these two rows with the subsequent row of AND operations. While not a meaningful optimisation in an unprotected implementation, in a masked and shuffled implementation it saves a substantial amount of instructions because it reduces the number of instruction sets to shuffle by one. This means one less round of computing jump offsets and at least one less load and store for every share. In addition, as a masked negation is performed simply by negating an arbitrary share, integrating the negation into the AND should in theory only cost a single instruction. In reality, a few more instructions are required, but still many less than if we were performing the operations separately. The S-Box we now compute then looks as follows.

$$x_0 = x_0 \oplus x_4; \quad x_4 = x_4 \oplus x_3; \quad x_2 = x_2 \oplus x_1;$$
$$t_0 = \neg x_0 \wedge x_1; \quad t_1 = \neg x_1 \wedge x_2; \quad t_2 = \neg x_2 \wedge x_3; \quad t_3 = \neg x_3 \wedge x_4; \quad t_4 = \neg x_4 \wedge x_0;$$
$$x_0 = x_0 \oplus t_1; \quad x_1 = x_1 \oplus t_2; \quad x_2 = x_2 \oplus t_3; \quad x_3 = x_3 \oplus t_4; \quad x_4 = x_4 \oplus t_0;$$
$$x_1 = x_1 \oplus x_0; \quad x_0 = x_0 \oplus x_4; \quad x_3 = x_3 \oplus x_2; \quad x_2 = \neg x_2;$$

The number of instructions we save with this implementation varies between the different shuffling schemes. We report them in Table 2. The savings from

Table 2: The instruction count of the masked S-Box without (Standard) and with (Optimised) combining the copy, NOT and AND per shuffling scheme.

	Shuffling Tuples	Shuffling Shares	Shuffling EL
Standard	2384	6175	6130
Optimised	2304	5797	5756

this optimisation are so substantial in the latter two schemes because it reduces the number of operation sets we shuffle by one. When using Shuffling Shares and Shuffling EL, this saves us one whole iteration of computing jump offsets. Moreover, as alluded to previously we need to load/store all values before/after each computation, meaning this also reduces the number of times the entire state is loaded into registers and written back into memory by one.

For a brief discussion of other optimisations avenues that we decided against, refer to Appendix A.

4.2 Shuffling

Generating the permutation. For generating a (theoretically) unbiased random permutation of a sequence $s = [0, n-1]$, the algorithm presented in [17] is a common choice that we also utilised for this work. It generates the permutation in $\mathcal{O}(n)$ time and requires $\Theta(n \log_2 n)$ bits of randomness. While there are caveats to using this algorithm, such as possible bias, they do not pose hazards to the security of our implementation, since the bias induced by side-channel leakage is typically higher.

Implementing Shuffling. In [32], Veyrat-Charvillon et al. give a taxonomy of three possible paths for implementing shuffling. In the first, called "Double Indexing", an operand and a permutation vector are stored in memory. At every shuffling step, the permutation vector is accessed and the retrieved value is used as the index to access the operand vector. The retrieved operands are then fed into the operation. For the second approach, the authors propose writing out the code for each operation and giving it a label to store in an array in memory at compile time. At runtime, the order of this array is randomised and every entry is used successively to jump to the stored address and to perform the operation at that address. Finally, while the previous two approaches determined the next operation to execute at runtime, the third approach is to use the self-programming capabilities some chips have to reorder the program memory after compiling so that at runtime, the program can be executed without additional control logic.

In our work, approach three was not considered, as the number of times the flash memory of a chip can be rewritten is prohibitively low for use in practice. Similarly, we decided against approach one due to the significant performance overhead of creating operand vectors and then accessing memory twice for every operation.

The approach we chose is a modification of the second approach. Instead of giving every operation a label and assembling the addresses of these labels in an array at compile time, we place the operations sequentially in the program, set one label before the beginning of the first operation, one label after the last operation and determine the code size of the operations. At runtime, we then access a permutation vector in memory and calculate an address offset by multiplying the operation code size by the permutation index and adding the resulting value to the initial label. After all operations of one shuffling set have been performed, a final jump to the label after the last operation is done and the next instruction can be executed. Our approach allows us to determine the next operation to execute in only five instructions without requiring significant logic at compile-time and saves us from randomising the address vectors at runtime. Additionally, it ensures by design that all shuffled operations have exactly the same length, preventing potential side-channel leakage due to different operation sizes.

What to shuffle. Our general motivation was to maximize the amount of shuffled components, meaning in particular that we try to shuffle the entire permutation. The bitsliced version of the S-Box as described in the Ascon specification has a natural structure of applying five operations at a time to the five state registers. Consequently, we chose to shuffle blocks of five masked operations. For the first and last XORs and the last NOT which do not comprise five operations, we introduced additional dummy operations. Accounting for the combining of the copy, NOT, and AND operations, this gives us five sets of operations across which we can shuffle. [2]

It is very important to note that shuffling significantly increases register pressure. Since there is no guarantee anymore that operations are performed in a particular order, a general rule is that results can not be written back to the same register immediately, but have to be stored in a temporary location until all operations of the current shuffling block have been performed. Take for example the first shuffling block of the S-Box. If we calculate $x_0 = x_0 \oplus x_4$ and then $x_4 = x_4 \oplus x_3$, everything is fine. Should we now shuffle with a permutation that performs these operations in inverted order, x_4 is updated before x_0 is, leading to incorrect results. Thus, we need to store the results of these operations in a temporary location before storing them back once all five have been completed. Naturally, these storing back operations can be (and are) shuffled, too.

In the linear layer, our use of bit interleaving posed a hurdle: Since the two bit interleaved registers need to be swapped for an odd rotation but not for an even rotation (cf. Section 2.2 or the visualisation in Appendix B), the

[2] The authors of [2] note that adding dummy operations in a masking-shuffling combination is rarely worth it from a security standpoint. We decided to add them regardless for two reasons: 1) Since in a levelled implementation a significant portion of the permutation will only be shuffled but not masked, the effect is more pronounced, and 2) With the dummy operations all shuffling sets will be of the same size, significantly simplifying the implementation.

operations needed to rotate by an even or an odd amount differ. Considering that we would like for all shuffled blocks to be of the same size to avoid distinguishing characteristics in time and power that can undo the benefit of shuffling, we introduced a dummy swap in the even rotation.

Due to the shape of the Ascon state, the linear layer also naturally lends itself to shuffling blocks of five operations. The granularity of the blocks to shuffle leaves more room for possibilities: For shuffling tuples, we considered each transformation of one state register as one block, thus giving us only a single set of operations to shuffle which each consists of two rotations and three XORs. For shuffling shares, we did separate the aforementioned blocks, giving us four sets of operations to shuffle: Two XORs and two rotations. As in the S-Box, we first shuffled the computation of the first 32-bit registers of the state registers and then performed the shuffled computation of the second. Shuffling Everything Light introduced a further difficulty, as the technique relies on the ability to process the two interleaved registers at the same time. Due to the two interleaved registers switching places in case of an uneven rotation, we can only have the rotation of both registers in the same shuffling set if the results are written to an intermediate location first and only moved to the correct place after all operations of the current shuffling set have been completed.

4.3 Randomness requirements

Section 3 already discussed the theoretical randomness requirements for the three shuffling approaches. We will now use these formulas to provide explicit values for the number of bits of randomness required for all five implemented variants of Ascon-128. Since the randomness requirement of the PINI-AND operations does not differ between shuffling schemes and only depends on the masking order, we calculate this value once and add it to the requirements of every (masked) scheme: Every round of the permutation contains five non-linear PINI-AND operations on the state. Since we split the registers in two, this results in ten 32-bit PINI AND operations. These operations require an additional $10 \cdot 16(d+1)d = 160 \cdot 4 \cdot 3 = 1920$ bits of randomness.

Shuffling Tuples The S-Box consists of three blocks of XORs, one block of PINI-AND and one block of NOTs we need to shuffle. On top of these, we need two blocks for shuffling the "storeback" operations mentioned in the previous section. Additionally, we have one further block for the linear layer. All eight of these blocks consist of five operations, meaning we have a rounded up randomness requirement of

$$8 \cdot (5 \log_2(5)) \approx 93 \text{ bits}$$

for generating the shuffling permutations for this approach.

Shuffling Shares For shuffling shares, we still only shuffle blocks of five operations. However, since we are now shuffling the processing of shares across

operations (in the case of linear operations) and separate the two halves of each state register into separate shuffling blocks, we now need eight shuffling blocks to complete one set of five operations in the S-Box. As we have the same amount of operations to perform as when shuffling tuples, this leaves us with $8 \cdot 6 = 48$ shuffling blocks in total for the linear operations of the S-Box. To shuffle the non-linear PINI-AND operations we require a total of 14 shuffling blocks to compute all intermediate values. Additionally, we need four more blocks to aggregate all intermediate values and to move them into the correct location, Finally, we need to double these numbers again to account for the separate processing of the interleaved halves of each state register. In total, we thus have 84 shuffling blocks in the S-Box. In the linear layer, each rotation and each subsequent XOR require one shuffling block, adding a further 16 blocks. This yields a total of

$$(84 + 16)(5 \log_2(5)) \approx 1161 \text{ bits}$$

to generate all permutations for shuffling in this approach.

Shuffling Everything Light For shuffling everything light, we now shuffle blocks of ten operations because we join the operations on each half of the state register into one shuffling block. This means that we only have six operations to shuffle for the S-Box. As we are masking with four shares, the total number of shuffling blocks for the S-Box is thus 24. In the linear layer we merged the rotations and following XOR into one operation, meaning we only have three operations to shuffle, resulting in a total of 12 shuffling blocks, Overall, we thus need

$$(24 + 12)(10 \log_2(10)) \approx 1196 \text{ bits}$$

of randomness in this scheme.

5 Analysis

5.1 Performance

Section 4.1 already provides information regarding the performance of the proposed schemes in terms of instruction counts. This section will also present the number of cycles our schemes need on a device since the correspondence between cycles and instructions is not always one-to-one. To obtain these numbers we used a QEMU simulation of a SiFive HiFive 1 Rev B [3], a board built around a SiFive FE310-G002 chip which provides the RV32IMAC ISA. Cycle measurements were performed through the RISC-V provided `RDCYCLE` control-and-status register (CSR).

In Figure 1 the number of cycles needed for one round of the permutation are shown per scheme. For comparison, we include an unmasked, but shuffled variant. In Table 3 we present the number of cycles needed to process a single block of plaintext, as well as the resulting throughput in bits per cycle.

[3] `https://www.sifive.com/boards/hifive1-rev-b`

Table 3: Number of clock cycles to process a single byte of plaintext and related throughput. All schemes are 3rd-order PINI masked.

	Unshuffled	ShuffleTuples	ShuffleShares	ShuffleEL	Levelled
No. Clock Cycles	16,395	20,842	50,326	51,481	7,301
Throughput (bits/cycle)	0.004	0.003	0.0013	0.0012	0.009

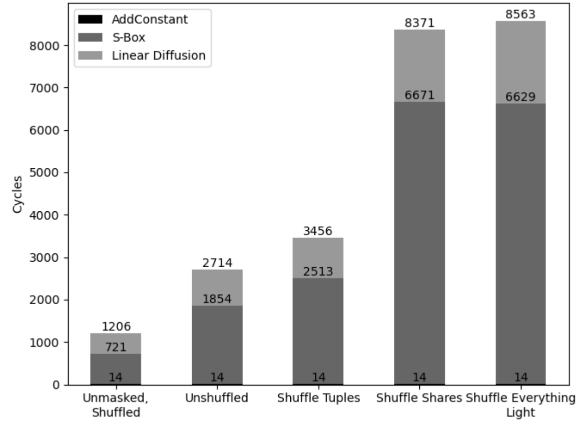

Fig. 1: The number of cycles needed to compute one round of the permutation per scheme, separated by the different steps. With the exception of the "Unmasked, Shuffled" variant, all others variants are 3rd-order PINI masked.

5.2 Security

To quantify what increases in security the implemented masking-shuffling combinations have brought, we performed an analysis using the Mutual Information framework and adapt the shortcut formulas from [2] to our customized shuffling variants.

When analysing a d-th order masked gadget, a potential attacker needs to obtain information about all $d+1$ shares of a key (Eq. 1). Instead, if attacking a shuffled gadget where a set of n operations are shuffled, the complexity of the attack increases linearly with the size of the permutation (Eq. 2).

$$N \geq \frac{c}{\prod_{j=0}^{d} \mathrm{MI}(K^j; L)} \quad (1) \qquad\qquad N \geq \frac{c \cdot n}{\mathrm{MI}_u(K; L)} \quad (2)$$

Here, $\mathrm{MI}_u(K;L)$ denotes the mutual information of an unshuffled implementation and we will assume that an adversary is attacking a single 32-bit chip register.

Shuffling Tuples. In the case of shuffling tuples, we are first masking an implementation and then shuffling the masked operations as a whole. An adversary thus needs to attack all n operations of a shuffling permutation to ensure that they obtain the specific value they are interested in. Furthermore, they need to obtain information about all shares of the value of interest, obtaining Eq. 3.

$$N \geq \frac{c \cdot n}{\prod_{j=0}^{d} \mathrm{MI}_u(K^j; L)} \tag{3}$$

Shuffling Shares. In the shuffling shares scenario, we no longer shuffle entire masked operations but instead shuffle across the shares of n operations. This implies that an attacker wishing to attack a particular masked value needs to perform multiple attacks on different shuffling permutations. The adversary has a probability of $\frac{1}{n}$ to obtain the share of the desired value in a shuffling permutation. Since they must additionally obtain all $d+1$ shares of the value, they must achieve this $d+1$ times, giving them a probability of $\left(\frac{1}{n}\right)^{d+1}$ to succeed.

$$N \geq \frac{c \cdot n^{d+1}}{\prod_{j=0}^{d} \mathrm{MI}_u(K^j; L)} \tag{4}$$

Shuffling Everything Light. Assuming an attacker recovering a single 32-bit register, this scheme effectively doubles the difficulty for an attacker to obtain the sought after register from the shuffling permutation. Therefore, all that changes from shuffling shares is the numerator increasing by a factor of two.

$$N \geq \frac{c \cdot (2n)^{d+1}}{\prod_{j=0}^{d} \mathrm{MI}_u(K^j; L)} \tag{5}$$

Scheme Comparison. We plotted the resulting number of traces needed according to these formulas for different values of $\mathrm{MI}_u(K;L)$. For this, we made two simplifying assumptions: First, c is a constant depending on the key entropy and the desired success rate of the attack and we simplify by equating it to the key entropy alone. Second, we assume that the mutual information between leakage and different shares of a value are equal simplifying all denominators from $\prod_{j=0}^{d} \mathrm{MI}_u(K^j;L)$ to $\mathrm{MI}_u(K^0;L)^{d+1}$. Table 4 also showcases the highest MI value such that the adversary needs 10^6 attack traces. We also show values for a lower security order of $d=1$ there to also account for the case that the masking order of an actual implementation is reduced due to (micro-)architectural leakages of share combinations.

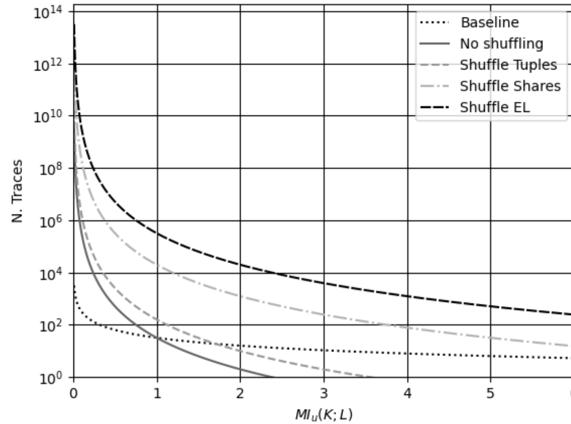

Fig. 2: Number of traces needed to mount a successful attack vs. the mutual information between masking shares and leakage. “Baseline” denotes an unmasked, unshuffled implementation while all others are 3rd-order PINI masked.

The plot is shown in Figure 2. The maximum value $\mathrm{MI}_u(K;L)$ can take is 32 bits, if the mutual information between the leakage and the key contains all information about the key. The baseline shown is an assumed implementation that is neither masked nor shuffled. A simple 3rd-order masked, unshuffled implementation already shows a significant improvement over the baseline, given sufficiently low values for mutual information. Shuffling Tuples yields a further, albeit not substantial, improvement over the unshuffled variant. Both Shuffling Shares and Shuffling Everything Light exhibit a significant jump in the number of traces needed, with these variants offering security levels between 10^4 and 10^6 traces for MI ≈ 1.

Table 4: Highest $\mathrm{MI}_u(K;L)$ value s.t. the adversary needs at least 1,000,000 attack traces. All implementations are 1st- or 3rd-order PINI masked.

	Unshuffled	Shuffle Tuples	Shuffle Shares	Shuffle EL
$d = 3$	0.0752	0.1125	0.376	0.7521
$d = 1$	0.0057	0.0126	0.0283	0.0566

6 Conclusion

Guided by the theoretical security evaluation we can conclude that all three shuffling schemes present notable increases in security compared to an implementation utilising only masking as a countermeasure. Overall, Shuffle Everything Light provides the best security gain in relation to performance cost, with a (theoretical) ten-fold increase in permissible mutual information at a roughly three-fold increase in clock cycles when compared to an only masked implementation. Shuffling Tuples proves to be a sound choice when there is not much spare performance available, with an almost 50% increase in security, again compared to an only masked implementation. while only increasing clock cycles by roughly 27%.

We also note that, while the schemes implemented are specific to the state structure of Ascon and the 32-bit RISC-V devices targetted, the general method of adapting countermeasures when the cipher state size is a multiple of 32 bits (and possibly interleaved) remains applicable.

Finally, we feel it is important to give a brief disclaimer: While the security estimates here are sound and the implementation should be secure from a theoretical standpoint, it is likely that through (micro-)architectural particularities such as overwrite- or memory remnant effects [26], the real security and masking order might be lower than the theoretical guarantees.

Future Work. While the theoretical analysis of the schemes devised in this work seems promising, an actual side-channel evaluation is an important next step to verify that the theoretical claims also hold up in practice, e.g. using countermeasure dissection [8]. Additionally, the exploration of possible schemes for combining masking and shuffling was far from exhaustive. It should certainly be possible to find schemes coming even closer to the "Shuffle Everything" scheme proposed in [2], thus further increasing the benefit to security. In the same vein, the existing approaches could be streamlined further, e.g. by unifying the computation of z_{ij} and z_{ji} for the PINI-AND gadgets into one shuffling block in the case of shuffling shares and shuffling everything light rather than using separate shuffling blocks for them. Finally, investigating avenues for reducing the randomness requirements for these schemes should prove useful to increase the feasibility of these implementations, as shown in [25].

Acknowledgements. This work was supported by an UvA starter grant.

References

1. Adomnicai, A., Fournier, J.J.A., Masson, L.: Masking the lightweight authenticated ciphers ACORN and Ascon in software. Cryptology ePrint Archive, Paper 2018/708 (2018), https://eprint.iacr.org/2018/708
2. Azouaoui, M., Bronchain, O., Grosso, V., Papagiannopoulos, K., Standaert, F.X.: Bitslice masking and improved shuffling: How and when to mix them in software? IACR Transactions on Cryptographic Hardware and Embedded Systems **2022**(2), 140–165 (Feb 2022). https://doi.org/10.46586/tches.v2022.i2.140-165, https://tches.iacr.org/index.php/TCHES/article/view/9484
3. Bellizia, D., Bronchain, O., Cassiers, G., Grosso, V., Guo, C., Momin, C., Pereira, O., Peters, T., Standaert, F.X.: Mode-level vs. implementation-level physical security in symmetric cryptography: A practical guide through the leakage-resistance jungle. CRYPTO 2020 (2020), https://eprint.iacr.org/2020/211
4. Bernstein, D.J.: Cache-timing attacks on AES (2005), https://api.semanticscholar.org/CorpusID:2217245
5. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G., Van Keer, R.: KECCAK implementation overview (2012), https://keccak.team/files/Keccak-implementation-3.2.pdf
6. Biham, E.: A fast new DES implementation in software. In: Fast Software Encryption. pp. 260–272. Springer Berlin Heidelberg, Berlin, Heidelberg (1997)
7. Bronchain, O., Hendrickx, J.M., Massart, C., Olshevsky, A., Standaert, F.X.: Leakage certification revisited: Bounding model errors in side-channel security evaluations. Cryptology ePrint Archive, Paper 2019/132 (2019), https://eprint.iacr.org/2019/132, https://eprint.iacr.org/2019/132
8. Bronchain, O., Standaert, F.: Side-channel countermeasures' dissection and the limits of closed source security evaluations. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2020**(2), 1–25 (2020). https://doi.org/10.13154/TCHES.V2020.I2.1-25, https://doi.org/10.13154/tches.v2020.i2.1-25
9. Cassiers, G., Standaert, F.X.: Trivially and efficiently composing masked gadgets with probe isolating non-interference. IEEE Transactions on Information Forensics and Security **15**, 2542–2555 (2020). https://doi.org/10.1109/TIFS.2020.2971153
10. Chari, S., Jutla, C.S., Rao, J.R., Rohatgi, P.: Towards sound approaches to counteract power-analysis attacks. In: Wiener, M. (ed.) Advances in Cryptology — CRYPTO' 99. pp. 398–412. Springer Berlin Heidelberg (1999)
11. de Chérisey, E., Guilley, S., Rioul, O., Piantanida, P.: Best information is most successful: Mutual information and success rate in side-channel analysis. IACR Transactions on Cryptographic Hardware and Embedded Systems **2019**(2), 49–79 (Feb 2019). https://doi.org/10.13154/tches.v2019.i2.49-79, https://tches.iacr.org/index.php/TCHES/article/view/7385
12. Coron, J.S.: Higher order masking of look-up tables. In: Nguyen, P.Q., Oswald, E. (eds.) Advances in Cryptology – EUROCRYPT 2014. pp. 441–458. Springer Berlin Heidelberg, Berlin, Heidelberg (2014). https://doi.org/10.1007/978-3-642-55220-5_25
13. Coron, J.S., Prouff, E., Rivain, M., Roche, T.: Higher-order side channel security and mask refreshing. Cryptology ePrint Archive, Paper 2015/359 (2015), https://eprint.iacr.org/2015/359
14. Dobraunig, C., Eichlseder, M., Mendel, F., Schläffer, M.: Ascon v1.2: Lightweight authenticated encryption and hashing. Journal of Cryptology **34**(3), 33 (Jun 2021). https://doi.org/10.1007/s00145-021-09398-9

15. Dobraunig, C., Eichlseder, M., Mendel, F., Schläffer, M.: Status update on Ascon v1.2 (2022), https://csrc.nist.gov/csrc/media/Projects/lightweight-cryptography/documents/finalist-round/status-updates/ascon-update.pdf
16. Duc, A., Faust, S., Standaert, F.X.: Making masking security proofs concrete or how to evaluate the security of any leaking device (extended version). Cryptology ePrint Archive, Paper 2015/119 (2015), https://eprint.iacr.org/2015/119, https://eprint.iacr.org/2015/119
17. Durstenfeld, R.: Algorithm 235: Random permutation. Commun. ACM **7**(7), 420 (jul 1964). https://doi.org/10.1145/364520.364540, https://doi.org/10.1145/364520.364540
18. Gao, S., Marshall, B., Page, D., Oswald, E.: Share-slicing: Friend or foe? IACR Transactions on Cryptographic Hardware and Embedded Systems **2020**(1), 152–174 (Nov 2019). https://doi.org/10.13154/tches.v2020.i1.152-174, https://tches.iacr.org/index.php/TCHES/article/view/8396
19. Gigerl, B., Mendel, F., Schläffer, M., Primas, R.: Efficient second-order masked software implementations of Ascon in theory and practice. Sixth NIST Lightweight Cryptography Workshop (2023), https://csrc.nist.gov/csrc/media/Events/2023/lightweight-cryptography-workshop-2023/documents/accepted-papers/04-efficient-second-order-masked-software.pdf
20. Herbst, C., Oswald, E., Mangard, S.: An AES smart card implementation resistant to power analysis attacks. In: Zhou, J., Yung, M., Bao, F. (eds.) Applied Cryptography and Network Security. pp. 239–252. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
21. Ishai, Y., Sahai, A., Wagner, D.: Private circuits: Securing hardware against probing attacks. In: Boneh, D. (ed.) Advances in Cryptology - CRYPTO 2003. pp. 463–481. Springer Berlin Heidelberg (2003). https://doi.org/10.1007/978-3-540-45146-4_27
22. Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) Advances in Cryptology — CRYPTO' 99. pp. 388–397. Springer Berlin Heidelberg (1999). https://doi.org/10.1007/3-540-48405-1_25
23. Mangard, S., Oswald, E., Popp, T.: Power Analysis Attacks. Springer New York (2007). https://doi.org/10.1007/978-0-387-38162-6
24. Mohajerani, K., Beckwith, L., Abdulgadir, A., Ferrufino, E., Kaps, J.P., Gaj, K.: SCA evaluation and benchmarking of finalists in the NIST lightweight cryptography standardization process. Cryptology ePrint Archive, Paper 2023/484 (2023), https://eprint.iacr.org/2023/484
25. Papagiannopoulos, K.: Low randomness masking and shuffling: An evaluation using mutual information. IACR Transactions on Cryptographic Hardware and Embedded Systems **2018**(3), 524–546 (Aug 2018). https://doi.org/10.13154/tches.v2018.i3.524-546, https://tches.iacr.org/index.php/TCHES/article/view/7285
26. Papagiannopoulos, K., Veshchikov, N.: Mind the gap: Towards secure 1st-order masking in software. Cryptology ePrint Archive, Paper 2017/345 (2017), https://eprint.iacr.org/2017/345
27. Prasad, S.H., Mendel, F., Schläffer, M., Nagpal, R.: Efficient low-latency masking of Ascon without fresh randomness. Cryptology ePrint Archive, Paper 2023/1914 (2023), https://eprint.iacr.org/2023/1914
28. Rivain, M., Prouff, E.: Provably secure higher-order masking of AES. Cryptology ePrint Archive, Paper 2010/441 (2010), https://eprint.iacr.org/2010/441, https://eprint.iacr.org/2010/441

29. Rivain, M., Prouff, E., Doget, J.: Higher-order masking and shuffling for software implementations of block ciphers. Cryptology ePrint Archive, Paper 2009/420 (2009), https://eprint.iacr.org/2009/420, https://eprint.iacr.org/2009/420
30. Standaert, F.X., Malkin, T.G., Yung, M.: A unified framework for the analysis of side-channel key recovery attacks (extended version). Cryptology ePrint Archive, Paper 2006/139 (2006), https://eprint.iacr.org/2006/139, https://eprint.iacr.org/2006/139
31. Sönmez Turan, M., McKay, K., Chang, D., Bassham, L.E., Kang, J., Waller, N.D., Kelsey, J.M., Hong, D.: Status report on the final round of the NIST lightweight cryptography standardization process. Nist interagency or internal report (ir) 8454, National Institute of Standards and Technology (2023). https://doi.org/10.6028/NIST.IR.8454
32. Veyrat-Charvillon, N., Medwed, M., Kerckhof, S., Standaert, F.X.: Shuffling against side-channel attacks: A comprehensive study with cautionary note. In: ASIACRYPT. vol. 7658, pp. 740–757. Springer (2012). https://doi.org/10.1007/978-3-642-34961-4_44, https://www.iacr.org/archive/asiacrypt2012/76580728/76580728.pdf
33. Weissbart, L., Picek, S.: Lightweight but not easy: Side-channel analysis of the Ascon authenticated cipher on a 32-bit microcontroller. Cryptology ePrint Archive, Paper 2023/1598 (2023), https://eprint.iacr.org/2023/1598

A Implementation Paths Not Taken

In light of the optimisation presented in section 4.1, we maintain that it is worthwhile to briefly discuss other means of implementing and optimising the encryption that we decided against using.

As discussed in Section 2.1, we decided against implementing the substitution layer as a masked lookup table [12]. While technically offering a performance improvement at runtime, the initial computation of these masked LUTs and the large memory requirement they impose for a four-share masked implementation make them infeasible for several resource-constrained devices.

An option for implementing the linear layer, and in particular circumventing the restriction of having no rotation instructions in RV32IM would have been to bitslice multiple encryption blocks by having one register for every bit of each state register and to then implement the rotations through "register renaming". Concretely, we could take 32 independent encryption blocks and bitslice them so that all first bits of each block go into one register, all second bits into the next, and so forth. In this case, the rotations of the linear layer would reduce to "renaming" e.g. the "register containing all first bits" would be renamed to the "register containing all second bits", in the case of a rotation by one. Apart from requiring a significant number of independent encryptions for efficient bitslicing (i.e. a bulk-encryption usecase), this approach is also infeasible due to the sheer number of registers needed. Since the Ascon state registers are 64 bits wide, we would need $5*64 = 320$ registers for storing all bits. The amount of loading and storing needed to realise this would very likely negate any performance gained from the free rotations.

Lastly, we could have approached the masking of the state differently in that we put multiple shares into one register, an approach also known as share-slicing. The benefit of this kind of slicing is that one can easily implement various operations on shares by rotating some of the share-sliced registers. This approach is especially appealing in architectures such as ARM, where certain instructions allow one of the operands to be rotated before the operation, effectively for free. Since such a construction does not exist for RV32IM, we cannot utilise this to our advantage. Furthermore, recent works have also discovered that share slicing also poses additional risks in a side-channel mitigation context [18].

B Additional figures

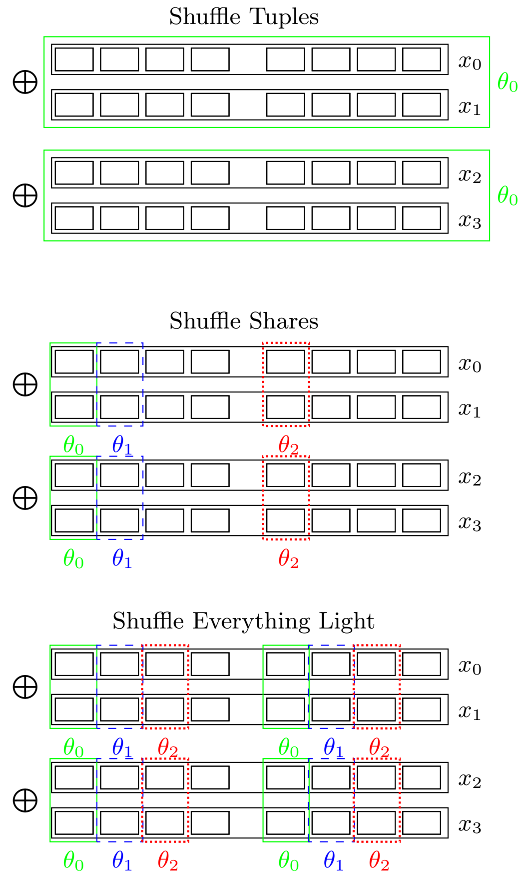

Fig. 3: A visualisation of the three different shuffling approaches selected for this implementation. Shown are two XOR operations on two 64-bit state registers, respectively split into two third-order masked 32-bit registers. Each θ denotes one set of operations to shuffle.

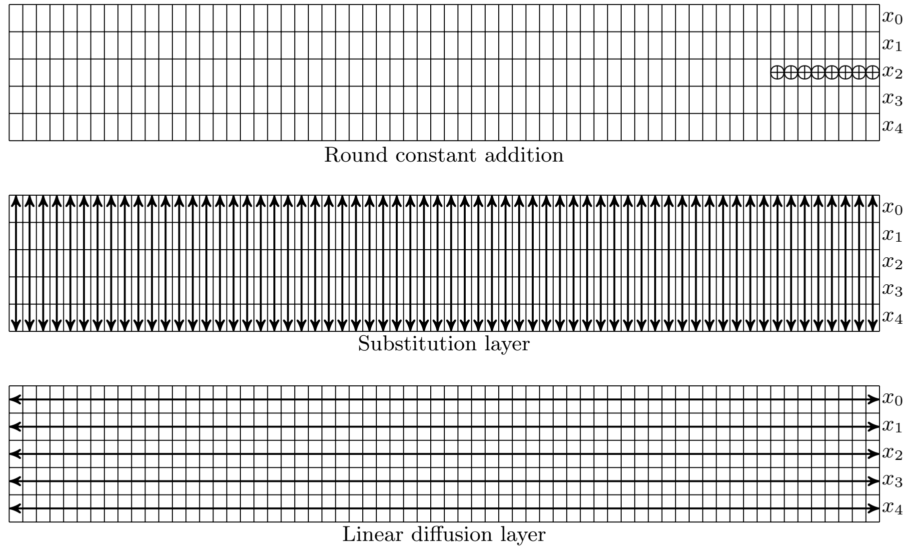

Fig. 4: The three steps performed on the state during each round of the Ascon permutation. Each rectangle represents one bit of the state.

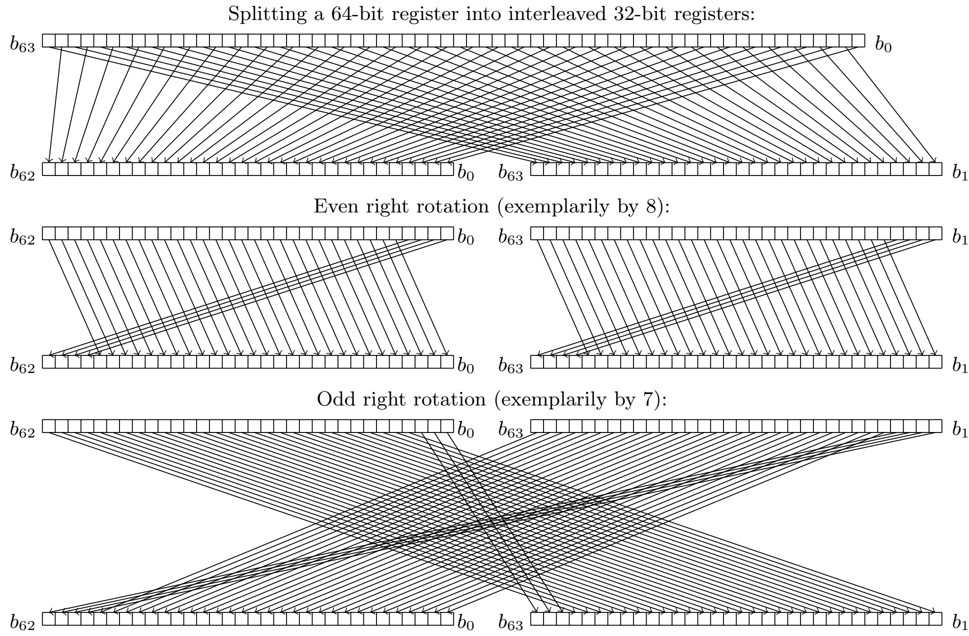

Fig. 5: A visualisation of interleaving and interleaved rotations.

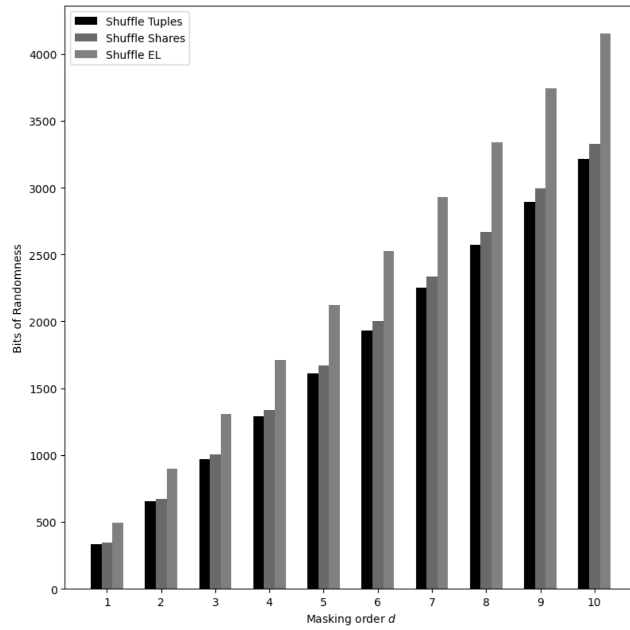

Fig. 6: Bits of randomness needed for linear operations per scheme depending on the chosen masking order d. Based on the formulas listed in Table 1 for $n = 5$ in the case of shuffle tuples and shuffle shares, and $n = 10$ for shuffle EL.

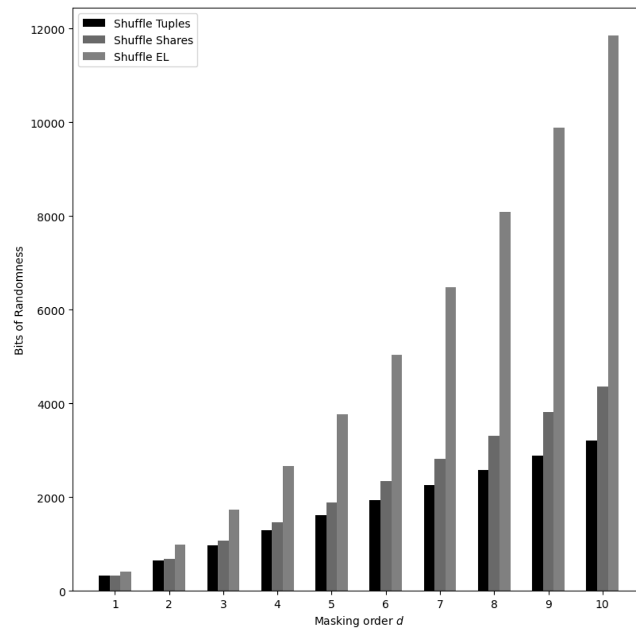

Fig. 7: Bits of randomness needed for non-linear operations per scheme depending on the chosen masking order d. Based on the formulas listed in Table 1 for $n = 5$ in the case of shuffle tuples and shuffle shares, and $n = 10$ for shuffle EL.

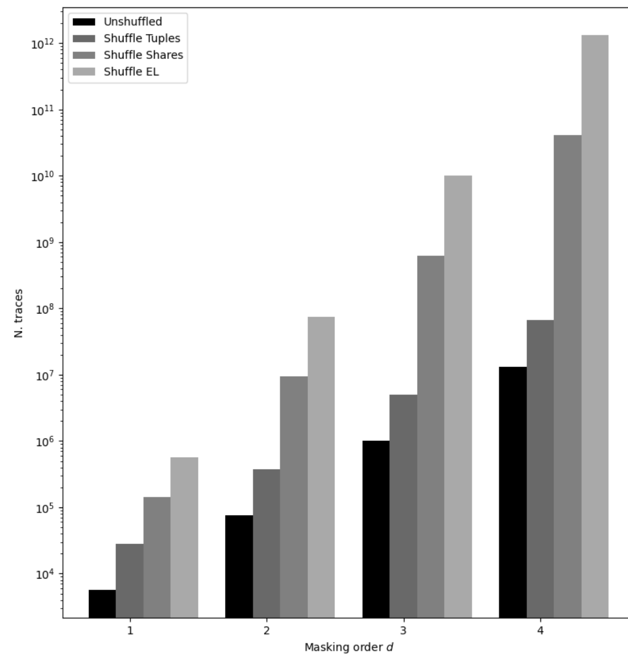

Fig. 8: The amount of traces needed to mount a successful attack per scheme and masking order. The $\mathrm{MI}(K; L)$ is fixed to 0.0752, the amount where a third-order masked but unshuffled implementation would require 10^6 traces to be broken.