IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2023, No. 4, pp. 344–366. DOI:10.46586/tches.v2023.i4.344-366

Low Trace-Count Template Attacks on 32-bit Implementations of ASCON AEAD

Shih-Chun You[1], Markus G. Kuhn[1], Sumanta Sarkar[2] and Feng Hao[2]

[1] University of Cambridge, Cambridge, UK, {scy27,mgk25}@cl.cam.ac.uk
[2] University of Warwick, Coventry, UK, {firstname.lastname}@warwick.ac.uk

Abstract. The recently adopted Ascon standard by NIST offers a lightweight authenticated encryption algorithm for use in resource-constrained cryptographic devices. To help assess side-channel attack risks of Ascon implementations, we present the first template attack based on analyzing power traces, recorded from an STM32F303 microcontroller board running Weatherley's 32-bit implementations of Ascon-128. Our analysis combines a fragment template attack with belief-propagation and key-enumeration techniques. The main results are three-fold: (1) we reached 100% success rate from a single trace if the C compiler optimized the unmasked implementation for space, (2) the success rate was about 95% after three traces if the compiler optimized instead for time, and (3) we also attacked a masked version, where the success rate was over 90% with 20 traces of executions with the same key, all after enumerating up to 2^{24} key candidates. These results show that suitably-designed template attacks can pose a real threat to Ascon implementations, even if protected by first-order masking, but we also learnt how some differences in programming style, and even compiler optimization settings, can significantly affect the result.

Keywords: ASCON · power analysis · template attack · SASCA

1 Introduction

Ascon [DEMS21], a family of algorithms for authenticated encryption with associated data (AEAD) and secure hashing, designed for resource-constrained devices, was in 2019 selected as a prime choice for lightweight applications in the CAESAR competition. In 2023, after multiple review rounds over five years, the National Institute of Standards and Technology (NIST) finally chose Ascon as the winner of NIST's lightweight cryptography standardisation process [LWC]. One may expect that Ascon could soon be implemented on millions of authentication chips, RFID tags and radio-controlled devices.

As Ascon becomes a new NIST standard, it is important to understand not only its theoretical properties but also potential implementation challenges, such as side-channel attacks (SCA). The designers of Ascon have already carefully considered side-channel protection. For example, Ascon does not use conditional branches or require any look-up table, which naturally prevents many timing attacks [DEMS21]. Furthermore, Ascon's permutation uses S-boxes of degree 2, which facilitates threshold implementations and masking as efficient countermeasures against some side-channel attacks [SD17]. And Ascon's mode of operation supports a so-called *leveled* implementation [BBC+20, VCS23], where counter-measures against Differential Power Analysis (DPA) are only required for its initial and final steps, and not for the processing of each message block, which reduces their performance impact.

Compared to the extensive published cryptanalysis work on Ascon, practical experiments on side-channel leakage have received less attention so far. Samwel and Daemen [SD17] presented Correlation Power Analysis (CPA) attacks and a threshold

Received: 2023-04-15 Accepted: 2023-06-15 Published: 2023-08-31

implementation of a toy-sized 20-bit version of Ascon. Gross et al. [GWDE17] showed that Ascon could resist first-order DPA based on simulated leakage. Abdulgadir et al. [ADK19] presented threshold implementations of Ascon on an Artix7 FPGA and demonstrated that they were effective in preventing DPA attacks. Diehl et al. [DAF+18] compared the cost of threshold implementations of Ascon and other selected authenticated ciphers against DPA. Recently, Luo et al. [LWL+22] presented a Soft Analytical Side-Channel Attack (SASCA) on Ascon based on simulation, using Hamming weights (HW) of 8-bit values with independent, identically distributed, added Gaussian noise as a leakage model. While attacks using simulated HWs can provide useful insights, real power traces can provide more information in the form of likelihoods for specific values. But real traces also require additional processing, such as interesting-point selection and dimensionality reduction, to deal with the much larger amount of leakage data, the longer word size, and potentially correlated switching noise from a real processing pipeline.

In this paper, we present template attacks using power traces from a 32-bit microcontroller STM32F303, with ARM Cortex-M4 core, running several Ascon-128 AEAD implementations. Our attack strategy combines three techniques: firstly, we use a *fragment template attack* [YK22], which uses a form of Linear Discriminant Analysis (LDA) [SA08, CK14a] modified for observing larger register sizes (Sec. 2.2), to obtain side-channel information leaked from this 32-bit device. Secondly, we use *belief propagation* (SASCA) [VCGS14] (Sec. 2.3) to improve the likelihood tables obtained from the template matches, by considering algebraic dependencies between the intermediate values observed. Finally we use *key enumeration* [VCGRS12] (Sec. 2.4) as an optimized brute-force search technique, to deal with residual errors. Combined, these measures optimize our analysis for the design of Ascon and the targeted 32-bit implementations.

We attack both unmasked (Sec. 3 and 4) and masked implementations (Sec. 6), and also consider the effect of two different compiler optimization settings on the success rate of key recovery (Sec. 5). We achieved fast key-recovery success rates of over 95% with fewer than 10, in some cases even single, power traces (Figure 9) based on the unmasked implementation [Wea21, `ASCON/`]. When it comes to the attack on the masked implementation [Wea21, `ASCON_masked/`], single traces did not help much, however, we succeeded in fast key recovery with 10 to 20 traces (Figure 13).

2 Preliminaries

2.1 Ascon

Ascon AEAD is based on a sponge mode, similar to MonkeyDuplex [BDPA12], but with stronger keyed initialization and finalization phases. The underlying permutations, denoted p^a and p^b, are obtained by iterating a 320-bit round function p for a or b times, respectively. Ascon AEAD first takes as inputs an initial vector IV (to identify the algorithm), a key K, and a nonce N, which it then combines with permutation p^a applied as a non-invertible key derivation function (KDF). It then invokes permutation p^b for blocks of associated data A and plaintext P, to absorb their content and generate the key stream for producing the blocks of ciphertext C. A final invocation of p^a serves as a tag generating function (TGF) to produce a message-authentication tag T. The encryption process of Ascon is illustrated in the following figure:

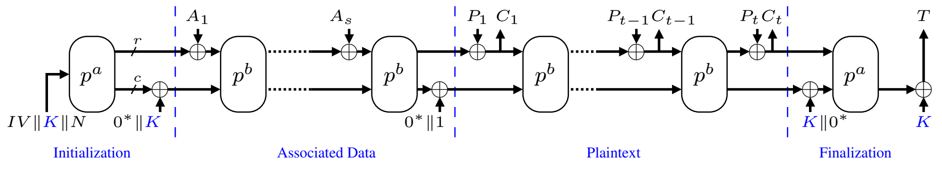


Table 1: Recommended parameters for ASCON

Cipher Variants	Bit size of						Rounds	
	State (S)	Rate (S_r)	Capacity (S_c)	Key (K)	Nonce (N)	Tag (T)	a	b
ASCON-128	320	64	256	128	128	128	12	6
ASCON-128a	320	128	192	128	128	128	12	8

During the execution of the AEAD mode, the output of the permutation is divided into two parts called *rate* (S_r) and *capacity* (S_c). The size of the rate is equal to the maximum number of data bits that an invocation of permutation will process. The two ASCON variants ASCON-128 and ASCON-128a differ according to their rate, capacity and number of rounds a and b (see Table 1). In this paper, we focus on ASCON-128 since it is the primary recommendation by the ASCON designers [DEMS21].

2.1.1 Ascon permutation

The permutation operates on a 320-bit state (S) that is divided into five 64-bit words (or five *lanes*), as in $S = L_0 \| L_1 \| L_2 \| L_3 \| L_4$.

The round function p follows the substitution and permutation network (SPN) design principle, and consists of three operations: constant addition p_{C}, substitution p_{S} and linear diffusion p_{L}, as $p = p_{\mathrm{L}} \circ p_{\mathrm{S}} \circ p_{\mathrm{C}}$.

Constant Addition: The operation p_{C} updates the state by XORing an 8-bit round constant to the least significant byte of L_2, where the round constant in each round is

Constant:		`0xf0`	`0xe1`	`0xd2`	`0xc3`	`0xb4`	`0xa5`	`0x96`	`0x87`	`0x78`	`0x69`	`0x5a`	`0x4b`
	p^{12}	0	1	2	3	4	5	6	7	8	9	10	11
round of	p^{8}	–	–	–	–	0	1	2	3	4	5	6	7
	p^{6}	–	–	–	–	–	–	0	1	2	3	4	5

Substitution operation: Step p_{S} is a nonlinear operation applying a 5-bit S-box, which operates on $(L_0[k], L_1[k], L_2[k], L_3[k], L_4[k])$ in parallel for bits $0 \le k \le 63$, as in

$$\begin{aligned}
&L_0 \leftarrow L_0 \oplus L_4, \quad L_4 \leftarrow L_4 \oplus L_3, \quad L_2 \leftarrow L_2 \oplus L_1 \\
&L_0' \leftarrow L_0 \oplus ((\neg L_1) \wedge L_2) \\
&L_1' \leftarrow L_1 \oplus ((\neg L_2) \wedge L_3) \\
&L_2' \leftarrow L_2 \oplus ((\neg L_3) \wedge L_4) \\
&L_3' \leftarrow L_3 \oplus ((\neg L_4) \wedge L_0) \\
&L_4' \leftarrow L_4 \oplus ((\neg L_0) \wedge L_1) \\
&L_1' \leftarrow L_1' \oplus L_0', \quad L_0' \leftarrow L_0' \oplus L_4', \quad L_3' \leftarrow L_3' \oplus L_2', \quad L_2' \leftarrow \neg L_2'
\end{aligned} \tag{1}$$

This S-box has the following cryptographic properties: maximum differential and linear probability $\frac{1}{4}$, differential and linear branch number 3 and algebraic degree 2. This substitution operation allows an efficient bit-sliced implementation and its low degree also facilitates efficient threshold implementations and masking.

Linear Diffusion operation: Step p_{L} provides diffusion by applying five different linear operations Σ_i for $0 \le i \le 4$, and each Σ_i performs XOR ($\oplus$) and right rotations ($\ggg$) on the word L_i as in

$$\begin{aligned}
L_0' &\leftarrow \Sigma_0(L_0) = L_0 \oplus (L_0 \ggg 19) \oplus (L_0 \ggg 28) \\
L_1' &\leftarrow \Sigma_1(L_1) = L_1 \oplus (L_1 \ggg 61) \oplus (L_1 \ggg 39) \\
L_2' &\leftarrow \Sigma_2(L_2) = L_2 \oplus (L_2 \ggg 1) \oplus (L_2 \ggg 6) \\
L_3' &\leftarrow \Sigma_3(L_3) = L_3 \oplus (L_3 \ggg 10) \oplus (L_3 \ggg 17) \\
L_4' &\leftarrow \Sigma_4(L_4) = L_4 \oplus (L_4 \ggg 7) \oplus (L_4 \ggg 41)
\end{aligned} \tag{2}$$

In later sections, we refer to the internal states of the Ascon p^{12} permutation as

$$\textbf{Input} = \beta_{-1} \xrightarrow{p_C, p_S} \alpha_0 \xrightarrow{p_L} \beta_0 \xrightarrow{p_C, p_S} \alpha_1 \xrightarrow{p_L} \beta_1 \xrightarrow{p_C, p_S} \cdots \xrightarrow{p_L} \beta_{11} = \textbf{Output}.$$

We will use symbols L_0, L_1, L_2, L_3, L_4 to represent the five lanes in intermediate states α_Ω or β_Ω, where Ω represents the round index in the p^{12} permutation.

2.2 Template attack

The Template Attack (TA), introduced by Chari et al. [CRR03], is a powerful profiled side-channel exploitation technique. The attacker first profiles the target device, while operating it in a training mode where they know the data being processed. Traces are recorded at this stage to build Gaussian multivariate trace templates that model the leakage of each of the different known values being processed. Then, during the attack stage, the attacker records an attack trace while an unknown secret is being processed, and compares that trace against each template. The unknown secret is obtained based on the candidate template that is most similar to the attack trace.

Several variants of the template attack have been proposed to improve the efficiency and accuracy of the profiling procedure. Schindler et al. [SLP05] introduced their $\mathcal{F}_9$ "stochastic model", where each bit in a targeted byte is treated as an independent variable of a multivariate linear-regression model, to predict the expected values of single points on a trace. Standaert and Archambeau [SA08] proposed using Fisher's Linear Discriminant Analysis (LDA) in template attacks for dimensionality reduction. Choudary and Kuhn [CK14a] combined both techniques to provide a predicted probability for each possible value for a target byte, instead of only the probability for the HW of possible values. Such LDA-based dimensionality reduction has several benefits: the projection can increase the signal-to-noise ratio, the reduced dimensionality leads to better covariance estimates, and LDA-based templates have shown better portability across different devices [CK14b].

Template attack with LDA In this template building approach, we first group the profiling traces according to the targeted byte value $b \in \{0, \ldots, 255\}$. A profiling trace observing target value b is denoted as $\mathbf{x}_{b,t}$, where $t \in \{1, \ldots, n_b\}$ enumerates the traces in group b. With the $\mathcal{F}_9$ stochastic model, we treat each member bit ($b[0]$ to $b[7]$) as an independent variable in a multivariate linear regression to calculate coefficients c_0 to $c_7 \in \mathbb{R}$ and a constant $c_8 \in \mathbb{R}$ to predict the expected values of samples as $\bar{x}_b = \sum_{\ell=0}^{7}(b[\ell] \cdot c_\ell) + c_8$ for each possible value b. To represent the expected vector of an entire m-sample trace, we write $\bar{\mathbf{x}}_b = \sum_{\ell=0}^{7}(b[\ell] \cdot \mathbf{c}_\ell) + \mathbf{c}_8$, where $\mathbf{c}_0, \ldots, \mathbf{c}_8 \in \mathbb{R}^m$.

Next, we perform LDA dimensionality reduction to project longer traces to shorter vectors. To prepare this step, we first build two covariance matrices, $\mathbf{B}$ and $\mathbf{\Sigma}$, which represent the signal and noise, respectively, as

$$\mathbf{B} = \frac{1}{\sum_b n_b} \sum_b n_b (\bar{\mathbf{x}}_b - \bar{\mathbf{x}})(\bar{\mathbf{x}}_b - \bar{\mathbf{x}})^\mathsf{T}, \quad \mathbf{\Sigma} = \frac{1}{\sum_b n_b} \sum_b \sum_{t=1}^{n_b} (\mathbf{x}_{b,t} - \bar{\mathbf{x}}_b)(\mathbf{x}_{b,t} - \bar{\mathbf{x}}_b)^\mathsf{T},$$

where $\bar{\mathbf{x}}$ denotes the average of all 256 vectors $\bar{\mathbf{x}}_b$. We then project all the m-sample traces, including profiling traces $\mathbf{x}_{b,t}$, expected traces $\bar{\mathbf{x}}_b$, and the attack trace $\mathbf{x}_\mathrm{a}$, onto the m'

$(m' \ll m)$ eigenvectors with the largest eigenvalues of $\mathbf{\Sigma}^{-1}\mathbf{B}$, to obtain m'-sample traces $\mathbf{x}_{b,t,\text{proj}}, \bar{\mathbf{x}}_{b,\text{proj}}, \mathbf{x}_{\text{a},\text{proj}} \in \mathbb{R}^{m'}$.

In this new subspace, where the signal-to-noise ratio is larger, we can now build a pooled covariance matrix [CK14a]

$$\mathbf{S} = \frac{1}{\sum_b n_b} \sum_b \sum_{t=1}^{n_b} (\mathbf{x}_{b,t,\text{proj}} - \bar{\mathbf{x}}_{b,\text{proj}})(\mathbf{x}_{b,t,\text{proj}} - \bar{\mathbf{x}}_{b,\text{proj}})^\mathsf{T},$$

such that the probability density of the attack trace $\mathbf{x}_{\text{a},\text{proj}}$ can be modelled as

$$L(\mathbf{x}_{\text{a},\text{proj}}|\bar{\mathbf{x}}_{b,\text{proj}}, \mathbf{S}) = \frac{1}{\sqrt{(2\pi)^{m'}|\mathbf{S}|}} \exp\left(-\frac{1}{2}(\mathbf{x}_{\text{a},\text{proj}} - \bar{\mathbf{x}}_{b,\text{proj}})^\mathsf{T}\mathbf{S}^{-1}(\mathbf{x}_{\text{a},\text{proj}} - \bar{\mathbf{x}}_{b,\text{proj}})\right).$$

Having this likelihood calculated for all 256 possible values of b, we can sort them in descending order to generate a *ranking table* of all candidates; alternatively, we can normalize these likelihoods to build a *probability table.*

Fragment template attack on 32-bit devices While we can directly enumerate all 8-bit values in a target byte this way, many embedded target devices today use 32-bit buses. Recently, Cassiers et al. demonstrated how to efficiently build LDA-based templates that directly target a 32-bit word as a whole [CDSU23]. However, as belief propagation on probability tables with 2^{32} values is very time consuming, we followed You and Kuhn's earlier *fragment template attack* [YK22] to separate the 32-bit value into a few fragments (e.g., four bytes or two 16-bit fragments) and build a template targeting one fragment at a time, using a different LDA projection for each fragment to treat signals from the other fragments as noise.

For example, we can split a target value $v \in \mathbb{Z}_{2^{32}}$ into four byte fragments $v \mapsto (F_0(v), \ldots, F_3(v))$ with $F_f(v) = \sum_{\ell=0}^{7} v[8f+\ell] \cdot 2^\ell$. Let $V_{f,b} = \{v \mid F_f(v) = b\}$ be the set of all 32-bit values where fragment number f has value b. For each f, we apply the $\mathcal{F}_9$ stochastic model to obtain the 256 expected trace vectors $\bar{\mathbf{x}}_{f,b} = \sum_{\ell=0}^{7} b[\ell] \cdot \mathbf{c}_{f,\ell} + \mathbf{c}_{f,8}$, from the traces $\mathbf{x}_{v,t}$ with $v \in V_{f,b}$, respectively. For the LDA procedure, we calculate $\mathbf{B}$ and $\mathbf{\Sigma}$ separately for each fragment:

$$\mathbf{B}_f = \sum_{b=0}^{255} \sum_{v \in V_{f,b}} n_v (\bar{\mathbf{x}}_{f,b} - \bar{\mathbf{x}})(\bar{\mathbf{x}}_{f,b} - \bar{\mathbf{x}})^\mathsf{T} \Bigg/ \sum_{b=0}^{255} \sum_{v \in V_{f,b}} n_v,$$

$$\mathbf{\Sigma}_f = \sum_{b=0}^{255} \sum_{v \in V_{f,b}} \sum_{t=1}^{n_v} (\mathbf{x}_{v,t} - \bar{\mathbf{x}}_{f,b})(\mathbf{x}_{v,t} - \bar{\mathbf{x}}_{f,b})^\mathsf{T} \Bigg/ \sum_{b=0}^{255} \sum_{v \in V_{f,b}} n_v,$$

where n_v represents the number of traces in group v. $\mathbf{B}_f$ only contains signals from fragment number f, and signals from the other three bytes no longer count here, but instead contribute to $\mathbf{\Sigma}_f$. In other words, they are considered to be switching noise in this model.

After projecting the profiling traces and attack traces to the m'-dimensional subspace (e.g., we used $m' = 8$ for byte and $m' = 16$ for 16-bit fragments) via these two matrices, we can (as before) calculate the pooled covariance matrix and combine it with the projected expected traces into the template for this byte fragment of the target word.

Due to the noise sensitivity of the template attack, the correct candidate will not always top the ranking table. We can use two additional steps to make the attack more resilient to noise: 1) *belief propagation*, which takes algorithmic dependencies between the targeted values into account, and 2) *key enumeration*, which tests more combinations of target values than just the top-ranked ones. We elaborate these techniques in the following sections.

2.3 Belief propagation and SASCA

Veyrat-Charvillon et al. [VCGS14] introduced Soft Analytical Side-Channel Analysis (SASCA), an inference technique for template attacks on cryptographic algorithms based on the belief-propagation algorithm [Mac03, Chapter 26]. The idea behind SASCA is that all the probability information available to the attacker is represented as a *factor graph*, where there are two types of nodes: 1) *variables*, which represent the intermediate states of the cryptographic algorithm, and 2) *factors*, which represent how these intermediate states depend on each other and the observed traces. Information can flow bidirectionally through edges that connect variables with factors. We choose a factor graph to reflect the mathematical relations between the target parts in the cryptographic algorithm.

The variable nodes represent the intermediate values in the cryptographic algorithm. You and Kuhn distinguish two types of factor nodes [YK22]. *Observation factors* $f_m(x_n)$ represent observed probabilities of the values of their only connected variable x_n, here usually from a template-based likelihood. *Constraint factors* $f_m(\mathbf{x}_m)$ are connected to more than one variable $(x_{n_1}, \ldots, x_{n_{k_m}}) = \mathbf{x}_m$ (where $\mathcal{N}(m) = \{n_1, \ldots, n_{k_m}\}$ denotes the set of indices of these variables) with a mathematical equation as the constraint (see example in Figure 1).

The information flow can be thought of as messages passed between variable nodes x_n and factor nodes f_m, which in practice are stored in a table, and from which the marginal probabilities of all the candidate values of each variable can be calculated. A message from a variable x_n to a factor f_m is denoted as $q_{n\to m}$, and a message from a factor f_m to a variable x_n as $r_{m\to n}$. Each of these messages is a function of a value ξ of x_n. The probability of a candidate $x_n = \xi$ in message $q_{n\to m}$ is

$$q_{n\to m}(x_n = \xi) = \prod_{m' \neq m} r_{m'\to n}(x_n = \xi).$$

Meanwhile, the probability of a candidate $x_n = \xi$ in the message $r_{m\to n}$ is

$$r_{m\to n}(x_n = \xi) = \sum_{\mathbf{w}} \left[f_m(x_n = \xi, \mathbf{x}_m \backslash x_n = \mathbf{w}) \prod_{n' \in \mathcal{N}(m)\backslash n} q_{n'\to m}(x_{n'} = w_{n'}) \right],$$

where

$$f_m(\mathbf{x}_m = \mathbf{v}) = \begin{cases} 1, & \text{constraint holds with } \mathbf{x}_m = \mathbf{v}, \\ 0, & \text{otherwise.} \end{cases}$$

For the special case of an observation factor, this reduces to

$$r_{m\to n}(x_n = \xi) = f_m(x_n = \xi),$$

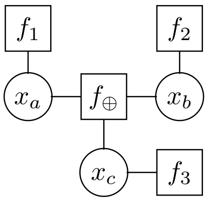
where $f_m(x_n)$ is the probability table observed from the templates, instead of a constraint function. Finally, we obtain the probability P_n of candidates $x_n = \xi$ by normalizing

$$P_n(x_n = \xi) = \frac{Z_n(x_n = \xi)}{\sum_{\xi'} Z_n(x_n = \xi')}, \quad \text{where} \quad Z_n(x_n = \xi) = \prod_m r_{m\to n}(x_n = \xi)$$

is the product of the probabilities of ξ in all the messages r passed to the same variable x_n.

If the graph is a tree structure, all the r and q probabilities can be calculated by traversing the tree recursively, visiting each edge once. However, factor graphs for cryptographic algorithms often feature loops, where this recursion would not terminate.

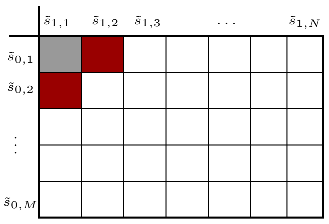
MacKay [Mac03, Chapter 26] describes a solution called loopy belief propagation (loopy BP). The main idea is to initialize all the values in the table for all messages q with one, then alternatingly update all the messages in the table for r and then q, with renormalization to prevent the probability values from becoming too small. Then the procedure terminates when it reaches a steady state. We count one update of all r followed by one update of all q as one *iteration*.



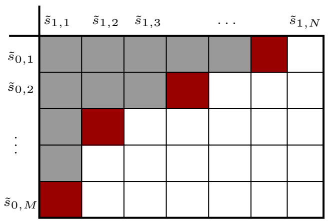
$$f_1(x_a) = \begin{cases} 0.8, & x_a = 0 \\ 0.2, & x_a = 1, \end{cases} \qquad f_2(x_b) = \begin{cases} 0.7, & x_b = 0 \\ 0.3, & x_b = 1, \end{cases}$$

$$f_\oplus(x_a, x_b, x_c) = \begin{cases} 1, & \text{if } x_a \oplus x_b = x_c \\ 0, & \text{otherwise,} \end{cases} \qquad f_3(x_c) = \begin{cases} 0.9, & x_c = 0 \\ 0.1, & x_c = 1. \end{cases}$$

Figure 1: An example of the factor graph for the operation $x_c = x_a \oplus x_b$. Circular nodes represent variables and square nodes represent factors. The three observation factors f_1, f_2, f_3 represent the template likelihoods and one constraint factor $f_\oplus$ represents the operation.



(a) The most likely combination must be the top-left element when a search begins.



(b) The frontier $\mathcal{F}$, blocks labeled in red, when the gray blocks have been enumerated.

Figure 2: The key enumeration array

2.4 Key enumeration

While the belief-propagation step reduces the impact of noise on the ranking tables, the correct candidates may still not top the tables, and more candidates further down will have to be tested as well. Veyrat-Charvillon et al. [VCGRS12] introduced an optimal key enumeration algorithm to search the correct key across *independent* ranked likelihood tables.

Assume that there are two independent secret variables s_0 and s_1 with M and N possible values, respectively. Through some side-channel analysis, we have already obtained their ranked (sorted) probability tables. Here the m^{th} and n^{th} most likely candidates of these two variables are denoted as $\tilde{s}_{0,m}$ and $\tilde{s}_{1,n}$, respectively, and their corresponding probabilities are denoted as $p_{0,m}$ and $p_{1,n}$. Figure 2a shows an $M \times N$ *joint ranked probability table*, where each block represents the joint probability, $p_{0,m} \times p_{1,n}$, of a combination $(\tilde{s}_{0,m}, \tilde{s}_{1,n})$.

Since $p_{0,m}$ and $p_{1,n}$ are from sorted tables, we have the following partial order of the joint probabilities $p_{0,m} \times p_{1,n}$ of $(\tilde{s}_{0,m}, \tilde{s}_{1,n})$:

$$p_{0,m} \geqslant p_{0,m'} \wedge p_{1,n} \geqslant p_{1,n'} \quad \Longrightarrow \quad p_{0,m} \times p_{1,n} \geqslant p_{0,m'} \times p_{1,n'}, \qquad \forall m' \geqslant m \wedge \forall n' \geqslant n.$$

Therefore, the top-leftmost block, representing $(\tilde{s}_{0,1}, \tilde{s}_{1,1})$, is the combination with the largest joint probability, and the combination with the second largest joint probability will pertain to one of $(\tilde{s}_{0,2}, \tilde{s}_{1,1})$ or $(\tilde{s}_{0,1}, \tilde{s}_{1,2})$. They will both be added into a set called *frontier*, $\mathcal{F}$, which includes all possible candidates for the combination with the next largest joint probability. Therefore, we only need to compare values from this set to find the next value pair to be enumerated. In Figure 2b, once all the combinations marked in gray have been enumerated, the frontier $\mathcal{F}$, marked in red, will be the set of all the pairs at the concave corners. While a combination $(\tilde{s}_{0,m}, \tilde{s}_{1,n})$ is being enumerated, we need to update $\mathcal{F}$ by removing $(\tilde{s}_{0,m}, \tilde{s}_{1,n})$, and then considering whether $(\tilde{s}_{0,m+1}, \tilde{s}_{1,n})$ or $(\tilde{s}_{0,m}, \tilde{s}_{1,n+1})$ or both shall be added to $\mathcal{F}$, respectively, by checking if one occupies a concave corner in the already enumerated gray part of the array.

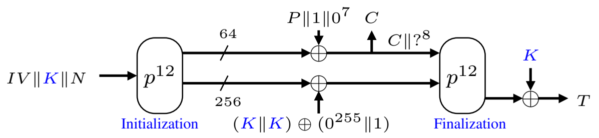


Figure 3: ASCON-128 with empty associated data A and seven-byte plaintext P.

Following this algorithm, the probability $p_{1,n}$ will only be queried after the combination $(\tilde{s}_{0,1}, \tilde{s}_{1,n-1})$ has been enumerated and then we need to add $(\tilde{s}_{0,1}, \tilde{s}_{1,n})$ into $\mathcal{F}$; similar operations apply to $p_{0,m}$. This means that we do not need all the values in the probability tables initially, and therefore we can build a tree of iterators recursively to search combinations of candidates from beyond two tables.

Key rank estimation In experiments aimed at evaluating the effectiveness of attacks, we know the correct key and merely need to determine its rank. Faster techniques, such as the histogram-based method by Glowacz et al. [GGP+15] can estimate the rank of a correct key in cases where enumeration would be too time consuming.

3 Building templates for Ascon AEAD

Our attack strategy consists of three main steps: fragment template attack, belief propagation, and key enumeration. In this section, we focus on the fragment template attack.

3.1 General experimental assumptions

We demonstrate a profiled fixed-length known-plaintext attack, only targeting the secret key K. In the profiling stage, our attacker can provide varying K, N, A, P, and can observe the corresponding C and T along with recorded power traces. In the attack stage, they can obtain values of N, A, P, C, T, and recorded power traces, to recover the secret key K. We demonstrate our attack by targeting ASCON-128. Note that while ASCON AEAD allows arbitrary-length associated data and plaintexts, in this attack demonstration, we used empty associated data and 7-byte plaintexts, to keep the traces aligned and minimize their length when covering the entire encryption process. In other words, we focus entirely on the two invocations of permutation p^{12} in the Initialization (KDF) and Finalization (TGF) phases, which process K. Figure 3 depicts this target encryption procedure.

3.2 Measurement setup

We targeted Weatherley's ASCON-128 implementations[1] [Wea21] with an optimized permutation for Cortex-M4, compiled with `arm-none-eabi-gcc` (v9.2.1) and uploaded into the 32-bit microcontroller STM32F303 on the SCA platform ChipWhisperer-Lite [OC14, CWL]. We recorded AC-coupled power traces with an NI PXIe-5160 10-bit oscilloscope with its highest sampling rate, 2.5 GHz, phase-locked to a function generator (NI PXIe-5423) supplying the target board with a 5 MHz square-wave clock signal. Therefore, there were exactly 500 points per clock cycle (500 PPC) on the raw traces.

Thanks to the lightweight structure of ASCON and our choice of short input size (Figure 3), it is practical to record power traces covering the full AEAD mode. Therefore, we built templates for target fragments of all the states $\alpha_0, \ldots, \alpha_{11}$ and $\beta_{-1}, \ldots, \beta_{11}$ of

[1]When we started these experiments in September 2021, they were the only open-source ARMv7m-optimized 32-bit implementations we could find, referred to by both the ASCON and NIST web sites.

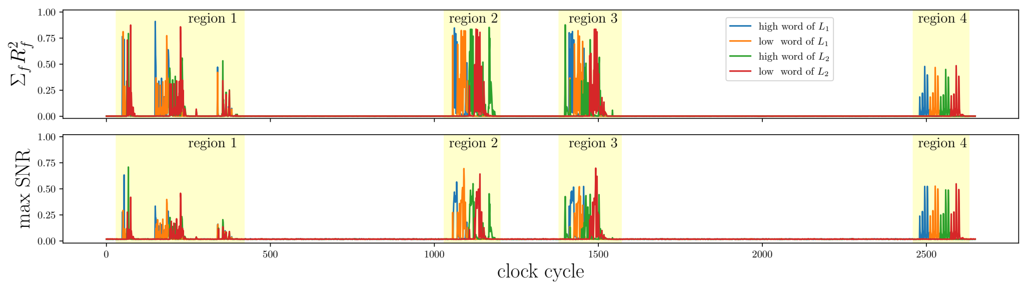


Figure 4: The top plot shows the $\Sigma_f R_f^2$ results for each 32-bit word of the 128-bit K for `U-Os`. The spikes lie in the marked regions corresponding to the four uses of K. The bottom plot shows the maximum of the SNR values among the four fragments.

permutation p^{12} in both Initialization and Finalization, except for fragments of known values: for IV, N, P, C, and T, we assign the probability of the actual fragment value to be 1 and all others 0. The two lanes L_1 and L_2 of the Initialization input contain the key fragments, which are our main targets.

For each experiment, we separated the recorded traces into the following categories, by purpose:

Category	#Traces	Purpose
Detection	16 000	interesting-clock-cycle detection
Profiling	64 000	template building
Validation	1 000	template-quality evaluation
Attack	10 000	key recovery

We recorded these number of traces for each of three experiments, which we refer to as `U-Os`, `U-O3`, and `M-Os`, respectively. The first two ran the unmasked implementation [Wea21, `ASCON/`], compiled either optimized for space (gcc option `-Os`) or for time (option `-O3`), whereas the third ran the masked implementation [Wea21, `ASCON_masked/`] optimized for space (option `-Os`).

We recorded 10 attack traces each from encryptions with the same key K, which we used for our experiments combining traces from multiple encryptions (Sec. 4.1). We varied nonce N and plaintext P randomly. For the `M-Os` experiment, we increased the number of recorded attack traces per key K to 100, i.e. there we recorded 100 000 attack traces in total. For the other categories, we varied the inputs K, N, and P randomly for each encryption recorded. In the rest of this section, we show data from the `U-Os` experiment.

3.3 Detecting interesting clock cycles

As the raw traces were very long, it is difficult to directly derive a distribution that describes the power traces for our target intermediate values. As not all the clock cycles are relevant to our target intermediate values, we can only consider the sample points of the clock cycles that are clearly correlated to these intermediate values. We refer to these as the *interesting clock cycles*.

We followed the method of finding interesting clock cycles for 32-bit key fragments from [YK22]. We divided all the intermediate states $(\beta_{-1}, \alpha_0, \ldots, \beta_{11})$ into 32-bit words and further divided these into four byte fragments (numbered 0, 1, 2 and 3), and then applied multivariate linear regression to model the correlation between the samples on power traces and each byte. After we built the regression model, we calculated its coefficient of determination $R^2 \in [0, 1]$, to see how well the samples fit the model. As we now have four different R^2 values, $R_0^2, R_1^2, R_2^2, R_3^2$, we applied their sum $\sum_f R_f^2$ to estimate the correlation between the model and the power traces for a 32-bit word. We selected those interesting

Table 2: Number of interesting clock cycles detected for 32-bit intermediate words in `U-Os`. The detection for IV and N is not needed since they are known values.

lane		L_0		L_1		L_2		L_3		L_4	
32-bit word		high	low	high	low	high	low	high	low	high	low
	input (β_{-1})	IV		244	230	310	252	N			
	α_0	32	27	111	128	43	36	34	24	89	90
	β_0	18	19	19	22	19	25	23	21	30	32
	α_1	17	17	17	13	33	31	23	24	27	23
	β_1	13	19	19	20	22	22	18	19	34	32
	α_2	17	19	12	13	29	30	29	19	25	21
	β_2	12	14	19	20	24	23	21	20	37	32
	α_3	18	19	12	15	29	27	24	24	22	22
	β_3	13	16	20	21	23	27	24	21	33	31
	α_4	18	39	16	18	33	29	23	23	24	27
	β_4	15	15	21	20	26	22	20	21	30	32
	α_5	17	17	15	14	33	27	20	21	50	28
Init.	β_5	12	14	21	18	21	21	20	20	34	33
	α_6	18	20	16	13	30	28	23	22	23	24
	β_6	17	21	23	23	22	21	19	18	35	31
	α_7	17	18	14	19	29	32	24	18	23	24
	β_7	17	14	18	22	25	27	21	23	38	34
	α_8	17	17	17	11	29	27	25	25	25	22
	β_8	13	15	21	22	22	23	24	21	35	30
	α_9	19	17	16	14	29	27	21	28	23	22
	β_9	13	28	22	21	26	23	20	21	34	32
	α_{10}	23	18	16	12	30	27	22	19	23	21
	β_{10}	15	17	26	22	23	26	22	21	33	33
	α_{11}	20	17	13	14	33	29	25	23	31	27
	β_{11}	30	101	28	36	67	62	26	24	48	46

lane		L_0		L_1		L_2		L_3		L_4	
32-bit word		high	low	high	low	high	low	high	low	high	low
	input (β_{-1})	102	133	40	41	45	46	40	41	76	80
	α_0	20	30	14	17	35	30	25	22	23	23
	β_0	18	14	21	20	25	25	25	20	31	32
	α_1	17	17	15	16	29	30	24	21	24	26
	β_1	12	13	20	20	21	22	20	21	34	31
	α_2	20	18	13	14	30	27	22	22	25	23
	β_2	21	15	21	23	23	23	18	29	32	32
	α_3	17	18	17	19	32	27	25	23	22	21
	β_3	16	12	24	19	22	21	22	19	31	33
	α_4	16	15	16	14	32	30	31	21	24	22
	β_4	15	17	22	19	20	25	28	21	33	30
	α_5	18	20	14	17	31	28	21	23	27	25
Fin.	β_5	19	17	22	21	23	22	20	19	35	34
	α_6	17	18	11	15	30	27	24	22	23	26
	β_6	15	15	20	20	26	22	23	22	32	35
	α_7	18	21	15	12	29	28	20	32	26	22
	β_7	18	16	22	34	22	21	26	20	36	33
	α_8	16	21	14	15	28	25	26	23	24	23
	β_8	17	15	24	22	23	22	21	22	34	35
	α_9	18	24	16	15	32	28	23	24	27	22
	β_9	15	13	30	21	21	25	19	19	37	33
	α_{10}	18	17	13	19	28	28	24	23	25	24
	β_{10}	11	15	21	20	18	23	19	21	33	34
	α_{11}	17	22	16	14	28	27	26	22	31	27
	β_{11}	65	63	62	65	63	65	98	99	116	124

clock cycles where $\sum_f R_f^2 > 0.004$.[2]

Our target [Wea21, `ASCON/`] implements a *bit-interleaved* [DEMS21, Sec. 4.1.1] version of Ascon, where a 64-bit lane is not just separated into a high and a low 32-bit word, but also sliced into its odd and even parts during the permutation, such that one 64-bit rotation becomes two 32-bit rotations. Therefore, data bits, especially the input and output, can be stored separated into both high and low words (H/L words), as well as sliced into even-bit and odd-bit words (E/O words). We decided to detect the interesting clock cycles for both the H/L and E/O words for a lane, and use their union set as the interesting clock sets for this lane, to consider both situations.

Table 2 shows the number of detected interesting clock cycles for each target 32-bit, H/L word of the intermediate states for the full AEAD process. We can see that there were more interesting clock cycles detected for those words closer to input or output (i.e., β_{-1} or β_{11}), as some of their interesting clock cycles were related to operations outside of the Ascon permutation, such as loading the initial states, XORing with P or K, or calculating T.

Among all the words, we detected the highest number of interesting clock cycles for L_1 and L_2 in β_{-1} of Initialization, since these two lanes are loaded with K, which is used four times in the full encryption. Figure 4 shows the $\Sigma_f R_f^2$ values for the H/L words from L_1 and L_2 in β_{-1} of Initializaion with the corresponding clock cycles. We found that the spikes were mainly located in four separate regions, indicating the clock cycles related to the four times when Ascon AEAD uses the key K.

We downsampled the selected interesting clock cycles from 500 to 10 PPC by replacing each 50 consecutive samples with their average, and concatenate them to form the traces $\mathbf{x}$ used for LDA-based template building (see Sec. 2.2).

3.4 Results of fragment template profiling

We evaluate the quality of our templates using two metrics defined by Standaert et

[2] We used the R^2 value as the selection metric rather than the more commonly used SNR, not only because it is a natural fit for a linear-regression model, but it also better fits a fragment-template attack, where we have to combine the quality metrics of multiple fragments. Here, the R^2 values of each fragment can simply be added up, whereas this is not meaningful for signal-to-noise ratios, as the noise is different for each fragment. For readers interested in comparing values: had we used the *maximum* SNR value among the four member bytes for each clock cycle, a threshold of SNR ≥ 0.02 would have given the closest results (see Figure 4 for both metrics side-by-side).

Table 3: Quality evaluation of selected templates from the `U-Os` experiment, using both the top-rank success rate (1-SR) and the base-2 logarithm of the guessing entropy (LGE).

word		high				low			
byte		0	1	2	3	4	5	6	7
L_1 of Init. **input**	1-SR	0.859	0.809	0.852	0.733	0.804	0.684	0.751	0.619
(1st lane of K)	LGE	0.182	0.278	0.203	0.415	0.295	0.500	0.362	0.666
L_2 of Init. **input**	1-SR	0.791	0.758	0.820	0.766	0.868	0.777	0.758	0.647
(2nd lane of K)	LGE	0.294	0.341	0.225	0.321	0.167	0.318	0.341	0.566
word		even				odd			
byte		0	1	2	3	4	5	6	7
L_3 of Fin. β_{11}	1-SR	0.126	0.095	0.165	0.165	0.137	0.144	0.170	0.119
	LGE	3.048	3.387	2.770	2.861	3.000	3.110	2.723	3.042
L_4 of Fin. β_{11}	1-SR	0.099	0.095	0.193	0.195	0.158	0.112	0.186	0.202
	LGE	3.287	3.644	2.621	2.479	2.833	3.295	2.440	2.427
L_0 of Init. α_6	1-SR	0.003	0.006	0.008	0.009	0.009	0.012	0.004	0.006
	LGE	6.229	6.140	5.907	6.311	6.096	6.168	6.226	6.199

Table 4: Quality evaluation of fragment templates for the key of ASCON AEAD with either all or only one part of the interesting clock cycles (`U-Os` experiment).

lane		L_1								L_2							
word		high				low				high				low			
byte		0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
all interesting clock cycles	1-SR	0.859	0.809	0.852	0.733	0.804	0.684	0.751	0.619	0.791	0.758	0.820	0.766	0.868	0.777	0.758	0.647
	LGE	0.182	0.278	0.203	0.415	0.295	0.500	0.362	0.666	0.294	0.341	0.225	0.321	0.167	0.318	0.341	0.566
region 1 only	1-SR	0.090	0.112	0.125	0.090	0.088	0.069	0.082	0.057	0.075	0.083	0.094	0.070	0.145	0.065	0.092	0.118
	LGE	3.480	3.363	2.933	3.577	3.599	4.031	3.643	4.161	3.801	3.526	3.573	3.963	2.778	3.825	3.578	3.066
region 2 only	1-SR	0.100	0.085	0.093	0.051	0.166	0.112	0.141	0.089	0.150	0.166	0.203	0.158	0.175	0.177	0.149	0.098
	LGE	3.551	3.706	3.627	4.421	2.659	3.332	2.726	3.656	2.711	2.821	2.384	2.854	2.462	2.620	2.857	3.576
region 3 only	1-SR	0.135	0.095	0.088	0.061	0.103	0.104	0.110	0.065	0.098	0.066	0.127	0.070	0.152	0.137	0.121	0.066
	LGE	2.955	3.317	3.467	4.073	3.414	3.450	3.320	4.170	3.370	3.977	2.975	3.921	2.668	2.916	3.174	3.913
region 4 only	1-SR	0.114	0.112	0.124	0.091	0.099	0.083	0.119	0.091	0.068	0.077	0.093	0.089	0.096	0.103	0.130	0.050
	LGE	3.264	3.006	2.984	3.176	3.337	3.336	3.020	3.401	3.675	3.445	3.202	3.324	3.286	3.178	2.827	4.021

al. [SMY09]: the n^{th}*-order success rate* (n-SR), which is the fraction of trials where the correct candidate has rank not larger than n, and the *guessing entropy* (GE), which is the expected value of the rank of the correct candidate (1 being the top rank). The *logarithmic guessing entropy* (LGE) is the arithmetic mean of the base-2 logarithm of those ranks.

After first shortening the traces according to the results of the interesting-clock-cycle detection and then building the LDA-based template parameters ($\mathbf{S}$, $\bar{\mathbf{x}}_{b,\mathrm{proj}}$ for all 256 values b of a fragment, etc.), we determined for the 1000 traces in our validation set both the LGE and 1-SR value, the latter being the fraction of those traces where the correct fragment (byte) candidate tops the ranking table of all 256 likelihood values $L(\mathbf{x}_{\mathrm{a,proj}}|\bar{\mathbf{x}}_{b,\mathrm{proj}}, \mathbf{S})$.

Table 3 shows these 1-SR and LGE values for a few example templates, while Figure 5 plots the results for all the target templates. Note that we built the H/L templates for the key, but E/O templates for the other intermediate values, to better match the implementation.

As we had expected, templates for the key fragments had the best quality among all the templates, as K fragments were built from the highest numbers of clock cycles. The results for templates of fragments in the last two lanes in state β_{11} in Finalization were also satisfactory, considering that these two lanes are part of the permutation output and then XORed with the key for the tag T, leading to more interesting clock cycles detected. The 1-SRs for fragments from the middle rounds, α_6 in Initialization for example, were much lower, while the corresponding LGEs were much higher than those values from either K or β_{11} in Finalization, as the optimized implementation of Weatherley appears to reduce the clock cycles that operate on a single intermediate value inside the permutation, whereas the input and output of a permutation will be involved in more operations across the permutations for AEAD mode.

We also show the results of quality evaluation when we built the templates for the key fragments with only one of the four regions of interesting clock cycles in Table 4. These

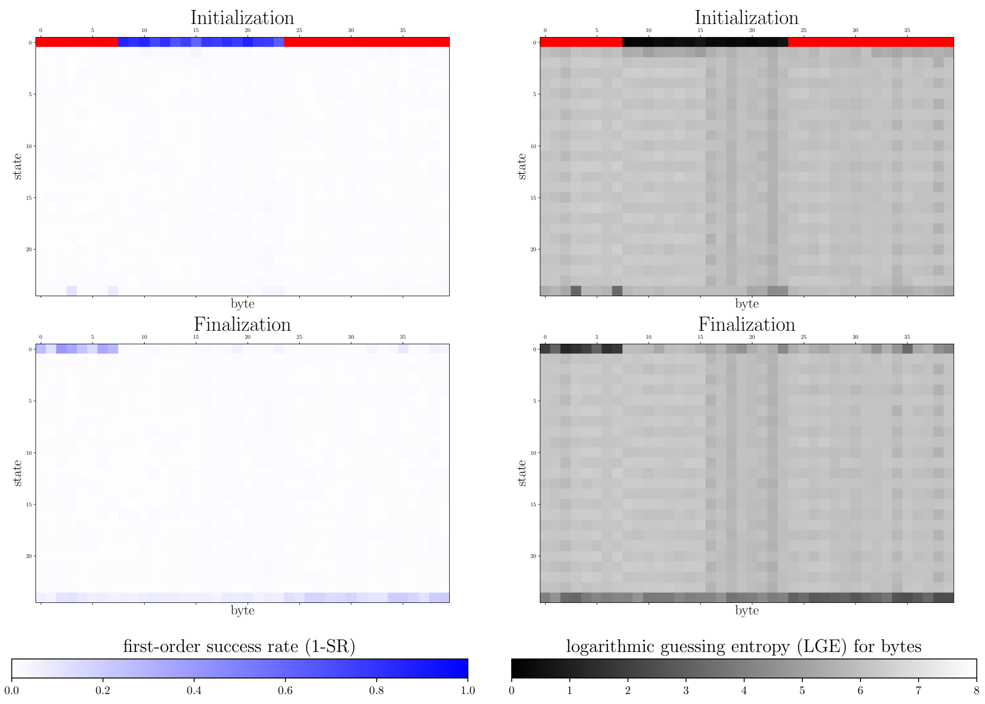


Figure 5: First-order success rate (left) and logarithmic guessing entropy (right) of all target fragment templates from `U-Os`. Each row represents a 40-byte state, e.g. state 0 is β_{-1}, state 1 is α_0, state 2 is β_0, etc., in chronological order. The red blocks represent bytes of the known values IV, N, for which no templates were needed.

results provide evidence that using the same key more than once in an Ascon AEAD significantly helps the attackers to build better templates.

4 Belief propagation and key enumeration for Ascon AEAD

Once we have obtained probability tables from the normalized likelihoods of attack traces matched against our fragment templates, we then apply belief propagation and key enumeration.

4.1 Factor graph for bitwise loopy BP across all intermediate states

In a factor graph covering the full Ascon AEAD mode with our assumptions, the connections within a single permutation and the connections across permutations form a loopy structure, so we apply a loopy-BP procedure to update the probability tables for K. With such a complicated loopy structure, we have to marginalize all the probability tables estimated by fragment templates into bitwise tables and perform belief propagation on individual bits as variables for efficiency. As for the tables from E/O templates, we only need a simple transposition step to move the bits to their original places in the H/L words after marginalization.

Graph for single encryptions We first build the factor graph covering a single round of the Ascon permutation. This small factor graph (Figure 6) includes three variable states and their corresponding observed factors: $\beta_{\Omega-1}$, α_Ω, β_Ω, and two types of constraint factors: f_S and f_L.

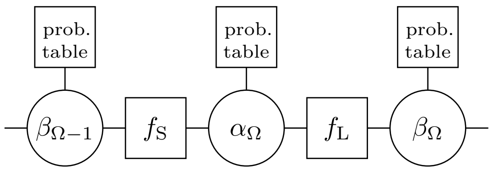


Figure 6: The factor graph for round Ω of the ASCON permutation. State variables $\beta_{\Omega-1}$, α_Ω, and β_Ω shown here each represent 320 single-bit variable nodes, respectively.

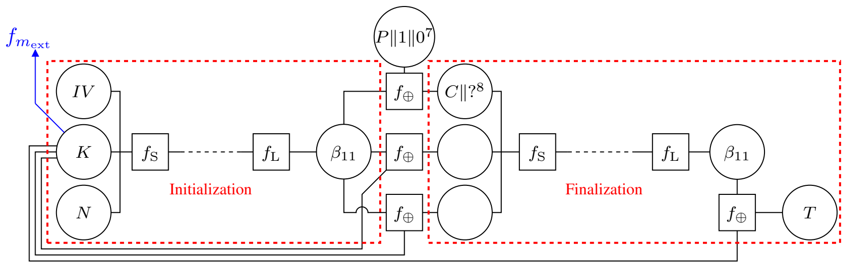


Figure 7: The factor graph at state level covering a full ASCON-128 encryption, with empty associated data and seven-byte plaintext (observation factors omitted).

Since $\alpha_\Omega = p_S(p_C(\beta_{\Omega-1}))$, the f_S constraint factors should update the information following the mathematical relations in functions p_C (constant addition) and p_S (non-linear substitution). We followed how Kannwischer et al. designed their constraint factors for the non-linear step and the constant addition step in KECCAK [KPP20, Sec. 4.1]: for the factors of p_S, we connect their five input bits and five output bits and use the S-box table as the mathematical constraint, while for p_C, we swap the probability values of the two candidates (0 and 1) when the value of the corresponding constant bit is 1. On the other hand, constraint factors f_L update the information following the mathematical relations in the linear function p_L, which are all XOR functions with three inputs and one output at the bitwise level (see eq. (2)). For example, $L'_0 \leftarrow \Sigma_0(L_0)$ can be realized by the constraint $\beta_\Omega[0,0] = \alpha_\Omega[0,0] \oplus \alpha_\Omega[0,64-19] \oplus \alpha_\Omega[0,64-28] = \alpha_\Omega[0,0] \oplus \alpha_\Omega[0,45] \oplus \alpha_\Omega[0,36]$, where $\alpha_\Omega[i,j]$ means bit j in lane i. Having built the factor graph for the first round of the p^{12} permutation, we can simply repeat the same construction for the remaining eleven rounds, only with different round constants, to cover all the states in one invocation of this permutation.

Considering ASCON AEAD applies a sequence of p^a and p^b permutations, the overall factor graph will be a concatenation of multiple factor graphs, one for each permutation, connected by constraint factors $f_\oplus$ for XOR functions, and variables for the inputs or outputs processed in between. Figure 7 shows the factor graph covering all the target states in our experiment. According to the encryption shown in Figure 3, the input state of the p^{12} in Finalization will be the output state (or state β_{11}) of p^{12} in Initialization, XORed with the state $P\|(\texttt{0x80})\|K\|K'$, where K' is the key K with the least significant bit flipped. Therefore, via a constraint factor $f_\oplus$, the two variables, respectively representing the bit in the first lane L_0 of the input state of Finalization and its counterpart in the output state of Initialization, will be connected with the corresponding variable for the bit in the padded plaintext $P\|(\texttt{0x80})$. Similarly, bits from $L_1\|L_2$ of the two states will be connected to the variables for K via constraint factor $f_\oplus$. Bits from $L_3\|L_4$ are also linked to the variables for K via $f_\oplus$, but the probability values for the least significant bit need to be swapped when the message is exchanged between the K variables and the $f_\oplus$. Likewise, the variables for the last 128 bits of the Finalization output are connected with the variables for K and T via $f_\oplus$.

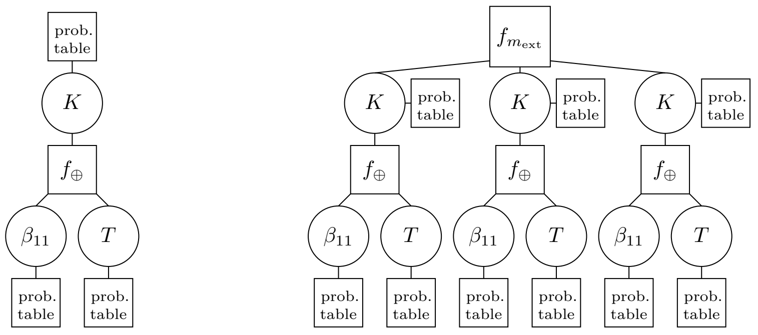


Figure 8: Tree-shaped factor graphs for single (left) and multiple encryptions (right).

Graph for multiple encryptions with the same key Ascon allows reuse of a key with different nonces. So for multiple encryptions with the same key, we introduce another external constraint factor f_{m_ext}, where the constraint is $K_1 = K_2 = \ldots = K_N$, to connect each key variable from a separate factor graph for a single encryption.

Key enumeration Finally, we apply the key enumeration algorithm [VCGRS12], as discussed in Sec. 2.4, to find the correct bit combination for the key, given the bit probability tables obtained from the belief propagation procedure. We can simply check the correctness of the enumerated key combinations since we know N, A, P, C, and T according to our assumptions in Section 3.1.

4.2 Loop-free alternative factor graph

As we can see from the results of the template evaluation in Figure 5, the templates for fragments in the middle states of both the permutations in Initialization and Finalization provide only little information. It may not be worth to perform belief propagation with a large factor graph covering all the middle states. Therefore, we also tried an alternative factor graph, where we removed the nodes for those middle states from the loopy factor graph, and only kept the nodes related to the XOR operation of the key K and the last 128 bits of β_{11} in Finalization to calculate the tag T. Figure 8 shows the new smaller factor graph for single encryptions and its expanded version for multiple encryptions with the same key. These smaller factor graphs will similarly output updated probability tables for key enumeration.

There are two advantages of this smaller graph design. The first one is that it is no longer a loopy structure, but a tree, so we can update the information recursively by accessing each node only once. On the other hand, thanks to the simplicity of the XOR operation, as well as the assumption that the tag T is already known by attackers, it will still be practical to perform belief propagation on byte tables or tables for even larger fragments, and therefore avoid the information loss caused by marginalization to bit tables. In these cases, the belief propagation procedure will output the updated probability tables for fragments instead of bits for enumeration.

4.3 Results for belief propagation and key enumeration

In the following, we report both the logarithmic guessing entropy and the n^th-order success rate as the evaluation metrics for our ability to recover the full key. For key ranks up to 2^{24}, we determined these through actual key enumeration, whereas higher key ranks we estimated using the histogram-based method of [GGP+15, Alg. 1]. We determined or estimated this way the rank of each of the 1000 keys used in our attack traces, and then

Table 5: Base-2 logarithmic guessing entropy (LGE) achieved for our key-recover attacks in the `U-Os` experiment with between 1 and 10 attack traces, using either no SASCA stage (using byte fragments), or with a SASCA stage using either the loopy belief propagation (using byte fragments marginalized into bits), or the tree-based belief propagation (using either 8-bit or 16-bit fragments, not marginalized).

# attack traces	1	2	3	4	5	6	7	8	9	10
no BP 8 bit	12.2189	4.9610	3.2118	2.4741	2.1625	1.9403	1.8147	1.7075	1.6175	1.5514
loopy BP 8 bit	11.6443	1.3345	0.1675	0.1069	0.0453	0.0516	0.0220	0.0000	0.0078	0.0261
tree BP 8 bit	4.4958	0.6850	0.2439	0.1176	0.0851	0.0690	0.0473	0.0379	0.0352	0.0322
tree BP 16 bit	2.3795	0.2850	0.1028	0.0498	0.0336	0.0316	0.0200	0.0156	0.0122	0.0076

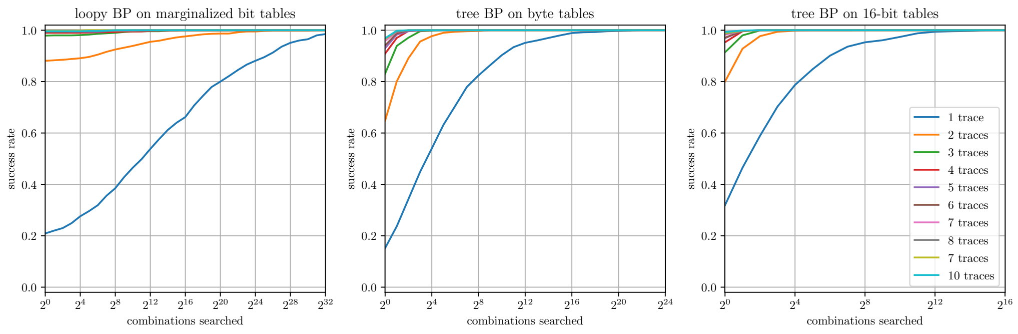


Figure 9: Key-recovery success rate as a function of the search depth (2^n-SR) for the `U-Os` experiment when performing SASCA with either our loopy or tree-shaped factor graphs.

estimated the LGE as the arithmetic mean of the binary logarithm of the rank of each key.

Table 5 shows the guessing entropy we achieved for all of our `U-Os` experiments, using either no belief propagation step, or after applying SASCA using either the loopy or tree-shaped factor graphs. Figure 9 plots for the latter experiments with SASCA the success rates after key enumeration with a given search depth. The results show that with loopy belief propagation on a single attack trace, a 2^{32} key enumeration will achieve a success rate of nearly 100%, and applying the tree-shaped belief propagation[3] reduces the cost of the key enumeration to well below 2^{20} steps. (With two or more attack traces, hardly any key enumeration is needed.)

To see whether a larger template-fragment size collects more information, we also repeated the tree-BP experiment with 16-bit fragments instead of bytes. The bottom two rows in Table 5 show that the guessing entropy drops very roughly by a factor two. Due to the high quality of the templates for our key fragments, belief propagation was not actually essential for extracting the key from the `U-Os` target.

5 Effect of compiler optimization on template attack

In the previous `U-Os` experiments, we left the `gcc` optimization set to option `-Os` (optimize for space), which was the default for the ChipWhisperer platform software. To see whether the compiler's code generation affects the performance of our attack, we decided to repeat the experiment with `gcc` option `-O3` (optimize for time), resulting in the `U-O3` recordings. Note that the different optimization options will not affect the execution of Weatherley's

[3]For tree-shaped belief propagation, without marginalization, probability tables from H/L and E/O templates are difficult to combine in the same factor graph. Therefore, we used there only H/L templates.

Table 6: Base-2 logarithmic guessing entropy achieved for our key-recover attacks in the `U-O3` experiment with between 1 and 10 attack traces, using either no SASCA stage (using byte fragments), or with a SASCA stage using the tree-based belief propagation (using either 8-bit or 16-bit fragments, not marginalized). Cf. Table 5.

# attack traces	1	2	3	4	5	6	7	8	9	10
no BP 8 bit	47.1043	37.9218	34.5498	32.6819	31.5794	30.8995	30.3895	29.9537	29.6220	29.4119
tree BP 8 bit	35.8763	23.6844	19.2180	16.8768	15.4783	14.6245	13.9832	13.5102	13.0639	12.7684
tree BP 16 bit	26.6637	15.2346	11.3605	9.4302	8.3368	7.6768	7.2024	6.8662	6.5471	6.3380

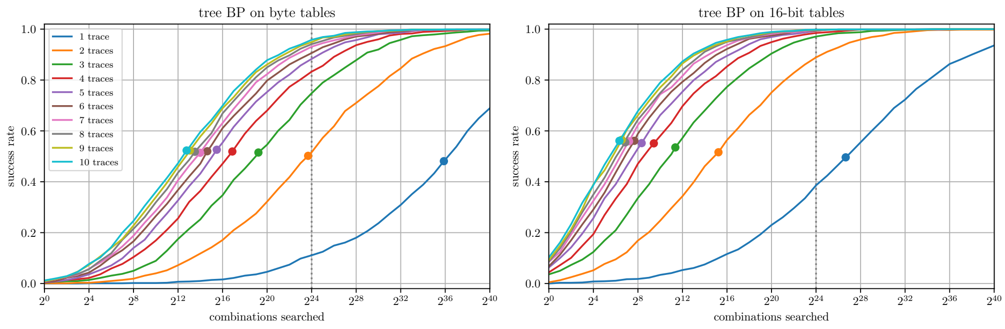
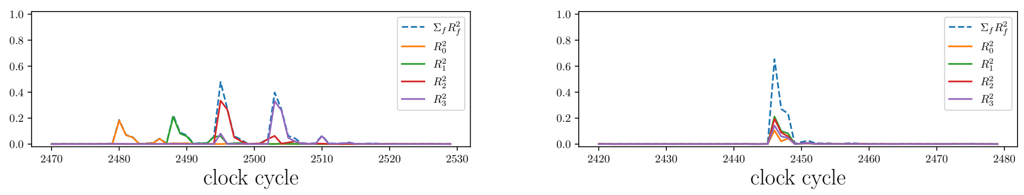


Figure 10: Key-recovery success rates on Ascon-128 with both 8 and 16-bit fragments in the `U-O3` experiment. The little circle marks the success rate if the search depth is limited to the respective guessing entropy from Table 6. Cf. Figure 9.

Ascon permutation [Wea21, `ASCON/internal-ascon-armv7m.S`], since its source code is manually optimized assembly code, which bypasses the optimizer. However they do affect the AEAD code outside the permutations, such as XORing the key K or calculating the tag T, as these mode-level operations are written in C. (Thus, we focus here only on the tree-BP experiment, as the middle rounds of the permutation won't be affected.)

Table 6 and Figure 10 show the equivalent information as Table 5 and Figure 9, respectively, but for compiler option `-O3` instead of `-Os`. Compared to the very successful single-trace attack in the `U-Os` experiment, the performance here is clearly worse: after our tree BP with 16-bit fragments, we need to search about 2^{27} key candidates to achieve a success rate higher than 50% (compared to previously 2^2), and we need to search about 2^{36} candidates with 8-bit fragments (compared to previously 2^4). In other words, the `U-O3` attack would hardly be practical without both BP *and* key enumeration.

A look into both the C source code of Weatherley's unmasked implementation, and the assembler listing produced by the compiler (with option `-Wa,-adhlns=file.lst`), revealed the reason. Although the handwritten assembler code for the permutation uses 32-bit registers, the surrounding C code XORs the key K with the state of the duplex construct.



(a) `U-Os` experiment

(b) `U-O3` experiment

Figure 11: The $\Sigma_f R_f^2$ results and the R_f^2 values for each byte fragment ($f = 0, 1, 2, 3$) of the high word of L_1 in K

Table 7: This LGE table compares the single-trace attack performance between including in the selected clock cycles only *one* of the four key-use regions 1, 2, 3, or 4, (from Figure 4, respectively), as opposed to all four together, all using tree BP with byte fragments.

traces \ region	all four	1	2	3	4
`U-Os`	4.4958	50.6751	44.5369	48.3660	41.6838
`U-O3`	35.8763	69.3538	76.4909	73.4121	88.3132

For example, it XORs two lanes of Finalization output (β_{11}) with K, to generate the tag T, using the macro `lw_xor_block_2_src()` in [Wea21, `ASCON/internal-util.h`], which is actually a loop processing individual bytes. When compiled to optimize code space (i.e., minimize the size of the executable) with `gcc` option `-Os`, the resulting ARMv7-M assembler code looks pretty exactly like the source code suggests, i.e., a loop over 16 bytes, which loads one byte from K and one from β_{11} into the 32-bit registers, XORs them, and stores one byte of T per iteration. In contrast, if we instead ask the compiler to optimize for time (`-O3`), it not only unrolls that loop, but also converts it into a sequence of just four repetitions of the operations for loading, XORing and storing 32-bit words. In other words, the optimizer converted here an 8-bit implementation of the key XOR operation into a 32-bit implementation.

We can also observe this difference on the recorded traces. Figure 11a and 11b show the results of the interesting clock-cycle detection for the high word of the first lane (L_1) of K during the calculation of T, when the code was compiled with options `-Os` and `-O3`, respectively. For `U-Os`, the peaks of the R^2_f values of each 8-bit fragment are located in four different clock cycles, indicating that their operations were not executed simultaneously, whereas for `U-O3`, the peaks are located in the same clock cycle.

Table 7 indicates how much the four-time use of the key in the Ascon AEAD mode helped our single-trace attacks against the unmasked version to succeed. Building the templates for key fragments with clock cycles related to only each single use of the key at a time, compared to those from all four times, increased the key enumeration cost by more than a factor 2^{32} for both our unmasked experiments.

6 Attacking a masked version

6.1 Attack strategy

Following our experiments above on the unmasked Ascon implementation (recordings `U-Os` and `U-O3`), we also tried to apply the combination of fragment template attack, belief propagation, and key enumeration on an implementation with masking (recording `M-Os`). Our target masked implementation of Ascon AEAD was from the same package by Weatherley [Wea21, `ASCON_masked/`]. This offers a C implementation of the permutation and protects the inputs (key, nonce, plaintext, etc.) with first-order Boolean masking [CJRR99], separating each of these values into two shares: one is the mask, varying per encryption, provided by a pseudo-random generator based on ChaCha [Ber08], and therefore the other share is the XOR of the input value and the mask. Throughout the encryption process, the intermediate values all remain likewise split into two shares, to randomize all the register values during execution. Compared to the unmasked version, this implementation also avoids some problems that may help side-channel attacks on the former. For example, it no longer XORs 8-bit values when calculating the tag T, and the two shares of the key are only sliced once, rather than three times.

Bronchain and Standaert [BS21] attacked Boolean masked implementations of AES and

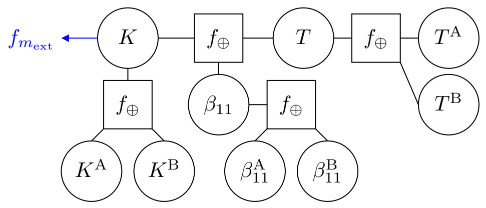


Figure 12: Factor graph for our `M-Os` experiment. (Each variable node also connects to an observation factor node, which is omitted in this graph.)

Clyde by extending the factor graph for the unmasked algorithm with nodes representing the original values connected to their shares in the masked version via a $f_\oplus$ factor. Following this idea, we introduce a multi-trace attack derived from the previous tree-shaped one, where the factor graph (Figure 12) will also cover the two shares of the original target states. Similar to the setting of the previous unmasked version, we use the empty associated data A and fix the size of the plaintext P to seven bytes. In the profiling stage, we assume that attackers can access all the input and output values (K, N, P, C, and T) as well as the seed of the pseudo-random generator, so that they can produce fragment templates for the key, its two shares, and all the other intermediate states in the factor graph. In the attack stage, we only use the probability tables obtained from the templates, and the known T values, to perform belief propagation and key enumeration, without knowledge of the seed.

Note that Figure 12 reflects the mathematical relations among the original values and their shares, not the actual steps in this masked implementation to calculate T. The implementation first calculates $T^{\text{A}} := K^{\text{A}} \oplus \beta_{11}^{\text{A}}$, $T^{\text{B}} := K^{\text{B}} \oplus \beta_{11}^{\text{B}}$, and finally $T := T^{\text{A}} \oplus T^{\text{B}}$. Therefore, we cannot build templates for the fragments of β_{11} since this value never appears. Instead we assign them a probability table with a uniform distribution (i.e., no information update). Besides, our assumption was that the attacker knows T, so we do not need the templates or probability tables of T^{A} and T^{B}, given that they will not affect the belief propagation.

6.2 Experiments

As mentioned in Sec. 3.2, while recording the `M-Os` traces from the masked version, we kept the setup the same as for the `U-Os` traces, except for the larger number of attack traces recorded, to have 100 encryptions each for the same key.

Table 8 shows the number of interesting clock cycles detected in the `M-Os` experiment, while Table 9 shows the results of the quality evaluation of the fragment templates. Here we built fragment templates for sliced registers (E/O words), since that is how the implementation represents most of our target states. We can see that the masking does protect the key K to some extent, as fewer interesting clock cycles (37 and 35 for the two lanes, respectively) were detected compared to the unmasked experiments (see Table 2, β_{-1} in Init.), leading to lower quality templates as evident from the higher guessing-entropy values for these fragments. However, for the two shares K^{A} and K^{B}, we still detected a large number of interesting clock cycles (144 and 139 for two lanes of K^{A}, 240 and 259 for K^{B}), and therefore the quality of their templates is still promising once attackers can calculate the random numbers for masking in the profiling stage. Note that there are more interesting clock cycles for K^{B}, the random mask, than for K^{A}, because for the former we can also detect leaks from where the masks are generated.

For the belief propagation and key enumeration, Figure 13 and Table 10 show the key-recovery guessing entropy and success rates achieved with different numbers of attack traces. Single-trace attacks did not succeed, however starting from around five attack traces recorded with the same key, a 2^{36} key enumeration is likely to succeed.

Table 8: Numbers of interesting clock cycle detected for the `M-Os` experiment.

target state		K		K^{A}		K^{B}		Fin. β_{11}^{A}		Fin. β_{11}^{B}	
Lane 1	high/low	21	26	113	109	161	157	41	42	46	40
	even/odd	16	28	113	106	213	207	36	33	28	39
	union	37		144		240		50		58	
Lane 2	high/low	20	26	107	109	203	174	36	36	44	43
	even/odd	30	33	114	116	230	227	17	41	16	39
	union	35		139		259		49		51	

Table 9: Quality evaluation of the templates from in the `M-Os` experiment, using both the top-rank success rate (1-SR) and the base-2 logarithm of the guessing entropy (LGE).

byte		Lane 1								Lane 2							
		even word				odd word				even word				odd word			
		0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
K	1-SR	0.006	0.004	0.012	0.016	0.013	0.018	0.020	0.011	0.003	0.006	0.012	0.013	0.016	0.016	0.013	0.021
	LGE	6.011	5.929	5.964	5.967	5.756	5.819	5.659	5.844	6.034	6.066	5.858	5.780	5.944	5.964	5.792	5.537
K^{A}	1-SR	0.179	0.112	0.245	0.206	0.115	0.108	0.129	0.162	0.177	0.099	0.246	0.149	0.142	0.102	0.173	0.214
	LGE	2.541	3.254	2.068	2.306	3.237	3.292	3.139	2.761	2.494	3.518	2.135	2.938	3.027	3.257	2.555	2.289
K^{B}	1-SR	0.308	0.400	0.440	0.511	0.354	0.535	0.340	0.598	0.330	0.390	0.419	0.509	0.313	0.431	0.307	0.622
	LGE	1.704	1.296	1.204	0.872	1.453	0.822	1.535	0.662	1.664	1.178	1.229	0.890	1.634	1.244	1.762	0.649
Fin. β_{11}^{A}	1-SR	0.014	0.014	0.021	0.017	0.013	0.016	0.013	0.026	0.017	0.011	0.016	0.019	0.011	0.015	0.014	0.023
	LGE	5.659	5.816	5.236	5.149	5.761	5.856	5.322	5.117	5.660	5.883	5.539	5.299	5.894	5.905	5.790	5.232
Fin. β_{11}^{B}	1-SR	0.007	0.010	0.014	0.015	0.016	0.016	0.018	0.019	0.014	0.011	0.007	0.007	0.011	0.007	0.010	0.020
	LGE	5.852	5.963	5.465	5.474	5.766	5.781	5.367	5.034	5.960	6.144	6.062	5.828	5.619	5.717	5.375	5.018

Table 10: Logarithmic guessing entropy achieved by our key-recovery attack with 8-bit fragment templates, using 1–100 encryption traces, respectively (tree BP, `M-Os` experiment). (The no BP control experiment, using only templates for K, found hardly any exploitable first-order leakage of K from the targeted implementation.)

# attack traces:	1	2	3	4	5	7	10	20	50	100
tree BP 8 bit	82.2678	65.3453	51.6035	42.3151	35.8232	27.1959	20.2351	12.4211	8.2922	7.1706
no BP 8 bit	120.6646	119.7556	119.2906	118.9934	118.8436	118.6460	118.4844	118.2426	118.0507	117.9981

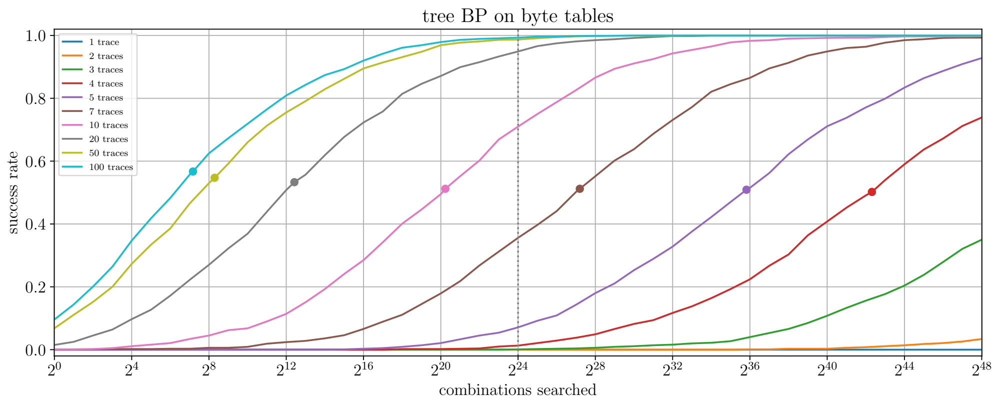


Figure 13: Key-recovery success rates in the `M-Os` experiments on the masked implementation as a function of the key-enumeration search depth. The little circle marks the success rate if the search depth is limited to the respective guessing entropy from Table 10.

7 Conclusion

ASCON's design supports a leveled implementation of side-channel countermeasures, meaning that only its Initialization and Finalization phases (two key applications and one permutation p^a each) need to be protected against DPA attacks, whereas for the remaining operations (message XORs and permutations p^b), SPA countermeasures are sufficient to ensure confidentiality of the encryption process, and no further countermeasures are required to ensure integrity [BBC+20, VCS23]. This was achieved by applying the key four times to the capacity, both before and after p^a during both Initialization and Finalization.

However, our results demonstrate that if such countermeasures are either absent (`U-Os`, `U-O3`), or not entirely effective against a template attack (`M-Os`), then this quadruple use of the key in ASCON AEAD mode actually increases the exposure of the key in profiled side-channel attacks (see Table 4), which may enable full key recovery (see Table 7), compromising both confidentiality and integrity. That higher exposure of the key, which in our loopy factor graph is directly connected to four different locations (Figure 7), enables the belief propagation algorithm to pass messages between Initialization and Finalization. Previous attack simulations by Luo et al. [LWL+22] did not exploit this higher key exposure and used only the mathematical relations around the first use of the key, at the start of Initialization.

Our successful single-trace attack (`U-Os`) benefited from some remaining 8-bit instructions in an open-source 32-bit adaption of the algorithm. Yet, even once these were fully converted to 32-bit instructions (`U-O3`), we still could recover the key used in this unmasked ASCON AEAD implementation, by belief propagation and key enumeration, with high success rates, from no more than 10 traces. Our attack procedure should also be applicable to ASCON-128a, with only minor modifications.

Our successful multi-trace attack on the more carefully written first-order Boolean masked ASCON AEAD implementation demonstrates how such protection, originally designed against CPA/DPA-style attacks, can be overcome by an appropriately designed template attack. However, a real-world application of such a profiled attack may still pose challenges considering our assumptions about the access to inputs that the attacker needs during the profiling phase.

An additional outstanding challenge remains to recover complete ASCON hashing inputs from a single trace, as was accomplished in [YK22] for SHA-3 (KECCAK), another sponge-based hash function. This will likely require better templates (e.g., directly built for 32-bit values, as recently proposed by Cassiers et al. [CDSU23]) for the internal states of the ASCON permutation. Our templates for these (e.g., Init. α_6 in Table 3 and Figure 5) were less effective than those reported for the KECCAK permutation in [YK22, Table 2]. However, even with the very similar hardware setting we used, such direct comparisons are still complicated by the fact that the KECCAK and ASCON target implementations came from different authors and had different programming styles. The former was entirely portable C code that left the 64-bit to 32-bit conversion to the compiler, whereas the latter offered a handcrafted assembler implementation of the permutation. That, but also the fact that ASCON's permutation is significantly simpler, for example it lacks an equivalent of KECCAK's complex θ step, overall appears to have resulted in less information leaking from the fewer instructions needed by ASCON to process its intermediate values.

We hope that our attack methodology can serve as a benchmark for the design of stronger masking protections, and other implementation guidance, specifically for protecting against profiled attacks on software implementations.

Data and source code used are available at:

https://www.cl.cam.ac.uk/research/security/datasets/ascon/

References

[ADK19] Abubakr Abdulgadir, William Diehl, and Jens-Peter Kaps. An open-source platform for evaluation of hardware implementations of lightweight authenticated ciphers. In *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–5, 2019. https://doi.org/10.1109/ReConFig48160.2019.8994788.

[BBC$^+$20] Davide Bellizia, Olivier Bronchain, Gaëtan Cassiers, Vincent Grosso, Chun Guo, Charles Momin, Olivier Pereira, Thomas Peters, and François-Xavier Standaert. Mode-level vs. implementation-level physical security in symmetric cryptography. In *Advances in Cryptology – CRYPTO 2020*, pages 369–400, Cham, 2020. Springer International Publishing. https://doi.org/10.1007/978-3-030-56784-2_13.

[BDPA12] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Permutation-based encryption, authentication and authenticated encryption. Presented at DIAC, 2012. https://keccak.team/files/KeccakDIAC2012.pdf.

[Ber08] Daniel J. Bernstein. ChaCha, a variant of Salsa20. In *Workshop record of SASC 2008*, pages 273–278. ECRYPT, 2008. https://cr.yp.to/chacha/chacha-20080120.pdf.

[BS21] Olivier Bronchain and François-Xavier Standaert. Breaking masked implementations with many shares on 32-bit software platforms: or when the security order does not matter. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(3):202–234, July 2021. https://tches.iacr.org/index.php/TCHES/article/view/8973.

[CDSU23] Gaëtan Cassiers, Henri Devillez, François-Xavier Standaert, and Balazs Udvarhelyi. Efficient regression-based linear discriminant analysis for side-channel security evaluations: Towards analytical attacks against 32-bit implementations. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(3):270–293, June 2023. https://tches.iacr.org/index.php/TCHES/article/view/10964.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 398–412. Springer, Berlin, Heidelberg, 1999. https://doi.org/10.1007/3-540-48405-1_26.

[CK14a] Marios O. Choudary and Markus G. Kuhn. Efficient stochastic methods: profiled attacks beyond 8 bits. In *International Conference on Smart Card Research and Advanced Applications*, pages 85–103. Springer, 2014. https://doi.org/10.1007/978-3-319-16763-3_6.

[CK14b] Omar Choudary and Markus G. Kuhn. Template attacks on different devices. In Emmanuel Prouff, editor, *Constructive Side-Channel Analysis and Secure Design*, pages 179–198, Cham, 2014. Springer International Publishing. https://doi.org/10.1007/978-3-319-10175-0_13.

[CRR03] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In Burton S. Kaliski, Çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2002*, pages 13–28. Springer, Berlin, Heidelberg, 2003. https://doi.org/10.1007/3-540-36400-5_3.

[CWL] CW1173: ChipWhisperer-Lite. accessed February 2018, https://media.newae.com/datasheets/NAE-CW1173_datasheet.pdf.

[DAF+18] William Diehl, Abubakr Abdulgadir, Farnoud Farahmand, Jens-Peter Kaps, and Kris Gaj. Comparison of cost of protection against differential power analysis of selected authenticated ciphers. *Cryptography*, 2(3), 2018. https://doi.org/10.3390/cryptography2030026.

[DEMS21] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1.2 submission to NIST, May 2021. https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/ascon-spec-final.pdf.

[GGP+15] Cezary Glowacz, Vincent Grosso, Romain Poussier, Joachim Schüth, and François-Xavier Standaert. Simpler and more efficient rank estimation for side-channel security assessment. In Gregor Leander, editor, *Fast Software Encryption*, pages 117–129. Springer Berlin Heidelberg, 2015. https://doi.org/10.1007/978-3-662-48116-5_6.

[GWDE17] Hannes Gross, Erich Wenger, Christoph Dobraunig, and Christoph Ehrenhöfer. Ascon hardware implementations and side-channel evaluation. *Microprocessors and Microsystems*, 52:470–479, 2017. https://www.sciencedirect.com/science/article/pii/S0141933116302721.

[KPP20] Matthias J. Kannwischer, Peter Pessl, and Robert Primas. Single-trace attacks on Keccak. Cryptology ePrint Archive, Paper 2020/371, 2020. https://eprint.iacr.org/2020/371.

[LWC] Lightweight cryptography. NIST Computer Security Resource Center (CSRC). accessed April 2023, https://csrc.nist.gov/projects/lightweight-cryptography.

[LWL+22] Sinian Luo, Weibin Wu, Yanbin Li, Ruyun Zhang, and Zhe Liu. An efficient soft analytical side-channel attack on Ascon. In Lei Wang, Michael Segal, Jenhui Chen, and Tie Qiu, editors, *Wireless Algorithms, Systems, and Applications*, pages 389–400. Springer, Cham, 2022. https://doi.org/10.1007/978-3-031-19208-1_32.

[Mac03] David J. C. MacKay. *Information theory, inference and learning algorithms*. Cambridge University Press, 2003.

[OC14] Colin O'Flynn and Zhizhang (David) Chen. ChipWhisperer: an open-source platform for hardware embedded security research. In Emmanuel Prouff, editor, *Constructive Side-Channel Analysis and Secure Design*, pages 243–260. Springer, Cham, 2014. https://doi.org/10.1007/978-3-319-10175-0_17.

[SA08] François-Xavier Standaert and Cédric Archambeau. Using subspace-based template attacks to compare and combine power and electromagnetic information leakages. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 411–425. Springer, 2008. https://doi.org/10.1007/978-3-540-85053-3_26.

[SD17] Niels Samwel and Joan Daemen. DPA on hardware implementations of Ascon and Keyak. *Proceedings of the Computing Frontiers Conference*, 2017. https://dl.acm.org/doi/abs/10.1145/3075564.3079067.

[SLP05] Werner Schindler, Kerstin Lemke, and Christof Paar. A stochastic model for differential side channel cryptanalysis. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 30–46. Springer, 2005. https://doi.org/10.1007/11545262_3.

[SMY09] François-Xavier Standaert, Tal G Malkin, and Moti Yung. A unified framework for the analysis of side-channel key recovery attacks. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 443–461. Springer, Berlin, Heidelberg, 2009. https://doi.org/10.1007/978-3-642-01001-9_26.

[VCGRS12] Nicolas Veyrat-Charvillon, Benoît Gérard, Mathieu Renauld, and François-Xavier Standaert. An optimal key enumeration algorithm and its application to side-channel attacks. In *International Conference on Selected Areas in Cryptography*, pages 390–406. Springer, 2012. https://doi.org/10.1007/978-3-642-35999-6_25.

[VCGS14] Nicolas Veyrat-Charvillon, Benoît Gérard, and François-Xavier Standaert. Soft analytical side-channel attacks. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 282–296. Springer, 2014. https://doi.org/10.1007/978-3-662-45611-8_15.

[VCS23] Corentin Verhamme, Gaëtan Cassiers, and François-Xavier Standaert. Analyzing the leakage resistance of the NIST's lightweight crypto competition's finalists. In Ileana Buhan and Tobias Schneider, editors, *Smart Card Research and Advanced Applications*, pages 290–308, Cham, 2023. Springer International Publishing. https://doi.org/10.1007/978-3-031-25319-5_15.

[Wea21] Rhys Weatherley. Finalists to the NIST lightweight cryptography competition, June 2021. https://github.com/rweather/lwc-finalists/tree/5d2b22c9ff7744be429cabda0c078ea5b7b6f79e/src/individual.

[YK22] Shih-Chun You and Markus G. Kuhn. Single-trace fragment template attack on a 32-bit implementation of Keccak. In Vincent Grosso and Thomas Pöppelmann, editors, *Smart Card Research and Advanced Applications*, pages 3–23. Springer, Cham, 2022. https://doi.org/10.1007/978-3-030-97348-3_1.