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

Shih-Chun You\textsuperscript{1}, Markus G. Kuhn\textsuperscript{1}, Sumanta Sarkar\textsuperscript{2} and Feng Hao\textsuperscript{2}

\textsuperscript{1} University of Cambridge, Cambridge, UK, \{scy27,mgk25\}@cl.cam.ac.uk
\textsuperscript{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 \cite{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 \cite{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 \cite{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 \cite{SD17}. And ASCON’s mode of operation supports a so-called leveled implementation \cite{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 \cite{SD17} presented Correlation Power Analysis (CPA) attacks and a threshold
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 generation function (TGF) to produce a message-authentication tag \( T \). The encryption process of Ascon is illustrated in the following figure:
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 $p = p_L \circ p_S \circ p_C$.

#### 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

<table>
<thead>
<tr>
<th>Round</th>
<th>0x0f0</th>
<th>0xda1</th>
<th>0xda2</th>
<th>0xda3</th>
<th>0xda4</th>
<th>0xda5</th>
<th>0xda6</th>
<th>0xda7</th>
<th>0xda8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1^a$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>$p_1^b$</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

#### 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 $0 \leq k \leq 63$, as in

\[
\begin{align*}
L_0 &\leftarrow L_0 \oplus L_4, & L_4 &\leftarrow L_4 \oplus L_3, & L_2 &\leftarrow L_2 \oplus L_1 \\
L'_0 &\leftarrow L_0 \oplus (\neg L_0 \land L_4) & L'_1 &\leftarrow L_1 \oplus (\neg L_0 \land L_3) & L'_2 &\leftarrow L_2 \oplus (\neg L_3 \land L_2) \\
L'_3 &\leftarrow L_3 \oplus (\neg L_4 \land L_0) & L'_4 &\leftarrow L_4 \oplus (\neg L_0 \land L_1) & L'_1 &\leftarrow L'_0 \oplus L'_0, & L'_2 &\leftarrow L'_0 \oplus L'_1 & L'_3 &\leftarrow L'_3 \oplus L'_2, & L'_4 &\leftarrow \neg L'_2 \\
\end{align*}
\]

This S-box has the following cryptographic properties: maximum differential and linear probability $\frac{1}{2}$, 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 $\Sigma_i$ for $0 \leq i \leq 4$, and each $\Sigma_i$ performs XOR ($\oplus$) and right rotations ($\gg$) on
the word $L_i$ as in

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

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

\[
\text{Input} = \beta_{-1} \xrightarrow{pc-ps} \alpha_0 \xrightarrow{pl} \beta_0 \xrightarrow{pc-ps} \alpha_1 \xrightarrow{pl} \beta_1 \xrightarrow{pc-ps} \cdots \xrightarrow{pl} \beta_{11} = \text{Output}.
\]

We will use symbols $L_0, L_1, L_2, L_3, L_4$ to represent the five lanes in intermediate states $\alpha_0$ or $\beta_1$, where $\Omega$ 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 $F_9$ “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 $x_b$, where $t \in \{1, \ldots, n_b\}$ enumerates the traces in group $b$. With the $F_9$ 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 $\bar{x}_b = \sum_{t=0}^{7} (b[t] \cdot c_t) + c_8$ for each possible value $b$. To represent the expected vector of an entire $m$-sample trace, we write $\bar{x}_b = \sum_{t=0}^{7} (b[t] \cdot c_t) + c_8$, where $c_0, \ldots, c_8 \in \mathbb{R}^m$.

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

\[
B = \frac{1}{\sum_b n_b} \sum_b n_b (\bar{x}_b - \bar{\bar{x}})(\bar{x}_b - \bar{\bar{x}})^T, \quad \Sigma = \frac{1}{\sum_b n_b} \sum_b \sum_{t=1}^{n_b} (x_{b,t} - \bar{x}_b)(x_{b,t} - \bar{x}_b)^T,
\]

where $\bar{\bar{x}}$ denotes the average of all 256 vectors $\bar{x}_b$. We then project all the $m$-sample traces, including profiling traces $x_{b,t}$, expected traces $\bar{x}_b$, and the attack trace $x_a$, onto the $m'$
(m' ≪ m) eigenvectors with the largest eigenvalues of $\Sigma^{-1}B$, to obtain $m'$-sample traces $x_{b,t,proj}, \bar{x}_{b,proj} \in \mathbb{R}^{m'}$.

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

$$S = \frac{1}{\sum_{b} n_{b}} \sum_{b} \sum_{t=1}^{n_{b}} (x_{b,t,proj} - \bar{x}_{b,proj})(x_{b,t,proj} - \bar{x}_{b,proj})^{T},$$

such that the probability density of the attack trace $x_{a,proj}$ can be modelled as

$$L(x_{a,proj}|\bar{x}_{b,proj},S) = \frac{1}{\sqrt{(2\pi)^{m}|S|}} \exp \left( -\frac{1}{2} (x_{a,proj} - \bar{x}_{b,proj})^{T}S^{-1}(x_{a,proj} - \bar{x}_{b,proj}) \right).$$

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 $V_{f,b} = \{v | F_f(v) = b\}$ be the set of all 32-bit values where fragment number $f$ has value $b$. For each $f$, we apply the $F_0$ stochastic model to obtain the 256 expected trace vectors $\bar{x}_{f,b} = \sum_{\ell=0}^{7} b[\ell] \cdot c_{f,\ell} + c_{f,8}$, from the traces $x_{v,t}$ with $v \in V_{f,b}$, respectively. For the LDA procedure, we calculate $B$ and $\Sigma$ separately for each fragment:

$$B_f = \frac{255}{\sum_{b=0}^{255} \sum_{v \in V_{f,b}} n_{v}(\bar{x}_{f,b} - \bar{x})(\bar{x}_{f,b} - \bar{x})^{T}} \sum_{b=0}^{255} \sum_{v \in V_{f,b}} n_{v},$$

$$\Sigma_f = \frac{255}{\sum_{b=0}^{255} \sum_{v \in V_{f,b}} \sum_{t=1}^{n_{b}} (x_{v,t} - \bar{x}_{f,b})(x_{v,t} - \bar{x}_{f,b})^{T}} \sum_{b=0}^{255} \sum_{v \in V_{f,b}} n_{v},$$

where $n_{v}$ represents the number of traces in group $v$. $B_f$ only contains signals from fragment number $f$, and signals from the other three bytes no longer count here, but instead contribute to $\Sigma_f$. 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 $f_m(x_m)$ are connected to more than one variable $(x_1, \ldots, x_{n_{km}}) = x_m$ (where $N(m) = \{n_1, \ldots, n_{km}\}$ 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 $q_{n \rightarrow m}$, and a message from a factor $f_m$ to a variable $x_n$ as $r_{m \rightarrow n}$. Each of these messages is a function of a value $\xi$ of $x_n$. The probability of a candidate $x_n = \xi$ in message $q_{n \rightarrow m}$ is

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

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

$$r_{m \rightarrow n}(x_n = \xi) = \sum_w \left[ f_m(x_n = \xi, x_m) \prod_{n' \in N(m) \setminus n} q_{n' \rightarrow m}(x_{n'} = w_{n'}) \right],$$

where

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

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

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

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

$$P_n(x_n = \xi) = \frac{Z_n(x_n = \xi)}{\sum_{\xi'} Z_n(x_n = \xi')},$$

where $Z_n(x_n = \xi) = \prod_m r_{m \rightarrow n}(x_n = \xi)$ is the product of the probabilities of $\xi$ 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.
While the belief-propagation step reduces the impact of noise on the ranking tables, the
most likely combination must be the top-left element when a search begins. 

Therefore, the top-leftmost block, representing \( x_c = x_a \oplus x_b \), is the combination with the largest joint probability, and the combination with the second largest joint probability will pertain to one of \((\tilde{s}_{0,1}, \tilde{s}_{1,0})\) 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 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 \( f_B \) represents the operation.

\[
f_1(x_a) = \begin{cases} 0.8, & x_a = 0 \\ 0.2, & x_a = 1, \end{cases} \quad f_2(x_b) = \begin{cases} 0.7, & x_b = 0 \\ 0.3, & x_b = 1, \end{cases} \quad f_3(x_c) = \begin{cases} 0.9, & x_c = 0 \\ 0.1, & x_c = 1. \end{cases}
\]

\[
f_B(x_a, x_b, x_c) = \begin{cases} 1, & \text{if } x_a \oplus x_b = x_c \\ 0, & \text{otherwise}, \end{cases}
\]

**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 \((\tilde{s}_{0,m}, \tilde{s}_{1,n})\):

\[ p_{0,m} \geq p_{0,m'} \land p_{1,n} \geq p_{1,n'} \quad \Rightarrow \quad p_{0,m} \times p_{1,n} \geq p_{0,m'} \times p_{1,n'}, \quad \forall m' \geq m \land \forall n' \geq n. \]

Therefore, the top-leftmost block, representing \((\tilde{s}_{0,1}, \tilde{s}_{1,0})\), 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.
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 $F$; 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.
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, \(R^2_0, R^2_1, R^2_2, R^2_3\), we applied their sum \(\sum R^2_j\) 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 AES.
The detection for IV and N is not needed since they are known values.

<table>
<thead>
<tr>
<th>Lane</th>
<th>Init.</th>
<th>L1 Low</th>
<th>L1 High</th>
<th>L2 Low</th>
<th>L2 High</th>
<th>Fin.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>L2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Fin</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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

Our target [Wea21, ASCON/1] 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., \( \tilde{\beta}_1 \) or \( \tilde{\beta}_{11} \)), 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 \( \beta_1 \) of Initialization, since these two lanes are loaded with \( K \), which is used four times in the full encryption. Figure 4 shows the \( \Sigma_f R^2_f \) values for the H/L words from \( L_1 \) and \( L_2 \) in \( \beta_1 \) of Initialization 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 \( x \) 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 al.

\footnote{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 \( \geq 0.02 \) 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-Ωs experiment, using both the top-rank success rate (1-SR) and the base-2 logarithm of the guessing entropy (LGE).

<table>
<thead>
<tr>
<th>word</th>
<th>high</th>
<th>low</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$L_1$ of Init. input</td>
<td>1-SR</td>
<td>0.839</td>
</tr>
<tr>
<td>(1st lane of $K$)</td>
<td>LGE</td>
<td>0.152</td>
</tr>
<tr>
<td>$L_2$ of Init. input</td>
<td>1-SR</td>
<td>0.791</td>
</tr>
<tr>
<td>(2nd lane of $K$)</td>
<td>LGE</td>
<td>0.294</td>
</tr>
</tbody>
</table>

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-Ωs experiment).

<table>
<thead>
<tr>
<th>lane</th>
<th>all interesting clock cycles</th>
<th>$L_1$ of Fin. $β_{11}$</th>
<th>$L_1$ of Fin. $β_{11}$</th>
<th>$L_4$ of Fin. $β_{11}$</th>
<th>$L_6$ of Init. $α_{11}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1-SR</td>
<td>0.839</td>
<td>0.809</td>
<td>0.852</td>
<td>0.733</td>
<td>0.804</td>
</tr>
<tr>
<td>LGE</td>
<td>0.152</td>
<td>0.215</td>
<td>0.203</td>
<td>0.419</td>
<td>0.299</td>
</tr>
<tr>
<td>3.287</td>
<td>3.644</td>
<td>2.624</td>
<td>2.479</td>
<td>2.833</td>
<td>3.295</td>
</tr>
<tr>
<td>0.086</td>
<td>0.099</td>
<td>0.098</td>
<td>0.092</td>
<td>0.092</td>
<td>0.093</td>
</tr>
<tr>
<td>0.096</td>
<td>0.112</td>
<td>0.125</td>
<td>0.109</td>
<td>0.108</td>
<td>0.109</td>
</tr>
<tr>
<td>0.083</td>
<td>0.095</td>
<td>0.099</td>
<td>0.102</td>
<td>0.103</td>
<td>0.094</td>
</tr>
<tr>
<td>0.100</td>
<td>0.112</td>
<td>0.125</td>
<td>0.109</td>
<td>0.108</td>
<td>0.109</td>
</tr>
<tr>
<td>0.100</td>
<td>0.112</td>
<td>0.125</td>
<td>0.109</td>
<td>0.108</td>
<td>0.109</td>
</tr>
<tr>
<td>0.096</td>
<td>0.112</td>
<td>0.125</td>
<td>0.109</td>
<td>0.108</td>
<td>0.109</td>
</tr>
<tr>
<td>0.096</td>
<td>0.112</td>
<td>0.125</td>
<td>0.109</td>
<td>0.108</td>
<td>0.109</td>
</tr>
<tr>
<td>LGE</td>
<td>0.294</td>
<td>0.341</td>
<td>0.225</td>
<td>0.321</td>
<td>0.167</td>
</tr>
</tbody>
</table>

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 ($S, x_{b,proj}$ 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 $L(x_{a,proj}|x_{b,proj}|, S)$.

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, $α_{16}$ 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 $\beta_{-1}$, state 1 is $\alpha_0$, state 2 is $\beta_0$, 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: $\beta_{\Omega-1}$, $\alpha_{\Omega}$, $\beta_{\Omega}$, and two types of constraint factors: $f_s$ and $f_l$. 
Initialization Finalization

\[ \text{XORed with the state} \]

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

Likewise, the variables for the last 128 bits of the Finalization output are connected with the variables for \( K \) and \( T \) via \( f_{\oplus} \).

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, \( L_0' \leftarrow \Sigma_\Omega(L_0) \) 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 \( \beta_{11} \)) of \( p^{12} \) in Initialization, XORed with the state \( P \parallel (0x80) \parallel K \parallel K' \), 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 \( P \parallel (0x80) \). Similarly, bits from \( L_1 \parallel L_2 \) of the two states will be connected to the variables for \( K \) via constraint factor \( f_\oplus \). Bits from \( L_3 \parallel L_4 \) 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 $f_{\text{ext}}$, 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 $\beta_{11}$ 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 $n^\text{th}$-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).

<table>
<thead>
<tr>
<th># attack traces</th>
<th>no BP 8 bit</th>
<th>loopy BP 8 bit</th>
<th>tree BP 8 bit</th>
<th>tree BP 16 bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12.2189</td>
<td>11.6443</td>
<td>4.4958</td>
<td>2.3795</td>
</tr>
<tr>
<td>2</td>
<td>4.9610</td>
<td>1.3345</td>
<td>0.6850</td>
<td>0.2850</td>
</tr>
<tr>
<td>3</td>
<td>3.2118</td>
<td>0.1675</td>
<td>0.2439</td>
<td>0.1028</td>
</tr>
<tr>
<td>4</td>
<td>2.4741</td>
<td>0.1069</td>
<td>0.1176</td>
<td>0.0498</td>
</tr>
<tr>
<td>5</td>
<td>2.1625</td>
<td>0.0453</td>
<td>0.0851</td>
<td>0.0336</td>
</tr>
<tr>
<td>6</td>
<td>1.9403</td>
<td>0.0516</td>
<td>0.0691</td>
<td>0.0316</td>
</tr>
<tr>
<td>7</td>
<td>1.8147</td>
<td>0.0220</td>
<td>0.0965</td>
<td>0.0222</td>
</tr>
<tr>
<td>8</td>
<td>1.7075</td>
<td>0.0000</td>
<td>0.0473</td>
<td>0.0156</td>
</tr>
<tr>
<td>9</td>
<td>1.6175</td>
<td>0.0078</td>
<td>0.0379</td>
<td>0.0122</td>
</tr>
<tr>
<td>10</td>
<td>1.5514</td>
<td>0.0261</td>
<td>0.0352</td>
<td>0.0076</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th># attack traces</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>no BP 8 bit</td>
<td>31.043</td>
<td>37.9218</td>
<td>41.5498</td>
<td>42.6815</td>
<td>31.5794</td>
<td>30.3890</td>
<td>30.3890</td>
<td>29.9637</td>
<td>29.6220</td>
<td>29.4119</td>
</tr>
</tbody>
</table>

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 $2^{26}$ 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.

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 $K$. 

(a) U-Os experiment

(b) U-O3 experiment
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.

<table>
<thead>
<tr>
<th>traces</th>
<th>region</th>
<th>all four</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>U-Os</td>
<td></td>
<td>4.4958</td>
<td>50.6751</td>
<td>44.5369</td>
<td>48.3660</td>
<td>41.6838</td>
</tr>
<tr>
<td>U-O3</td>
<td></td>
<td>35.8763</td>
<td>69.3538</td>
<td>76.4909</td>
<td>73.4121</td>
<td>88.3132</td>
</tr>
</tbody>
</table>

For example, it XORs two lanes of Finalization output ($\beta_{11}$) 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 $\beta_{11}$ 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
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 $T^A := K^A \oplus \beta^A_{11}$, $T^B := K^B \oplus \beta^B_{11}$, and finally $T := T^A \oplus T^B$.

Therefore, we cannot build templates for the fragments of $\beta_{11}$ 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, $\beta_{-1}$ 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 $\mathbb{M}$-0a experiment.

<table>
<thead>
<tr>
<th>target state</th>
<th>$\mathbb{K}$</th>
<th>$\mathbb{K}^*$</th>
<th>$\mathbb{K}^\beta$</th>
<th>Fin. $\beta_{1[0]}^\beta$</th>
<th>Fin. $\beta_{1[1]}^\beta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lane 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>high/low</td>
<td>21 26 113 109</td>
<td>161 157 41 42 46 40</td>
<td>213 207 30 33 28 39</td>
<td></td>
<td></td>
</tr>
<tr>
<td>even/odd</td>
<td>16 28 113 106</td>
<td>217 205 30 33 28 39</td>
<td>203 174 36 36 44 43</td>
<td></td>
<td></td>
</tr>
<tr>
<td>union</td>
<td>49 154</td>
<td>203 174 36 36 44 43</td>
<td>203 174 36 36 44 43</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lane 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>high/low</td>
<td>20 26 109 109</td>
<td>203 174 36 36 44 43</td>
<td>203 174 36 36 44 43</td>
<td></td>
<td></td>
</tr>
<tr>
<td>even/odd</td>
<td>30 34 114 116</td>
<td>203 174 36 36 44 43</td>
<td>203 174 36 36 44 43</td>
<td></td>
<td></td>
</tr>
<tr>
<td>union</td>
<td>35 139</td>
<td>203 174 36 36 44 43</td>
<td>203 174 36 36 44 43</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 9: Quality evaluation of the templates from in the $\mathbb{M}$-0s experiment, using both the top-rank success rate (1-SR) and the base-2 logarithm of the guessing entropy (LGE).

<table>
<thead>
<tr>
<th>byte</th>
<th>Lane 1 even word</th>
<th>Lane 1 odd word</th>
<th>Lane 2 even word</th>
<th>Lane 2 odd word</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathbb{K}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0.023</td>
<td>0.024</td>
<td>0.024</td>
<td>0.024</td>
</tr>
<tr>
<td>1</td>
<td>0.023</td>
<td>0.024</td>
<td>0.024</td>
<td>0.024</td>
</tr>
<tr>
<td>2</td>
<td>0.023</td>
<td>0.024</td>
<td>0.024</td>
<td>0.024</td>
</tr>
<tr>
<td>3</td>
<td>0.023</td>
<td>0.024</td>
<td>0.024</td>
<td>0.024</td>
</tr>
<tr>
<td>$\mathbb{K}^\beta$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0.023</td>
<td>0.024</td>
<td>0.024</td>
<td>0.024</td>
</tr>
<tr>
<td>1</td>
<td>0.023</td>
<td>0.024</td>
<td>0.024</td>
<td>0.024</td>
</tr>
<tr>
<td>2</td>
<td>0.023</td>
<td>0.024</td>
<td>0.024</td>
<td>0.024</td>
</tr>
<tr>
<td>3</td>
<td>0.023</td>
<td>0.024</td>
<td>0.024</td>
<td>0.024</td>
</tr>
<tr>
<td>Fin. $\beta_{1[0]}^\beta$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0.023</td>
<td>0.024</td>
<td>0.024</td>
<td>0.024</td>
</tr>
<tr>
<td>1</td>
<td>0.023</td>
<td>0.024</td>
<td>0.024</td>
<td>0.024</td>
</tr>
<tr>
<td>2</td>
<td>0.023</td>
<td>0.024</td>
<td>0.024</td>
<td>0.024</td>
</tr>
<tr>
<td>3</td>
<td>0.023</td>
<td>0.024</td>
<td>0.024</td>
<td>0.024</td>
</tr>
<tr>
<td>Fin. $\beta_{1[1]}^\beta$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0.023</td>
<td>0.024</td>
<td>0.024</td>
<td>0.024</td>
</tr>
<tr>
<td>1</td>
<td>0.023</td>
<td>0.024</td>
<td>0.024</td>
<td>0.024</td>
</tr>
<tr>
<td>2</td>
<td>0.023</td>
<td>0.024</td>
<td>0.024</td>
<td>0.024</td>
</tr>
<tr>
<td>3</td>
<td>0.023</td>
<td>0.024</td>
<td>0.024</td>
<td>0.024</td>
</tr>
</tbody>
</table>

Table 10: Logarithmic guessing entropy achieved by our key-recovery attack with 8-bit fragment templates, using 1–100 encryption traces, respectively (tree BP, $\mathbb{M}$-0a experiment). (The no BP control experiment, using only templates for $K$, found hardly any exploitable first-order leakage of $K$ from the targeted implementation.)

<table>
<thead>
<tr>
<th># attack traces</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>7</th>
<th>10</th>
<th>20</th>
<th>50</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>tree BP $\mathbb{K}$-bit</td>
<td>0.20</td>
<td>0.24</td>
<td>0.28</td>
<td>0.32</td>
<td>0.36</td>
<td>0.40</td>
<td>0.44</td>
<td>0.48</td>
<td>0.52</td>
<td>0.56</td>
</tr>
<tr>
<td>no BP $\mathbb{K}$-bit</td>
<td>0.09</td>
<td>0.13</td>
<td>0.17</td>
<td>0.21</td>
<td>0.25</td>
<td>0.30</td>
<td>0.35</td>
<td>0.40</td>
<td>0.45</td>
<td>0.50</td>
</tr>
</tbody>
</table>

Figure 13: Key-recovery success rates in the $\mathbb{M}$-0s 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 \cite{BBC20, 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 \cite{U-Os, U-03}, or not entirely effective against a template attack \cite{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. \cite{LWL22} 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 \cite{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 \cite{U-03}, 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 \cite{YK22} for SHA-3 \cite{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. \cite{CDSU23}) for the internal states of the Ascon permutation. Our templates for these (e.g., Init. $\alpha_6$ in Table 3 and Figure 5) were less effective than those reported for the Keccak permutation in \cite[Table 2]{YK22}. 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 $\theta$ 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:

\url{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.


