A Template Attack to Reconstruct the Input of SHA-3 on an 8-bit Device

Shih-Chun You* and Markus G. Kuhn

Department of Computer Science and Technology,
University of Cambridge, Cambridge CB3 0FD, UK
{scy27,mgk25}@cl.cam.ac.uk

Abstract. We present an enumeration procedure based on a template attack to recover the complete input text of a SHA-3 implementation on an 8-bit microprocessor from a single trace of a power-analysis side channel. This attack targets 600 bytes of triple-redundant internal state in each invocation of the permutation used by SHA-3. We first build templates that can generate for each of these bytes a rank table of all 256 candidates. The templates we obtained for our 8-bit target CPU nearly identified the correct value of most target bytes directly, rather than just gathering information about their Hamming weights. We then search the full intermediate state of the Keccak permutation to eliminate remaining uncertainties about the recovered byte values. From the resulting intermediate states we finally reconstruct both the input and output of SHA-3 and verify the output. In our experimental evaluation of this procedure we achieved success rates higher than 99%.

Keywords: Template attack · SHA-3 · Keccak · enumeration trees.

1 Introduction

In 2015, the National Institute of Standards and Technology (NIST) standardized *Secure Hash Algorithm 3* (SHA-3) [16], which is based on the Keccak sponge function and the Keccak-f permutation designed by Bertoni et al. [2,3]. Keccak-f consists of multiple rounds, each of which consists of five steps known as θ, ρ, π, χ and ι. The Keccak-f permutation is not only the main building block of the SHA-3 family of hash functions, but is also used in the SHAKE family of extendable-output functions, and can be used in many other contexts, such as key-derivation functions, message-authentication codes, and key-agreement schemes (e.g., NewHope [1]), where either its inputs or outputs can be confidential data for which side-channel attacks may be a concern.

Previous papers discussed side-channel attacks to recover keys used in the generation of Keccak-based message authentication codes (MAC-Keccak). Taha and Schaumont mainly used Differential Power Analysis (DPA) to attack step θ to recover a fixed-length key and discussed the relationship between key-length

* supported by the Cambridge Trust and the Ministry of Education, Taiwan

presented at COSADE 2020, 5–7 October 2020, to appear in LNCS

and the DPA resilience of MAC-Keccak [23]. They later applied similar attacks to recovering MAC-Keccak keys with arbitrary length [22]. Luo et al. modified this attack to determine the intermediate state after a complete round of Keccak-f [10], applying DPA after the non-linear step χ.

Such DPA-style attacks can effectively recover a MAC-Keccak key K, but they do not extend to other applications where there is no fixed key K, as they require leakage traces of many thousand repeated executions of SHA-3($K\|M$) with known variable input message M. For example, a DPA-style attack could not reconstruct the complete input of MAC-Keccak. Instead, we focus here on attacking a *single* invocation of Keccak-f in order to reconstruct both its input and output. To achieve this, we require a template attack (TA) [4]. We then use this capability to demonstrate recovery of a complete SHA-3 input given a single power trace and then verify the results with the given output of SHA-3. Our technique therefore not only can recover MAC-Keccak keys of arbitrary length without prior knowledge of the message M. It naturally also extends to other Keccak-f applications with confidential inputs or outputs, such as random-bit generation.

Since each step of the Keccak-f permutation is invertible, given its full output state we can calculate the input state of the step. Likewise, if we can determine a complete intermediate state, we can calculate from that both the input and output of the entire permutation. Having reconstructed the output of one Keccak-f invocation and the input of the next, we merely have to XOR these together in order to reconstruct one block of input of a SHA-3 execution.

We first tried to use the template attack to determine the value of every byte in a single full intermediate state. However, there is no room for mistakes: the diffusion of Keccak-f means that even a single bit error will result in a completely different input or output. Therefore, we combined a kind of template attack with an enumeration technique around the mathematical structure of Keccak-f to correct errors. We first use a template attack to estimate the likelihood of each of the 256 possible values of each byte in *three* consecutive intermediate states. Since the state of Keccak-f contains 200 bytes (1600 bits), there will be 200 per-byte rank tables associated with each observed intermediate state, that is 600 rank tables in total. In a pair of (nearly) consecutive intermediate states, each byte will only depend on a small number of bytes in neighboring states: the avalanche effect takes multiple rounds to come into effect. This makes it possible to eliminate errors by combining likelihood information from neighboring intermediate states and using the result to build rank tables for combinations of bytes. We repeat this until we obtain the (top of the) rank table for the entire state.

In this paper, we discuss the details of the template attack we performed on SHA-3 to obtain rank tables of all bytes of three consecutive intermediate states (Section 4), and then present the search procedure we used to recover the complete intermediate states (Section 5). Finally, we evaluate the success probability of recovering the inputs of SHA-3 by this method (Section 5.4).

2 Preliminaries and notation

2.1 Keccak-f[1600] and SHA-3

Our terminology and notation related to SHA-3 and the Keccak-f permutation closely follow NIST FIPS 202 [16]. The SHA-3 algorithm is based on the Keccak-f[1600] permutation, which consists of a sequence of five steps that iterates 24 times on a 1600-bit state.

Each of the steps θ, ρ, π, χ and ι results in an intermediate state of 1600 bits. In this paper, we refer to these intermediate states as α_ω, α'_ω, β_ω and β'_ω as follows:

$$\textbf{Input} \xrightarrow{\theta} \alpha_0 \xrightarrow{\rho,\pi} \alpha'_0 \xrightarrow{\chi} \beta_0 \xrightarrow{\iota} \beta'_0 \xrightarrow{\theta} \alpha_1 \xrightarrow{\rho,\pi} \cdots \xrightarrow{\chi} \beta_{23} \xrightarrow{\iota} \textbf{Output}$$

The round index ω runs from 0 to 23 in Keccak-f[1600]. We use the term *intermediate byte* to refer to one of the 200 bytes in an intermediate state of Keccak-f[1600]. The SHA-3 standard describes these states as a $5 \times 5 \times 64$-bit cube with an x, y and z axis. Since we used an 8-bit processor in our experiments, we refer to the 64 bits along the z axis as 8 bytes. For example, we describe an intermediate byte in state α_0 as $\alpha_0[i,j,k]$, where i,j,k are the x, y, z coordinates with $0 \leq i \leq 4$, $0 \leq j \leq 4$, $0 \leq k \leq 7$. The least significant bit in this byte we denote by $\alpha_0[i,j,k][0]$ and its most significant bit by $\alpha_0[i,j,k][7]$. We call the five bytes with the same y and z coordinates a *byte row*, and the 25 bytes with the same z coordinate a *byte slice*.

All five steps in a Keccak-f[1600] round are practical to invert [2] and the Keccak team provides C++ implementations of the corresponding inverse functions [9]. In other words, the input, output, and all intermediate states of a Keccak-f[1600] execution can be converted into each other efficiently.

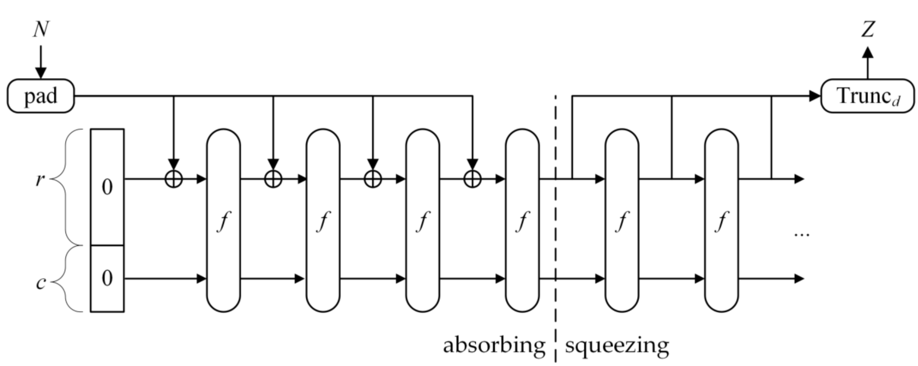

Fig. 1. The diagram of the Keccak sponge function from NIST FIPS 202 [16]. In this diagram, N is the arbitrary-length input sequence and Z is the d-bit output sequence.

The Keccak[c](N, d) function is based on the Keccak-f[1600] permutation [16]. It first "absorbs" an arbitrary-length input bit sequence into its internal state and then can "squeeze" out an arbitrary-length output bit sequence, and so is described as a "sponge function". Figure 1 shows how Keccak[c](N, d) absorbs the input bit string N and squeezes out a d-bit result. Input message N is first padded and then split into blocks of $r = 1600 - c$ bits, where parameter c is called the *capacity* and parameter r the *rate*. The input and output of Keccak-f[1600] each consist of $r + c = 1600$ bits, which we denote accordingly by $\mathbf{R}\|\mathbf{C}$. After all r-bit blocks have been absorbed, in the squeezing stage the output sequence is generated by concatenating the $\mathbf{R}$ fragment being output by each iteration of Keccak-f[1600] until the concatenated sequence is at least of the required length d, and it is then truncated to d bits.

The SHA-3 family is finally defined for input messages M using Keccak[c] for the output sizes $d \in \{224, 256, 384, 512\}$ bits as

$$\textbf{SHA3-}d(M) = \textbf{Keccak}[2d](M\|01, d).$$

In addition, SHA-3 defines two extendable-output functions (XOFs) as

$$\begin{aligned}\textbf{SHAKE128}(M, d) &= \textbf{Keccak}[256](M\|1111, d)\\ \textbf{SHAKE256}(M, d) &= \textbf{Keccak}[512](M\|1111, d)\end{aligned}$$

where users have free choice over the output length d.

We ran all experiments in this paper on **SHA3-512**(M) because this is the SHA-3 algorithm with the largest capacity c, i.e. the largest security margin. The technique works equally well on the other SHA-3 algorithms.

2.2 Template attack

The traditional template attack. Chari et al. introduced a powerful side-channel exploitation technique called Template Attack (TA) [4]. It consists of two stages, profiling and attack. During profiling, we build templates that model the leakage traces of different candidate secrets from traces recorded while a known secret is processed. Then, we record an attack trace while an unknown secret is processed. We compare that with all the templates, and predict the secret as the candidate with the template most similar to the attack trace.

In this approach, attackers need to collect a sizable number of profiling traces. These will be separated into subsets according to the secret value targeted. If we target one intermediate byte, the number of subsets will be 256. From the trace subset corresponding to intermediate byte b, we construct a template consisting of an expected trace $\bar{\mathbf{x}}_b \in \mathbb{R}^m$ and a covariance matrix $\mathbf{S}_b \in \mathbb{R}^{m\times m}$, as

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

where n_b is the number of profiling traces in this subset, and $\mathbf{x}_{b,t}$ is the t^{th} profiling trace with corresponding intermediate byte b, each trace containing m points in time.

Later, when we obtain an attack trace $\mathbf{x}_{\mathrm{a}}$, we can calculate as a likelihood function a probability-density value for each template with

$$f(\mathbf{x}_{\mathrm{a}}|\bar{\mathbf{x}}_b, \mathbf{S}_b) = \frac{1}{\sqrt{(2\pi)^m |\mathbf{S}_b|}} \exp\left(-\frac{1}{2}(\mathbf{x}_{\mathrm{a}} - \bar{\mathbf{x}}_b)^{\mathsf{T}} \mathbf{S}_b^{-1} (\mathbf{x}_{\mathrm{a}} - \bar{\mathbf{x}}_b)\right).$$

Then we can sort the 256 results into a rank table, where the top entry is the most likely candidate.

The template attack with stochastic models. The previous approach, where the arithmetic mean of the traces in *each* subset is used to estimate their expected value, needs a large total number of profiling traces. Based on the stochastic model $\mathcal{F}_9$ by Schindler et al. [19], Choudary and Kuhn used an alternative solution [6]. They treat each bit, $b[0]$ to $b[7]$, in the targeted intermediate byte as an independent variable and then use multivariate linear regression to calculate coefficients c_0 to c_7 and a constant c_8 for predicting the expected values of single points on a trace as $\hat{x}_b = \sum_{l=0}^{7}(b[l] \cdot c_l) + c_8$ and equivalently as

$$\hat{\mathbf{x}}_b = \sum_{l=0}^{7}(b[l] \cdot \mathbf{c}_l) + \mathbf{c}_8$$

for an entire trace, where $\mathbf{c}_0, \ldots, \mathbf{c}_8 \in \mathrm{R}^m$ are the vectors of coefficients and constants previously estimated by multivariate linear regression.

They also modified the way to calculate the covariance matrices $\mathbf{S}_b$ as

$$\mathbf{S}_b = \frac{1}{n_b - 1}\sum_{t=1}^{n_b}(\mathbf{x}_{b,t} - \hat{\mathbf{x}}_b)(\mathbf{x}_{b,t} - \hat{\mathbf{x}}_b)^{\mathsf{T}}, \quad \mathbf{S}_{\mathrm{pooled}} = \frac{1}{\sum_{b=0}^{255} n_b}\sum_{b=0}^{255}(n_b - 1)\mathbf{S}_b.$$

Instead of a different $\mathbf{S}_b$ in each template, they used one single *pooled* covariance matrix estimate, $\mathbf{S}_{\mathrm{pooled}}$, which is the weighted average of the $\mathbf{S}_b$, because previous studies [17,8] had suggested this is a more effective estimate when the actual covariance matrix can be assumed to be independent of the targeted value b. The function to calculate the probability density value then becomes

$$f(\mathbf{x}_{\mathrm{a}}|\hat{\mathbf{x}}_b, \mathbf{S}_{\mathrm{pooled}}) = \frac{1}{\sqrt{(2\pi)^m |\mathbf{S}_{\mathrm{pooled}}|}} \exp\left(-\frac{1}{2}(\mathbf{x}_{\mathrm{a}} - \hat{\mathbf{x}}_b)^{\mathsf{T}} {\mathbf{S}_{\mathrm{pooled}}}^{-1} (\mathbf{x}_{\mathrm{a}} - \hat{\mathbf{x}}_b)\right)$$

Data compression with linear discriminant analysis. Choudary and Kuhn also integrated Fisher's Linear Discriminant Analysis (LDA), as proposed by Standaert and Archambeau [20], into their approach [6]. This is a procedure to project the traces onto a subspace with higher signal-to-noise ratio (SNR), as determined by two covariance matrices $\mathbf{B}$ and $\mathbf{\Sigma}$, where $\mathbf{B}$ is the inter-class scatter representing the signal, while $\mathbf{\Sigma}$ is the total intra-class scatter representing the noise. When recovering 8-bit secrets, these two matrices can be calculated from the profiling traces as

$$\mathbf{B} = \frac{1}{\sum_{b=0}^{255} n_b} \sum_{b=0}^{255} n_b (\hat{\mathbf{x}}_b - \bar{\mathbf{x}})(\hat{\mathbf{x}}_b - \bar{\mathbf{x}})^\mathsf{T},$$

$$\mathbf{\Sigma} = \frac{1}{\sum_{b=0}^{255} n_b} \sum_{b=0}^{255} \sum_{t=1}^{n_b} (\mathbf{x}_{b,t} - \hat{\mathbf{x}}_b)(\mathbf{x}_{b,t} - \hat{\mathbf{x}}_b)^\mathsf{T} = \mathbf{S}_\text{pooled},$$

where $\bar{\mathbf{x}} = 256^{-1} \sum_{b=0}^{255} \hat{\mathbf{x}}_b = \mathbf{c}_8 + \frac{1}{2} \sum_{l=0}^{7} \mathbf{c}_l$ is the arithmetic mean of the expected values $\hat{\mathbf{x}}_b$.

We then build a matrix $\mathbf{A} \in \mathbb{R}^{m \times m'}$ where the columns are the m' normalized eigenvectors of the matrix $\mathbf{\Sigma}^{-1}\mathbf{B}$ corresponding to its m' largest eigenvalues (see also [7, footnote 6]). The LDA projection of a raw trace $\mathbf{x}_\text{a}$ onto the resulting m'-dimensional subspace is then $\mathbf{x}_\text{proj} = \mathbf{A}^\mathsf{T}\mathbf{x}_\text{a}$.

In our experiments, we follow Choudary and Kuhn's approach [6] as outlined above, firstly using multivariate linear regression to build matrices $\mathbf{\Sigma}$ and $\mathbf{B}$, secondly calculating the projection matrix $\mathbf{A}$, then using that to project all profiling traces onto the subspace with high SNR. From these projected traces, we then build very compact templates, again using multivariate linear regression. The resulting template information consists of a new covariance matrix $\mathbf{S}_\text{proj} \in \mathbb{R}^{m' \times m'}$, 256 new expected traces $\hat{\mathbf{x}}_{b,\text{proj}} \in \mathbb{R}^{m'}$, along with $\mathbf{A}$.

2.3 Combining multiple likelihood tables

With ideal templates, attackers should find the full state of a secret by simply taking the most likely candidate from each part of the secret and concatenating them. However, template attacks are noise sensitive, so the correct candidate will not always top the rank table. Therefore, Veyrat-Charvillon et al. introduced an optimal key enumeration algorithm to search the correct key across several ranked likelihood tables of the sub-keys of AES [24]. Given two rank tables in descending order of likelihood, each with 2^8 values, there will be 2^{16} possible combinations. Their approach searches the 2^{16} possible combinations in descending order of their joint likelihood until the correct combination is found, without calculating the joint likelihoods of all 2^{16} combinations. They generalized this method using a recursive tree structure that combines two tables at a time to combine the results of more than two rank tables. With this algorithm, it becomes practical to search the correct combination of the sub-keys when correct candidates do not top the tables. This increases the noise resiliency of the attack significantly.

We applied their method in our experiments to enumerate the intermediate states instead of any key. We will refer to their tree-structured algorithm as an *enumeration tree* in this paper.

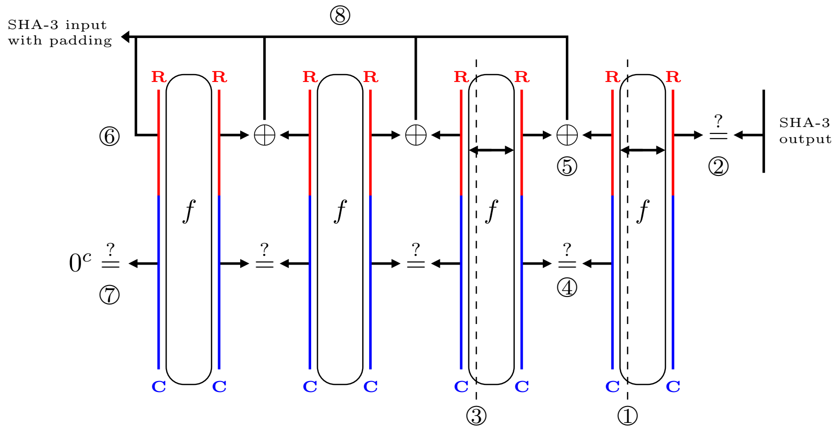

Fig. 2. The procedure to reconstruct SHA-3 inputs by template attack: ① reconstruct an intermediate state of the last Keccak-f[1600] permutation and calculate its input and output; ② verify the correctness by checking whether the first 512 bits in the output match the SHA3-512 output; ③ repeat ① on other permutations but ④ verify the correctness by checking whether the **C** of the output matches that of the input in the following permutation; ⑤ XOR the **R** of the two consecutive permutations to calculate each part of the SHA-3 input; ⑥ in the special case of the first r bits of the SHA-3 input, that part is identical to the **R** part of the input of the first Keccak-f[1600] permutation and ⑦ the **C** part of that permutation should be c 0 bits; ⑧ concatenate each part to form the complete SHA-3 input with padding.

3 Attack Strategy

Because of the invertibility of every step in Keccak, attackers can access not only the output but also the input of a Keccak-f[1600] permutation once they obtain any intermediate state. Figure 2 depicts how we can use this to recover an input of SHA3-512.

First, we use template attacks to reconstruct all the bytes in an intermediate state of the last Keccak-f[1600] permutation. After, for example, state α'_0 is reconstructed, we can calculate the inverses of π, ρ, and θ to find out the input of this Keccak-f[1600] permutation, and then its output. We can verify the correctness of the latter by checking whether its first 512 bits match the SHA3-512 output.

Second, we can repeat what we have done on the last Keccak-f[1600] permutation for its predecessor, and verify the correctness of its output by checking whether its last $c = 1024$ bits **C** match those of the input of its successor. The input of the first Keccak-f[1600] permutation has **C** equal to an all-zero string.

Third, we can calculate each part of the SHA-3 input by XOR-ing the **R** part of the input and the output of two consecutive Keccak-f[1600] permutations. In the special case of the first r bits of the SHA-3 input, that is identical to the **R** part of the input of the first Keccak-f[1600] permutation. Finally, after concatenating all the parts and removing the padding, the input of SHA3-512 is recovered.

To target SHAKE128 or SHAKE256, we only need to attack permutations in the absorbing stage, as the squeezing stage fully depends on the output of the former, and recall that SHAKE uses slightly different padding.

4 Template Attack on SHA-3

Now the problem remains how to successfully recover at least one intermediate state in each invocation of the Keccak-$f[1600]$ permutation in the SHA3-512 procedure by template attack. We chose the intermediate states α'_0, β_0, and α_1 to build our templates, in order to cover a non-linear step (χ) while limiting the dependency on bits from other slices. (Any other choice of target round should work equally well.)

4.1 Target hardware device and measurement setup

Our SHA3-512 implementation is based on the Keccak-$f[1600]$ implementation in the official C reference code, the Extended Keccak Code Package [25]. We ran it on a power-analysis test board designed by Choudary [5, Section 2.2.2].

The target processor is the 8-bit microcontroller ATxmega256A3U [12]. We supply it with an external 2 MHz square wave clock signal generated by a National Instruments PXIe-5423 [15] wave generator that is configured to use the same reference clock as the NI PXIe-5160 [14] oscilloscope that we used to record the traces of power consumption. This way, with a sampling rate of 250 MHz, each clock cycle contains exactly 125 data points, with phase jitter about 8 ps standard deviation. The power supply was an NI PXI-4110 [13].

We recorded 32 000 profiling traces and 1000 evaluation traces of the Keccak-$f[1600]$ permutation with random inputs to build the templates and evaluate their quality. For testing, we also recorded two sets of SHA3-512 traces. The first one contains 1000 random inputs with length shorter than 71 bytes, so it needs one Keccak-$f[1600]$ permutation to absorb the input. The second set contains 1000 random inputs whose lengths range from 216 to 287 bytes, so they need four Keccak-$f[1600]$ permutations to be absorbed. Since our target states are α'_0, β_0, and α_1, we only recorded the traces covering the power consumption of the first two rounds of one Keccak-$f[1600]$ permutation, and each raw trace contained 40 000 clock cycles or 5 000 000 samples.

4.2 Interesting clock cycle detection

Since our raw traces were too long for building templates directly, we first determined the clock cycles that contain information about the targeted intermediate states, which in the Keccak-$f[1600]$ permutation each contain 200 intermediate bytes. We tested each clock cycle to find out whether it is related to any of the intermediate bytes we target. We used the 8 bits in the intermediate bytes as 8 binary variables in a multivariate linear regression to analyze their correlation with the peak current in each clock cycle.

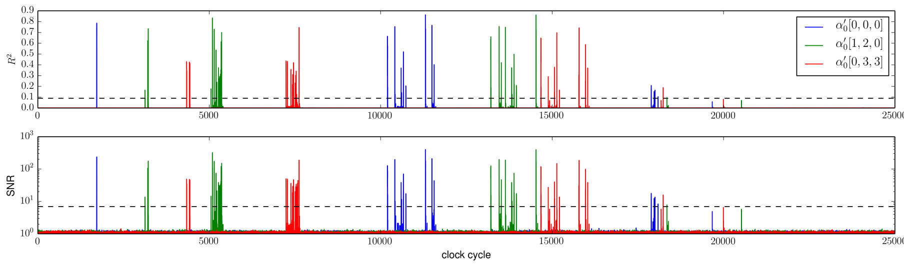

Fig. 3. Comparison of the highest R^2 coefficient and SNR value in each clock cycle.

We decided whether the correlation is sufficiently high via the coefficient of determination (R^2), as estimated by the regression. The clock cycles with R^2 higher than a threshold were added to the set of interesting clock cycles. Since traditionally the interval $-0.3 < R < 0.3$ indicates a variable of low correlation, we selected clock cycles based on the threshold $R^2 > 0.09$. The multivariate linear regression and R^2 were calculated using the `LinearRegression` class in the Python library `scikit-learn` [18]. Fig. 3 shows the resulting highest R^2 value occuring in each clock cycle, along with SNR value [11]

$$\mathrm{SNR}(s) = \frac{\sum_{b=0}^{255} n_b(\bar{\mathbf{x}}_b[s] - \bar{\mathbf{x}}[s])^2}{\sum_{b=0}^{255}\sum_{t=0}^{n_b}(\mathbf{x}_{b,t}[s] - \bar{\mathbf{x}}_b[s])^2}$$

at each per-clock-cycle peak time s. (Our $R^2 > 0.09$ threshold is approximately equivalent to an SNR > 7 threshold.)

Let $\mathcal{A}'_{0,[i,j,k]}$ be the set of interesting clock cycles for intermediate byte $\alpha'_0[i,j,k]$, $\mathcal{B}_{0,[i,j,k]}$ that of $\beta_0[i,j,k]$, and $\mathcal{A}_{1,[i,j,k]}$ that of $\alpha_1[i,j,k]$. The clock cycles that leak these $3 \times 200 = 600$ intermediate bytes should be sufficient for building working templates, but we found a method to consider more clock cycles at the same time. Between the intermediate states α_0 and α'_0 are the steps ρ and π, which are both transposition steps. We give an example here how the eight bits in $\alpha'_0[2,1,1]$ match those from up to two bytes in α_0:

$$\begin{aligned}
\alpha'_0[2,1,1][0] &= \alpha_0[0,2,0][5], & \alpha'_0[2,1,1][1] &= \alpha_0[0,2,0][6],\\
\alpha'_0[2,1,1][2] &= \alpha_0[0,2,0][7], & \alpha'_0[2,1,1][3] &= \alpha_0[0,2,1][0],\\
\alpha'_0[2,1,1][4] &= \alpha_0[0,2,1][1], & \alpha'_0[2,1,1][5] &= \alpha_0[0,2,1][2],\\
\alpha'_0[2,1,1][6] &= \alpha_0[0,2,1][3], & \alpha'_0[2,1,1][7] &= \alpha_0[0,2,1][4].
\end{aligned}$$

Therefore we extend the set of interesting clock cycles for $\alpha'_0[2,1,1]$ from $\mathcal{A}'_{0,[2,1,1]}$ to $\mathcal{A}'_{0,[2,1,1]} \cup \mathcal{A}_{0,[0,2,0]} \cup \mathcal{A}_{0,[0,2,1]}$. This similarly applies to the intermediate state α_1, but the other way round.

Table 1. The number of interesting clock cycles for each byte in $\alpha'_0[i,j,k]$ (left) and $\beta_0[i,j,k]$ (right). The numbers for α_1 (omitted here) look similar to those for α'_0.

(i,j) \ k	0	1	2	3	4	5	6	7
(0, 0)	36	38	33	33	32	33	42	33
(1, 0)	112	114	102	96	100	109	98	106
(2, 0)	107	103	96	96	98	103	94	98
(3, 0)	115	122	103	84	78	89	92	103
(4, 0)	134	124	82	74	74	87	95	100
(0, 1)	110	116	102	94	80	91	93	105
(1, 1)	109	117	95	83	77	88	97	102
(2, 1)	107	87	75	75	72	82	94	108
(3, 1)	109	109	96	93	97	102	92	100
(4, 1)	118	112	97	93	88	106	122	121
(0, 2)	90	75	75	73	69	70	84	97
(1, 2)	113	99	82	73	77	85	98	110
(2, 2)	86	86	94	85	70	69	76	81
(3, 2)	50	38	35	33	32	30	51	37
(4, 2)	103	99	87	71	65	72	80	100
(0, 3)	99	101	98	91	82	88	91	97
(1, 3)	108	112	104	99	95	97	97	103
(2, 3)	110	99	77	73	70	78	89	96
(3, 3)	127	114	79	70	73	87	89	99
(4, 3)	44	44	45	41	46	45	60	45
(0, 4)	127	119	104	98	97	112	127	125
(1, 4)	117	109	98	92	96	110	112	111
(2, 4)	115	110	100	103	94	89	94	98
(3, 4)	87	88	88	87	98	95	86	83
(4, 4)	93	87	89	83	72	80	90	104

(i,j) \ k	0	1	2	3	4	5	6	7
(0, 0)	34	39	34	31	30	29	37	33
(1, 0)	25	26	23	23	30	26	32	27
(2, 0)	28	28	25	29	27	24	31	30
(3, 0)	26	32	30	25	27	24	34	28
(4, 0)	29	38	24	25	24	24	31	30
(0, 1)	27	25	25	27	24	24	34	29
(1, 1)	27	29	23	25	23	24	34	29
(2, 1)	27	28	23	25	24	27	36	37
(3, 1)	26	30	25	26	28	29	34	31
(4, 1)	30	29	24	27	28	22	34	35
(0, 2)	27	27	23	24	23	23	35	34
(1, 2)	30	24	22	24	21	21	29	30
(2, 2)	27	28	28	25	21	21	30	28
(3, 2)	32	24	23	24	23	23	30	31
(4, 2)	28	28	21	23	21	23	29	29
(0, 3)	28	26	26	29	26	26	33	28
(1, 3)	25	25	22	26	27	28	32	28
(2, 3)	32	26	23	25	25	25	35	33
(3, 3)	31	36	22	28	24	25	35	30
(4, 3)	30	29	25	27	29	29	45	34
(0, 4)	28	36	23	27	24	26	36	36
(1, 4)	27	32	25	25	27	29	42	30
(2, 4)	28	32	26	31	31	25	35	30
(3, 4)	27	29	25	30	28	22	35	28
(4, 4)	26	33	26	30	56	32	35	40

Table 1 lists the number of interesting clock cycles selected for each intermediate byte after that extension. In state α'_0, the numbers in lanes $(0,0)$, $(3,2)$, and $(4,3)$ are smaller because step ρ rotates the bits in these lanes by multiples of eight. For example, we always have $\alpha'_0[3,2,0] = \alpha_0[4,3,7]$, which implies that $\mathcal{A}'_{0,[3,2,0]} = \mathcal{A}_{0,[4,3,7]} = \mathcal{A}'_{0,[3,2,0]} \cup \mathcal{A}_{0,[4,3,7]}$, and that does not extend the set of clock cycles.

4.3 Building templates

Pre-processing. When targeting a specific byte, we select only the samples in the interesting clock cycle set of this byte. For example, when building the template for $\alpha'_0[2,1,1]$, the profiling traces reassembled this way cover 87 clock cycles with $87 \times 125 = 10875$ samples.

Since the 125 samples per clock cycle still lead to too long execution times for building the templates, we reduced the sampling rate further by a factor 5, averaging five consecutive samples into a new sample.

Templates with LDA compression. After the detection and pre-processing steps, we now have shorter traces for building templates for each of 600 bytes. We apply Choudary et al.'s method [6] (see Section 2.2). In the LDA compression, we chose only the first $m' = 8$ eigenvectors to form the projection matrices since the other eigenvalues are negligible. Besides the projection matrices, our templates therefore contain 8×8 covariance matrices and 8-point expected traces.

4.4 Evaluating the quality of templates

Having built the templates, we use the 1000 evaluation traces to estimate template quality, resulting in 600 rank tables for each evaluation trace.

As figures of merit, we use both the first-order success rate and the guessing entropy as defined by Standaert et al [21]. Table 2 shows the resulting success rates for states α'_0 and β_0, i.e. the fraction of these 1000 evaluation where the correct candidate topped the rank table. Table 3 shows the guessing entropy for each byte of states α'_0 and β_0, i.e. the average rank of the correct candidates in these 1000 evaluations (top rank = 1).

5 Searching the correct intermediate states

The results of the template evaluations show that it is improbable that all 200 bytes of an intermediate state can be directly recovered by combining only the top-ranking candidates. Therefore attackers will need a search scheme to find the correct combination of high-ranking candidates. One obvious choice is to build an enumeration tree [24] to successively combine the rank tables for individual target bytes into tables for larger byte sequences, until the high-ranking combinations of all 200 bytes of an intermediate state are determined. While this

Table 2. Success rates on $\alpha'_0[i,j,k]$ (left) and $\beta_0[i,j,k]$ (right). The rates for α_1 (omitted here) look similar to those for α'_0.

(i,j) \ k	0	1	2	3	4	5	6	7
(0, 0)	0.924	0.924	0.598	0.749	0.485	0.542	0.946	0.931
(1, 0)	0.995	0.994	0.931	0.957	0.971	0.965	0.999	0.991
(2, 0)	0.993	0.978	0.937	0.936	0.963	0.918	0.981	0.992
(3, 0)	0.999	0.997	0.983	0.787	0.771	0.878	0.967	0.969
(4, 0)	0.999	0.999	0.769	0.736	0.669	0.831	0.979	0.995
(0, 1)	1.000	1.000	0.982	0.956	0.846	0.780	0.999	0.986
(1, 1)	0.995	0.997	0.931	0.905	0.794	0.903	0.984	0.991
(2, 1)	1.000	0.925	0.811	0.819	0.655	0.879	0.987	0.998
(3, 1)	0.997	0.978	0.923	0.946	0.995	0.949	0.988	0.988
(4, 1)	1.000	0.975	0.877	0.921	0.896	0.943	0.998	1.000
(0, 2)	0.998	0.951	0.829	0.803	0.657	0.695	0.999	1.000
(1, 2)	0.998	0.997	0.836	0.726	0.669	0.838	0.995	0.998
(2, 2)	0.972	0.989	0.984	0.853	0.719	0.664	0.969	0.990
(3, 2)	0.998	0.816	0.642	0.536	0.579	0.616	0.973	0.991
(4, 2)	0.997	0.977	0.810	0.679	0.677	0.747	0.984	0.997
(0, 3)	1.000	1.000	0.968	0.945	0.816	0.846	0.994	0.980
(1, 3)	0.990	0.996	0.941	0.979	0.959	0.945	0.988	0.994
(2, 3)	0.999	0.942	0.823	0.728	0.703	0.658	0.986	1.000
(3, 3)	0.999	1.000	0.732	0.715	0.632	0.834	0.964	0.994
(4, 3)	0.911	0.878	0.791	0.759	0.850	0.972	0.997	0.987
(0, 4)	1.000	1.000	0.897	0.889	0.880	0.961	1.000	1.000
(1, 4)	1.000	0.998	0.879	0.895	0.896	0.978	1.000	0.991
(2, 4)	0.992	0.996	0.935	0.984	0.984	0.749	0.970	0.991
(3, 4)	0.982	0.939	0.905	0.977	0.992	0.832	0.972	0.989
(4, 4)	0.991	0.947	0.914	0.959	0.727	0.768	0.999	1.000

(i,j) \ k	0	1	2	3	4	5	6	7
(0, 0)	0.803	0.872	0.718	0.587	0.413	0.528	0.801	0.677
(1, 0)	0.530	0.654	0.255	0.226	0.354	0.274	0.522	0.314
(2, 0)	0.487	0.592	0.334	0.262	0.263	0.355	0.475	0.351
(3, 0)	0.529	0.683	0.309	0.220	0.294	0.275	0.498	0.355
(4, 0)	0.526	0.651	0.299	0.207	0.235	0.351	0.490	0.353
(0, 1)	0.373	0.365	0.286	0.305	0.274	0.306	0.536	0.483
(1, 1)	0.293	0.348	0.327	0.280	0.272	0.376	0.608	0.449
(2, 1)	0.259	0.353	0.262	0.240	0.291	0.298	0.596	0.533
(3, 1)	0.290	0.346	0.290	0.267	0.352	0.376	0.544	0.485
(4, 1)	0.358	0.385	0.295	0.390	0.362	0.259	0.619	0.437
(0, 2)	0.277	0.300	0.340	0.322	0.200	0.263	0.569	0.325
(1, 2)	0.289	0.300	0.309	0.354	0.216	0.259	0.553	0.341
(2, 2)	0.224	0.299	0.339	0.358	0.197	0.258	0.541	0.281
(3, 2)	0.275	0.244	0.327	0.269	0.233	0.270	0.508	0.341
(4, 2)	0.284	0.230	0.236	0.293	0.173	0.263	0.530	0.315
(0, 3)	0.301	0.252	0.291	0.289	0.444	0.319	0.638	0.374
(1, 3)	0.312	0.256	0.260	0.257	0.438	0.344	0.700	0.336
(2, 3)	0.383	0.225	0.274	0.268	0.347	0.328	0.661	0.396
(3, 3)	0.379	0.285	0.270	0.265	0.311	0.307	0.695	0.340
(4, 3)	0.337	0.262	0.260	0.247	0.425	0.340	0.696	0.401
(0, 4)	0.351	0.413	0.241	0.225	0.256	0.326	0.612	0.474
(1, 4)	0.338	0.393	0.260	0.216	0.228	0.332	0.593	0.332
(2, 4)	0.299	0.350	0.282	0.299	0.302	0.318	0.616	0.493
(3, 4)	0.303	0.326	0.271	0.290	0.253	0.262	0.649	0.400
(4, 4)	0.319	0.783	0.528	0.516	0.828	0.601	0.587	0.670

Table 3. Guessing entropy on $\alpha'_0[i,j,k]$ (left) and $\beta_0[i,j,k]$ (right). The entropy for α_1 (omitted here) look similar to those for α'_0.

(i,j) \ k	0	1	2	3	4	5	6	7
(0, 0)	1.095	1.109	2.336	1.616	3.215	2.592	1.074	1.096
(1, 0)	1.005	1.006	1.085	1.049	1.033	1.048	1.001	1.009
(2, 0)	1.007	1.024	1.074	1.070	1.044	1.102	1.022	1.008
(3, 0)	1.001	1.003	1.018	1.377	1.424	1.185	1.035	1.034
(4, 0)	1.001	1.001	1.452	1.575	1.680	1.297	1.028	1.005
(0, 1)	1.000	1.000	1.021	1.053	1.255	1.440	1.002	1.014
(1, 1)	1.005	1.003	1.084	1.127	1.353	1.147	1.020	1.009
(2, 1)	1.000	1.089	1.325	1.347	1.756	1.208	1.014	1.002
(3, 1)	1.003	1.022	1.092	1.066	1.006	1.056	1.013	1.012
(4, 1)	1.000	1.027	1.187	1.107	1.158	1.076	1.002	1.000
(0, 2)	1.003	1.057	1.294	1.377	1.833	1.819	1.001	1.000
(1, 2)	1.002	1.003	1.275	1.565	1.670	1.269	1.005	1.002
(2, 2)	1.031	1.012	1.020	1.274	1.625	1.947	1.035	1.010
(3, 2)	1.002	1.341	2.042	2.546	2.370	2.100	1.027	1.009
(4, 2)	1.003	1.026	1.395	1.709	1.832	1.508	1.019	1.003
(0, 3)	1.000	1.000	1.035	1.075	1.297	1.294	1.008	1.026
(1, 3)	1.010	1.004	1.068	1.024	1.053	1.072	1.012	1.008
(2, 3)	1.001	1.072	1.355	1.575	1.710	1.812	1.015	1.000
(3, 3)	1.001	1.000	1.594	1.618	1.959	1.324	1.050	1.006
(4, 3)	1.121	1.194	1.443	1.525	1.301	1.054	1.003	1.013
(0, 4)	1.000	1.000	1.140	1.175	1.156	1.054	1.000	1.000
(1, 4)	1.000	1.002	1.216	1.177	1.142	1.024	1.000	1.009
(2, 4)	1.010	1.005	1.083	1.020	1.022	1.491	1.030	1.009
(3, 4)	1.023	1.078	1.131	1.028	1.008	1.318	1.032	1.012
(4, 4)	1.009	1.060	1.122	1.052	1.652	1.492	1.001	1.000

(i,j) \ k	0	1	2	3	4	5	6	7
(0, 0)	1.296	1.178	1.622	2.351	3.931	2.629	1.391	1.715
(1, 0)	2.643	1.954	7.313	9.001	5.537	7.692	2.752	5.906
(2, 0)	2.675	2.241	4.973	8.000	6.842	4.567	2.914	4.949
(3, 0)	2.371	1.778	7.058	8.803	6.444	6.724	2.959	5.089
(4, 0)	2.433	1.794	6.284	9.404	6.959	4.883	3.105	5.764
(0, 1)	4.583	5.037	6.780	7.534	5.965	6.288	2.697	3.360
(1, 1)	6.258	5.443	5.074	7.012	7.183	4.046	2.053	3.480
(2, 1)	6.325	5.132	7.682	8.731	6.660	6.622	2.468	2.980
(3, 1)	6.103	5.088	6.765	7.806	5.521	4.701	2.317	3.210
(4, 1)	5.267	4.972	6.526	5.000	4.129	7.227	2.214	3.897
(0, 2)	7.704	6.183	5.059	5.273	9.640	7.801	2.431	6.919
(1, 2)	5.800	7.270	6.671	4.691	9.212	6.722	2.723	5.457
(2, 2)	8.800	7.315	5.902	4.676	9.164	7.875	2.852	7.929
(3, 2)	6.875	8.534	6.677	6.691	8.061	8.670	2.906	6.216
(4, 2)	7.238	8.397	8.326	6.095	9.477	9.050	2.687	7.163
(0, 3)	5.747	7.825	6.600	6.936	3.231	5.893	2.140	4.747
(1, 3)	5.547	8.029	7.555	7.707	3.502	5.444	1.716	5.898
(2, 3)	4.549	8.766	7.473	6.990	4.631	5.860	1.899	3.982
(3, 3)	4.746	6.739	7.764	7.300	5.486	6.208	1.648	5.044
(4, 3)	5.313	8.414	8.048	7.751	3.531	5.413	1.796	4.470
(0, 4)	5.294	3.874	7.979	9.418	8.310	6.139	2.309	3.309
(1, 4)	5.309	3.939	7.766	8.770	7.162	6.030	2.335	5.722
(2, 4)	5.261	4.359	6.343	6.365	6.494	6.079	2.259	3.364
(3, 4)	6.766	4.995	7.510	7.268	7.313	7.794	1.929	4.508
(4, 4)	5.753	1.355	2.426	3.045	1.295	2.164	2.393	2.405

approach is practical to search for moderately-sized states (e.g., 16-byte AES keys), we found that, when it comes to our much larger 200-byte states, it would still require unrealistically accurate templates for the search time to be tolerable.

To avoid directly combining the rank tables of our 200 target bytes, we built a three-layer scheme that can gradually combine the probabilistic information available about these bytes into a full state. In addition, rather than targeting just 200 bytes, our scheme actually takes 600 rank tables into consideration, to consider per-byte likelihoods from three intermediate states: α_0', β_0, and α_1. At the bottom, Layer 1 first merges the rank tables associated with five bytes in the same byte row, updates the likelihood of each combination, and then generates in total 40 new rank tables that cover entire byte rows. Layer 2 then combines five byte rows in the same byte slice, updates their likelihood values, and then generates eight new rank tables for byte slices. Finally, Layer 3 just concatenates the eight top candidates from each byte-slice rank table, and verifies the correctness of the resulting full intermediate state.

5.1 Layer 1: generating tables for byte rows

Between the intermediate states α_0' and β_0 is step χ. It can be calculated within a byte row, without any influence from other byte rows, which allows us to split the combination of these intermediate states into 40 mutually independent parts. Therefore we can combine per-byte rank tables using a practical enumeration tree that covers only five bytes at a time. We use the first byte row ($j = 0, k = 0$) here to demonstrate this.

First, we initialize the number T of combinations we want to collect in the resulting byte-row rank table to $T = 2500$. The five bytes of state α_0' in the first byte row are $\alpha_0'[0,0,0]$, $\alpha_0'[1,0,0]$, $\alpha_0'[2,0,0]$, $\alpha_0'[3,0,0]$, $\alpha_0'[4,0,0]$, and we use the five variables $A_0', A_1', A_2', A_3', A_4'$ to represent their values. As likelihood functions we use the Gaussian multivariate probability-density values provided by the template attack: $\mathcal{L}(\alpha_0'[0,0,0] = A_0') = f_{\alpha_0'[0,0,0]}(\mathbf{x}_{\text{proj}}|\hat{\mathbf{x}}_{A_0',\text{proj}}, \mathbf{S}_{\text{proj}})$, etc. With the rank tables of these five bytes, we build an enumeration tree to search the first T combinations in descending order of joint likelihood of a byte row. Assuming independence, our first estimate of their joint likelihood is

$$\mathcal{L}_{\text{row}}(\alpha_0'[\cdot,0,0] = (A_0', A_1', A_2', A_3', A_4')) := \prod_{i=0}^{4} \mathcal{L}(\alpha_0'[i,0,0] = A_i').$$

Now the top-T combinations and their corresponding joint likelihoods form a truncated rank table for this byte row.

For these T combinations, we calculate the values of state β_0 in this byte row as

$$(B_0, B_1, B_2, B_3, B_4) = \chi(A_0', A_1', A_2', A_3', A_4').$$

Since we also have ranked likelihood tables for all bytes in state β_0, we now can similarly calculate the likelihood for any combination $(B_0, B_1, B_2, B_3, B_4)$, and update the above top-T joint likelihoods by multiplying with the likelihood of β_0, that is

$$\mathcal{L}^{\text{new}}_{\text{row}}(\alpha'_0[\cdot,0,0]=(A'_0,A'_1,A'_2,A'_3,A'_4)) := \prod_{i=0}^{4} \mathcal{L}(\alpha'_0[i,0,0]=A'_i)\mathcal{L}(\beta_0[i,0,0]=B_i).$$

Then, we sort these T combinations again in descending order of their updated joint likelihood, and obtain the new rank table of this byte row.

5.2 Layer 2: generating tables for byte slices

We then use a method similar to Layer 1 to combine five byte-row rank tables into a byte-slice rank table. We use here the first byte slice ($k=0$) to demonstrate this. Let R'_j represent a byte row value of state $\alpha'_0[\cdot,j,0]$ in this byte slice, such that it contains five bytes, where $R'_j=(A'_{0,j},A'_{1,j},A'_{2,j},A'_{3,j},A'_{4,j})$.

We use the rank tables of the five byte rows again to build an enumeration tree, and search the first T combinations in descending order of joint likelihood of a byte slice. Number T is as in Layer 1. Our initial joint likelihood estimate for a byte slice is

$$\mathcal{L}_{\text{slice}}(\alpha'_0[\cdot,\cdot,0]=(R'_0,R'_1,R'_2,R'_3,R'_4)) := \prod_{j=0}^{4}\mathcal{L}^{\text{new}}_{\text{row}}(\alpha'_0[\cdot,j,0]=R'_j)=\prod_{j=0}^{4}\prod_{i=0}^{4}\mathcal{L}(\alpha'_0[i,j,0]=A'_{i,j})\mathcal{L}(\beta_0[i,j,0]=B_{i,j}).$$

Similar as in Layer 1, we now update these joint likelihoods by taking the rank tables of α_1 into account. We use variable $A_{i,j}$ to represent the candidates of intermediate byte $\alpha_1[i,j,0]$, and with $R_j=(A_{0,j},A_{1,j},A_{2,j},A_{3,j},A_{4,j})$ have

$$(R_0,R_1,R_2,R_3,R_4)=\theta^*(\iota^*_{0,k}(\chi(R'_0),\chi(R'_1),\chi(R'_2),\chi(R'_3),\chi(R'_4)),\tau).$$

where $\iota^*_{0,k}$ is ι in round 0 with input and output truncated to byte slice k, and $\theta^*(\ldots,\tau)$ is θ applied to just one byte slice, where $\tau\in\{0,1\}^5$ represents the five bits of column-parity information taken by θ from the previous byte slice. Since step χ operates within a byte row, it will not use any data outside the byte slice. Likewise, step ι XORs with a round constant, so it too is independent of other byte slices. However, when executing step θ on only a byte slice, we will lack information about five bits, because bit rotations are involved in step θ and hence these five bits come from another byte slice. Without that information τ, step θ^* on only one byte slice will have 32 possible outcomes. It is reasonable to choose the combination τ that maximizes the joint likelihood of $\alpha_1[\cdot,\cdot,0]$, which is

$$\max_{\tau\in\{0,1\}^5}\prod_{j=0}^{4}\prod_{i=0}^{4}\mathcal{L}(\alpha_1[i,j,0]=A_{i,j}).$$

Then, we can update the joint likelihood of this byte slice by multiplying with the joint likelihood of α_1, that is

$$\mathcal{L}^{\text{new}}_{\text{slice}}(\alpha'_0[\cdot,\cdot,0] = (R'_0, R'_1, R'_2, R'_3, R'_4)) := \prod_{j=0}^{4}\prod_{i=0}^{4} \mathcal{L}(\alpha'_0[i,j,0] = A'_{i,j})\mathcal{L}(\beta_0[i,j,0] = B_{i,j})\mathcal{L}(\alpha_1[i,j,0] = A_{i,j}).$$

We then again sort these T combinations in descending order of the updated joint likelihoods to form a new rank table for this byte slice.

5.3 Layer 3: consistency checking

In Layer 3, we could again form an enumeration tree to combine the top-T entries in the eight byte-slice rank tables from Layer 2 into a single top-T rank table of the full 200-byte state. In practice, however, we found that this was never necessary, as in all our experiments if a byte-slice rank table contained the correct combination, it was already ranked top. Therefore, Layer 3 actually only needed to concatenate the top-ranked combinations from all eight byte-slice tables together and then can calculate the corresponding input and output of the Keccak-f[1600] permutation. Then, we can check consistency of these with available SHA-3 data, as described in Section 3 and Fig. 2.

If the top combination fails that consistency check, most likely the correct candidate was already missing in the tables produced by layers 1 or 2. Therefore we quadruple T and restart the search from Layer 1 in such cases. (In our experiments, we gave up after still not finding a correct solution with $T = 640\,000$, but this limit can of course be raised given sufficient computing resources.)

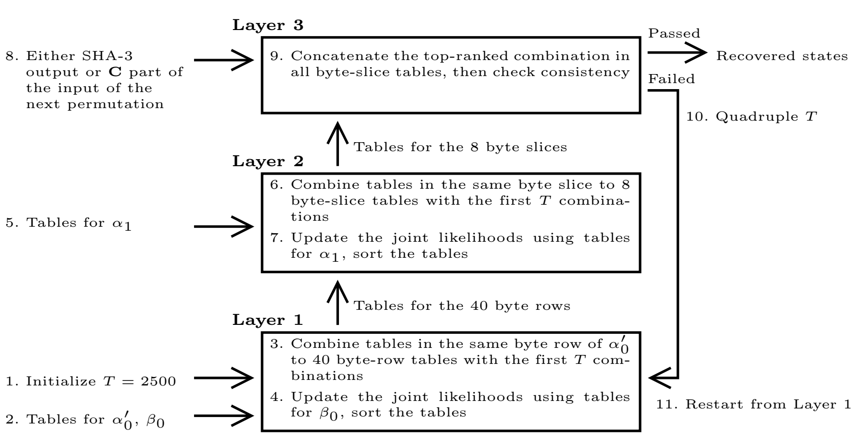

Fig. 4. The procedure to combine a full state from 600 tables.

5.4 Results

SHA3-512 with only one Keccak-f[1600] invocation. We first evaluated our attack using 1000 test traces of SHA3-512 executions, each with a random input shorter than 72 bytes. This is the simplest case, where a SHA3-512 execution invokes Keccak-f[1600] only once to digest the input. We only need to apply the template attack here to obtain the 600 rank tables of intermediate bytes in that one Keccak-f[1600] invocation, apply our three-layer search to find the correct combination, and calculate the input and output of the Keccak-f[1600] invocation. Its correctness can be verified by checking whether the first 512 bits of the output match the SHA-3 output and whether the last 1024 bits of the input are all zero. If both checks pass, the input of SHA-3 can be reconstructed by removing the padding from the first 576 bits of the recovered Keccak-f[1600] input.

In these 1000 tests, we successfully reconstructed the SHA3-512 input 999 times, while we failed to recover one remaining one even with $T = 640\,000$. The number of additional traces for which we recovered the correct input was for each T value

T	2500	10 000	40 000	160 000	640 000	failed
new traces recovered	873	77	33	11	5	1
cumulative %age	87.3	95.0	98.3	99.4	99.9	100
CPU time avr. [s]	8.20	24.98	90.53	431.35	2605.88	N/A
CPU time std. [s]	0.23	0.44	1.70	7.65	23.86	N/A

SHA3-512 with multiple Keccak-f[1600] invocations. Generally, where the input is longer than 72 bytes, it takes multiple Keccak-f[1600] invocations to digest. There we need to use the templates to obtain the 600 rank tables of the three intermediates states in every invocation, and we then start the three-layer search for each, from the last invocation to the first. We verify the correctness and calculate the SHA-3 input as described in Section 3.

While the success probability for each Keccak-f[1600] invocation is the same, the success rate of reconstructing the entire SHA-3 input should drop with increasing number of invocations, as the failure to recover the state of any Keccak-f[1600] invocation means that two SHA-3 input blocks cannot be recovered. If the success rate of reconstructing the state of one Keccak-f[1600] permutation is p, then the success rate of reconstructing SHA3-512 inputs of L bytes length will be $p^{\lceil \frac{L+1}{72} \rceil}$.

We also tried to recover SHA3-512 inputs ranging from 216 bytes to 287 bytes, where Keccak-f[1600] was invoked four times. Of 1000 attempted traces, we successfully reconstructed the SHA-3 input 999 times, while in the only unsuccessful one the search failed for one invocation of the permutation. While we would normally expect the success rate of attacking SHA-3 with shorter input to be higher than with longer inputs, in these experiments the success rates were both too close to 1 to be distinguishable.

6 Discussion and conclusion

Search time and success rate may be optimized further by adjusting the rank-table length T for each byte row or slice separately, depending on the relative likelihoods involved. So far we used the same T for all 40 byte rows in Layer 1 and all eight byte slices. From the numbers in Table 2, it is evident that the success rates are much better for some byte locations, and for these, smaller initial values of T may lead to a faster hit.

Our method could be extended by also building templates of intermediate states in later rounds, such as a combination of α'_1, β_1, α_2. When attackers fail to recover the state in the first round, they could then try to search other rounds and do a similar search as they have done in the first round. Although ι is different in each round, there may be scope for reusing at least some templates across rounds. In total there should be 23 combinations of intermediate states that attackers could target using our search method.

With our method, we demonstrated that it is practical to reconstruct the inputs of an unprotected SHA-3 software implementation on an ATxmega256A3U 8-bit microcontroller using a template attack, even where the templates fail to rank some correct bytes highest. In future work, we hope to extend this attack procedure to work on 32-bit devices, where the success rates of templates can be far worse.

References

1. Alkim, E., et al.: NewHope: Algorithm specifications and supporting documentation (2019), `https://newhopecrypto.org/`
2. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: The Keccak reference (Jan 2011)
3. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Keccak. In: Johansson, T., Nguyen, P.Q. (eds.) Advances in Cryptology – EUROCRYPT 2013. pp. 313–314. Springer, Berlin, Heidelberg (2013)
4. Chari, S., Rao, J.R., Rohatgi, P.: Template attacks. In: International Workshop on Cryptographic Hardware and Embedded Systems. pp. 13–28. Springer (2002)
5. Choudary, M.O.: Efficient multivariate statistical techniques for extracting secrets from electronic devices. Tech. Rep. UCAM-CL-TR-878, University of Cambridge (2015), PhD thesis
6. Choudary, M.O., Kuhn, M.G.: Efficient stochastic methods: profiled attacks beyond 8 bits. In: International Conference on Smart Card Research and Advanced Applications. pp. 85–103. Springer (2014)
7. Choudary, M.O., Kuhn, M.G.: Efficient, portable template attacks. IEEE Transactions on Information Forensics and Security **13**(2), 490–501 (Feb 2018)
8. Choudary, O., Kuhn, M.G.: Efficient template attacks. In: International Conference on Smart Card Research and Advanced Applications. pp. 253–270. Springer (2013)
9. KeccakTools, `https://github.com/KeccakTeam/KeccakTools`
10. Luo, P., Fei, Y., Fang, X., Ding, A.A., Kaeli, D.R., Leeser, M.: Side-channel analysis of MAC-Keccak hardware implementations. In: Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy (HASP '15). Association for Computing Machinery (2015)

11. Mangard, S.: Hardware countermeasures against DPA – a statistical analysis of their effectiveness. In: Topics in Cryptology – CT-RSA 2004. Springer, Berlin, Heidelberg (2004)
12. Microchip: ATxmega256A3U, accessed February 2020, `https://www.microchip.com/wwwproducts/en/atxmega256a3u`
13. National Instruments: PXI-4110 programmable power supply, `http://www.ni.com/en-gb/support/model.pxi-4110.html`
14. National Instruments: PXIe-5160 oscilloscope, `http://www.ni.com/en-gb/support/model.pxie-5160.html`
15. National Instruments: PXIe-5423 waveform generator, `http://www.ni.com/en-gb/support/model.pxie-5423.html`
16. NIST: SHA-3 standard: permutation-based hash and extendable-output functions (Aug 2015), `http://dx.doi.org/10.6028/NIST.FIPS.202`, FIPS PUB 202
17. Oswald, D., Paar, C.: Breaking Mifare DESFire MF3ICD40: power analysis and templates in the real world. In: International Workshop on Cryptographic Hardware and Embedded Systems. pp. 207–222. Springer (2011)
18. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research **12**, 2825–2830 (Oct 2011)
19. Schindler, W., Lemke, K., Paar, C.: A stochastic model for differential side channel cryptanalysis. In: International Workshop on Cryptographic Hardware and Embedded Systems. pp. 30–46. Springer (2005)
20. Standaert, F.X., Archambeau, C.: Using subspace-based template attacks to compare and combine power and electromagnetic information leakages. In: International Workshop on Cryptographic Hardware and Embedded Systems. pp. 411–425. Springer (2008)
21. Standaert, F.X., Malkin, T.G., Yung, M.: A unified framework for the analysis of side-channel key recovery attacks. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 443–461. Springer (2009)
22. Taha, M., Schaumont, P.: Differential power analysis of MAC-Keccak at any key-length. In: Sakiyama, K., Terada, M. (eds.) Advances in Information and Computer Security. pp. 68–82. Springer, Berlin, Heidelberg (2013)
23. Taha, M., Schaumont, P.: Side-channel analysis of MAC-Keccak. In: 2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST). pp. 125–130. IEEE (2013)
24. Veyrat-Charvillon, N., Gérard, B., Renauld, M., Standaert, F.X.: An optimal key enumeration algorithm and its application to side-channel attacks. In: International Conference on Selected Areas in Cryptography. pp. 390–406. Springer (2012)
25. Extended Keccak code package, `https://github.com/XKCP/XKCP`, accessed April 2019, `lib/low/KeccakP-1600/Compact64/KeccakP-1600-compact64.c`