Single-Trace Fragment Template Attack on a 32-bit Implementation of Keccak

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. Template attacks model side-channel leakage information using Gaussian multivariate distributions. They have been quite successful in directly reconstructing individual bits of 8-bit parallel buses and registers from power traces. However, extending their use to larger word sizes, such as 32-bit buses, becomes impractical. Here we show that it is possible to use an LDA-based stochastic model to independently build templates for just byte fragments of such a word, to predict the exact values of its four member bytes, instead of only overall Hamming weights. We demonstrate this technique to reconstruct the arbitrary-length inputs of SHA3-512 and some other Keccak sponge functions implemented on a 32-bit Cortex-M4 device. The quality of these templates was high enough such that remaining errors in their predictions could be eliminated via belief-propagation on a factor-graph network (SASCA). In our experiments, we already reliably recovered SHA3-512 inputs up to 719 bytes long (10 invocations of the permutation), and reconstructing even longer inputs should be just a matter of making longer recordings.

Keywords: Template attack · SASCA · Keccak · SHA-3 · 32-bit device

1 Introduction

1.1 Motivation and Background

Since the National Institute of Standards and Technology (NIST) standardized *Secure Hash Algorithm 3* (SHA-3) [14] in 2015, several variants of Differential Power Analysis (DPA) [19,18,10] have been used to reconstruct the keys in Keccak-based message authentication codes (MAC-Keccak). These attacks require multiple accesses to the SHA-3($K\|M$) function, with a known and varying message M, and their recorded power traces, to recover the fixed and unknown key K.

Later, in 2020, two different approaches for single-trace recovery strategies appeared. Kannwischer et al. [7] used Soft Analytical Side-Channel Analysis (SASCA) [20] to recover a 128 or 256-bit secret S used in Keccak-f[1600]($S\|M$), given known message M, based on simulated noisy Hamming-weight information

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

of intermediate values in this permutation. They concluded that their method was very successful on a simulated 8-bit or 16-bit device, but the situation was not yet clear on 32-bit devices, where they successfully recovered 128-bit keys only under some conditions, such as a lower noise level, but not 256-bit keys. They also suggested their SASCA approach may reach a higher success rate with a leakage model bearing more information than just Hamming weights.

We introduced the other approach in [22]. On an ATxmega256A3U [1] 8-bit device, we used an enumeration procedure based on 600 rank tables for the intermediate bytes. Each rank table lists all 256 candidate bytes in descending order of probability, as predicted by LDA-based stochastic-model templates. This enumeration technique could reconstruct a complete intermediate state from a single trace and then invert it to determine all Keccak-f[1600] input and output bits. By repeating the same procedure on every invocation of Keccak-f[1600] in the absorbing stage of the Keccak sponge function, we recovered arbitrary-length SHA3-512 inputs.

Therefore, encouraged by both these results, we now target a more ambitious goal, namely to reconstruct the complete arbitrary-length input of SHA-3 or SHAKE functions implemented on a 32-bit device, from a single trace. To achieve this target, we will have to figure out how to practically build templates for a 32-bit bus that can obtain far more information about a 32-bit state than just the Hamming weight.

Choudary and Kuhn [3] used template attacks based on Linear Discriminant Analysis (LDA) to directly recover from a single load instruction the exact value of a byte, and not just its Hamming weight. They also looked at extending their method to states with more than 8 bits. However, directly building templates for a 32-bit value is not practical this way.

1.2 Contributions and Paper Structure

We introduce the *fragment template attack*, to extract information about individual bits from power traces that observe activity on 32-bit parallel data buses. To achieve this, we apply the LDA technique to project the data onto subspaces where the projected data are only related to a fragment (e.g., a byte or a nibble) of the full 32-bit word, and then independently build templates for these fragments, to enable us to reconstruct their values independently and within a reasonable run time.

Having built fragment templates for some important intermediate states in the Keccak-f[1600] permutation implemented on a 32-bit device, we targeted the STM32F303RCT7 CPU on a ChipWhisperer-Lite board [15]. We find that their quality is good enough for a SASCA attack, i.e. to error-correct the template-attack information with the help of a loopy belief propagation factor graph based on the structure of the rounds of the Keccak-f[1600] permutation.

In this paper, we first introduce and review some of the prior work that our technique is based on (Section 2), namely the LDA-based stochastic model templates and SASCA. Section 3 then explains our methodology, including how we build fragment templates and our modification of a previous use of SASCA

against Keccak, and how to combine the results from multiple invocations of Keccak-$f[1600]$ in the absorbing stage of a Keccak sponge function to calculate full arbitrary-length inputs. The evaluation of our experiments in Section 4 shows how parameters such as the number of rounds observed and the number of known bits at the input of Keccak-$f[1600]$ affect the success probability of our attack.

2 Preliminaries and Notation

2.1 LDA-based Templates on Keccak

The template attack with stochastic models. Following the original template attack (TA) introduced by Chari et al. [2], the "stochastic" model $\mathcal{F}_9$ by Schindler et al. [16], and the use of Fisher's Linear Discriminant Analysis (LDA) by Standaert and Archambeau [17] for dimensionality reduction of traces, Choudary and Kuhn [3] combined these into an LDA-based template profiling stage for a $\mathcal{F}_9$ model as follows. Firstly, record the traces and group them according to the target byte value $b \in \{0, \ldots, 255\}$, where trace $\mathbf{x}_{b,t}$ observed target value b, with $t \in \{1, \ldots, n_b\}$ enumerating the traces in that group. When building a template for a target byte b, treat each member bit ($b[0]$ to $b[7]$) as an independent variable, and then use a multivariate linear regression to calculate for each point in time coefficients c_0 to c_7 and a constant c_8 to predict the expected values of samples as $\bar{x}_b = \sum_{l=0}^{7}(b[l] \cdot c_l) + c_8$, the $\mathcal{F}_9$ stochastic model. We write

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

to represent the expected vector of an entire m-sample trace, where $\mathbf{c}_0, \ldots, \mathbf{c}_8 \in \mathbb{R}^m$. From these, build two covariance matrices, $\mathbf{B}$ representing the signal, and $\mathbf{\Sigma}$ representing the noise, as

$$\mathbf{B} = \frac{1}{\sum_b n_b} \sum_b n_b (\bar{\mathbf{x}}_b - \bar{\mathbf{x}})(\bar{\mathbf{x}}_b - \bar{\mathbf{x}})^\mathsf{T},$$

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

where $\bar{\mathbf{x}}$ is the average of all 256 expected vectors $\bar{\mathbf{x}}_b$.

Dimensionality reduction. In the LDA step, project the m-sample traces $\mathbf{x}_{b,t}$ onto the m' largest eigenvectors of $\mathbf{\Sigma}^{-1}\mathbf{B}$, to obtain dimensionality-reduced m'-sample traces $\mathbf{x}_{b,t,\mathrm{proj}}$, where $m' \ll m$, and the signal-to-noise ratio in the new subspace is larger. Likewise, the expected traces $\bar{\mathbf{x}}_b$ as well as the attack trace $\mathbf{x}_\mathrm{a}$ can be projected into the same subspace as $\bar{\mathbf{x}}_{b,\mathrm{proj}}, \mathbf{x}_{\mathrm{a},\mathrm{proj}} \in \mathbb{R}^{m'}$.

With all these traces projected into the new subspace, we now can build a pooled covariance matrix

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

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

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

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

If the originally recorded trace length and sampling frequency are very high, prior sample selection or resampling steps are needed to make the above LDA matrix operations feasible. Like in [22], we therefore used sample-rate reduction and sample selection based on multivariate linear regression as initial dimensionality reduction steps before applying LDA compression (see Section 4.2).

Template attack on an 8-bit implementation of Keccak. In [22] we used the above LDA-based template attack already to recover 600 intermediate bytes from single invocations of the Keccak-$f[1600]$ permutation on an ATxmega256A3U 8-bit microcontroller. This permutation is the sequence of five steps $(\theta, \rho, \pi, \chi, \iota)$, which iterate for 24 rounds. We reuse here the same notation for the intermediate states α_Ω, α'_Ω, β_Ω and β'_Ω between these steps, defined for the Ω^{th} round as

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

We use three variables, $i, j \in \mathbb{Z}_5$, $h \in \mathbb{Z}_8$ to label the 200 bytes of these states, where "$\in \mathbb{Z}_n$" shall imply arithmetic modulo n. For example, the first byte in the first lane of α'_0 is $\alpha'_0[0,0,0]$, and its least significant bit is $\alpha'_0[0,0,0][0]$. In addition to this bytewise notation, we also use a bitwise notation with bit index $k \in \mathbb{Z}_{64}$ and a "$\hat{}$" on the variable, as in $\hat{\alpha}'_0[i,j,k] = \alpha'_0[i,j,h][l]$ for $k = 8 \times h + l$ and $l \in \{0, \ldots, 7\}$.

Given an attack trace, in [22] we used 600 templates to generate the rank table for each of the 200 intermediate bytes in each of the three states α'_0, β_0 and α_1, so that all the correct candidates in α'_0 can be found through the three-level enumeration.

2.2 Soft Analytical Side-Channel Analysis on Keccak

Belief propagation and SASCA. Veyrat-Charvillon et al. [20] introduced SASCA, which is an inference technique for template attacks on cryptographic

algorithms based on the belief-propagation algorithm [11, 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 called "variable", representing the intermediate states of the cryptographic algorithm, and "factor", representing how these intermediate states depend on each other and on the observed traces. Each of these nodes is only connected to nodes of the respective other type (i.e., the factor graph is a bipartite graph), and information can flow through these connections. The factor graph therefore reflects the mathematical structure of the cryptographic algorithm, which then influences the updating of the probability estimates of the variables accordingly during the execution of the belief-propagation or sum-product message-passing algorithm.

While the variable nodes represent the intermediate values in the cryptographic algorithm, we can separate the factor nodes into two subtypes, "observation factors" and "constraint factors". Observation factors $f_m(x_n)$ represent observed probabilities of the values of their only connected variable x_n, here usually from a template-based likelihood. Constraint factors $f_m(\mathbf{x}_m)$ are connected to more than one variable $(x_{n_1}, \ldots, x_{n_{k_m}}) = \mathbf{x}_m$ (where $\mathcal{N}(m) = \{n_1, \ldots, n_{k_m}\}$ shall denote the set of indices of these variables) with a mathematical equation as the constraint. 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. On a connection, the information flow is bidirectional, where a message from a variable x_n to a factor f_m is denoted as $q_{n\to m}$, and a message from a factor f_m to a variable x_n as $r_{m\to n}$. Each of these messages is a function of a value ξ of x_n. The probability of a candidate $x_n = \xi$ in message $q_{n\to m}$ is:

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

which means the probability passing from a variable to a factor is the product of the probabilities of the same candidate in all the messages r passing from all other factors connected to this variable. Meanwhile, the probability of a candidate $x_n = \xi$ in the message $r_{m\to n}$ is:

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

where

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

In other words, the probability passed from factor f_m to variable x_n is the sum of the product of the probabilities of the candidates in the messages q passed from the other variables $x_{n'}$ connected to factor f_m, where these candidates combined

with the candidate $x_n = \xi$ match the constraint in f_m. For the special case of an observation factor this reduces to:

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

where $f_m(x_n)$ is the probability table observed from the templates, instead of a constraint function. To obtain the final probability P_n of candidates $x_n = \xi$, we need the product

$$Z_n(x_n = \xi) = \prod_m r_{m \to n}(x_n = \xi)$$

of the probabilities in all the messages r passed to the same variable x_n and then normalize the result as

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

This is how the probabilities can be updated recursively through a tree structure. The algorithm terminates on tree-shaped factor graphs once the number of steps has reached the diameter of the tree. However, in most SASCA networks of cryptographic algorithms, the factor graph is less likely to be a tree structure. Instead, it probably features loops, which means that this recursive belief propagation will not terminate to output exact probabilities.

MacKay describes an easy solution [11, Chapter 26] 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 when the probability values become too small. Then terminate when a steady state has been reached.

Apply loopy belief propagation to Keccak. Kannwischer et al. [7] demonstrate how they use loopy-BP given noisy Hamming-weight information of intermediate values. Their simulated attacks targeted the secret first 128 or 256 bits of the input of a Keccak-f[1600] permutation, with the remaining input bits being known. They first introduce a bitwise (i.e., $\xi \in \{0, 1\}$) loopy-BP network. In this case, many constraint factors and variables in the bit permutation step ρ and π are no longer needed: firstly, we can simply connect the output of step θ to the input of step χ following the permutation rules of the two steps instead, and secondly, step ι XORs some round constant in the first lane, so we only need to swap the output probabilities corresponding to 0 and 1 of step χ there. Therefore, we only need to include one of the two states α_Ω and α'_Ω in the factor graph, and one of β_Ω and β'_Ω.

As for the most complicated step, θ, the corresponding equation is

$$\hat{\alpha}_\Omega[i, j, k] = \bigoplus_{j=0}^{4} \hat{\beta}'_{\Omega-1}[i-1, j, k] \oplus \bigoplus_{j=0}^{4} \hat{\beta}'_{\Omega-1}[i+1, j, k-1] \oplus \hat{\beta}'_{\Omega-1}[i, j, k].$$

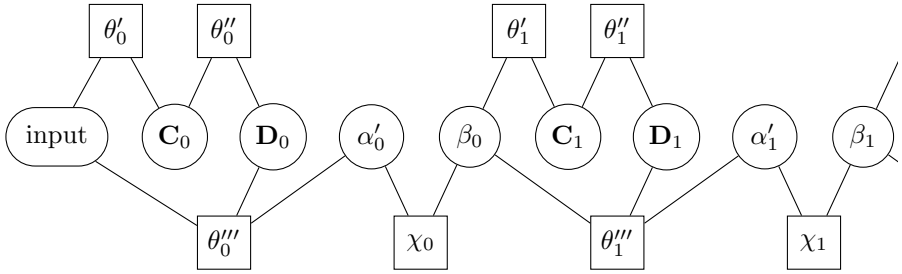

Fig. 1. The loopy-BP graph structure for the Keccak-f permutation, showing the node relations for the first two rounds. Variable nodes are in circles, constraint factors in squares. Observation factors are not shown here. Each state variable shown here actually represents 1600 or 320 single-bit variable nodes, respectively.

If we directly designed a constraint factor following this equation, it would connect to 12 variables. Instead, Kannwischer et al. [7, Fig. 1] separated it into three equations

$$\hat{\mathbf{C}}_\Omega[i,k] = \bigoplus_{j=0}^{4} \hat{\beta}'_{\Omega-1}[i,j,k], \tag{θ'}$$

$$\hat{\mathbf{D}}_\Omega[i,k] = \hat{\mathbf{C}}_\Omega[i-1,k] \oplus \hat{\mathbf{C}}_\Omega[i+1,k-1], \tag{θ''}$$

$$\hat{\alpha}_\Omega[i,j,k] = \hat{\mathbf{D}}_\Omega[i,k] \oplus \hat{\beta}'_{\Omega-1}[i,j,k], \tag{θ'''}$$

where $\hat{\mathbf{C}}$ and $\hat{\mathbf{D}}$ are additional 320-bit intermediate states (which we will also refer to as $\mathbf{C}$ and $\mathbf{D}$ bytewise). They then use these three substeps of θ to build the constraint factors in their graph.[1]

For step χ, they suggest to combine the five-bit input and output in a row (where j and k are fixed) into a single constraint factor node, instead of connecting these ten bits with five separate nodes connecting to three input bits and one output bit. They claim this will increase the efficiency of information transmission from $\hat{\beta}$ to $\hat{\alpha}'$ nodes. Fig. 1 shows the resulting factor graph.

They terminate the loopy-BP procedure if the total entropy of all the variables drops to 0, or if the probabilities in the network no longer change, or after 50 iterations.

They simulated attacks on devices with 8, 16, or 32-bit words, of which their leakage model provides noisy Hamming weights. They state that the bitwise factor graph is not suitable for processing Hamming weights because marginalization will discard the information in the joint distribution of the bits in the target word, leading to bad attack performance. Therefore, they developed a "clustering" technique to deal with Hamming-weight information, which combines e.g. eight bits into one variable (i.e., $\xi \in \mathbb{Z}_{256}$).

[1] $\hat{\beta}'[i,j,k]$, $\hat{\mathbf{C}}[i,k]$, $\hat{\mathbf{D}}[i,k]$, $\hat{\alpha}[i,j,k]$ here are equivalent to I, P, T, O, respectively in [7].

3 Our Attack Strategy

At a high level, our attack has three main steps. We first split each 32-bit target word into several fragments and build a set of templates targeting each fragment independently. We use these profiled fragment templates to generate a probability table for every fragment in the words of the intermediate states that we target in an invocation of the Keccak-$f[1600]$ permutation. Secondly, we marginalize these probability tables for fragments into binary probability tables for each bit. We then feed these, as well as the known bits in the capacity part of the input, into the loopy-BP network for error correction. Recall that the capacity input has all 0 bits in the first invocation in a Keccak sponge function, and in later invocations it is the same as the capacity output of the previous invocation. The third step is to calculate the complete input and output of this invocation. Repeat this for each invocation. In the end, by XORing consecutive rate parts, we find the complete padded input of the Keccak sponge function.

3.1 Template Attack on Word Fragments

If we were to directly apply an LDA-based stochastic-model template [3] on each intermediate 32-bit word, we first would use multivariate linear regression, treating the 32 member bits as independent variables, to calculate the expected value for each candidate. We could then build templates for these candidates, to which the attack traces can be compared. However, with 2^{32} candidates, this approach is neither efficient nor practical. Therefore, we instead separate an intermediate word into fragments, here four bytes, and independently build templates for each. We hope that by limiting the candidate set to just the values of one fragment f at a time, treating the values of the other fragments as noise, based on the resulting per-fragment inter-class scatter $\mathbf{B}_f$ and total (pooled) intra-class scatter $\mathbf{\Sigma}_f$, the LDA can project the traces onto different subspaces, where each projection maximizes the signal-to-noise ratio for just one byte at a time.

More specifically, applying the LDA procedure directly on an intermediate 32-bit word, of value v, the matrices $\mathbf{B}$ and $\mathbf{\Sigma}$ would be

$$\mathbf{B} = \sum_{v=0}^{2^{32}-1} n_v(\bar{\mathbf{x}}_v - \bar{\mathbf{x}})(\bar{\mathbf{x}}_v - \bar{\mathbf{x}})^{\mathsf{T}} \Bigg/ \sum_{v=0}^{2^{32}-1} n_v,$$

$$\mathbf{\Sigma} = \sum_{v=0}^{2^{32}-1} \sum_{t=1}^{n_v} (\mathbf{x}_{v,t} - \bar{\mathbf{x}}_v)(\mathbf{x}_{v,t} - \bar{\mathbf{x}}_v)^{\mathsf{T}} \Bigg/ \sum_{v=0}^{2^{32}-1} n_v,$$

where $\bar{\mathbf{x}}_v$ is the expected value of traces corresponding to v with

$$\bar{\mathbf{x}}_v = \sum_{l=0}^{31} (v[l] \cdot \mathbf{c}_l) + \mathbf{c}_{32}, \tag{1}$$

where $\mathbf{c}_l$ is the coefficient vector of bit $v[l]$, and $\mathbf{c}_{32}$ is the constant vector.

Instead, our LDA procedure takes the same training traces but profiles the template with only eight bits at a time. We split each word value $v \in \mathbb{Z}_{2^{32}}$ into four byte fragments $v \mapsto (F_0(v), \ldots, F_3(v))$ with $F_f(v) = \sum_{l=0}^{7} v[8f+l] \cdot 2^l$. 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 can apply the $\mathcal{F}_9$ stochastic model to obtain the 256 expected trace vectors

$$\bar{\mathbf{x}}_{f,b} = \sum_{l=0}^{7} b[l] \cdot \mathbf{c}_{f,l} + \mathbf{c}_{f,8}, \tag{2}$$

from the traces $\mathbf{x}_{v,t}$ with $v \in V_{f,b}$, respectively. We then calculate the inter-class scatter $\mathbf{B}_f$ and the total intra-class scatter $\mathbf{\Sigma}_f$:

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

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

Now the inter-class scatter $\mathbf{B}_f$ only contains the signals from fragment number f, and the signals from the other three bytes no longer count in the inter-class scatter, but instead contribute to the total intra-class scatter $\mathbf{\Sigma}_f$. In other words, they are considered to be switching noise in this model.

After we project the profiling and attack traces via these two matrices to the new m'-dimensional subspace ($m' = 8$ in this paper), we can calculate the pooled covariance matrix and combine it with the projected expected traces as the template for this target byte in the intermediate word.

Note that in practice, with far less than 2^{32} profiling traces acquired, an efficient implementation will exploit the fact that many n_v will be zero, by iterating over recorded traces rather than all v. Alternative schemes for partitioning a 32-bit word into fragments might be useful as well, such as 11+11+10 bits, or grouping bits into fragments by distance of coefficient $\mathbf{c}_l$.

3.2 Bitwise Loopy Belief Propagation on Factor Graphs

After our templates generate the per-fragment probability tables for the selected intermediate states, we marginalize these tables to eight binary tables of their member bits and then use a bitwise loopy-BP network as the error-correction procedure. Kannwischer et al. [7] state that the probability of a bit calculated by marginalizing the Hamming weight will lose much information available in the joint distribution of the unit's member bits, but we believe that the information loss caused by marginalization may not be a severe problem in our experiments: our templates are based on the stochastic model $\mathcal{F}_9$ [16], where bits in the target bytes are viewed as independent binary variables. With the assumption of mutual independence, this model already, to some extent, gives up exploiting information from a joint distribution across bits. Since we have already bitwisely marginalized probabilities, the clustering technique is not required.

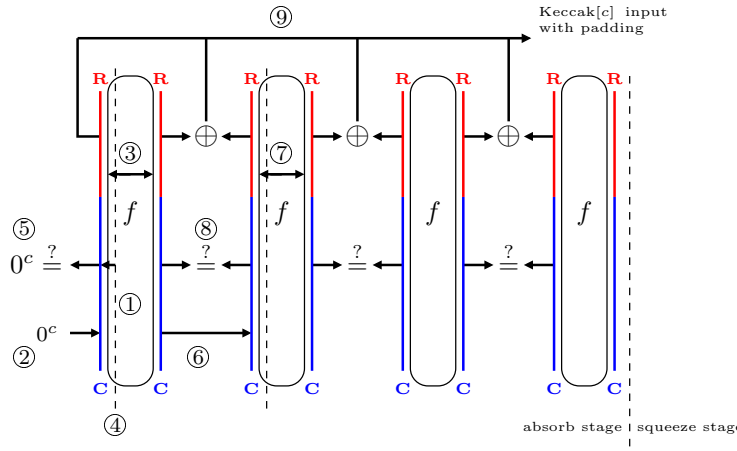

Fig. 2. The procedure to reconstruct input (and output) of sponge function Keccak[c] by template attack: ① generate the probability tables for the target intermediate states in the first Keccak-f[1600] permutation and marginalize them to binary tables; ② add the observation-factor for the capacity part of the input, which is all 0; ③ run the loopy-BP network, terminate and calculate the input and output of this invocation from state α'_0 ④, and then ⑤ check consistency of the input capacity part; ⑥ add the observation-factor for the capacity part of the input, where the bits match the capacity part of the output from the previous invocation; ⑦ repeat template recovery, table marginalization, and loopy-BP on latter invocations in the absorb stage; ⑧ repeat step ⑤; ⑨ XOR the rate parts of consecutive invocations and concatenate these XOR results to find the padded Keccak[c] input.

Besides that main difference, we made a number of other changes compared to Kannwischer et al. We terminate the loopy-BP algorithm after either reaching a steady state, with the maximum iteration count increased from 50 to 200. We found that checking the total entropy value helped little, so we dropped this termination check. Their factor graph appears to cover only the first two rounds [8, slide 14] whereas we tested different factor graphs that cover the first two, three, or four rounds, respectively, to take more side-channel information into account. Another modification is that we do not use the technique of damping, but that can be a future optimization.

We did not acquire any side-channel observations for the input. Instead its observation factors set the capacity part of the Keccak-f[1600] input according to the sponge construct with probability one to all-zero for the first invocation, and, also with probability one, to the verified output of the previous invocation in subsequent invocations. The rate-part bits of the input are the only variables without any observation factor connected.

3.3 Dealing with Multiple Invocations

We slightly modify the procedure to recover the full padded input of a Keccak sponge function from [22] as follows.

After the loopy-BP algorithm reaches a steady state, we select in α'_0 for each bit variable the candidate with the higher probability to decide on our prediction for that intermediate state. However, the correctness of that state is not yet ensured. Therefore, we feed the predicted α'_0 bits into the inverse functions of π, ρ, and θ, to calculate the corresponding input. Then we check if its capacity part matches the expected value (e.g., all zero at the first invocation). If it passes this check, we accept our α'_0 prediction, and calculate from that the predicted output of the invocation. Otherwise, we consider the attempt to have failed and terminate. The reason for not using the loopy-BP results of the rate part in the input variable node directly is that it does not benefit from this consistency check against the capacity part.

For a sponge function with more than one invocation, we repeat what we have done for the first invocation, however now the capacity of the input is verified instead against the capacity of the output of the previous invocation.

After recovering the input and output of every invocation, the remaining steps for calculating the complete padded sponge-function input are straightforward, involving XORing the rate-part inputs and outputs, as described in [22] and Fig. 2.

4 Experiments

4.1 Keccak Implementation and the Target Board

Our experiments target the 32-bit processor STM32F303RCT7, which has one ARM Cortex-M4 core, on a ChipWhisperer-Lite (CW-Lite) board [5]. Our Keccak implementation is based on the official reference C code [21] and our test application implements the four SHA-3 functions (SHA3-224, SHA3-256, SHA3-384, SHA3-512) and two extendable output functions (SHAKE128, SHAKE256). This device stores the intermediate states that we target as a sequence of fifty 32-bit words. We used the default compiler settings of the ChipWhisperer 5.2.1 software, such as optimization for space (`-Os` with `arm-none-eabi-gcc` v9.2.1).

4.2 Trace Recording

The ChipWhisperer-Lite board also includes a power-analysis oscilloscope, but that can record no more than 24 kilosamples per trace (at up to 105 MS/s). However, we wanted to record at least 15,000 clock cycles per trace, to cover at least four rounds of the Keccak-f[1600] permutation. That would have left us with very few points per clock cycle (PPC). To separate signals from 32 data bits processed in parallel, more samples per clock cycle will give us more dimensions in the signal space to achieve this. At the same time we wanted to preserve the phase lock between the oscilloscope's sampling clock and the CPU clock. Therefore, we used instead an NI PXIe-5160 [12] 10-bit oscilloscope, which can sample at 2.5 GS/s into 2 GB of sampling memory, and an NI PXIe-5423 [13] wave generator, as an external clock signal source, to supply the target board with a

5 MHz square wave signal. We installed the oscilloscope and waveform generator in the same PXIe chassis and configured both to use a common 100 MHz reference clock signal from the latter. With this setting, we collected traces at the highest sampling rate, at 500 points per clock cycle (500 PPC). This provided us with the flexibility to later digitally downsample to different PPC values, as needed.

After not using the on-board oscilloscope, we had to create an impedance-matched connection for the power signal. We used a 50 Ω coaxial cable to connect the oscilloscope and the CW-Lite's measure connector (JP10) [6]. However, JP10 taps the VDD connection of the CPU after a 13 Ω source impedance (R66+R67). This posed a problem: the 3.3 V DC level would have lead to a high current drain with the oscilloscope input configured to 50 Ω impedance and DC coupling, but if we don't have a 50 Ω impedance match on at least one end of the transmission line, reflections will add a lot of ripples to the recorded waveform. Therefore, we connected the coaxial cable to JP10 via a 37 Ω resistor (to better match the 50 Ω impedance of the cable) and a 10 nF capacitor (to block the 3.3 V DC component). Together with the 50 Ω impedance of the oscilloscope input, this capacitor forms a high-pass filter with a time constant of 0.5 µs (2.5 clock cycles), or a 3 dB cutoff frequency of about 320 kHz. This way, we both avoid ringing on the cable, by terminating it at both ends, and use AC coupling with an impulse response that decays within a few clock cycles.

We recorded traces while the device executed SHA3-512 on random inputs that each require 10 invocations of the Keccak-f[1600] permutation. At 2.5 GS/s, each 7,500,000-sample trace we recorded covers the first four complete rounds of Keccak-f[1600], and we recorded that for each invocation of the permutation. To exclude the possibility of trigger accidents (none were detected), we checked that all traces recorded have a Pearson correlation of at least 0.98 with the mean trace. Overall, we recorded 16 000 traces for interesting-clock-cycle detection, 64 000 for template building, and 1 000 for model evaluation. For the traces recorded for testing, see Sec. 4.4.

4.3 SASCA Model Building and Evaluation

Interesting clock cycle detection. Treating each bit in the intermediate byte as an independent variable, in [22] we had used multivariate linear regression to find the coefficient of determination (R^2) of these eight variables, and for the voltage-peak point in each clock cycle, we had evaluated the correlation with the intermediate byte. Using a selection threshold of $R^2 > 0.09$, we had created far shorter training traces for each intermediate byte to build its LDA-based template.

To detect the interesting clock cycle sets (ICs) for a 32-bit device, we assume that the four bytes in the same word will share the same sets. Therefore, we make a small change to our method for 8-bit devices. Rather than estimating the correlation between the samples and the 32-bit intermediate value with a 32-bit linear regression, as in eq. (1), which would need more traces to build, we instead estimate the correlation by adding the four R_f^2 values calculated from

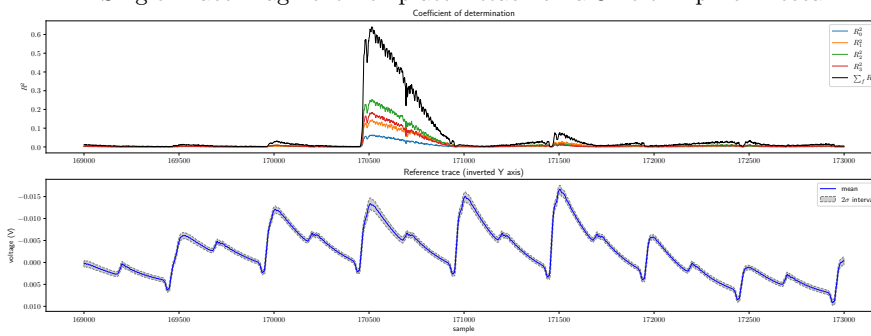

Fig. 3. The corresponding four R_f^2 values of $(\alpha'_0[0,0,0], \ldots, \alpha'_0[0,0,3])$ for each sample based on the 16 000 detection traces and their sum representing the detection results of the full 32-bit word (above), as well as the mean trace and the 2σ interval (below) at the same time samples.

the independently built 8-bit model (2) of each fragment byte in this 32-bit intermediate value. While this may be less accurate, due to slight overfitting, it significantly reduces the number of traces required.

Fig. 3 shows a small part of a mean trace, covering the 32-bit word consisting of $(\alpha'_0[0,0,0], \ldots, \alpha'_0[0,0,3])$, along with the corresponding four R_f^2 values for each point, based on the 16 000 detection traces. Most of the data dependency is limited to one clock cycle in the time interval shown. We also can see that the R values peak near the voltage peek, and can use this to speed up the selection of samples from our 500 PPC data. We sum 50 voltage samples around each voltage peak, and calculate $\sum_f R_f^2$ for that to decide whether this entire clock cycle should be included. Table 1 shows the number of interesting clock cycles selected for each intermediate word in the first round, with two different thresholds (0.04 and 0.01); the results of the omitted other three rounds are similar. We used the lower threshold $\sum_f R_f^2 > 0.01$. The SNR values of the points selected were in the range 0.01 to 3.43.

Template building and validation Considering the run time for building templates, we only wanted to deal with at most 2000 samples per trace after selecting the ICs. Given the numbers in Table 1, we therefore decided to resample the training traces from 500 PPC down to 10 PPC, by averaging 50 consecutive samples into one, effectively reducing the sampling rate to 50 MHz.

Using the 1000 traces in the validation set, Table 2 shows the resulting success rate and guessing entropy (as in [22]) for α'_0, while Table 3, 4, 5 show the corresponding results for intermediate states β_0, $\mathbf{C}_0$ and $\mathbf{D}_0$, respectively. The omitted data for other rounds looks similar. Our results for α'_0 and β_0 are not as good as the ones for the 8-bit processor in [22], and possibly not good enough for our enumeration procedure there, but they are suitable for SASCA. Note that, similar to the 8-bit experiments in [22], the results for the first lane of state α' in every round are worse than those for the other lanes in the same state. This

Table 1. Numbers of interesting clock cycles selected in round 0 with thresholds $\sum_f R_f^2 > 0.04$ (left) and $\sum_f R_f^2 > 0.01$ (right)

Lane$[i]$	$\mathbf{C}_0$		$\mathbf{D}_0$	
	first word	second word	first word	second word
[0]	13	15	3	2
[1]	12	16	3	1
[2]	10	16	3	1
[3]	11	17	3	2
[4]	12	16	3	1

Lane$[i,j]$	α'_0		β_0	
	first word	second word	first word	second word
[0, 0]	21	35	28	39
[1, 0]	73	90	54	68
[2, 0]	67	89	53	68
[3, 0]	68	88	49	66
[4, 0]	71	88	54	68
[0, 1]	64	85	47	61
[1, 1]	71	87	56	69
[2, 1]	67	80	46	61
[3, 1]	71	89	53	70
[4, 1]	69	74	48	55
[0, 2]	61	90	49	70
[1, 2]	68	84	51	67
[2, 2]	66	87	48	64
[3, 2]	73	84	52	68
[4, 2]	73	91	59	69
[0, 3]	64	88	47	64
[1, 3]	63	88	43	61
[2, 3]	71	90	54	69
[3, 3]	68	89	55	73
[4, 3]	77	85	50	58
[0, 4]	75	74	50	62
[1, 4]	79	90	49	67
[2, 4]	64	86	50	65
[3, 4]	65	91	52	70
[4, 4]	65	82	45	60

Lane$[i]$	$\mathbf{C}_0$		$\mathbf{D}_0$	
	first word	second word	first word	second word
[0]	31	35	36	30
[1]	31	33	25	33
[2]	32	35	25	26
[3]	31	38	17	32
[4]	35	36	34	55

Lane$[i,j]$	α'_0		β_0	
	first word	second word	first word	second word
[0, 0]	55	69	48	66
[1, 0]	130	139	91	114
[2, 0]	125	141	88	112
[3, 0]	120	142	88	111
[4, 0]	136	158	96	111
[0, 1]	120	147	86	111
[1, 1]	124	144	92	111
[2, 1]	129	143	85	103
[3, 1]	127	141	91	110
[4, 1]	141	144	100	103
[0, 2]	143	166	87	113
[1, 2]	121	135	89	110
[2, 2]	126	142	90	113
[3, 2]	133	148	92	109
[4, 2]	134	162	101	116
[0, 3]	120	145	87	112
[1, 3]	115	140	84	112
[2, 3]	131	146	96	112
[3, 3]	116	144	90	115
[4, 3]	143	158	106	112
[0, 4]	133	146	102	106
[1, 4]	134	146	104	117
[2, 4]	122	137	83	111
[3, 4]	131	140	87	110
[4, 4]	135	153	83	126

is because this lane is not rotated in steps π or ρ, resulting in fewer interesting clock cycles for the bits in this lane.

Since we use the marginal probabilities in the Loopy-BP network, we also show in Table 6 the average number of correct bits in different intermediate states from the 1000 validation traces. Because the probability tables are binary after marginalization, we define whether a bit is successfully predicted by checking if the probability of the correct candidate bit is higher than 0.5. The marginalized results also show that our templates predicted the state α'_Ω more successfully in these four rounds than the other states.

We also tried other choices of fragment size besides 4×8 bits: $11 + 11 + 10$ bits, 8×4 bits, 16×2 bits and 32×1 bit. We found that the choice of fragment size plays little role in the results after marginalization. As an example, Table 8 compares the performance of these different fragment sizes for the first bit in α'_0 after marginalization. Therefore, the fragment size can be chosen here to optimize computation time. For 11-bit fragments, calculating probability tables for 2^{11} candidates dominates the testing stage. On the other hand, with 32 1-bit fragments, the profiling stage takes longer, as we need to calculate a separate $\mathbf{\Sigma}_f$ for each fragment for LDA, the most time-consuming profiling step. Therefore, for our experiments with single-bit marginalization, the use of 4×8-bit fragment templates seemed a good compromise.

Table 2. Success rates (left) and guessing entropy (right) of templates in α'_0

(i,j) \ h	0	1	2	3	4	5	6	7
(0, 0)	0.036	0.046	0.021	0.023	0.029	0.050	0.012	0.015
(1, 0)	0.534	0.580	0.192	0.203	0.176	0.426	0.338	0.463
(2, 0)	0.459	0.558	0.259	0.152	0.206	0.457	0.352	0.386
(3, 0)	0.376	0.213	0.248	0.469	0.289	0.306	0.291	0.612
(4, 0)	0.522	0.377	0.370	0.246	0.275	0.384	0.506	0.351
(0, 1)	0.450	0.273	0.133	0.348	0.412	0.393	0.145	0.405
(1, 1)	0.473	0.242	0.435	0.449	0.342	0.373	0.347	0.487
(2, 1)	0.878	0.358	0.109	0.149	0.791	0.389	0.151	0.163
(3, 1)	0.360	0.332	0.259	0.279	0.173	0.358	0.366	0.531
(4, 1)	0.598	0.337	0.140	0.447	0.432	0.230	0.068	0.307
(0, 2)	0.717	0.292	0.110	0.140	0.790	0.427	0.162	0.284
(1, 2)	0.807	0.457	0.182	0.135	0.610	0.539	0.173	0.196
(2, 2)	0.423	0.214	0.110	0.789	0.383	0.277	0.176	0.777
(3, 2)	0.789	0.554	0.233	0.164	0.608	0.423	0.219	0.242
(4, 2)	0.435	0.255	0.533	0.357	0.268	0.390	0.601	0.537
(0, 3)	0.517	0.240	0.112	0.424	0.387	0.364	0.168	0.554
(1, 3)	0.740	0.318	0.118	0.124	0.577	0.460	0.217	0.305
(2, 3)	0.599	0.609	0.248	0.195	0.358	0.709	0.256	0.230
(3, 3)	0.359	0.295	0.362	0.277	0.271	0.388	0.559	0.382
(4, 3)	0.517	0.263	0.228	0.807	0.263	0.187	0.132	0.885
(0, 4)	0.635	0.424	0.122	0.290	0.445	0.288	0.061	0.183
(1, 4)	0.522	0.234	0.282	0.747	0.211	0.160	0.164	0.845
(2, 4)	0.767	0.504	0.151	0.138	0.411	0.503	0.273	0.267
(3, 4)	0.633	0.571	0.148	0.140	0.250	0.621	0.265	0.382
(4, 4)	0.860	0.359	0.111	0.178	0.838	0.397	0.146	0.203

(i,j) \ h	0	1	2	3	4	5	6	7
(0, 0)	47.918	37.603	72.631	66.417	58.449	36.038	84.323	69.128
(1, 0)	2.914	2.228	9.998	9.484	10.852	3.305	5.991	3.168
(2, 0)	3.296	2.191	7.754	13.492	10.111	2.793	4.998	4.433
(3, 0)	3.878	10.928	7.214	3.287	6.476	5.613	5.836	2.142
(4, 0)	2.576	4.329	4.455	7.112	8.444	4.172	2.976	5.131
(0, 1)	3.304	6.886	21.260	3.147	3.947	3.788	13.872	2.868
(1, 1)	2.725	7.374	2.801	3.769	5.946	4.577	5.000	3.175
(2, 1)	1.161	4.434	21.005	16.938	1.381	4.054	16.640	12.926
(3, 1)	4.909	4.014	7.500	7.730	13.265	3.903	5.013	2.675
(4, 1)	2.005	4.753	18.085	3.258	3.421	8.237	30.208	3.685
(0, 2)	1.573	5.369	22.824	15.404	1.378	3.447	12.988	7.555
(1, 2)	1.295	3.155	12.805	16.964	2.118	2.141	13.294	12.928
(2, 2)	3.110	8.532	21.392	1.404	5.061	6.394	13.671	1.291
(3, 2)	1.308	2.049	9.743	14.054	2.401	3.065	11.262	8.953
(4, 2)	2.866	6.688	2.319	5.176	8.416	4.756	1.986	2.902
(0, 3)	2.555	8.155	22.583	2.758	4.980	4.951	14.281	2.157
(1, 3)	1.509	5.179	17.242	16.478	2.061	3.089	9.198	6.468
(2, 3)	2.029	1.885	8.480	12.055	5.119	1.573	8.126	8.616
(3, 3)	4.863	5.171	4.425	6.750	9.046	4.186	2.511	5.356
(4, 3)	2.509	7.140	9.502	1.275	7.513	11.743	17.919	1.167
(0, 4)	1.866	3.518	19.914	4.703	3.229	6.271	33.439	7.764
(1, 4)	2.620	9.101	8.051	1.582	10.651	13.080	14.079	1.306
(2, 4)	1.549	2.537	13.825	18.494	3.763	2.569	7.648	8.671
(3, 4)	2.066	2.134	15.311	16.691	7.488	1.926	8.879	4.935
(4, 4)	1.212	4.708	25.596	12.427	1.255	4.436	19.452	9.656

Table 3. Success rates (left) and guessing entropy (right) of templates in β_0

(i,j) \ h	0	1	2	3	4	5	6	7
(0, 0)	0.063	0.060	0.026	0.034	0.035	0.039	0.022	0.017
(1, 0)	0.067	0.084	0.039	0.034	0.035	0.065	0.035	0.058
(2, 0)	0.055	0.073	0.049	0.043	0.046	0.070	0.039	0.052
(3, 0)	0.061	0.049	0.030	0.057	0.052	0.058	0.046	0.052
(4, 0)	0.045	0.066	0.059	0.044	0.056	0.080	0.048	0.051
(0, 1)	0.054	0.053	0.028	0.055	0.056	0.050	0.036	0.054
(1, 1)	0.062	0.052	0.061	0.054	0.049	0.043	0.043	0.043
(2, 1)	0.096	0.045	0.034	0.041	0.063	0.068	0.022	0.029
(3, 1)	0.047	0.063	0.043	0.055	0.045	0.055	0.038	0.064
(4, 1)	0.081	0.055	0.032	0.063	0.073	0.047	0.020	0.049
(0, 2)	0.055	0.062	0.029	0.033	0.067	0.056	0.029	0.035
(1, 2)	0.070	0.059	0.032	0.050	0.059	0.054	0.025	0.033
(2, 2)	0.064	0.049	0.028	0.065	0.049	0.057	0.029	0.067
(3, 2)	0.076	0.073	0.050	0.028	0.049	0.057	0.029	0.040
(4, 2)	0.064	0.072	0.080	0.053	0.051	0.064	0.065	0.058
(0, 3)	0.048	0.061	0.031	0.054	0.051	0.058	0.025	0.055
(1, 3)	0.088	0.062	0.031	0.030	0.051	0.085	0.032	0.050
(2, 3)	0.065	0.079	0.046	0.049	0.043	0.080	0.042	0.033
(3, 3)	0.055	0.067	0.053	0.038	0.044	0.065	0.050	0.050
(4, 3)	0.062	0.067	0.043	0.066	0.051	0.056	0.018	0.061
(0, 4)	0.063	0.080	0.028	0.063	0.050	0.044	0.022	0.031
(1, 4)	0.064	0.056	0.045	0.060	0.049	0.048	0.032	0.057
(2, 4)	0.073	0.074	0.037	0.025	0.048	0.072	0.044	0.054
(3, 4)	0.057	0.084	0.030	0.045	0.032	0.083	0.025	0.054
(4, 4)	0.077	0.061	0.020	0.041	0.163	0.144	0.038	0.055

(i,j) \ h	0	1	2	3	4	5	6	7
(0, 0)	29.099	31.206	66.016	55.164	49.097	41.215	77.061	72.161
(1, 0)	41.296	27.599	51.756	51.166	51.967	35.532	52.942	43.689
(2, 0)	45.505	31.928	47.914	52.142	52.209	34.579	52.971	48.676
(3, 0)	46.225	38.049	48.516	43.621	45.427	39.180	53.513	44.922
(4, 0)	44.973	33.773	41.436	53.657	45.342	29.965	45.826	47.460
(0, 1)	42.920	37.477	55.296	43.037	47.861	39.370	62.768	47.697
(1, 1)	44.062	41.569	43.692	47.447	49.794	41.370	51.363	48.475
(2, 1)	37.942	35.927	58.811	53.895	39.371	37.991	61.425	57.728
(3, 1)	49.000	33.408	47.756	51.132	54.896	38.350	52.300	44.095
(4, 1)	39.393	34.967	55.991	43.244	38.900	35.159	70.819	49.384
(0, 2)	40.071	34.954	57.510	54.016	40.775	36.531	61.750	54.218
(1, 2)	37.223	35.369	53.714	52.559	43.220	38.293	61.419	59.342
(2, 2)	42.703	38.837	58.793	42.058	44.949	41.459	63.710	40.046
(3, 2)	38.453	34.161	51.291	54.249	44.727	36.590	60.970	54.708
(4, 2)	40.264	35.120	43.146	49.255	43.270	33.627	44.398	47.326
(0, 3)	41.919	38.271	58.745	46.345	45.974	39.711	64.287	47.035
(1, 3)	35.889	34.639	56.862	54.783	43.745	32.077	56.285	52.072
(2, 3)	39.466	29.259	47.707	52.404	47.964	28.582	53.567	55.279
(3, 3)	47.758	34.515	41.710	50.016	45.893	35.975	51.737	50.468
(4, 3)	36.454	33.344	46.953	38.291	38.767	35.921	63.725	36.496
(0, 4)	36.658	30.434	57.476	47.138	45.926	38.289	73.197	47.420
(1, 4)	41.344	35.637	47.295	43.026	50.578	45.520	64.383	42.099
(2, 4)	38.915	32.309	55.223	55.883	42.468	35.518	55.496	49.946
(3, 4)	42.077	29.945	53.072	53.553	51.580	35.255	56.883	48.758
(4, 4)	37.359	38.467	61.073	50.593	15.980	21.212	53.895	33.795

Table 4. Success rates (left) and guessing entropy (right) of templates in $\mathbf{C}_0$

(i,j) \ h	0	1	2	3	4	5	6	7
(0, 0)	0.027	0.036	0.016	0.030	0.041	0.060	0.019	0.042
(1, 0)	0.025	0.044	0.020	0.039	0.034	0.066	0.015	0.036
(2, 0)	0.027	0.043	0.027	0.039	0.047	0.051	0.018	0.043
(3, 0)	0.032	0.047	0.017	0.045	0.045	0.056	0.015	0.046
(4, 0)	0.026	0.048	0.022	0.037	0.066	0.075	0.018	0.048

(i,j) \ h	0	1	2	3	4	5	6	7
(0, 0)	58.015	39.596	65.977	51.890	42.605	31.637	76.724	49.012
(1, 0)	58.307	40.936	69.313	49.246	43.310	32.581	77.534	46.917
(2, 0)	56.889	42.208	66.796	51.466	36.740	33.989	72.759	47.559
(3, 0)	59.543	41.348	68.157	51.589	38.406	31.291	74.440	44.055
(4, 0)	60.075	39.145	69.823	49.706	33.487	29.861	65.547	43.852

Table 5. Success rates (left) and guessing entropy (right) of templates in $\mathbf{D}_0$

(i,j) \ h	0	1	2	3	4	5	6	7
(0, 0)	0.013	0.020	0.006	0.012	0.016	0.010	0.008	0.013
(1, 0)	0.013	0.016	0.010	0.016	0.016	0.016	0.008	0.015
(2, 0)	0.008	0.016	0.011	0.014	0.012	0.021	0.005	0.016
(3, 0)	0.010	0.020	0.009	0.013	0.016	0.019	0.012	0.011
(4, 0)	0.017	0.006	0.011	0.012	0.020	0.020	0.009	0.019

(i,j) \ h	0	1	2	3	4	5	6	7
(0, 0)	91.069	84.318	92.714	87.537	84.127	73.385	93.005	85.368
(1, 0)	87.800	84.453	89.139	86.089	78.383	78.650	90.992	80.381
(2, 0)	89.727	86.831	89.815	88.058	76.028	78.165	92.148	84.787
(3, 0)	93.462	83.278	92.638	84.953	84.579	70.239	92.599	82.877
(4, 0)	91.890	81.804	90.937	88.506	80.511	76.263	91.385	76.724

Table 6. Average (μ) and standard deviation (σ) of the number of correct bits found after marginalization of the byte tables (out of 1600 bits in α'_Ω and β_Ω, and 320 bits in $\mathbf{C}_\Omega$ and $\mathbf{D}_\Omega$, respectively).

State	α'_0	β_0	α'_1	β_1	α'_2	β_2	α'_3	β_3
μ	1353.432	1093.831	1352.345	1094.108	1353.010	1095.214	1353.998	1095.555
σ	15.854	17.746	16.313	17.103	16.028	17.255	15.243	17.265
State	$\mathbf{C}_0$	$\mathbf{D}_0$	$\mathbf{C}_1$	$\mathbf{D}_1$	$\mathbf{C}_2$	$\mathbf{D}_2$	$\mathbf{C}_3$	$\mathbf{D}_3$
μ	211.007	187.974	211.480	187.722	211.509	187.489	211.051	187.565
σ	7.992	9.049	8.181	7.999	8.230	7.774	8.077	8.189

Table 7. Results of terminating bitwise SASCA on the 32-bit device

Network	#Steady	#Iteration				#Correct Traces								
		Median	Mean	σ	Max	Input	α'_0	β_0	α'_1	β_1	α'_2	β_2	α'_3	β_3
4-round	1000	25	25.421	0.573	28	1000	1000	1000	1000	1000	1000	1000	1000	1000
3-round	1000	30	30.331	1.247	34	1000	1000	1000	1000	1000	1000	1000	N/A	N/A
2-round	1000	51	51.710	4.391	72	1000	1000	1000	1000	1000	N/A	N/A	N/A	N/A

Table 8. Fragment size had little influence on accuracy of bit prediction, as illustrated here for the first bit in α'_0, using several metrics: predicted marginalized probability of correct candidate from the first trace (Prob.), number of correct bit predictions over 1000 validation traces (#Success), maximum and average deviation ($|\epsilon|$) of probability among these 1000 trials from the predictions made by four-byte fragment templates.

Fragments	$11+11+10$ bits	4×8 bits	8×4 bits	16×2 bits	32×1 bit
Prob.	0.752437	0.750506	0.752002	0.752274	0.751888
#Success	729	730	733	733	732
Max $\|\epsilon\|$	0.026377	–	0.010587	0.013578	0.013906
Average $\|\epsilon\|$	0.002809	–	0.001652	0.001872	0.002043

Evaluation on different networks. We now evaluate how well the loopy-BP algorithm works when fed with marginalized binary probability tables from a single validation trace, along with 1024 known bits in the capacity part of the input. Table 7 shows the number of validation traces reaching a steady state, along with statistics on the number of iterations required, and the number of validation traces where all intermediate bits were recovered. We provide results from three networks, covering two, three, and four rounds, respectively. Although intermediate values of all the validation traces are successfully recovered in these three networks, we can see that we will need fewer iterations to reach a steady state with the four-round network. Fig. 4 (left) shows the percentage of successfully recovered traces (defined as all the bits of α'_0 are recovered correctly) out of the 100 validation traces for these three factor-graph networks as a function of the number of loopy-BP iterations. It takes fewer iterations to completely recover state α'_0 than it takes for the network to stabilize. It appears that the two-round network takes more iterations to recover all validation traces correctly than the larger networks.

Fig. 4 (right) shows the percentage of successfully recovered traces out of 1000 validation traces when we provide different numbers of known bits (not just 1024), to explore the situation when the size of the rate (unknown) and capacity (known) parts of the permutation input vary in different sponge functions. When the number of unknown bits increases beyond half of the full state, including up

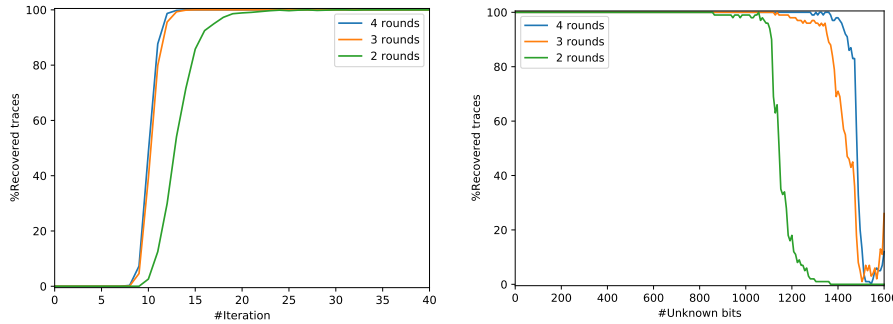

Fig. 4. Percentage of successfully recovered traces for the different factor-graph networks (with different numbers of rounds observed), as a function of the number of loopy-BP iterations (left) and the number of unknown input bits (right).

to the $1600-128\times2 = 1344$ unknown bits in SHAKE128, the four-round network performs better than the others. Therefore we chose the four-round network for our final version of the attack.

4.4 Loopy Belief-Propagation Results

Results for the SHA-3 and SHAKE functions. We recorded five groups of 1000 test traces. Each group had a different range of SHA3-512 input lengths, requiring 1, 2, 4, 5, or 10 invocations of Keccak-f[1600] to absorb, respectively. Table 9 shows the number of successfully recovered inputs for each of these test traces, and related statistics on the number of iterations required. We can see that all the inputs were successfully recovered, after about 25–30 iterations.[2]

Apart from SHA3-512, we also recorded test traces for other Keccak[c] sponge functions, including the other three SHA-3 variants and the two SHAKE extendable output functions. It is noteworthy that, because our SASCA network of Keccak-f[1600] relies on the capacity part of the output of the previous invocation, the functions with a shorter capacity part (c bits) may encounter a lower success rate or may require more iterations to reach a steady state. Table 10 shows some results of these five functions with inputs that can be absorbed in one or two invocations. We can see the results meet our expectation that the shorter the capacity part, the lower the number of inputs we successfully recover, and the more iterations we need to reach a steady state, despite all success rates remaining close to 1. It is also noteworthy that in the same function, if the success rate for inputs requiring one invocation is p, that for inputs requiring two invocations should be p^2, which is also consistent with our results.

[2] Recall that Kannwischer et al.'s results [7] for their *all-zero public input* set, which is similar to our experiments with very short Keccak[c] input, were worse than those for their *random public input* set. We did not observe such variability in our setting, i.e. the success rates or the number of iterations required did not significantly vary with the input length of Keccak[c], even down to just one byte.

Table 9. Results of recovering the SHA3-512 inputs with multiple invocations of Keccak-f[1600] permutation.

#Invocations	#Traces Recovered	#Iteration			
		Median	Mean	σ	Max
1	1000	25	25.399	0.804	28
2	1000	26	25.629	0.619	29
4	1000	26	25.575	0.611	29
5	1000	26	25.615	0.621	31
10	1000	25	25.364	0.552	28

Table 10. Results of recovering the functions in the SHA-3 family with one and two invocations by the four-round network.

Function	c	#Inv.	#Rec.	#Iteration*			
				Median	Mean	σ	Max
SHA3-512	1024	1	1000	25	25.399	0.804	28
		2	1000	26	25.629	0.619	29
SHA3-384	768	1	1000	27	26.838	0.942	29
		2	1000	27	27.061	0.662	30
SHA3-256	512	1	1000	29	28.646	1.246	32
		2	998	29	28.679	0.761	33
SHAKE256		1	997	29	29.054	1.272	34
		2	996	29	28.996	0.926	37
SHA3-224	448	1	1000	29	29.106	1.255	33
		2	996	29	29.440	0.971	37
SHAKE128	256	1	979	31	30.897	1.512	39
		2	971	31	31.206	1.212	39

* Only invocations that reached a steady state are taken into account.

Table 11. Results of recovering the functions in the SHA-3 family with one invocation by the three-round network.

Function	c	#Rec.	#Iteration*			
			Median	Mean	σ	Max
SHA3-512	1024	1000	30	30.064	1.720	35
SHA3-384	768	1000	34	34.066	2.057	41
SHA3-256	512	999	38	38.023	2.924	46
SHAKE256		999	39	38.789	2.727	50
SHA3-224	448	992	39	39.284	2.947	52
SHAKE128	256	921	43	43.512	5.033	107

* Only invocations that reached a steady state are taken into account.

Apart from our final 4-round version, we have also tried these experiments with the three-round network. Table 11 shows the results of recovering 1000 inputs with one invocation from the test traces of the six SHA-3 or SHAKE functions. It appears that the four-round network provides better results, suggesting that recording longer traces covering more rounds helps to push the success rate much closer to 1.

5 Conclusion and Outlook

With the help of LDA-based dimensionality reduction, we successfully built fragment templates that generate separate probability tables for each byte in the 32-bit words of the targeted intermediate states. The quality of our templates is sufficient for creating per-bit marginalized observation factors from which a bitwise loopy-BP network can reconstruct the full input and output of each

invocation of Keccak-f[1600], using also knowledge about a part of its input, as given by the sponge construction. From that we can easily reconstruct the padded arbitrary-length inputs of the Keccak sponge functions. Interestingly, our results so far indicate that, although the Keccak[c] functions with a longer capacity have cryptographically a higher security margin, that actually helps in our attack strategy. Our results suggest that this method will also work for Keccak-based sponge functions with a shorter capacity, especially when observing more rounds by recording longer traces. We also expect that this attack strategy can easily be applied to other SHA-3-derived functions, such as cSHAKE, KMAC, TupleHash and ParallelHash, defined in NIST Special Publication 800-185 [9], which also use the Keccak[256] or Keccak[512] functions, except for different padding methods.

Our fragment templates reconstruct full-state information stored in larger word sizes (such as 32 bits) than are practical with regular template or stochastic-method attacks, by using the LDA technique to project traces onto subspaces that are only related to a manageable part of the state. Further improvements should be possible, for example lowering the R^2 threshold to include more interesting clock cycles may help to build templates with even higher success rates, at the expense of more computational time required for profiling. We expect this fragment-template technique can be extended beyond attacks on SHA-3 related functions. Also, so far we have only demonstrated this technique using the same board for profiling and attack, therefore its portability remains to be investigated; however LDA-based techniques have previously already been shown to help with portability of templates across boards [4].

References

1. Atmel Corporation: AVR XMEGA Microcontrollers, `http://www.atmel.com/products/microcontrollers/avr/avr_xmega.aspx`, accessed March 2014
2. Chari, S., Rao, J.R., Rohatgi, P.: Template attacks. In: International Workshop on Cryptographic Hardware and Embedded Systems. pp. 13–28. Springer (2002)
3. Choudary, M.O., Kuhn, M.G.: Efficient stochastic methods: profiled attacks beyond 8 bits. In: International Conference on Smart Card Research and Advanced Applications (CARDIS). pp. 85–103. Springer (2014)
4. Choudary, M.O., Kuhn, M.G.: Efficient, portable template attacks. IEEE Transactions on Information Forensics and Security **13**(2), 490–501 (Feb 2018)
5. CW1173: ChipWhisperer-Lite product data sheet (13 Feb 2018). `https://media.newae.com/datasheets/NAE-CW1173_datasheet.pdf`
6. ChipWhisperer-Lite arm edition, schematic, rev 03. `https://github.com/newaetech/chipwhisperer/raw/develop/hardware/capture/chipwhisperer-lite-32bit/cw-lite-arm-main.pdf`
7. Kannwischer, M.J., Pessl, P., Primas, R.: Single-trace attacks on Keccak. IACR Transactions on Cryptographic Hardware and Embedded Systems **2020**(3), 243–268 (Jun 2020). https://doi.org/10.13154/tches.v2020.i3.243-268
8. Kannwischer, M.J., Pessl, P., Primas, R.: Single-trace attacks on Keccak. CHES 2020 presentation slides. `https://iacr.org/submit/files/slides/2020/tches/ches2020/30391/slides.pdf`

9. Kelsey, J., Chang, S., Perlner, R.: SHA-3 derived functions: cSHAKE, KMAC, TupleHash and ParallelHash (2016). https://doi.org/10.6028/NIST.SP.800-185
10. Luo, P., Fei, Y., Fang, X., Ding, A.A., Kaeli, D.R., Leeser, M.: Side-channel analysis of MAC-Keccak hardware implementations. IACR Cryptology ePrint Archive **2015**, 411 (2015)
11. MacKay, D.J.C.: Information theory, inference and learning algorithms. Cambridge University Press (2003)
12. NI PXIe-5160, `http://www.ni.com/en-gb/support/model.pxie-5160.html`
13. NI PXIe-5423, `http://www.ni.com/en-gb/support/model.pxie-5423.html`
14. 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
15. O'Flynn, C., Chen, Z.D.: ChipWhisperer: an open-source platform for hardware embedded security research. In: Prouff, E. (ed.) Constructive Side-Channel Analysis and Secure Design. pp. 243–260. Springer, Cham (2014)
16. 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)
17. 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)
18. Taha, M., Schaumont, P.: Differential power analysis of MAC-Keccak at any keylength. In: Sakiyama, K., Terada, M. (eds.) Advances in Information and Computer Security. pp. 68–82. Springer, Berlin, Heidelberg (2013)
19. 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)
20. Veyrat-Charvillon, N., Gérard, B., Standaert, F.X.: Soft analytical side-channel attacks. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 282–296. Springer (2014)
21. Extended Keccak code package, `https://github.com/XKCP/XKCP`, accessed April 2019, `lib/low/KeccakP-1600/Compact64/KeccakP-1600-compact64.c`
22. You, S.C., Kuhn, M.G.: A template attack to reconstruct the input of SHA-3 on an 8-bit device. In: Bertoni, G.M., Regazzoni, F. (eds.) Constructive Side-Channel Analysis and Secure Design. pp. 25–42. Springer International Publishing, Cham (2021)