Single-trace template attacks on permutation-based cryptography

Shih-Chun You

University of Cambridge
Department of Computer Science and Technology
Girton College

December 2022

This thesis is submitted for
the degree of Doctor of Philosophy

Declaration

This thesis is the result of my own work and includes nothing which is the outcome of work done in collaboration except as declared in the Preface and specified in the text. I further state that no substantial part of my thesis has already been submitted, or, is being concurrently submitted for any such degree, diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. It does not exceed the prescribed word limit for the relevant Degree Committee.

Single-trace template attacks on permutation-based cryptography

Shih-Chun You

Summary

The *Template Attack* introduced by Chari, Rao, and Rohatgi has been widely used in *Side-Channel Attacks* on cryptographic algorithms running on microcontrollers. In 2014, Choudary and Kuhn successfully optimized a variant of this technique, based on *Linear Discriminant Analysis* (LDA), to reconstruct the actual values of a byte handled by a single microcontroller machine instruction, instead of only its *Hamming weight*. While their attack targeted single `LOAD` instructions, I believe this method can be even more powerful when attackers target intermediate values inside a cryptographic algorithm, for such values can be related to more than single instructions, and further mathematical tools can be applied for value enumeration or error correction when multiple target values can be checked against one another.

In my dissertation, I first describe how I successfully built LDA-based templates for full-state recovery on target intermediate bytes in the SHA3-512 hash function implemented on an 8-bit device, which I combined with a three-layer enumeration technique for error correction to recover all the input values of this hash function from a single trace recording. To demonstrate an alternative technique, I also combined these template recovery results with a modified belief-propagation procedure for error recovery, adapting a 2020 design by Kannwischer et al. In combination, these techniques reached success rates near 100% in recovering all SHA3-512 input bytes.

Secondly, I introduce the *fragment template attack* to make this technique feasible for targeting 32-bit microcontrollers. It cuts a 32-bit intermediate value into smaller pieces, applying the LDA-based template attack by independently building templates for these pieces. For a SHA-3 implementation on a 32-bit device, the quality of these fragment templates is good enough that their predictions can reconstruct the full arbitrary-length SHA-3 or SHAKE inputs with a very high success rate when combined with belief propagation. Thirdly, I also show that a combination of fragment template attack, belief propagation, and key enumeration can recover the key used in an Ascon-128 implementation.

My experiments show how LDA-based templates can pose a threat to cryptographic algorithms once it is combined with belief propagation and key enumeration, even when they are implemented on a 32-bit device and in applications where keys are only used once. Therefore, we should not underestimate these risks and it is important to analyze the resilience against template attacks, in addition to DPA-style correlation attacks, when designing or implementing cryptographic algorithms and evaluating their security level.

Acknowledgements

Firstly, I would like to thank my principal supervisor Markus Kuhn for inspiring me with ideas in research, helping me to improve my English writing and speaking, and supporting me whenever I needed in these four years. I also had a good time with my second advisor Sergei Skorobogatov, especially when we collaborated on an ECC project and I therefore gained much experience with hardware. The administrative and technical staff in Computer Lab also provided me with lots of support when I was working in the department. Also thanks Sumanta Sarkar and Feng Hao for the collaboration in the Ascon project, as well as Eric Chun-Yu Peng and Dimitrije Erdeljan for sharing the office.

Secondly, I would like to thank Jiun-Peng Chen, Chen-Mou Cheng, and Bo-Yin Yang for referencing me four years ago for my Ph.D. application to Cambridge. In these four years, Cambridge Trust and the Ministry of Education, Taiwan were the main financial supporters of my tuition fee and living cost, as well as my parents, my grandma, and my uncle also provided me with some additional "pocket money" for traveling in Europe and living with better quality.

Thirdly, I would like to thank my parents and my little brother for the weekly family Skype meeting, and the co-founders Dr. Chiang and Mr. Huang of my beloved online chatting group, Gekifujomon, with my close friends during the lockdowns.

Last but not the least, I would like to thank future readers of my thesis and technical report.

Contents

Notation and glossary

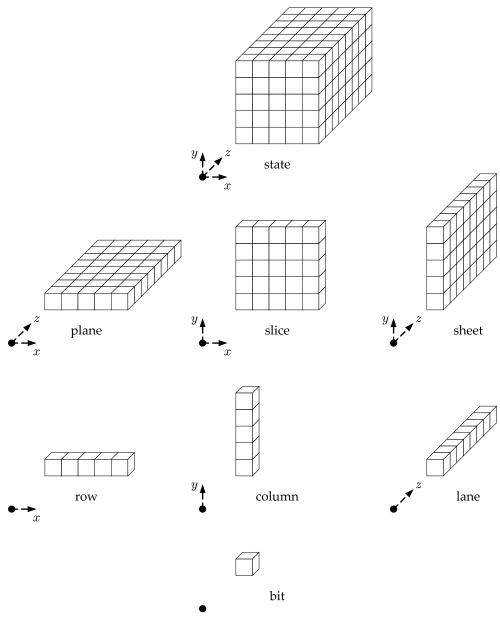
General operations

$X \| Y$	concatenation of two bitstrings
$X \oplus Y$	bitwise XOR
$X \vee Y$	bitwise OR
$X \wedge Y$	bitwise AND
$\neg X$	NOT, operating on any size of bitarray by flipping all the bits
$\mathbf{Rot}(X, n)$	a function that rotates a bitstring X to the right by n bit, no matter the endianness, e.g. given a 3-bit sequence $Y = Y[0] \| Y[1] \| Y[2]$, $\mathbf{Rot}(Y, 1) = Y[2] \| Y[0] \| Y[1]$
$\mathbf{Trunc}(X, n)$	a function that truncates a bitstring X to it left-most n bits, e.g. given a 4-bit sequence $Y = Y[0] \| Y[1] \| Y[2] \| Y[3]$, $\mathbf{Trunc}(Y, 2) = Y[0] \| Y[1]$

In this thesis, $\oplus, \vee, \wedge$ can operate on bits, bitstrings, two- or three-dimensional bit-arrays if they have the same sizes or shapes. For example, if there are two 3-bit strings A and B. $A \oplus B = (A[0] \oplus B[0]) \| (A[1] \oplus B[1]) \| (A[2] \oplus B[2])$

KECCAK notation

KECCAK-f	KECCAK permutation
KECCAK-$f[1600]$	KECCAK permutation with a 1600-bit state
S	a 1600-bit sequence, representing the input or output of KECCAK-$f[1600]$
state	a 5-by-5-by-64-bit array where KECCAK-$f[1600]$ is executed
row	a 5-bit sequence along the x-axis in a state



i	the coordinate of bits in a row, where additions and subtractions of i will be on $\mathbb{Z}_5$
column	a 5-bit sequence along the y-axis in a state
j	the coordinate of bits in a column, where additions and subtractions of j will be on $\mathbb{Z}_5$
lane	a bit sequence along the z-axis in a state.
k	the coordinate of bits in a lane; additions and subtractions of the coordinates will be on $\mathbb{Z}_{64}$
${}^{\mathbf{4}}k$, ${}^{\mathbf{8}}k$, ${}^{\mathbf{16}}k$, ${}^{\mathbf{32}}k$	the coordinates of nibbles, byte, 16-bit and 32-bit words in a lane, respectively, where additions and subtractions of these coordinates will be on $\mathbb{Z}_{16}$, $\mathbb{Z}_8$, $\mathbb{Z}_4$, and $\mathbb{Z}_2$
$L_{(i,j)}$	the lane in a state with (i, j), e.g., $L_{(0,0)}$ is the first lane in a state
plane	a two-dimensional space, on which all the bits have the same y coordinate in a state
sheet	a two-dimensional space, on which all the bits have the same x coordinate in a state
slice	a two-dimensional space, on which all the bits have the same z coordinate in a state
Ω	the round index of the Keccak permutation, ranging from 0 to 23 in Keccak-$f[1600]$
rate	the number of bits that each invocation of Keccak-$f[1600]$ will absorb or squeeze out in a sponge function, denoted as r
S_r	the bitstring representing the rate part of S
capacity	the number of bits other than the rate part of S, denoted as c and therefore $r + c = 1600$
S_c	the bit sequence representing the capacity part of S, where $S = S_r \| S_c$
pad10*1	a bitstring starting with 1, then an arbitrary number of 0s, and ending with 1 ($1\|0^*\|1$), of which the minimal size is two, used for padding in the Keccak family
Keccak$[c](N, d)$	a Keccak sponge function with capacity c, where N is the bitstring absorbed by the function and d is the length of the function's output

SHA-3	a family of hash functions based on the KECCAK sponge function, including SHA3-512, SHA3-384, SHA3-256, and SHA3-224
SHAKE	extendable-output functions (XOFs) based on the KECCAK sponge function: SHAKE256 and SHAKE128

ASCON notation

S	a 320-bit sequence, representing the input or output of the ASCON permutation
state	a 5-by-64-bit array where the ASCON permutation is executed, which is in the same shape as a plane in KECCAK-$f[1600]$
row	a 5-bit sequence along the x-axis in a state, where the coordinate i is defined the same way as the one in KECCAK-$f[1600]$
lane	a 64-bit sequence along the z-axis in a state, where the coordinates k, $^{\mathbf{4}}k$, $^{\mathbf{8}}k$, $^{\mathbf{16}}k$, and $^{\mathbf{32}}k$ are defined the same way as those in KECCAK-$f[1600]$
L_i (or L'_i)	the lane with the x coordinate i, e.g. L_0 is the first lane in a state
Ω	the round index of the ASCON permutation
rate	the number of bits that each invocation of the ASCON permutation will absorb or squeeze out in a sponge function, denoted as r, and S_r represents the rate part of S
capacity	the number of bits other than the rate part of S, denoted as c and therefore $r + c = 1600$, and S_c represents the capacity part of S
ASCON-128	one of the ASCON functions for authenticated encryption with associated data (AEAD)

Abbreviations

BP	belief propagation
CPA	correlation power analysis (or correlation power attack)
DPA	differential power analysis (or differential power attack)
GE	guessing entropy

HD	Hamming distance
HW	Hamming weight
LDA	linear discriminant analysis
loopy-BP	loopy belief-propagation procedure
LWC	lightweight cryptography
PoI	points of interest
PPC	points per clock cycle
PQC	post-quantum cryptography
SASCA	soft analytical side-channel attack
SCA	side-channel attack (or side-channel analysis)
SR	success rate
TA	template attack

Chapter 1

Introduction

Recent years have been a critical period for the cryptography community in that it is about to standardize at least two new types of cryptographic algorithms for the next generation. The National Institute of Standards and Technology (NIST) [2] published in 2016 a call [3] for proposals for *Post-Quantum Cryptography* (PQC) algorithms that can survive the potential risk posed by quantum computing. There, Shor's algorithm [4] promises to dramatically reduce the time complexity of solving the problems of integer factorization and finding discrete logarithms [5] in certain cyclic groups, whereas the computational infeasibility of these problems is a prerequisite for the security of existing asymmetric-cryptography standards based on *RSA* [6], *Diffie–Hellman* [7] and *Elliptic Curve Cryptography* (ECC) [8, 9] constructs. Secondly, NIST published another call [10] in 2018 for *Lightweight Cryptography* (LWC), which focuses on authenticated encryption and secure hash constructs optimized for use in highly resource-constrained applications, such as radio-frequency identification (RFID [11]) devices or authentication chips embedded in other components.

Since many of these algorithms may be used in devices that are expected to be physically tamper-resistant, it is important to fully understand not only the security risks posed by different types of cryptanalysis but also by *Side-Channel Attacks* (SCA) [12, 13, 14], which exploit physical information leaked from hardware devices during the execution of these algorithms, such as power-supply current variations and electromagnetic emissions.

Modern cryptographic constructs and implementations pose more challenges for side-channel attackers than before, in many ways. Better randomization of algorithms and protocols makes it more difficult to observe the same key being used many times, a prerequisite for correlation-based side-channel attacks such as *Differential Power Analysis* (DPA) [13]. In addition, the mathematical structure of some of these new algorithms is more complicated, for example through larger internal state spaces. Furthermore, embedded devices nowadays mainly rely on 32-bit rather than 8-bit microcontrollers, where more bits processed in parallel make it more difficult for attackers to distinguish one bit from another in the leakage signal. The focus of this thesis is to explore techniques available to attackers to tackle some of these challenges and to demonstrate that it is still possible for experienced attackers to recover the secrets given these more complex hardware and software conditions.

I, therefore, present here examples of advanced side-channel attacks on a 32-bit processor, running different cryptographic algorithms related to a recently standardized hash-function family, *Secure Hash Algorithm 3* (SHA-3) [15]. I got particularly interested in the SHA-3 family not only on its own, for its increasing use as a hash function and random-bit stream generator, but also because it appears as a component in more than one of the recent post-quantum candidate algorithms. And at least one of the lightweight authenticated encryption candidates, Ascon, is also based on a permutation function closely related to the Keccak permutation at the heart of SHA-3.

1.1 Side-channel attacks

In contrast to other types of cryptanalysis [16, 17, 18], the main characteristic of side-channel attacks is to acquire information about intermediate values during the execution of cryptographic algorithms from certain unintentional, noisy *side channels*. Then attackers can use this information, possibly along with some known ciphertexts or plaintexts, to reconstruct targeted secrets, such as the key. Such side channels can be observations of the computation time [12], high-frequency fluctuations of the power consumption [13, 19, 20], or other accidentally emitted electro-magnetic signals [21, 22] from a working device.

Precursors of Side-Channel Attacks (SCA) can be traced back to some early-to-mid 20$^{\text{th}}$ century eavesdropping attacks on military [23] and diplomatic communication systems [24, Chapter 8, pp. 109–112]. In 1996, Kocher introduced his *Timing Attack* [12] on implementations of RSA. His idea was to repeatedly time the modular exponentiation function required for RSA decryption, and then use an adaptive chosen-ciphertext attack to recover key bits in the secret exponent, facilitated by preparing test ciphertexts for which the time needed to perform a particular modular multiplication differs in the square-and-multiply algorithm used. Later, in 1998, Kocher, Jaffe, and Jun demonstrated with their Differential Power Analysis (DPA) attack how to recover the key used in *Data Encryption Standard* (DES) [25], with information observed from power-consumption traces measured from the device during the execution of the decryption procedure [13]. After these successful attacks on two side channels of standardized cryptographic algorithms, implementers increasingly realized that Side-Channel Attacks may pose serious security threats.

1.1.1 Categories of side-channel attacks

Mangard et al. [20, Sec. 1.2] categorize side-channel attacks by two criteria: passive v.s. active and invasive v.s. semi-invasive v.s. non-invasive.

Passive and Active attacks As mentioned previously, apart from the intended I/O channels, hardware devices can also interact with the environment through other channels. These interactions can be in both directions, i.e., the device may leak some information through such channels, while some information from the environment may also be passed into the device through these channels. Therefore, sometimes attackers can not only *passively* collect the side-channel information, but also *actively* affect the running device via these channels. Mangard et al. describe *passive attacks* as the case where attackers recover secrets by observing the effects of computation, such as the execution time or the power consumption when the device is working largely or even entirely within its specification. On the other hand, they describe *active attacks* as the situation where attackers manipulate the inputs and/or the environment of a device to make it work abnormally, and then reveal the secret by exploiting the abnormal behavior of the device [20, Sec. 1.2].

Invasive, semi-invasive, and non-invasive attacks Since side-channel attacks exploit those channels other than the normal ways of communication with target devices, attackers may cause temporary changes or even damage the devices when they access their target interface. Skorobogatov describes three types of side-channel attacks, categorized into invasive, semi-invasive, and non-invasive attacks [26]. Non-invasive attacks only exploit interfaces that can be directly accessed, i.e., there are no permanent changes to the device; semi-invasive attacks require depackaging or decapsulation of the device, but no direct electrical contact to a chip surface; while invasive attacks include all the other more aggressive attacks, essentially without limits [20, Sec. 1.2], including microprobing and circuit modification.

Common means of attacks Here I provide some examples of side-channel attacks. Based on some early ideas [27, 28], Biham and Shamir introduced their *Differential Fault Analysis* (DFA) in 1997 [29]. With the assumption that a fault occurs for one single intermediate bit each time during DES encryption, they built a model to recover the key of DES by comparing a few resulting faulty ciphertexts. Their method has been generalized and used to attack other cryptographic algorithms (e.g., AES [30]), and it also gradually became a major methodology for active attacks, and it can be used with a variety of means to inject faults. In 2002, Skorobogatov and Anderson introduced their *Optical Fault Induction Attacks* [31]. They demonstrated this semi-invasive attack by flipping individual bits in the SRAM array of a depackaged microcontroller (Microchip PIC16F84) with a $30 photoflash lamp. In 2009, Fukunaga and Takahashi attempted to supply *glitchy clock signals* for devices running their target block ciphers and then collected faulty results to reduce the key candidate space [32].

On the other hand, various side channels have been studied for passive attacks. Assuming that the temperature leakage is linearly correlated to the power consumption, Hutter and Schmidt collected leaked power information from the dissipated heat of devices [33], which is also known as the *Temperature Side Channel*. Genkin et al. introduced their *Acoustic Cryptanalysis* in 2014 [34]. Monitoring the sound generated by the computer, they extracted full 4096-bit RSA keys from the decryption of their chosen ciphertexts within one hour.

Besides the above interesting, but relatively niche side-channel attacks, most researchers, however, collect the side-channel information by monitoring the current flows in their target device. Based on their different monitoring means, the side channels that these attacks exploit can be further categorized into *Power Side Channels* [13, 20] and *Electromagnetic (EM) Side Channels* [22], and therefore the corresponding analysis of information from these channels are usually referred to as *Power Analysis* and *EM Analysis*, respectively.

According to Kocher's description [13], when conducting an attack via power side channels, we can insert a small resistor into the GND or V_{DD} line of the working device, where the voltage drop across this resistor is proportional to the current flowing into or out of the device. With an oscilloscope recording that voltage drop, I refer to the resulting one-dimensional array of time samples as a *power trace*, or also simply a *trace* in this thesis. Such a recording method can provide an aggregated view of the current flow and power consumption of the working device. Meanwhile, Agrawal et al. suggested that we can go beyond this aggregated view with more flexible EM side channels [22], where attackers apply EM probes to detect the EM signal induced from the (sometimes decapsulated [35]) circuit in their target devices. An example of the flexibility of EM analysis was demonstrated in 2012 by Heyszl et al, whose experiments showed that it is possible to detect localized electromagnetic signals induced from current flows from a small region of a circuit [36].

However, from my perspective, such flexibility also makes the EM attack experiments more difficult to control. For instance, the positioning of the probes can significantly affect the recording [36, 37]. In addition, many researchers preferred power analysis rather than EM analysis when they developed most of the currently popular passive attack methods [13, 38, 39] and studied SCA resilience of new candidates for standard cryptographic algorithms [40, 41]. As a result, I chose power analysis as my experimental method in this thesis.

1.1.2 Extracting information from power traces

After we collect power traces from a working device, the next step will be the analysis of that raw data. In general, the goal is to find the relations between the value of samples in recorded traces and the targeted secrets (e.g., the key) used by the program executed on the device.

Horizontal leakage and vertical leakage Since early publications of attacks [12, 13], researchers have been commonly using the term *leakage* to describe secret-related information *leaked* through side channels. As each sample of a power trace indicated the voltage (proportional to the current) at a given time, leakage may be observed in two dimensions. The first one is the *timing leakage*, also known as the *horizontal leakage*, since we usually plot the time on the horizontal axis.

Classic timing leakage, as exploited by a timing attack, happens when the number of clock cycles for a program part varies with the value of a secret. Kocher's timing attack on implementations of RSA in 1996 [12] targeted the modular exponentiation, which executes a modular squaring when being provided a bit of the secret key with value 0, whereas it executes a modular squaring followed by a modular multiplication if that bit is 1. This additional modular multiplication leads to a longer execution time, and therefore it causes some timing leakage correlated to the bit value for the secret key. However, this flaw in cryptographic implementations can be prevented in a few ways. Joye and Yen proposed the *Montgomery powering ladder* [42], which adapts an idea by Montgomery [43], for modular exponentiation in an abelian group for RSA as a timing-leakage free substitution for the square-and-multiply algorithm. Besides, in cryptographic applications, designers and implementers also avoid conditional statements (e.g., if/else) to make the number of instructions constant [44]. Thereafter, the threat of timing attacks focused on micro-architectural leakages, such as the timing variation from cache hits and misses [45].

On the other hand, attackers can also find some *vertical leakage*, such as the *power leakage*, to extract information about the secret. This kind of leakage mainly results from the difference between the power consumption when target devices are processing different values of the secret. For example, when the device flips a byte from all "0" to all "1", it may require more energy compared to flipping only four bits. Attacks exploiting power leakage remain a main threat to many embedded cryptographic implementations with untrusted device users, e.g. smartcards and other hardware tokens. Unlike the approach of avoiding conditional branches to prevent timing leakage, there are no well-recognized means to fundamentally eliminate power leakage at the software design level. Therefore, current research still focuses on the development of vertical-leakage-related attacks and their countermeasures.

Attacks on vertical leakage We commonly categorize the attacks exploiting power (vertical) leakage into *profiling attacks* and *non-profiling attacks*. In general, non-profiling attacks need a relatively large number (i.e., normally more than a few hundred for the best cases) of attack power traces, but usually rely on only a few samples from each one. On the other hand, profiling attacks require only several or even single attack power traces, but need a profiling device and phase.

Among the non-profiling attacks, *Differential Power Analysis* (DPA) attacks are the most common type. DPA attacks reveal the secrets by comparing the samples of a large number of power traces being recorded when different data blocks are fed as the input for a process, e.g., encryption or decryption for a cryptographic application. This requires that attackers know at least a part of the input or output (e.g., the ciphertext of DES [25] decryption), and the secret (e.g., the key of DES decryption) remains unchanged during the recording procedure. The sample analysis may involve some statistical measures, such as the *difference of means* (DoM) [13] or the *Pearson Correlation Coefficient* [46, 38].

In Kocher et al.'s initial DPA attack [13], they assumed that attackers have recorded power traces with the corresponding ciphertexts from several decryptions of DES, and that there will be a sample in these power traces, the value of which is correlated to a bit, b, of an intermediate value, V, in the penultimate round. This intermediate value can be found from a subkey, C, of the last round key and its corresponding part of the ciphertext, C, by calculating the linear **AddRoundKey** (`XOR`) function and the non-linear **SubstitutionBox** (Sbox) function of DES:

$$\left.\begin{array}{r}\text{guessed secret subkey } K\\ \text{known cipher fragment } C\end{array}\right] \oplus \longrightarrow \boxed{\text{Sbox}} \longrightarrow \text{predicted intermediate value } V.$$

Along with the known ciphertexts, once attackers correctly guess the value of the subkey, they can successfully predict the corresponding intermediate value for each trace, and therefore the bit b. In this case, if they separate all the power traces into two groups, according to the predicted b value, they will obtain a relatively large estimate for the DoM between the two groups. Otherwise, the value will not be significant with the wrong grouping according to the other incorrectly guessed subkey candidates.

Correlation Power Analysis (CPA), introduced by Brier, Clavier, and Olivier in 2004 [38], became the most commonly used DPA-style technique. A CPA attack assumes that power consumption is related to the number of "1" bits changing in a target intermediate data unit (e.g. a byte on an 8-bit device or a 32-bit word on a 32-bit device), while the device processes all bits in such a unit in the same clock cycle. Therefore, the power consumption can be modeled as a noisy linear function of the *Hamming Distance* (HD) between two target intermediate units (e.g. bytes) or even the *Hamming Weight* (HW) of a single target unit. Given several different plaintexts and their corresponding power-consumption traces recorded, we can choose as a target an intermediate value calculated from a single key byte and a single byte from these plaintexts before further diffusion, e.g. the state after the **AddRoundKey** and **SubstitutionBox** in the first round of AES [47, 48]. Then, if we have guessed the key byte correctly, a few time samples on the traces will be highly correlated to the HW of a state or the HD between two states, and consequently, there will be peaks in the Pearson Correlation Coefficient [46] at the corresponding time samples. Meanwhile, there will be no such peaks for the 255 other, wrong key-byte candidates, and this way the correct key-byte value is identified.

Profiling attacks The advantage of the previous DPA or CPA attacks is that they work well with very generic leakage models, which distinguish only larger groups of values, such as HW or HD, but work across many devices. Meanwhile, it is also possible to build far more detailed leakage models, based on the observation that each bit in a register or on a bus has an individual leakage signal. Such models require careful *profiling* of the type of hardware being targeted, which adds to the complexity of the attack, but opens the possibility of correctly identifying individual unit values, rather than just groups. Such attacks are known as *profiling attacks*.

In my opinion, the security risks posed by profiling attacks warrant particular attention. For example, profiling attacks may succeed to recover the actual value of secrets after just one single observation of the execution of an algorithm, known as a *single-trace attack*. These may help to circumvent many countermeasures targeted mainly at correlation-style attacks that need hundreds of traces. An early and influential profiling technique, the *Template Attack* (TA), was introduced by Chari, Rao, and Rohatgi in 2002 [39]. The Template Attack is a two-stage procedure. In the *profiling* stage, attackers build a *template* including the expected value and a covariance matrix of the selected samples from pre-recorded power traces where a particular candidate of a target state appears. By repeating the same procedure on each candidate, they can complete the template set for the target state. Then, in the *attack* stage, they compare the selected samples from the attack trace against each template in the set by calculating a likelihood value, using this multivariate Gaussian model with its expected value and covariance matrix. The larger the likelihood value, the more likely the corresponding candidate of the template is the secret value hidden in the attack traces. Chari et al. [39] already mentioned their early attempts to distinguish between key bytes with the same Hamming weight values, given that this still required that the attacker records multiple attack traces for one target encryption.

1.1.3 Template attack to reconstruct the full state

Before I started my Ph.D. project, an advanced template attack on 8-bit processors was introduced by Choudary and Kuhn [49] to distinguish all the 256 candidates of a byte value being processed by a `LOAD` instruction. Their approach is based on Schindler et al.'s *stochastic model* [50], which uses linear regression to build templates for individual bits, which they combined with *Fisher's Linear Discriminant Analysis* (LDA), as introduced by Standaert and Archambeau [51] for dimensionality reduction of traces. Their templates provided a likelihood value for each candidate, and then they calculated the likelihood ranking of the correct candidate. They nearly reach a 0-bit guessing entropy, i.e. the binary logarithm of the mean rank of the correct candidate, from about 100 attack traces with the same secret value. In this thesis, I will refer to this type of template attack, which can provide a likelihood prediction for each possible actual value of the target, instead of just its Hamming weight, as a *full-state* recovery.

In their attack, they focus on a handful of clock cycles, mainly covering the `LOAD` instruction. I expected that a similar attack, targeting an intermediate value processed in a cryptographic algorithm, may achieve an even better result, extending it down to single attack traces, considering that there will be more than one instruction handling such an intermediate value. Besides, Choudary's Ph.D. thesis [52] left one issue open that I was also curious about: how to apply the attack to situations where more than 8 bits of data are processed simultaneously, e.g. in devices with 32-bit buses. Since 32-bit cores, such as ARM's Cortex-M family, now dominate the microcontroller market, exploring this type of full-state template attack remains of particular interest, especially on known cryptographic algorithms.

1.2 Post-processing side-channel information

When we apply a full-state template attack to recover a single byte, the procedure ends once attackers obtain a predicted likelihood value for each candidate. In contrast, when we attempt to attack a cryptographic algorithm, a single TA (or DPA) procedure usually reveals only a small piece of the target secret, such as a key byte. Therefore, after attackers have repeated the procedure to collect all the pieces of the secret, they will need some post-processing steps to predict a full secret. Most simply, the authors in some early studies implied that they just concatenated the key byte candidates, each chosen according to the highest likelihood value in TA [39], or with the most significant Pearson correlation coefficient value in CPA [38], into a full key. This means that there will be no room for mistakes to occur in the prediction of each piece of the secret.

However, especially for template attacks (e.g., previously mentioned Choudary's LDA-based attack), it is not easy to build a model that can always provide the correct candidate with the highest likelihood value. Therefore, methods such as *Key Enumeration* [53], *Algebraic Side-Channel Analysis* (ASCA) [54], *Tolerant Algebraic Side-Channel Analysis* (TASCA) [55], and *Soft Analytical Side-Channel Analysis* (SASCA) [56] have been developed and can apply to the ambiguous and even misleading information provided by non-perfect templates so that attack procedures can be more compatible with situations in the real world. These mathematical tools usually require that attackers can access other values involved in the target cryptographic algorithm.

I was particularly interested in two of the above methods. The first one is using key enumeration, which requires the knowledge of at least one pair of plaintext and ciphertext. Veyrat-Charvillon et al. introduced their algorithm to efficiently search the correct key-byte combination in 2012 [53]. They started by building a search scheme to optimize the enumeration of the combinations of two key bytes with their respective side-channel-predicted probability for each candidate, and then generalized it into a recursive way so that it can be used for searching the correct combination of the 16-byte key in AES-128. For another type of approach, Veyrat-Charvillon et al. later presented their SASCA [56] in 2014. This methodology requires side-channel information, normally probability tables, from a few more intermediate values in addition to those originally targeted, and then mutually updates the probability tables following the algorithm-specific mathematical relations (usually represented by a *factor graph*) between the target and the additional intermediate values, so that attackers can make more reliable predictions of the target intermediate values.

1.3 Target algorithms

In 2015, NIST published the Secure Hash Algorithm 3 (SHA-3) in NIST FIPS 202 [15]. SHA-3 is based on the Keccak permutation designed by Bertoni et al. [57]. With this permutation, they described the concept of *permutation-based cryptography* [58], where various cryptographic applications can be constructed by different modes (e.g., sponge mode or duplex mode [58]) that consists of multiple invocations of the same permutation. Therefore, the Keccak permutation is not only the main building block of the standardized SHA-3 family of hash functions (e.g., SHA3-512) and extendable-output functions (e.g., SHAKE256), but can also be used in many other contexts, such as pseudorandom function (e.g., Farfalle [59]), authenticated encryption (e.g., Keyak [60]), and key-agreement schemes (e.g., SHAKE functions used in NewHope [61] and CRYSTALS–Kyber [62]), where either its inputs or outputs can be confidential data for which side-channel attacks may be a concern. As more and more applications rely on SHA-3 or the Keccak permutation, it is important to understand their resilience against template attacks and the need for countermeasures.

One characteristic of hash functions, including SHA-3, is that their output data will not contain all the information about the secret inputs, so secret inputs of hash functions cannot be inverted only with the output data. However, it is possible to obtain lost information if attackers perform template attacks to recover some intermediate values. For example, the secret key in some implementations of HMAC-SHA-1 can be recovered by a template attack [63].

Before I started my Ph.D. program, 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 one step (θ) to recover a fixed-length key and discussed the relationship between key length and the DPA resilience of MAC-Keccak [64]. They later applied similar attacks to recovering MAC-Keccak keys with arbitrary length [65]. Luo et al. modified this attack to determine the intermediate state after a complete round of Keccak permutation [66], applying DPA after the non-linear step (χ). These attacks have not yet applied a full-state template attack on Keccak. Given that in some proposed applications [62, 67], the Keccak functions will not be executed multiple times on common inputs, a single-trace template attack is more likely to pose a threat to these applications than multi-trace DPA attacks.

Attack concepts that successfully target Keccak may also threaten permutation-based cryptographic algorithms built on other permutations, such as Ascon [68], with appropriate modifications. Following NIST's call for LWC algorithms [10] in 2018, they chose Ascon as one of the 10 candidates in 2021 for the last round competition [69] and announced it as the final winner in 2023 [70]. Therefore, I selected Ascon as my next target after Keccak. Both algorithms have some mathematical similarities, particularly in their non-linear operations.

Apart from other cryptanalysis [71], early published side-channel attacks on Ascon, still, mostly focus on DPA-style attacks [72, 73]. This again shows the need to analyze the impact of template attacks on KECCAK, ASCON, and other permutation-based cryptographic algorithms.

1.4 Countermeasures against power analysis

Since the issues of SCA have been highlighted for more than two decades, modern implementations of cryptographic applications are mostly protected by some countermeasures, such as the previously mentioned Montgomery ladder against timing attacks in Sec. 1.1.2. Mangard et al. categorized various types of countermeasures against power analysis into *hiding* and *masking* [20].

For hiding countermeasures, the goal is to decrease the signal-to-noise ratio (SNR) so that the side-channel information can be hidden behind the noise [74]. These countermeasures also include means such as inserting random dummy operations or shuffling some independent operations within the target cryptographic algorithms to fool side-channel attackers [20, Ch. 7].

On the other hand, masking countermeasures date back to 1999, when Chari et al. [75] introduced their technique to split secrets into $N + 1$ shares by providing N independent random values (N *masks* in this thesis), which is later commonly referred to as N^{th}-order masking. This makes it more difficult to reconstruct secrets since now attackers will need to correctly predict every share. Following their work, Prouff and Rivain provided a security analysis of masking [76]. Since this technique originally aims to mitigate CPA/DPA-style attacks, it freshly masks the key at the start of each encryption.

Masking then gradually became a widely implemented countermeasure, and most symmetric cryptographic algorithms, such as Rijndael [20, Sec. 9.2] and other AES candidates [77], can implement *Boolean* masking [75, Sec. 3.3] to split the secret keys, where the mathematical relation between the key K and the $N + 1$ shares S_0 to S_N is XOR ($K = \bigoplus_{n=0}^{N} S_n$).

However, it is expensive to implement Boolean masking on non-linear steps (e.g., **SubstitutionBox**) in symmetric cryptographic algorithms, and sometimes it requires hybrid use of masking (i.e., Boolean masking for linear steps and other types for non-linear steps) for efficiency [78, 79]. To address this problem, Bertoni et al. designed the non-linear step of KECCAK with only binary operations `NOT`, `XOR`, and `AND`, such that Boolean masking can easily be applied [80]. This core of the non-linear function was later also used in ASCON. Therefore, I expect that Boolean masking will be widely used in the future implementations of these permutation-based algorithms, and to what extent of protection this countermeasure can provide remains an important issue.

1.4.1 Attack Boolean-masked implementations

Faced with this protection, despite the difficulty as the order increases, attackers can still apply an $(N+1)^{\text{th}}$-order DPA to attack an implementation with N^{th}-order masking [13, 81, 82], which is to consider $N + 1$ samples in a power trace, each representing a share for the N^{th}-order-masked key, in the DPA statistic model at the same time.

On the other hand, among previous proposals for template attacks on masked cryptographic implementations (e.g., [83, 84, 85, 86]), there is still not yet a widely recognized strategy for countermeasures, unlike what is the case for DPA. However, some previous studies suggested that we can still apply belief propagation to a masked cryptographic implementation by also considering the mathematical relation between the original intermediate values and their Boolean masking shares (i.e., the XOR operation for this case.) [87, 88]. This requires attackers to build templates for each share of their target secret, and therefore additional access to the random generator may be necessary as well in the profiling stage, complicating this attack even more.

1.5 Contributions

I introduce a methodology, *fragment template attack* (FTA), to extract information about individual bits from power traces that observe activity on 32-bit parallel data buses. To achieve this, I 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 build templates for these fragments, to enable us to reconstruct their values independently and within a reasonable run time. Within the various types of side-channel attacks introduced previously, my FTA technique is a passive, non-invasive, power-trace profiling attack. In this thesis, my survey of this technique was still in its early stage, so my attack was implemented in a more laboratory-controlled environment, which involves phase-locking clock sources of the oscilloscope and the target devices to avoid the unalignment problem, using target boards designed for side-channel research, and using the same board for both profiling and attack stages.

With the assistance of two algorithmic SCA tools, the optimal key enumeration and SASCA, this FTA technique can seriously threaten permutation-based cryptographic algorithms such as Keccak and Ascon.

1.5.1 Thesis structure

In this thesis, Chapter 2 discusses more details of available SCA tools, including the LDA-based template attack, the optimal key enumeration, and SASCA. This chapter also describes the mathematical structure of Keccak and Ascon, and introduces the experiment platforms. Chapter 3 presents how I used the LDA-based template attack to target a SHA3-512 implementation on an 8-bit device (Section 3.2) and how I designed a three-layer enumeration to search the arbitrary-length input of SHA3-512, and its performance in experiments (Section 3.3). Then, given that Kannwischer et al. [89] published a SASCA procedure originally designed for simulated HW information approximately at the same time as my enumeration, I also introduce how I modified their methodology to make it compatible with the information observed from my templates in Section 3.4.

Chapter 4 introduces perhaps the most important part of my research, the fragment template attack. I demonstrate the feasibility of this attack through three different experiments. The first one reused the previous SHA3-512 datasets recorded from the 8-bit device, but built templates for two nibble fragments of a target byte instead of a template for the byte directly. The second experiment is to apply the fragment template attack to recover secrets of a toy stream cipher implemented on a 32-bit device, and then the last experiment is to attack an implementation including all six functions in the KECCAK family on the same hardware device. Chapter 5 applies the fragment template technique to attack both unmasked and masked ASCON implementations also on the 32-bit device, to show that the threats of this attack can be a more general issue beyond just KECCAK. Finally, I discuss some side issues and future work before concluding in Chapter 6.

I published earlier versions of much of the methodology and experimental results from Chapter 3, 4 and 5 in two peer-reviewed papers and one poster, and I also published the final version of my attack of ASCON in one peer-reviewed paper:

[90] *A template attack to reconstruct the input of SHA-3 on an 8-bit device*,
COSADE 2020, LNCS vol. 12244.

[91] *Single-trace fragment template attack on a 32-bit implementation of KECCAK*,
CARDIS 2021, LNCS vol. 13173.

[92] *A template attack on ASCON AEAD*,
CHES 2022, poster. `https://ches.iacr.org/2022/acceptedposters.php`.

[93] *Low trace-count template attacks on 32-bit implementations of ASCON AEAD*,
CHES 2023, pre-print version, to appear in TCHES 2023/4.

Chapter 2

Preliminaries

2.1 Template attack on current traces

Chari et al. introduced a powerful side-channel exploitation technique called Template Attack (TA) [39]. It consists of two stages, profiling and attack. During profiling, attackers build templates that model the leakage traces of different candidate secrets from traces recorded while a known secret is being processed. Then, they record an attack trace while an unknown secret is being processed, and then compare that with all the templates and predict the secret according to the candidate with the template most similar to the attack trace.

In this thesis, I am in particular interested in information leaked from the power consumption of a device, observed via direct coupling, to minimize measurement noise. According to Mangard et al.'s description [20, Sec. 3.4.2], we can insert a small resistor into the GND or V_{DD} line of the working device, where the voltage drop across this resistor is proportional to the current flowing into or out of the device. With an oscilloscope recording that voltage drop, we refer to the resulting one-dimensional array of time samples as a *current trace*, or also simply a *trace* in this thesis.

When the oscilloscope and the target device are synchronized, the same sample index on each trace will represent the current measured during the same phase of the same instruction during the execution of a constant-time cryptographic algorithm. Implementations of cryptographic algorithms generally avoid using conditional branches, to prevent timing attacks. This makes it easier for attackers to obtain synchronized templates, unless there are countermeasures introduced, such as random delays, to hinder synchronized recordings. In the latter case, additional steps would have to be taken to align traces.

2.1.1 The basic template attack

In this approach, attackers need to collect a sizeable number of traces in the profiling stage. 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.

When we obtain an attack trace $\mathbf{x}_\text{a}$, we can then calculate as a likelihood function a multivariate Gaussian probability-density value for each template with

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

We can normalize these likelihoods to build a *probability table* by

$$p(b=\xi) = \frac{f(\mathbf{x}_\text{a}|\bar{\mathbf{x}}_\xi,\mathbf{S}_\xi)}{\sum_{b'=0}^{255}f(\mathbf{x}_\text{a}|\bar{\mathbf{x}}_{b'},\mathbf{S}_{b'})}.$$

2.1.2 The template attack with linear regression 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. [50], Choudary and Kuhn used an alternative solution [49] that is more efficient regarding the number of traces recorded. They treat each bit, $b[0]$ to $b[7]$, in the targeted intermediate byte as an independent variable and then use multiple 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 $\hat{x}_b = \sum_{\ell=0}^{7}(b[\ell]\cdot c_l) + c_8$ and equivalently as

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

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 multiple linear regression.

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

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

Instead of a different $\mathbf{S}_b$ in each template, they used one single *pooled* covariance matrix estimate, $\mathbf{S}_\text{pooled}$, which is the weighted average of the $\mathbf{S}_b$, because previous studies [94, 95] 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

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

With $\mathbf{S}_\mathrm{pooled}$ staying constant for all 256 candidates, we can normalize these likelihoods to build a probability table by merely

$$p(b = \xi) = \frac{\hat{f}(\mathbf{x}_\mathrm{a}|\hat{\mathbf{x}}_\xi)}{\sum_{b'=0}^{255} \hat{f}(\mathbf{x}_\mathrm{a}|\hat{\mathbf{x}}_{b'})}, \quad \hat{f}(\mathbf{x}_\mathrm{a}|\hat{\mathbf{x}}_b) = \exp\left(-\frac{1}{2}(\mathbf{x}_\mathrm{a} - \hat{\mathbf{x}}_b)^\mathsf{T} {\mathbf{S}_\mathrm{pooled}}^{-1} (\mathbf{x}_\mathrm{a} - \hat{\mathbf{x}}_b)\right).$$

Alternatively, we can also represent this distribution as a *logarithmic likelihood table* by

$$p_\mathrm{log}(b = \xi) = -\frac{1}{2}(\mathbf{x}_\mathrm{a} - \hat{\mathbf{x}}_\xi)^\mathsf{T} {\mathbf{S}_\mathrm{pooled}}^{-1} (\mathbf{x}_\mathrm{a} - \hat{\mathbf{x}}_\xi).$$

Then we can sort the 256 results from either a probability table or a logarithmic likelihood table into a *ranking table*, where the top entry is the most likely candidate.

2.1.3 Data compression with linear discriminant analysis

Choudary and Kuhn also integrated Fisher's Linear Discriminant Analysis (LDA), as proposed by Standaert and Archambeau [51], into their approach [49, 52]. This is a procedure to project the traces onto a subspace with a higher signal-to-noise ratio (SNR), as determined by two covariance matrices $\mathbf{B}$ and $\mathbf{W}$, where $\mathbf{B}$ is the inter-class scatter, representing the signal, while $\mathbf{W}$ 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

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

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 $\mathbf{W}^{-1}\mathbf{B}$ corresponding to its m' largest eigenvalues (see also [96, footnote 6]). The LDA projection of a raw trace $\mathbf{x}_\mathrm{a}$ onto the resulting m'-dimensional subspace is then $\mathbf{x}_\mathrm{proj} = \mathbf{A}^\mathsf{T}\mathbf{x}_\mathrm{a}$.

Following Choudary and Kuhn's approach as outlined above, the complete procedure of template profiling will be: firstly using multiple linear regression to build matrices $\mathbf{W}$ 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 multiple linear regression. The resulting template information consists of a new pooled covariance matrix $\mathbf{S}_\mathrm{proj} \in \mathbb{R}^{m' \times m'}$, 256 new expected traces $\hat{\mathbf{x}}_{b,\mathrm{proj}} \in \mathbb{R}^{m'}$, along with $\mathbf{A}$.

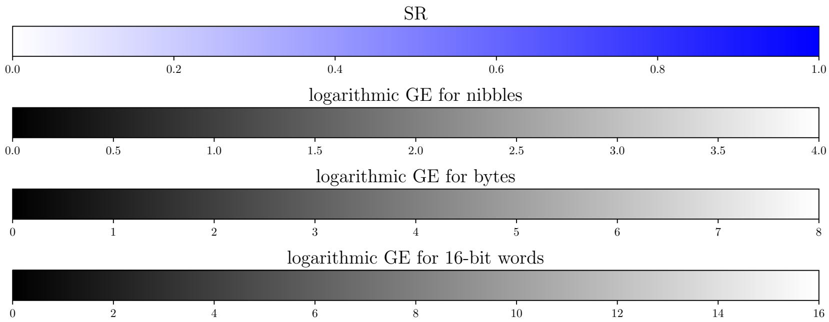


Figure 2.1: Examples for SR and logarithmic GE values in color matrices of this thesis.

Dimensionality of projected traces However, how to determine the dimensionality, m', of the projected trace remains an open question. Choudary integrated at least two different ways [52, Sec. 3.9.1], originally used in *Principle Component Analysis* [97, 98], into his LDA-based model [52, Sec. 3.10]. Given the size, m, of the original trace is smaller than the total number of profiling traces, we will observe at most m non-zero, normalized eigenvectors $\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_m$, sorted in descending order of their corresponding absolute eigenvalues $|\lambda_1| \geq |\lambda_2| \geq \ldots \geq |\lambda_m|$. One of the methods selects the smallest value m', such that the *cumulative percentage of total variation*,

$$\phi(m') = \frac{\sum_{g'=1}^{m'} |\lambda_{g'}|}{\sum_{g=1}^{m} |\lambda_g|},$$

will be larger than a chosen threshold (e.g., 0.9). In my experiments, I started by using a slightly modified criterion, where I select each eigenvector $\mathbf{a}_{g'}$ with an eigenvalue larger than one-thousandth of the total variation ($|\lambda_{g'}| > 0.001 \times \sum_{g=1}^{m} |\lambda_g|$) into the projected matrix $\mathbf{A}$ until I observed the relation between the number of non-zero eigenvalues and the size of target variable (see Section 4.4.5). Note that when I applied the latter criterion to my experiments for profiling templates for bytes with the $\mathcal{F}_9$ model, this will select $m' = 8$ eigenvectors.

2.1.4 Template quality evaluation

It had been an open question of how to evaluate a Gaussian template model in a template attack so that we know whether it can provide us with reliable predictions. Therefore, Standaert et al. [99] defined their *success rate* and *guessing entropy*, which have already been widely used in related studies [86, 100].

Success rate (SR) Given a ranking table for all the possible candidates of a variable predicted by a side-channel distinguisher function, such as the likelihood function from the templates introduced above, they defined the n^{th}-order success rate as the probability that the correct candidate is located within the first n candidates in the ranking table. In this thesis, I only use the first-order success rate ($n = 1$) for template quality evaluation, where only the case of the correct candidate topping the ranking table is considered to be successful. Apart from tables listing the success rates (e.g. Table B.4), I also plot the success rate as color matrices (e.g. Figure 3.3) to provide an overview of how template quality differs across a larger set of targeted intermediate values. In these matrices, I map the success rate value from white to blue, where the darker the color, the higher the value.

Guessing entropy (GE) Given the same ranking table, they also defined the guessing entropy as the expected value of the ranking of the correct candidate. In this thesis, I use the arithmetic mean value of the rankings from several trials to estimate this value. When plotted in a color matrix, I map the logarithmic guessing entropy values from white to black, where the darker the color, the lower the value and the more information the templates provide.

Figure 2.1 shows how SR and logarithmic GE values are represented in color matrices in this thesis. Unless stated otherwise, both these values are estimated here by ranking tables from 1000 different trials. Note that these two values have their respective natural benchmarks when compared to a model providing no information. Given a ranking table with 2^n candidates, the success rate will converge to $\frac{1}{2^n}$ for random guessing, while the guessing entropy will converge to $\frac{(1+2^n)}{2} \approx 2^{n-1}$. Once templates provide some information for our target, the success rate will be higher than $\frac{1}{2^n}$, while the guessing entropy will be lower than 2^{n-1}.

2.2 Key enumeration

With ideal templates and in the absence of noise, attackers should find the full state of a secret by simply taking the most likely candidate from each part of the secret (e.g., a byte) and concatenating them. However, template attacks are noise sensitive, so the correct candidate will not always top the ranking table. Therefore, Veyrat-Charvillon et al. introduced an optimal key enumeration algorithm to search the correct key across the *independent* ranked likelihood tables of the 16 key bytes of AES [53]. Given two ranking 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 ranking tables. With this algorithm, it becomes practical to search for the correct combination of the key bytes when the correct candidates do not top the tables. This increases the noise resiliency of the attack significantly.

In this thesis, I also refer to this procedure as *secret enumeration* when it is used to enumerate some intermediate values other than keys.

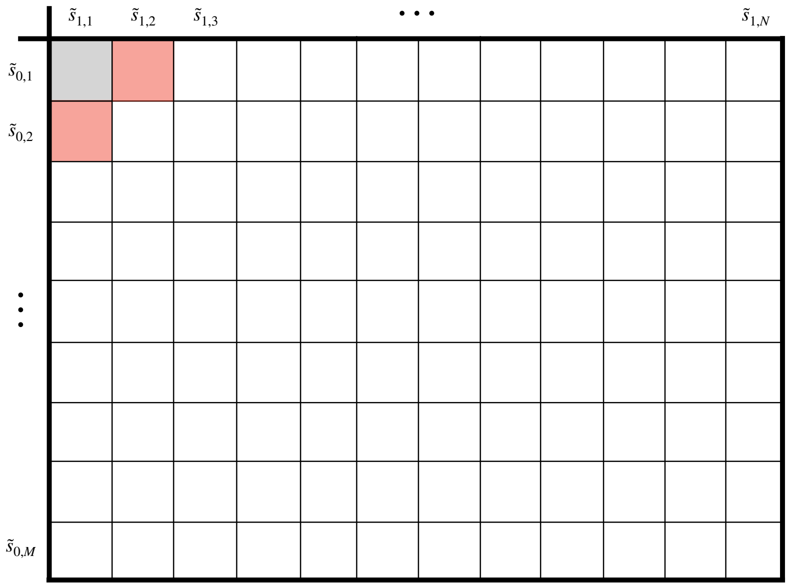


Figure 2.2: The combination with the largest joint probability must be the top-left element when the search begins.

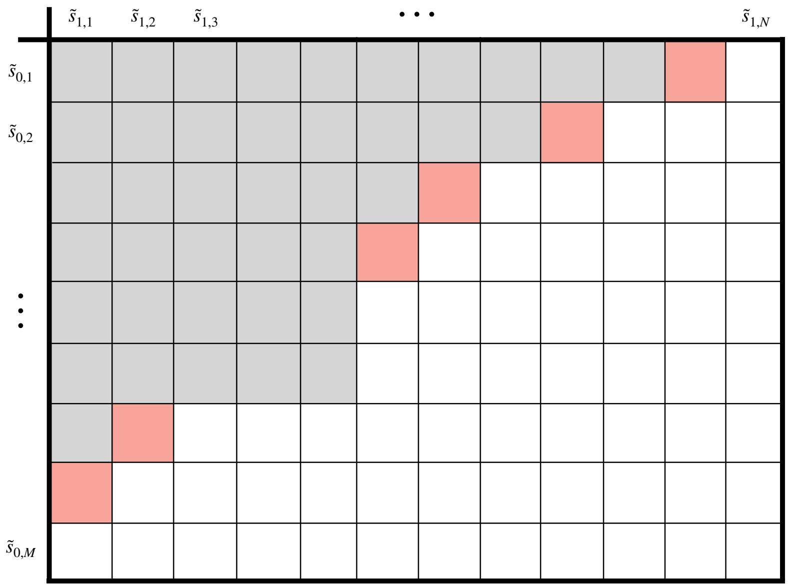


Figure 2.3: The frontier $\mathcal{F}$, blocks labeled in red, when the gray blocks have been enumerated.

2.2.1 Search within two ranking tables

Assume that there are two independent secret variables s_0 and s_1 with M and N possible values respectively. Through some side-channel attacks, we have already obtained their probability tables and sorted them into their ranking tables. Here the m^{th} and n^{th} most likely candidates of these two variables are denoted as $\tilde{s}_{0,m}$ and $\tilde{s}_{1,n}$, respectively, and their corresponding probabilities are denoted as $p_{0,m}$ and $p_{1,n}$. Figure 2.2 shows an $M \times N$ array, where each block represents the joint probability, $p_{0,m} \times p_{1,n}$, of a combination $(\tilde{s}_{0,m}, \tilde{s}_{1,n})$.

An efficient comparing rule: In this array, as the tables have been sorted, there is an important rule that a probability represented by a block is always greater than (or equal to) another represented by not only any blocks to its right in the same row, but also any blocks to its bottom in the same column. For their transitive relation, given a block represents $(\tilde{s}_{0,m}, \tilde{s}_{1,n})$, we have

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

This means that the joint probability of a block is greater than or equal to the value of any other blocks to its bottom right. This also implies that a block will never be considered to be a candidate of the next enumerated combination before all the blocks to its top left have been enumerated.

With this rule, the most top-left block represents the combination with the largest joint probability, which is $(\tilde{s}_{0,1}, \tilde{s}_{1,1})$. Later, the combination with the second largest joint probability can only be either $(\tilde{s}_{0,2}, \tilde{s}_{1,1})$ or $(\tilde{s}_{0,1}, \tilde{s}_{1,2})$, given the comparing rule will eliminate all the other combinations, but it cannot apply to compare the joint probabilities of these two. They will both be added into a set called *frontier*, $\mathcal{F}$, which includes all candidate combinations that cannot eliminate one another simply via the comparing rule. Therefore, we only need to compare values from this set, instead of from all the remaining combinations, to find the next enumerated value pair with the next largest joint probability.

For a more general case, see Figure 2.3: once all the combinations marked in gray have been enumerated, the frontier $\mathcal{F}$, marked in red, will be a set of all the combinations at the concave corners. For these combinations, their joint probability must be larger than those of any other unenumerated combinations to their bottom-right, according to the comparing rule, but they cannot mutually eliminate each other because they are either to the top-right or to the bottom-left of one another. Therefore, we have to search in the frontier by comparing their probability values, but that is still better than searching through all the unenumerated combinations.

While a combination $(\tilde{s}_{0,m}, \tilde{s}_{1,n})$ is being enumerated, we need to update $\mathcal{F}$ by removing $(\tilde{s}_{0,m}, \tilde{s}_{1,n})$ (marking it gray here), and then considering whether $(\tilde{s}_{0,m+1}, \tilde{s}_{1,n})$ or $(\tilde{s}_{0,m}, \tilde{s}_{1,n+1})$ or both shall be added to $\mathcal{F}$, respectively. Only if one occupies a concave corner in the already enumerated gray part of the array, will it become a new member of $\mathcal{F}$ at this time.

2.2.2 Search with a recursive structure

Following this algorithm, we can see that the probability $p_{1,n}$ will only be referenced after the combination $(\tilde{s}_{0,1}, \tilde{s}_{1,n-1})$ has been enumerated and then we need to add $(\tilde{s}_{0,1}, \tilde{s}_{1,n})$ into $\mathcal{F}$, while this similarly applies to $p_{0,m}$. This means that we do not need all the values in the probability tables in the beginning, and therefore we can separate the search procedure into three nodes, two of which I call *table* nodes and one *combining* node. As a child node with a ranking table, a table node will provide the next candidate (e.g. $\tilde{s}_{1,n}$) and its corresponding probability (e.g. $p_{1,n}$) from its table to its parent combining node once the latter requests the

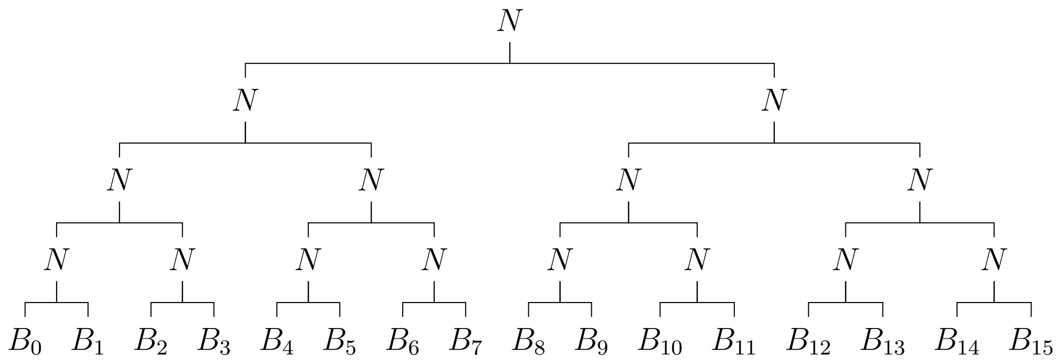


Figure 2.4: The recursive enumeration in AES key combination, where N represents a combining node, and B_n represents a table node with the ranking table of an AES subkey.

information for updating $\mathcal{F}$. From this design, we can build a tree of iterators recursively, given a child node of a combining node can be another combining node, to search combinations of candidates from beyond two tables.

Figure 2.4 shows the example of how Veyrat-Charvillon et al. [53] apply their algorithm to combine key bytes into a round or master key used in AES. At the top level, the master combining node will return the next combination of 16 key bytes with the next largest joint probability among those not yet being enumerated once we ask for it. At the second level, the two combining nodes will return the next combination of eight key bytes with the largest joint probability among those not yet being enumerated once being called by the master node. This similarly applies to other middle levels. As for the bottom level, there are 16 table nodes each with one ranking table for a key byte from template attacks.

Note that this is a well-balanced tree structure because the number of key bytes, 16, is a power of 2, but this algorithm can also be used for combining any number, besides powers of 2, of target variables with an unbalanced structure. When I use this enumeration algorithm, I prefer logarithmic likelihood tables (described in Section 2.1) so that we can use addition instead of multiplication to calculate logarithmic joint probability values more efficiently.

2.3 Belief propagation and SASCA

Veyrat-Charvillon et al. [56] introduced Soft Analytical Side-Channel Analysis (SASCA), an inference technique for template attacks on cryptographic algorithms based on the belief-propagation algorithm [101, 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 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, I prefer to classify 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 bi-directional, 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

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

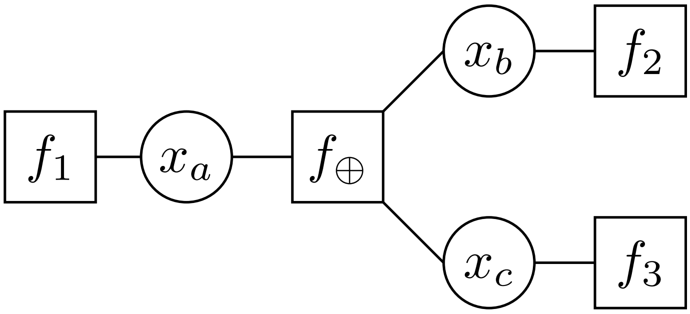


Figure 2.5: A factor graph covering three variable x_a, x_b, x_c. The square nodes represent variables, while the circle nodes represent factors. In this figure, f_1, f_2, f_3 are observation factors, and $f_\oplus$ is a constraint factor.

I provide a small example in the following scenario: an attacker uses a template attack to recover three binary variables x_a, x_b, x_c, where the mathematical relation between these variables is $x_a = x_b \oplus x_c$. Observed from the templates, the probabilities of their two candidates $\{0, 1\}$ are $\{0.8, 0.2\}, \{0.7, 0.3\}, \{0.9, 0.1\}$ respectively. Figure 2.5 depicts the factor graph covering these three variables. With the information above, the tables in observation factor nodes f_1, f_2, f_3 are:

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

and the constraint function in factor node $f_\oplus$ is:

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

When calculating the probability $P_a(x_a = 0)$, we first find the value of $Z_a(x_a = 0)$ by:

$$\begin{aligned} Z_a(x_a = 0) &= r_{1 \to a}(x_a = 0) \times r_{\oplus \to a}(x_a = 0) \\ &= f_1(x_a = 0) \times \sum_{x_a = 0, x_b, x_c} [f_\oplus(x_a, x_b, x_c) \times q_{b \to \oplus}(x_b) \times q_{c \to \oplus}(x_c)] \\ &= 0.8 \times [q_{b \to \oplus}(0) \times q_{c \to \oplus}(0) + q_{b \to \oplus}(1) \times q_{c \to \oplus}(1)] \,, \end{aligned}$$

where we can keep following the rules to update q tables:

$$\begin{aligned} q_{b \to \oplus}(x_b = \xi) = r_{2 \to b}(x_b = \xi) = f_2(x_b = \xi), \\ q_{c \to \oplus}(x_c = \xi) = r_{3 \to c}(x_c = \xi) = f_3(x_c = \xi). \end{aligned}$$

Therefore, $Z_a(x_a = 0)$ will be:

$$\begin{aligned} Z_a(x_a = 0) &= 0.8 \times [f_2(x_b = 0) \times f_3(x_c = 0) + f_2(x_b = 1) \times f_3(x_c = 1)] \\ &= 0.8 \times [0.7 \times 0.9 + 0.3 \times 0.1] = 0.528. \end{aligned}$$

Likewise, $Z_a(x_a = 1)$ will be:

$$\begin{aligned} Z_a(x_a = 1) &= f_1(x_a = 1) \times [f_2(x_b = 0) \times f_3(x_c = 1) + f_2(x_b = 1) \times f_3(x_c = 0)] \\ &= 0.2 \times [0.7 \times 0.1 + 0.3 \times 0.9] = 0.068. \end{aligned}$$

Finally, we can normalize the probability table $P_a(0) = 0.528 \div (0.528 + 0.068) = 0.8859$, and $P_a(1) = 0.068 \div (0.528 + 0.068) = 0.1141$.

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 cases 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 a solution [101, 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 to prevent the probability values from becoming too small. Then the procedure terminates when it reaches a steady state. We call it an *iteration* that updates r and then q once for each.

2.4 Keccak

2.4.1 Keccak-$f[1600]$ permutation

Bertoni et al. [57] define a family of Keccak-$f[n]$ permutations, where n denotes the number of bits they operate on. The six standardized SHA-3 and SHAKE functions are 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 thesis, I 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-$f[1600]$.

The SHA-3 standard describes these states as a $5 \times 5 \times 64$-bit cube with an x, y, and z axis with little-endian bit order along the z axis. I use coordinates $i \in \mathbb{Z}_5$, $j \in \mathbb{Z}_5$, and $k \in \mathbb{Z}_{64}$ to denote each bit inside such states, e.g. an intermediate bit in state α_0 will be referred to as $\alpha_0[i, j, k]$. I closely follow the notation in paper [57], where the bits with the same coordinates j and k are in the same *row*, the same i and k in the same *column*, and the same i and j in the same *lane*, while bits with the same coordinate i are on the same *sheet*, the same j on the same *plane*, and the same k on the same *slice*.

Considering the frequent use of an 8-bit unit in my experiments, I also refer to the 64 bits along the z axis as eight *intermediate bytes*. For example, we describe an intermediate byte in state α_0 as $\alpha_0[i, j, {}^{\mathbf{8}}k]^{\mathbf{8}}$, where $i \in \mathbb{Z}_5, j \in \mathbb{Z}_5, {}^{\mathbf{8}}k \in \mathbb{Z}_8$ are the coordinates in this case. Note that because of the little-endian bit order, the least significant, or the left-most, bit of an intermediate *byte* $\alpha_0[i, j, {}^{\mathbf{8}}k]^{\mathbf{8}}[0]$ is the intermediate *bit* $\alpha_0[i, j, k] = \alpha_0[i, j, 8 \times {}^{\mathbf{8}}k]$, while the most significant bit $\alpha_0[i, j, {}^{\mathbf{8}}k]^{\mathbf{8}}[7]$ can also be referred to as $\alpha_0[i, j, k] = \alpha_0[i, j, 8 \times {}^{\mathbf{8}}k + 7]$. In this situation, I 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* in this thesis. Similarly, coordinates ${}^{\mathbf{4}}k$, ${}^{\mathbf{16}}k$, and ${}^{\mathbf{32}}k$ represent the cases of nibble, 16-bit words, and 32-bit words respectively.

For translation between an input (or output) bitstring S and a state $\mathbf{state}$, the left-most bit in S will be the bit $\mathbf{state}[0, 0, 0]$, then being filled along the z axis, then along the x axis, and finally along the y axis.

The five steps are introduced as follows. Note that a lane in these states will be denoted as $L_{(i,j)}$ or $L'_{(i,j)}$, e.g. $L_{(i,j)}[k] = \mathbf{state}[i, j, k]$ or $L_{(i,j)}[k] = \alpha_0[i, j, k]$, considering its frequent use as the operational unit in these steps. Here '$\neg$' denotes the operation to flip all the bits in the following bitarray, while '$\oplus$', '$\vee$', and '$\wedge$' denote bitwise `XOR`, `OR`, and `AND` operations on two bitarrays with the same shape respectively. Meanwhile, function $\mathbf{Rot}(X, n)$ rotates the one-dimensional bitarray X to the right by n bits, regardless of its endianness.

Step θ As described in Algorithm 1, step θ first assigns a new internal plane $\mathbf{C}$ by calculating the column parity of the input state, and then it applies a linear transform to calculate another new plane $\mathbf{D}$. Each plane in the input state will be XORed with $\mathbf{D}$ to calculate its corresponding plane in the output state. This is the most complicated linear step inside KECCAK-$f[1600]$ permutation, as more bits are used here than in any other step to calculate a one-bit result.

Step ρ and Step π These two steps are both transpositions on bits. Step ρ is a rotation procedure within a lane, while step π is a transposition among whole lanes. Algorithm 2 shows the procedure of these two steps. In some implementations [102], these two transposition steps do not strictly follow their original order, but can be mixed for optimization.

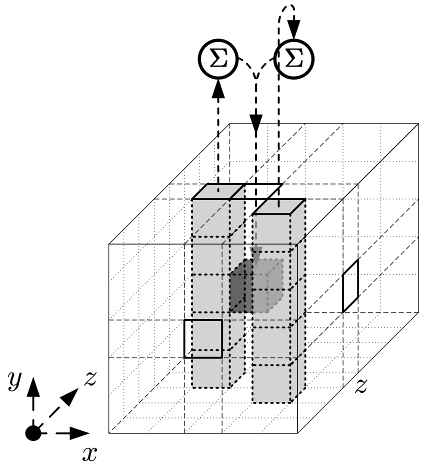
Step χ This is the only non-linear step in the KECCAK-$f[1600]$ permutation. It applies `NOT`, `AND`, and `XOR` instructions on five bits in the same row to calculate the output state. Table B.2 in Appendix B.1 provides the input and corresponding output values when this step is seen as a 5-bit-in and 5-bit-out substitution box. Because the instructions in these steps are all bitwise, we can also execute it in parallel among five lanes on the same plane as described in Algorithm 3.

Step ι This is to XOR the first lane of the state (i.e. $L_{(0,0)}$) with a round constant, which can be calculated given the round number (Ω) [15], but a pre-computed round constant table is used in most implementations of SHA-3, including one of the official C reference codes, XKCP [102]. The round constant tables used in KECCAK-$f[1600]$ are listed in Table B.1 in Appendix B.1.

All five steps in a KECCAK-$f[1600]$ round are practical to invert [103] and the KECCAK team provides C++ implementations of the corresponding inverse functions [104]. In other words, the input, output, and all intermediate states of a KECCAK-$f[1600]$ execution can be converted into each other efficiently.

Algorithm 1 Step θ for round Ω

procedure $\theta(\beta'_{\Omega-1})$
  internal plane $\mathbf{C}, \mathbf{D}$
  for $i \leftarrow 0$ to $4, k \leftarrow 0$ to 63 **do**
    $\mathbf{C}[i,k] \leftarrow \bigoplus_{j=0}^{4} \beta'_{\Omega-1}[i,j,k]$
  end for
  for $i \leftarrow 0$ to $4, k \leftarrow 0$ to 63 **do**
    $\mathbf{D}[i,k] \leftarrow \mathbf{C}[i-1,k] \oplus \mathbf{C}[i+1,k-1]$
  end for
  for $i \leftarrow 0$ to $4, j \leftarrow 0$ to $4, k \leftarrow 0$ to 63 **do**
    $\alpha_\Omega[i,j,k] \leftarrow \beta'_{\Omega-1}[i,j,k] \oplus \mathbf{D}[i,k]$
  end for
  return $\alpha_\Omega[i,j,k]$
end procedure



Algorithm 2 Step ρ and π for round Ω

procedure $\pi \circ \rho(\alpha_\Omega)$
  $(L_{(0,0)}, L_{(1,0)}, L_{(2,0)}, \ldots, L_{(3,4)}, L_{(4,4)}) \coloneqq \alpha_\Omega$ ▷ step ρ starts here
  $(i,j) \leftarrow (1,0)$
  for $t \leftarrow 0$ to 23 **do**
    $r \leftarrow (t+1)(t+2)/2$
    $L_{(i,j)} \leftarrow \mathbf{Rot}(L_{(i,j)}, r)$
    $(i,j) \leftarrow (j, (2i+3j))$
  end for
  for $i \leftarrow 0$ to $4, j \leftarrow 0$ to 4 **do** ▷ step π starts here
    $L'_{(i,j)} \leftarrow L_{((i+3j),i)}$
  end for
  $\alpha'_\Omega \coloneqq (L'_{(0,0)}, L'_{(1,0)}, L'_{(2,0)}, \ldots, L'_{(3,4)}, L'_{(4,4)})$
  return α'_Ω
end procedure

Algorithm 3 Step χ for round Ω

procedure $\chi(\alpha'_\Omega)$
  $(L_{(0,0)}, L_{(1,0)}, L_{(2,0)}, \ldots, L_{(3,4)}, L_{(4,4)}) \coloneqq \alpha'_\Omega$
  for $i \leftarrow 0$ to $4, j \leftarrow 0$ to 4 **do**
    $L'_{(i,j)} \leftarrow L_{(i,j)} \oplus ((\neg L_{(i+1,j)}) \wedge L_{(i+2,j)})$
  end for
  $\beta_\Omega \coloneqq (L'_{(0,0)}, L'_{(1,0)}, L'_{(2,0)}, \ldots, L'_{(3,4)}, L'_{(4,4)})$
  return β_Ω
end procedure

Algorithm 4 Step ι for round Ω

1: **procedure** $\iota(\beta_\Omega)$
2: $\beta'_\Omega \leftarrow \beta_\Omega$
3: $\mathbf{rc} \leftarrow \mathbf{RCTable}[\Omega]$
4: **for** $k \leftarrow 0$ to 63 **do**
5: $\beta'_\Omega[0,0,k] \leftarrow \beta'_\Omega[0,0,k] \oplus \mathbf{rc}[k]$
6: **end for**
7: **return** β'_Ω
8: **end procedure**

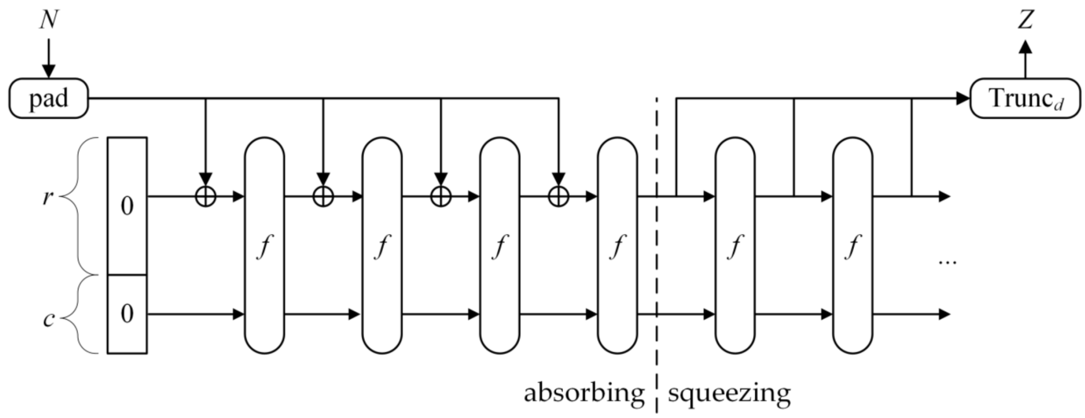


Figure 2.6: The diagram of the Keccak sponge function from NIST FIPS 202 [15]. In this diagram, N is the arbitrary-length input sequence and Z is the d-bit output sequence.

2.4.2 Keccak sponge functions: SHA-3 and SHAKE

A Keccak sponge function, Keccak$[c](N,d)$, consists of sequenced Keccak-$f[1600]$ permutations. It first *absorbs* an arbitrary-length input bitstring into its internal state and then can *squeeze out* an arbitrary-length output bitstring, and so is described as a *sponge function*. The input or output bitstring S of each invocation of Keccak-$f[1600]$ can be separated into two parts, S_r and S_c. In other words, $S = S_r \| S_c$, where '$\|$' denotes the operation to concatenate one-dimensional bitarrays. S_r is the part used in the sponge function to absorb or squeeze out the bitstring, while S_c is the part that stays unchanged for the input of the next Keccak-$f[1600]$ permutation. We refer to the length of S_r as *rate* and denote it as r, while the length of S_c is called *capacity* and denoted as c.

Figure 2.6 shows how Keccak$[c](N,d)$ absorbs the input bitstring N and squeezes out a d-bit result. Input message N is first padded ($N \leftarrow N\|$pad10*1) to a sequence with a length equal to a multiple of r and then split into blocks of r bits. After all r-bit blocks have been absorbed, in the squeezing stage, the output sequence is generated by concatenating the S_r 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

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

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

$$\text{SHAKE128}(M, d) = \text{KECCAK}[256](M \| 1111, d),$$
$$\text{SHAKE256}(M, d) = \text{KECCAK}[512](M \| 1111, d),$$

where users have free choice over the output length d.

2.5 ASCON

2.5.1 ASCON permutation

The ASCON team first introduces a family of 320-bit ASCON permutations. They describe the 320-bit state S as a two-dimensional structure, which is in the same shape with a plane (5×64) in the KECCAK-$f[1600]$ state. Therefore, the state can be separated into five 64-bit words (or five lanes), starting with the rate part (S_r) of the state and followed by the capacity part (S_c) when this permutation is used in a sponge construction, as in

$$S = L_0 \| L_1 \| L_2 \| L_3 \| L_4 = S_r \| S_c.$$

Note that, unlike KECCAK, ASCON interprets the state S as a big-endian byte array (or a bit-string) when needed. In this situation, the most significant byte of L_0 will be labeled as byte 0, and the least significant byte of L_4 will be labeled as byte 39.

ASCON performs either 6, 8, or 12 rounds of a substitution-permutation-network-based (SPN-based) transformation p to update the state. These three permutations are referred to as p^6, p^8, and p^{12}, respectively. Each SPN-based transformation p consists of three steps: Constant addition p_{C}, Substitution p_{S}, and Linear diffusion p_{L} in chronological order:

$$p = p_{\mathrm{L}} \circ p_{\mathrm{S}} \circ p_{\mathrm{C}}.$$

Constant Addition The step p_{C} updates the state by XORing an 8-bit round constant with the least significant byte of L_2 (byte 23). These round constants and their corresponding round Ω in the three ASCON permutations are as follows:

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

Substitution Like the step χ in Keccak-$f[1600]$, step p_S applies a non-linear substitution function on five bits in each row of the ASCON state. Table B.3 in Appendix B.1 show the substitution table with five-bit input and output, e.g., I_0 represents a bit from L_0 and I_1 represents a bit from L_1, etc. Similarly, we can execute step p_S in parallel on the five 64-bit lanes with Algorithm 5. Note that lines 6 to 10 are the same as step χ in Keccak-$f[1600]$.

Algorithm 5 Step p_S of ASCON

1: **procedure** p_S(**state**)
2: $(L_0, L_1, L_2, L_3, L_4) \leftarrow \textbf{state}$
3: $L_0 = L_0 \oplus L_4$
4: $L_4 = L_4 \oplus L_3$
5: $L_2 = L_2 \oplus L_1$
6: $L'_0 = L_0 \oplus ((\neg L_1) \wedge L_2)$
7: $L'_1 = L_1 \oplus ((\neg L_2) \wedge L_3)$
8: $L'_2 = L_2 \oplus ((\neg L_3) \wedge L_4)$
9: $L'_3 = L_3 \oplus ((\neg L_4) \wedge L_0)$
10: $L'_4 = L_4 \oplus ((\neg L_0) \wedge L_1)$
11: $L'_1 = L'_1 \oplus L'_0$
12: $L'_0 = L'_0 \oplus L'_4$
13: $L'_3 = L'_3 \oplus L'_2$
14: $L'_2 = \neg L'_2$
15: $\textbf{state} \leftarrow (L'_0, L'_1, L'_2, L'_3, L'_4)$
16: **return** **state**
17: **end procedure**

Linear Diffusion Step p_L provides linear diffusion within each 64-bit word in the state via Algorithm 6.

Algorithm 6 Step p_L in Ascon permutation

procedure p_L(**state**)
$(L_0, L_1, L_2, L_3, L_4) \leftarrow \textbf{state}$
$L'_0 = L_0 \oplus \textbf{Rot}(L_0, 19) \oplus \textbf{Rot}(L_0, 28)$
$L'_1 = L_1 \oplus \textbf{Rot}(L_1, 61) \oplus \textbf{Rot}(L_1, 39)$
$L'_2 = L_2 \oplus \textbf{Rot}(L_2, 1) \oplus \textbf{Rot}(L_2, 6)$
$L'_3 = L_3 \oplus \textbf{Rot}(L_3, 10) \oplus \textbf{Rot}(L_3, 17)$
$L'_4 = L_4 \oplus \textbf{Rot}(L_4, 7) \oplus \textbf{Rot}(L_4, 41)$
$\textbf{state} \leftarrow (L'_0, L'_1, L'_2, L'_3, L'_4)$
return **state**
end procedure

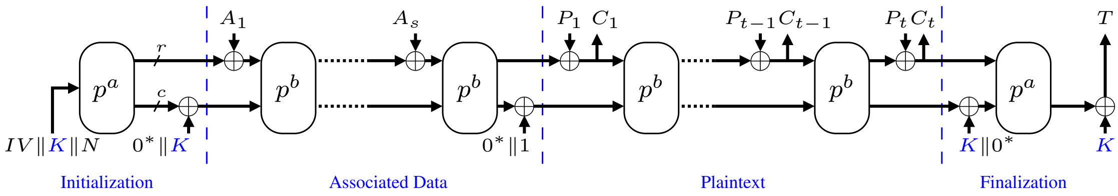


Figure 2.7: Encryption of Ascon AEAD

2.5.2 Ascon authenticated encryption with associated data

Based on the Ascon permutations, Dobraunig et al. designed the Ascon *authenticated encryption with associated data* (AEAD) and Ascon hashing. For Ascon AEAD, they defined two encryptions Ascon-128 and Ascon-128a. They both take four inputs: a 128-bit key K, a 128-bit nonce N, an arbitrary-length associated data bitstring A, and an arbitrary-length plaintext P, and then calculated the ciphertext C with the same size of the plaintext. The mathematical structures of these two are very similar but with some differences in the choices of parameters. I use '$|X|$' to denote the length of a one-dimensional bitstring X in bits, such as $|N| = 128$.

Figure 2.7 shows the encryption of Ascon AEAD, which includes four processes: initialization, processing associated data, processing plaintext, and finalization. In my experiments, I demonstrated my attack only on Ascon-128. For this function, there are four important parameters used: the key size ($|K| = 128$ bits[1]), the rate size ($r = 64$ bits), the round number ($a = 12$) for the invocations of p^a in initialization and finalization, and the round number ($b = 6$) for the invocations of p^b when processing associated data and plaintext. The constant 64-bit initial vector IV records these four parameters in its first four bytes, then being padded with an all-zero bitstring, so the value of the initial vector is:

$$IV = \texttt{0x80400c0600000000}.$$

Initialization In this process, we first concatenate the initial vector with the key and the nonce, and then we input the bitstring into permutation p^{12}. Then, we calculate the output of the process by XORing the key into the last 128-bit of the permutation output. We can summarize the process by the following equation:

$$S_r \| S_c = S \leftarrow p^{12}(IV \| K \| N) \oplus (0^{192} \| K).$$

Processing associated data If the associated data bitstring is null, there will not be any invocations of permutations in this process. For any other associated data A, we first pad the data with a single '1' and the smallest number of repeated '0's such that the size of the padded

[1]The official document uses 'k' to denote the size of the key, but I use '$|K|$' to distinguish it from the z coordinate k in this thesis.

data will equal a multiple of the rate size r (64-bit), and then splitting the padded data into r-bit bitstrings:

$$A_1, \ldots, A_s \leftarrow \begin{cases} r\text{-bit blocks of } A \| 1 \| 0^{r-1-(|A| \bmod r)} & \text{if } |A| > 0, \\ \emptyset & \text{if } |A| = 0. \end{cases}$$

Then, for each block, we XOR the associated data into the rate part of the state and update the state by p^6:

$$S_r \| S_c \leftarrow S \leftarrow p^6((S_r \oplus A_\tau) \| S_c), \text{ for } 1 \leq \tau \leq s.$$

After processing the last block (also in the case of null associated data), we flip the last bit of the state:

$$S \leftarrow S \oplus (0^{319} \| 1).$$

Processing plaintext Similar to the padding step processing associated data, we first pad the plaintext P with a single '1' and the smallest number of repeated '0's such that the size of the padded data will equal a multiple of the rate size r (64-bit), and then splitting the padded data into r-bit bitstrings:

$$P_1, \ldots, P_t \leftarrow r\text{-bit blocks of } P \| 1 \| 0^{r-1-(|P| \bmod r)}.$$

Then, for each block, we XOR the plaintext into the rate part of the state for calculating cipher blocks, and update the state by p^6, except for the last block:

$$C_\tau \leftarrow S_r \oplus P_\tau, \text{ for } 1 \leq \tau \leq t, \quad S_r \| S_c \leftarrow S \leftarrow \begin{cases} p^6(C_\tau \| S_c), & \text{for } 1 \leq \tau \leq t-1, \\ C_\tau \| S_c, & \text{for } \tau = t. \end{cases}$$

Then we concatenate the cipher blocks and truncate the result to the length of the plaintext for the output ciphertext bitstring:

$$C \leftarrow \mathbf{Trunc}(C_1 \| ... \| C_t, |P|),$$

where function $\mathbf{Trunc}(X, n)$ truncates the one-dimensional bitarray X to its first n bits.

Finalization. At the beginning of finalization, we use the key for the third time, by XORing the key into the first 128 bits of the capacity part (S_c) of the state:

$$S_c \leftarrow S_c \oplus (K \| 0^{128}),$$

and then update the state by the permutation p^{12}:

$$S \leftarrow p^{12}(S_r \| S_c).$$

Finally, we can calculate the 128-bit tag T by XORing the key and the last 128 bits of the state:

$$T \leftarrow S[192:320] \oplus K.$$

Now the ciphertext C and the tag T are the output data of the ASCON-128 encryption.

Algorithm 7 in Appendix B.1 shows the procedure of ASCON-128 encryption. Meanwhile, a very similar procedure of its decryption is shown in Algorithm 8 in Appendix B.1, where the initialization and the associated data processing are the same, but here we XOR each input ciphertext block with the rate part of the state in the same invocation of the p^6 permutation to find the plaintext:

$$P_\tau \leftarrow S_r \oplus C_\tau, \text{ for } 1 \leq \tau \leq t.$$

Another difference is that we need to compare the tag T' calculated in the finalization process with the input tag T, and will reject the decryption if they are different, to protect the system from chosen-cipher attacks.

ASCON-128a shares the same encryption and decryption procedure with ASCON-128 and only differs in parameters: $|K| = 128$, $r = 128$, $a = 12$, $b = 8$, and $IV =$ `0x80800c0800000000`.

In the later chapters, I refer to the internal states of Ascon p^{12} permutation as follows:

$$\textbf{Input} = \beta_{-1} \xrightarrow{p_C, p_S} \alpha_0 \xrightarrow{p_L} \beta_0 \xrightarrow{p_C, p_S} \alpha_1 \xrightarrow{p_L} \beta_2 \cdots \xrightarrow{p_L} \beta_{11} = \textbf{Output},$$

and the bits, nibbles, bytes, or 16-bit fragments in a state will be denoted in the same style in Keccak, except that there is no y coordinate j, such as $\alpha_0[i,k]$ or $\beta_1[i, {}^{\mathbf{8}}k]$[8]

2.6 General experimental setting

This section explains the hardware and general setup for all my experiments, including the Keccak experiment on an 8-bit device (Section 3.2), as well as the toy stream cipher experiment (Section 4.3), the Keccak experiment (Section 4.4), and the Ascon experiments (Chapter 5) on a 32-bit device.

2.6.1 Measurement setting

Recording equipment For power-trace recording, I used an NI PXI platform, including an NI PXIe-5160 [105] 10-bit oscilloscope, which can sample at 2.5 GS/s into 2 GB of memory, for recording the supply-current traces, a PXIe-5423 [106] arbitrary waveform generator to supply the clock signal for the target devices, and a PXI-4110 [107] power supply, all installed in the same PXIe chassis. I configured both the oscilloscope and waveform generator to use a common 100 MHz reference clock signal from the latter. This helps me to preserve the phase lock between the oscilloscope's sampling clock and the CPU clock of the two target devices, to avoid misaligned recordings.

Choudary's 8-bit board I used a power-analysis test board designed by Choudary [52, Section 2.2.2] as my 8-bit target, which will also be referred to as Choudary's board. Its processor is the 8-bit microcontroller ATxmega256A3U [108]. For the schematics of this target board, see [52, Figure 2.8].

When recording traces for my experiments on this board, I powered it using the PXI-4110 [107] power supply. The target 8-bit processor was supplied with an external 2 MHz square-wave clock signal generated by the PXIe-5423 waveform generator, configured to use the same reference clock as the PXIe-5160 oscilloscope. I used a coaxial cable with 50 Ω to connect the oscilloscope and connector `SMA_MATCH` on the board [52, Figure 2.8], where Choudary already provided an impedance-matched connection point.

This way, the samples on the traces captured by the oscilloscope will be proportional to the current flowing through the 1 Ω resistor, `RP` [52, Figure 2.8], which is inserted between system GND and the GND pin of the processor. With a sampling rate of 250 MHz, each clock cycle in the recorded traces contains exactly 125 data points (125 points per clock cycle or 125 PPC), with a phase jitter of about 8 ps standard deviation.

ChipWhisperer-Lite 32-bit board My target 32-bit processor was the STM32F303RCT7, which has one ARM Cortex-M4 core, on a ChipWhisperer-Lite (CW-Lite) board [109]. I will also refer to this device as the CW-Lite board.

Note that this board includes a power-analysis oscilloscope, but that could not record more than 24 kilosamples per trace (at up to 105 MS/s). Considering that this duration would only cover a very small part of my target algorithms, I decided to use an external oscilloscope instead of the onboard one. At the same time, I wanted to preserve the phase lock between the oscilloscope's sampling clock and the CPU clock. Therefore, I used again the PXIe-5160 oscilloscope and the PXIe-5423 waveform generator as an external clock signal source to supply the target board with a 5 MHz square wave signal, which is the lowest frequency for the board to work stably. On the other side, I configured the oscilloscope at the highest sampling rate, 2.5 GS/s, so it collected traces with 500 points per clock cycle (PPC). Compared to recording the trace directly with a lower sampling rate, this setting provided me with the flexibility to later digitally downsample to different PPC values, as needed, by using the sum of consecutive raw samples as a new sample. Note that the USB cable between the NI controller and the CW-Lite board was not only the I/O channel but also the power source of the device, so it did not require the PXI-4110 power supply.

Since I did not use the onboard oscilloscope, I had to create an impedance-matched connection for the power signal. It used a 50 Ω coaxial cable to connect the oscilloscope and the CW-Lite's measure connector (JP10) [110]. However, JP10 taps the V_{DD} connection of the CPU after a 13 Ω source impedance (R66+R67), unlike the case of measuring from the GND side on Choudary's board. This posed a problem: the 3.3 V DC level would have led to a high current drain with the oscilloscope input configured to 50 Ω impedance and DC coupling, where the DC level should

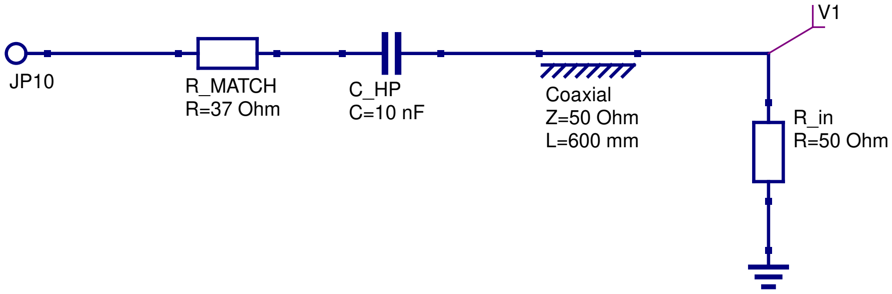


Figure 2.8: My impedance-matched connection with AC coupling for CW-Lite board measurement. Note that `V1` is the value being recorded by the oscilloscope.

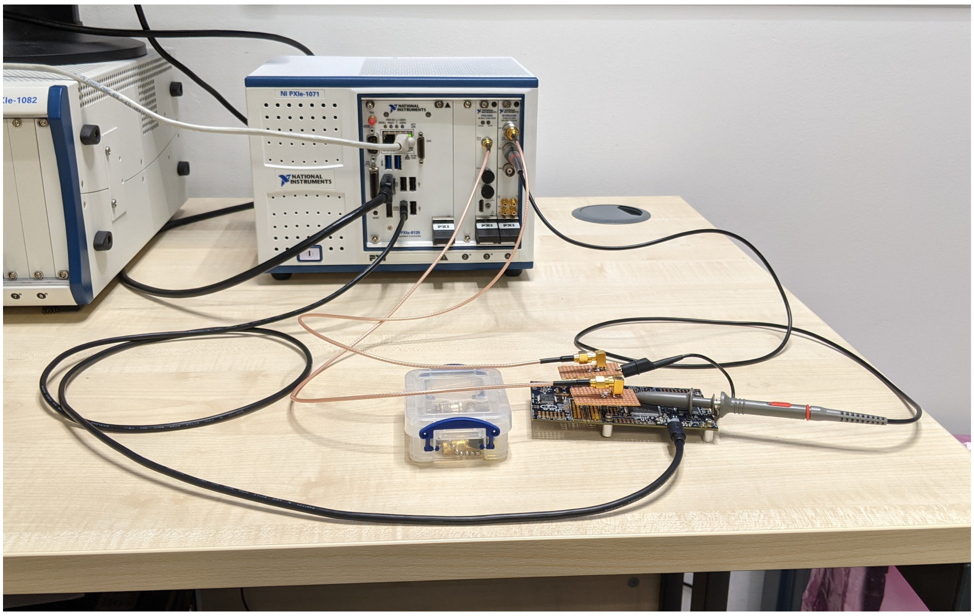


Figure 2.9: Measurement setup for the experiments on the 32-bit device.

be confined to between ±2.5 V. However, if there is no 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, as shown in Figure 2.8, I connected the coaxial cable to JP10 via a 37 Ω resistor[2] `R_MATCH` (to better match the 50 Ω impedance of the cable) and a 10 nF capacitor `C_HP` (to block the 3.3 V DC component). Together with the 50 Ω impedance (`R_in`) 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, I avoid ringing on the cable by terminating it at both ends, but at the same time use AC coupling[3] with an impulse response that decays within a few clock cycles. Figure 2.9 shows the connected recording platform for the CW-Lite 32-bit device.

[2]It was actually a 15 Ω and a 22 Ω resistor in series.
[3]The oscilloscope remained configured in DC-coupling mode with 50 Ω termination.

2.6.2 Recorded traces

As implied in Section 1.1.2, we need to record at least two sets of traces for the profiling stage and the attack stage, respectively, in template attack experiments. However, for some early checking purposes and for preventing overfitting issues, I further split the recorded traces into the following sets in the profiling stage:

Reference I used the mean array of the relatively small number of traces in this set to detect and then exclude possible errors such as trigger accidents, misalignment, and other abnormal recordings. For my experiments on Choudary's board, I checked that all traces recorded in other sets have a Pearson correlation coefficient of at least 0.98 against this mean array, or they would be seen as abnormal traces. I applied similar checks to my experiments on the CW-Lite board, but no errors were detected.

Detection When we target a cryptographic algorithm, the recorded raw traces are usually too long for profiling templates directly, even with the LDA dimensionality reduction. Therefore we should first determine the samples that contain information about the target intermediate values, where these samples were referred to as *points of interest* (PoI) by Chari et al.'s first template attack [39, Sec. 3.1]. I used the traces in this set with some statistical methods (see Section 3.2.2) to determine m points of interest in my experiments.

Profiling Once the m PoI have been selected with the detection set, I concatenate the corresponding samples from the traces in the profiling set into new m-sample traces, and then use these to profile LDA-based templates in my attacks. Separating the detection and profiling sets avoids the risk of overfitting.

Validation I used these traces to calculate the two metrics, SR and GE, introduced in Section 2.1.4, for template-quality evaluation and fine-tuning parameters.

Table 2.1 shows the number of traces for each set in the profiling state of my main experiments. The number of instructions covered by each trace depends on the targeted algorithm. In all my experiments on Keccak, I focused more on the Keccak-$f[1600]$ permutation, and a trace covers only the first few rounds of a single permutation, considering that the size of a trace covering a full permutation could be very large. Therefore, we can collect more than one such trace with a SHA-3 function with multiple Keccak-$f[1600]$ permutations in the profiling stage. For example, with Choudary's board, I collected four traces from each SHA3-512 function with the input that requires four invocations of Keccak-$f[1600]$ to absorb, while I collected 10 each time there with the CW-Lite board. In contrast, in all my experiments on Ascon, a trace covers

Table 2.1: The number of traces in each set of the profiling stage in my main experiments.

	Experiments			
Algorithm	KECCAK	toy stream cipher	KECCAK	ASCON
Board	Choudary's	CW-Lite	CW-Lite	CW-Lite
Reference	1 600	1 600	1 600	1 600
Detection	16 000	16 000	16 000	16 000
Profiling	32 000	64 000	64 000	64 000
Validation	1 000	1 000	1 000	1 000
Source	SHA3-512 with four KECCAK-$f[1600]$	A full encryption of stream cipher	SHA3-512 with ten KECCAK-$f[1600]$	A full encryption of ASCON-128

the full AEAD mode thanks to the fewer clock cycles it requires. In other words, I collected only one trace for each ASCON-128 encryption. For the toy stream cipher experiment, a trace covers one full encryption as well.

Regarding the traces in the attack set, their categorization depends on the different goals of each experiment. Therefore, I introduce this later in the corresponding sections, respectively.

2.6.3 Computing resources

Although I ran some of my early template attack experiments on other computers, I provide all the run-time estimation in this thesis with the last machine I used, which is the server `dev-cpu-1` in my department. This machine was equipped with 32 Intel® Xeon® Gold 5218 processors (22M Cache, 2.30 GHz) [111] and 252 GB memory.

On this machine, I used `Python` (v3.10.6) for all my experiments. My template profiling procedure highly relies on `NumPy` (v1.21.5 [112, 113]) for matrix multiplication, inverse calculation, finding eigenvalues and eigenvectors, etc. It also relies on `scikit-learn` (v1.1.3 [114, 115, 116]) to build multiple linear regression models. These also rely on the Intel® oneAPI Math Kernel Library [117] to accelerate and parallelize matrix computations.

Limited points of interest In my experiments, template profiling is the most resource-consuming step compared to other steps such as belief propagation or secret enumeration. Therefore, I had to take my computing resources and their limitations into consideration when I chose the parameters, points of interest m in particular, used in this stage.

Excluding the data loading and saving, I measured the following time intervals during template profiling, to show where the bottleneck is:

T_0: build the multiple linear-regression model

T_1: calculate the inter-class scatter matrix $\mathbf{B}$ and the total intra-class scatter matrix $\mathbf{W}$

T_2: calculate $\mathbf{W}^{-1}\mathbf{B}$

T_3: find eigenvalues and eigenvectors of $\mathbf{W}^{-1}\mathbf{B}$

T_4: calculate the projected pooled covariance matrix $\mathbf{S}_{\text{proj}}$, the projected expected traces $\hat{\mathbf{x}}_{b,\text{proj}}$, and the projection matrix $\mathbf{A}$

As in Section 2.1, we now profile a template with Schindler's linear-regression model for an l-bit target intermediate value from N profiling traces (i.e., $N = \sum_{b=0}^{2^l-1} n_b$, where n_b is the number of traces for each of the 2^l values b). Given m points of interest, we apply an LDA reduction down to $m' \ll m$ dimensions.

In my experiments, m (points of interest) plays the most important role when it comes to the run time, because it varies most across different target intermediate values, while l, N, and m' are fixed in a single experiment. Table 2.2 shows the run time, where the values are averaged over 100 profiling runs with different m, given fixed $l = 8$, $N = 64000$, and $m' = 8$. We can see that the multiple linear regression and eigenvector decomposition take the longest.

Table 2.3 shows the run time for some larger m given the same other fixed parameters, based on single trials. We can see the run time increases superlinearly with m, but remains acceptable when we profile only a few templates. Considering that I target hundreds of intermediate bytes in my attacks (e.g., 600 for Keccak on the 8-bit device, and 1920 for Keccak on the 32-bit device), I will first keep m below about 3500, as long as that achieves a satisfactory success rate. My template profiling used about 45 GB RAM in my largest case $m = 15\,000$, which is still well below the 252 GB RAM available on `dev-cpu-1`, so memory use did not constrain my choice of m.

Unlike in the profiling stage, multithreading did not help much in the attack stage on this machine, so my template recovery procedure remained single-threaded in the later attack experiments. Table 2.2 and Table 2.3 also show the attack-stage run time, averaged over 1000 trials for each value. The attack-stage run time is far faster compared to the profiling stage. We can observe that m still affects the attack-stage run time, but not as significantly as in the profiling stage.

Table 2.2: The run-time estimation of template profiling and attack with different m ($l = 8$, $N = 64000$, $m' = 8$), where the results for profiling are estimated with the average of 100 trials and the results for attack are estimated with 1000 trials.

Profiling								
#Samples (m)		1000	1500	2000	2500	3000	3500	4000
T_0	CPU time (s)	175.953	190.565	228.531	271.053	320.039	356.163	460.848
	Wall time (s)	16.600	23.427	34.323	41.283	54.288	64.174	134.086
T_1	CPU time (s)	24.094	37.473	51.549	62.027	76.201	84.926	100.794
	Wall time (s)	2.617	4.192	5.958	7.880	9.477	11.011	12.534
T_2	CPU time (s)	3.341	7.483	9.054	11.088	13.989	16.876	21.294
	Wall time (s)	0.119	0.387	0.657	1.153	1.534	2.330	3.113
T_3	CPU time (s)	30.167	68.268	114.539	190.557	304.332	440.430	558.884
	Wall time (s)	0.961	2.593	5.117	11.905	23.368	41.545	58.023
T_4	CPU time (s)	0.589	1.105	1.642	2.748	3.483	3.263	3.916
	Wall time (s)	0.112	0.287	0.493	0.767	1.081	1.425	1.847
Total	CPU time (s)	234.144	304.894	405.316	537.473	718.044	901.659	1145.736
	Wall time (s)	20.409	30.886	46.548	62.988	89.747	120.486	209.603
Attack								
#Samples (m)		1000	1500	2000	2500	3000	3500	4000
Total	CPU time (µs)	300.087	326.923	353.150	360.314	353.743	365.349	388.008
	Wall time (µs)	314.850	347.421	372.640	385.972	389.504	381.505	432.831

Table 2.3: The run-time estimation of template profiling with selected larger m values. The results for profiling are estimated with only one trial and the results for attack are estimated with the average of 1000 trials.

Profiling					
#Samples (m)		5000	7000	10000	15000
Total	CPU time (s)	1687.176	2878.112	6176.754	14579.546
	Wall time (s)	236.241	483.436	1129.878	2842.976
Attack					
#Samples (m)		5000	7000	10000	15000
Total	CPU time (µs)	398.024	456.010	466.810	549.702
	Wall time (µs)	420.910	481.697	493.438	566.547

Chapter 3

LDA-based template attack on a Keccak 8-bit implementation

I started my investigation by extending Choudary's LDA-based template attack to perform a full-state recovery on three intermediate states of the Keccak-$f[1600]$ permutations in a Keccak sponge function, SHA3-512. With support from the techniques of secret enumeration or belief propagation, the full input of the SHA3-512 function can be recovered.

As mentioned in Chapter 2, previous 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 SHA3-$d(K\|M)$ with known variable input message M and fixed key K. For example, a DPA-style attack could not reconstruct the complete input of MAC-Keccak. Instead, I demonstrate here a template attack on a single invocation of Keccak-$f[1600]$ to reconstruct both its 1600-bit input and output bitstrings. Using this capability, I then demonstrate recovery of a complete SHA3-512 input given a single power trace and then verify the results with the given output of SHA3-512. My technique therefore not only can recover a MAC-Keccak arbitrary-length key K without prior knowledge of the message M, but also can naturally extend to other Keccak-f applications with confidential inputs or outputs, such as random-bit generation.

3.1 Attack strategy

3.1.1 On a full Keccak sponge function

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 in any middle rounds in Keccak-f, we can calculate the input, output, and even any other intermediate states of the permutation from this successfully recovered state. Once we

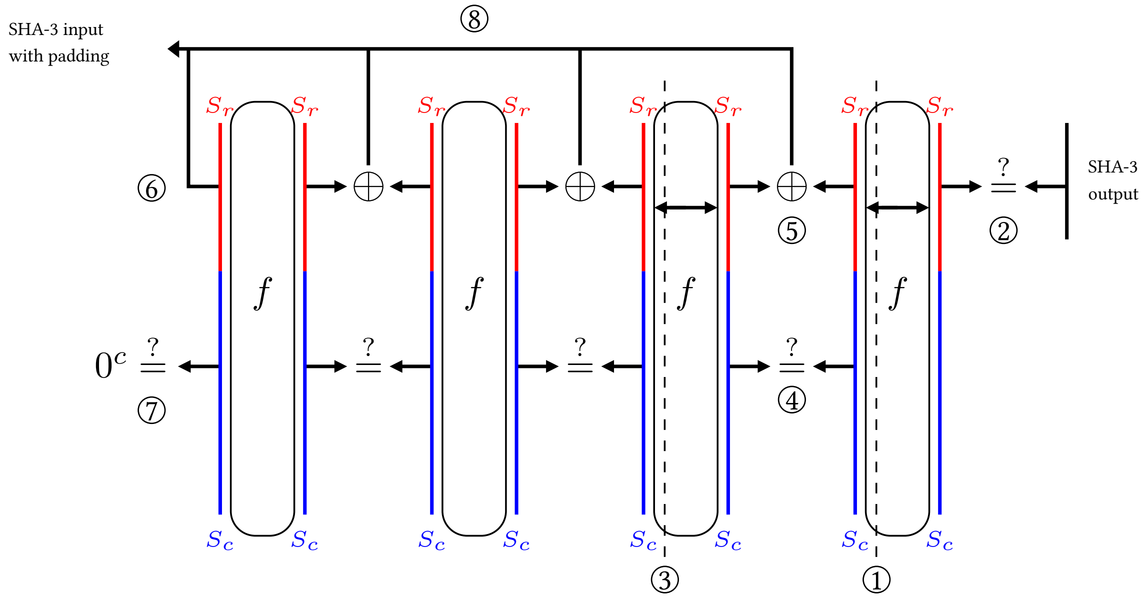


Figure 3.1: The procedure to reconstruct SHA3-d inputs by template attack: ① reconstruct an intermediate state of the last Keccak-$f[1600]$ permutation and calculate its input and output; ② verify the correctness by checking whether the first d bits in the output match the SHA3-d output; ③ repeat ① on other permutations but ④ verify the correctness by checking whether the S_c of the output matches that of the input in the following permutation; ⑤ XOR the S_r of the two consecutive permutations to calculate each part of the SHA3-d input; ⑥ in the special case of the first r bits of the SHA3-d input, that part is identical to the input Sr of the first Keccak-$f[1600]$ permutation and ⑦ the input S_c of that permutation should be c 0 bits; ⑧ concatenate each part to form the complete SHA3-d input with padding.

have a partially public input or output state, we can easily verify whether we have correctly determined the state.

Given the fact described above, Figure 3.1 depicts the high-level procedure of my attack strategy to recover the input of a SHA3-d function. Firstly, we can use LDA-based template attacks as well as some mathematical tools 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-$f[1600]$ invocation, and then its output. We can verify the correctness of the latter by checking whether its first d bits match the SHA3-d output.

Secondly, 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 $c = 2d$ bits, also known as S_c, match those of the input of its successor. Without a predecessor, the input of the first Keccak-$f[1600]$ permutation has S_c equal to an all-zero string, following the Keccak specification.

Thirdly, we can calculate each part of the SHA3-d input by XORing the S_r of the input and the output of two consecutive KECCAK-$f[1600]$ permutations. In the special case of the first r bits of the SHA3-d input, that is identical to the S_r of the input of the first KECCAK-$f[1600]$ permutation. Finally, after concatenating all the parts and removing the padding, the input of this SHA3-d function can be recovered.

3.1.2 On a single invocation of KECCAK-$f[1600]$

With the procedure introduced in the previous subsection, now the question is how to successfully reconstruct the values of all the bytes in an intermediate state. I first tried to use the LDA-based template attack on every byte in a single full intermediate state α_0'. With the logarithmic likelihood tables predicted by these templates, we can sort the tables in descending order of the likelihood into their ranking tables, and then determine their values by selecting their top candidates. However, the diffusion of KECCAK-f means that even a single bit error will result in a completely different input or output, and the quality of these templates was still too low that their predicted rank tables were not good enough for secret enumeration.

Therefore, I combined the LDA-based template attack on *three* consecutive intermediate states, α_0', β_0, and α_1, with an enumeration technique around the mathematical structure of KECCAK-f to correct errors. Since the state of KECCAK-$f[1600]$ contains 200 bytes, there will be 600 per-byte likelihood tables in total, which are then sorted into 600 ranking tables. 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 of diffusion takes multiple rounds to come into effect. This makes it possible to combine likelihood information from neighboring intermediate states to build better rank tables for secret enumeration, and I built a *three-layer* scheme that can gradually combine the probabilistic information available about these bytes into a full state. At the bottom, Layer 1 first merges the rank tables associated with five bytes from α_0' in the same byte row, enumerates a limited number of combinations, updates the likelihood of each enumerated combination with the tables of β_0, and then generates in total 40 new rank tables that cover entire byte rows. Layer 2 then repeats the steps in layer 1 to combine and enumerate five byte rows in the same byte slice, update their likelihood values by the tables of α_1, and then generate eight new rank tables for byte slices. Finally, Layer 3 just concatenates the eight top candidates from each byte-slice ranking table, and verifies the correctness of the resulting full intermediate state α_0'. The detailed description of this three-layer scheme is in 3.3.

3.2 Template attack on SHA3-512

I demonstrate all experiments in this chapter on **SHA3-512**(M), considering that this is the SHA3-d variant with the largest capacity c, i.e. the largest security margin.

3.2.1 Target implementation and measurement setup

The SHA3-512 implementation targeted here is based on the KECCAK-$f[1600]$ implementation in the official C reference code, the Extended KECCAK Code Package (XKCP) [102], and I ran it on Choudary's 8-bit board [52, Sec. 2.2.2], following the description in Section 2.6.1 to record power traces. Each raw trace contained 40 000 clock cycles (or 5 000 000 samples), covering the power consumption of the first two rounds of one KECCAK-$f[1600]$ permutation, which include the target states α'_0, β_0, and α_1.

For the attack stage, I also recorded two sets of SHA3-512 traces. The first one contains 1000 random inputs with a 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 it needs four KECCAK-$f[1600]$ permutations to absorb these inputs, which is the same setting as the traces in the profiling stage.

3.2.2 Interesting clock cycle detection

Since the recorded raw traces were too long for profiling templates directly, we should first determine the clock cycles that contain information about the targeted intermediate states, which in the KECCAK-$f[1600]$ permutation each contain 200 intermediate bytes. Given a time sample s and target intermediate values, we can use some statistical metrics, such as using Welch's t-test [118, 119], signal-to-noise ratio (SNR) [120], or normalized inter-class variance (NICV) [121], as selection criteria. Once the value of these statistical metrics exceeds a chosen threshold, indicating a relatively high correlation, the corresponding time sample will be determined as a point of interest (PoI) of the target intermediate values.

However, since the recorded traces were very long, I detected only the peak current in each clock cycle and determined whether all the samples in such a clock cycle should be selected as PoIs. In other words, I selected the *interesting clock cycles*. Moreover, I decided to build a multiple linear regression with a stochastic model $\mathcal{F}_9$ for the target intermediate values and the peak current samples, and decide whether the correlation is sufficiently high via the coefficient of determination (R^2), as estimated by the regression. The choice of R^2 better fits my experiments than the other statistic metrics, as now the $\mathcal{F}_9$ model applies to both the interesting-clock-cycle detection and template profiling. In comparison, Welch's t-test is more suitable if there are just two groups of intermediate values, and both SNR and NICV will consider the non-linear part of signals, which is not used in the linear regression with $\mathcal{F}_9$ model.

Considering that traditionally the interval $-0.3 < R < 0.3$ indicates a variable of low correlation in many research fields [122, Table 1], I selected clock cycles based on the threshold $R^2 > 0.09$, but the threshold can be lowered to select more clock cycles if needed. The multiple linear regression and R^2 were calculated using the `LinearRegression` class in the Python

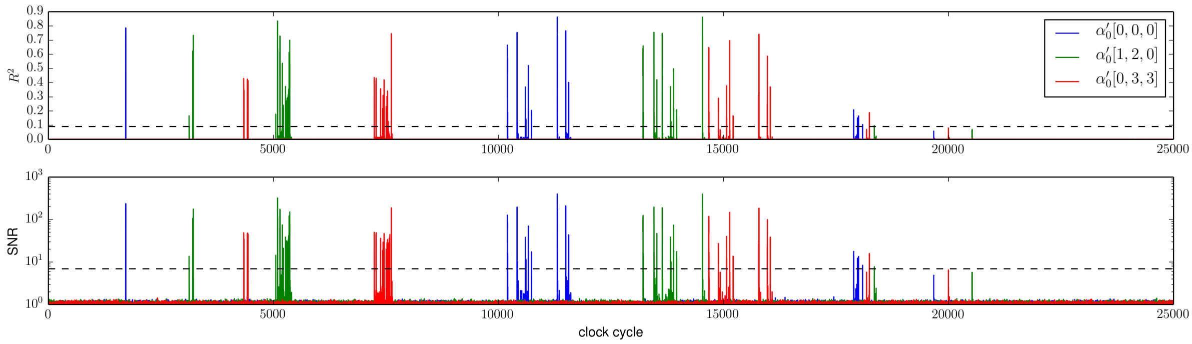


Figure 3.2: Comparison of the highest R^2 coefficient and SNR value in each clock cycle.

library `scikit-learn` [123]. Fig. 3.2 shows the resulting highest R^2 value occurring in each clock cycle, along with the SNR value [120]

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

at each per-clock-cycle peak time s for comparison. This $R^2 > 0.09$ threshold is approximately equivalent to an $\mathrm{SNR} > 7$ threshold.

Let $\mathcal{A}'_{0,[i,j,{}^{\mathbf{8}}k]^{\mathbf{8}}}$ be the set of interesting clock cycles for intermediate byte $\alpha_0'[i,j,{}^{\mathbf{8}}k]^{\mathbf{8}}$, $\mathcal{B}_{0,[i,j,{}^{\mathbf{8}}k]^{\mathbf{8}}}$ that of $\beta_0[i,j,{}^{\mathbf{8}}k]^{\mathbf{8}}$, and $\mathcal{A}_{1,[i,j,{}^{\mathbf{8}}k]^{\mathbf{8}}}$ that of $\alpha_1[i,j,{}^{\mathbf{8}}k]^{\mathbf{8}}$. The clock cycles that leak these $3 \times 200 = 600$ intermediate bytes should be sufficient for profiling 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. Here is an example of how the eight bits in byte $\alpha_0'[2,1,1]^{\mathbf{8}}$, labeled here as from $\alpha_0'[2,1,1]^{\mathbf{8}}[0]$ to $\alpha_0'[2,1,1]^{\mathbf{8}}[7]$, match those from up to two bytes in α_0:

$$\begin{aligned}
\alpha_0'[2,1,1]^{\mathbf{8}}[0] &= \alpha_0[0,2,0]^{\mathbf{8}}[5], & \alpha_0'[2,1,1]^{\mathbf{8}}[1] &= \alpha_0[0,2,0]^{\mathbf{8}}[6],\\
\alpha_0'[2,1,1]^{\mathbf{8}}[2] &= \alpha_0[0,2,0]^{\mathbf{8}}[7], & \alpha_0'[2,1,1]^{\mathbf{8}}[3] &= \alpha_0[0,2,1]^{\mathbf{8}}[0],\\
\alpha_0'[2,1,1]^{\mathbf{8}}[4] &= \alpha_0[0,2,1]^{\mathbf{8}}[1], & \alpha_0'[2,1,1]^{\mathbf{8}}[5] &= \alpha_0[0,2,1]^{\mathbf{8}}[2],\\
\alpha_0'[2,1,1]^{\mathbf{8}}[6] &= \alpha_0[0,2,1]^{\mathbf{8}}[3], & \alpha_0'[2,1,1]^{\mathbf{8}}[7] &= \alpha_0[0,2,1]^{\mathbf{8}}[4].
\end{aligned}$$

Therefore we can extend the set of interesting clock cycles for $\alpha_0'[2,1,1]^{\mathbf{8}}$ from $\mathcal{A}'_{0,[2,1,1]^{\mathbf{8}}}$ to $\mathcal{A}'_{0,[2,1,1]^{\mathbf{8}}} \cup \mathcal{A}_{0,[0,2,0]^{\mathbf{8}}} \cup \mathcal{A}_{0,[0,2,1]^{\mathbf{8}}}$. This similarly applies to the intermediate state α_1, but the other way round.

Table 3.1 lists the number of interesting clock cycles selected for each intermediate byte after that extension. In state α_0', the numbers in lanes $L_{(0,0)}$, $L_{(3,2)}$, and $L_{(4,3)}$ are smaller because step ρ rotates the bits in these lanes by multiples of eight. For example, we always have $\alpha_0'[3,2,0]^{\mathbf{8}} = \alpha_0[4,3,7]^{\mathbf{8}}$, which implies that $\mathcal{A}'_{0,[3,2,0]^{\mathbf{8}}} = \mathcal{A}_{0,[4,3,7]^{\mathbf{8}}} = \mathcal{A}'_{0,[3,2,0]^{\mathbf{8}}} \cup \mathcal{A}_{0,[4,3,7]^{\mathbf{8}}}$, and that does not extend the set of clock cycles.

Table 3.1: The number of interesting clock cycles for each byte in $\alpha_0'[i, j, {}^{\mathbf{8}}k]^{\mathbf{8}}$ (left) and $\beta_0[i, j, {}^{\mathbf{8}}k]^{\mathbf{8}}$ (right). The numbers for α_1 (omitted here) look similar to those for α_0'.

(i,j) \ $^{\mathbf{8}}k$	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) \ $^{\mathbf{8}}k$	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

3.2.3 Profiling templates

Pre-processing When targeting a specific byte, we shall select only the samples in the interesting clock cycle set of this byte to profile its template. For example, when profiling the template for $\alpha_0'[2, 1, 1]^{\mathbf{8}}$, the profiling traces reassembled this way cover 87 clock cycles with $m = 87 \times 125 = 10875$ samples.

Since the 125 samples per clock cycle still lead to too long execution times for profiling templates for all the 600 target intermediate bytes (see Section 2.6.3 for the discussion of run time), I reduced the sampling rate further by a factor of 5, averaging five consecutive samples into a new sample, for a more feasible number of m for each target byte.

Templates with LDA compression After the detection and pre-processing steps, now there were shorter traces for profiling templates for each of 600 bytes. I then apply Choudary et al.'s method [49] described in Section 2.1. In the LDA compression, I selected the eigenvectors by the criterion introduced in Section 2.1.3, leading to using only the $m' = 8$ eigenvectors with the largest eigenvalues to form a projection matrix $\mathbf{A}$ for each byte, whereas the other eigenvalues are negligible. Besides the projection matrices, the templates, therefore, contain 8×8 pooled covariance matrices and 8-point expected traces.

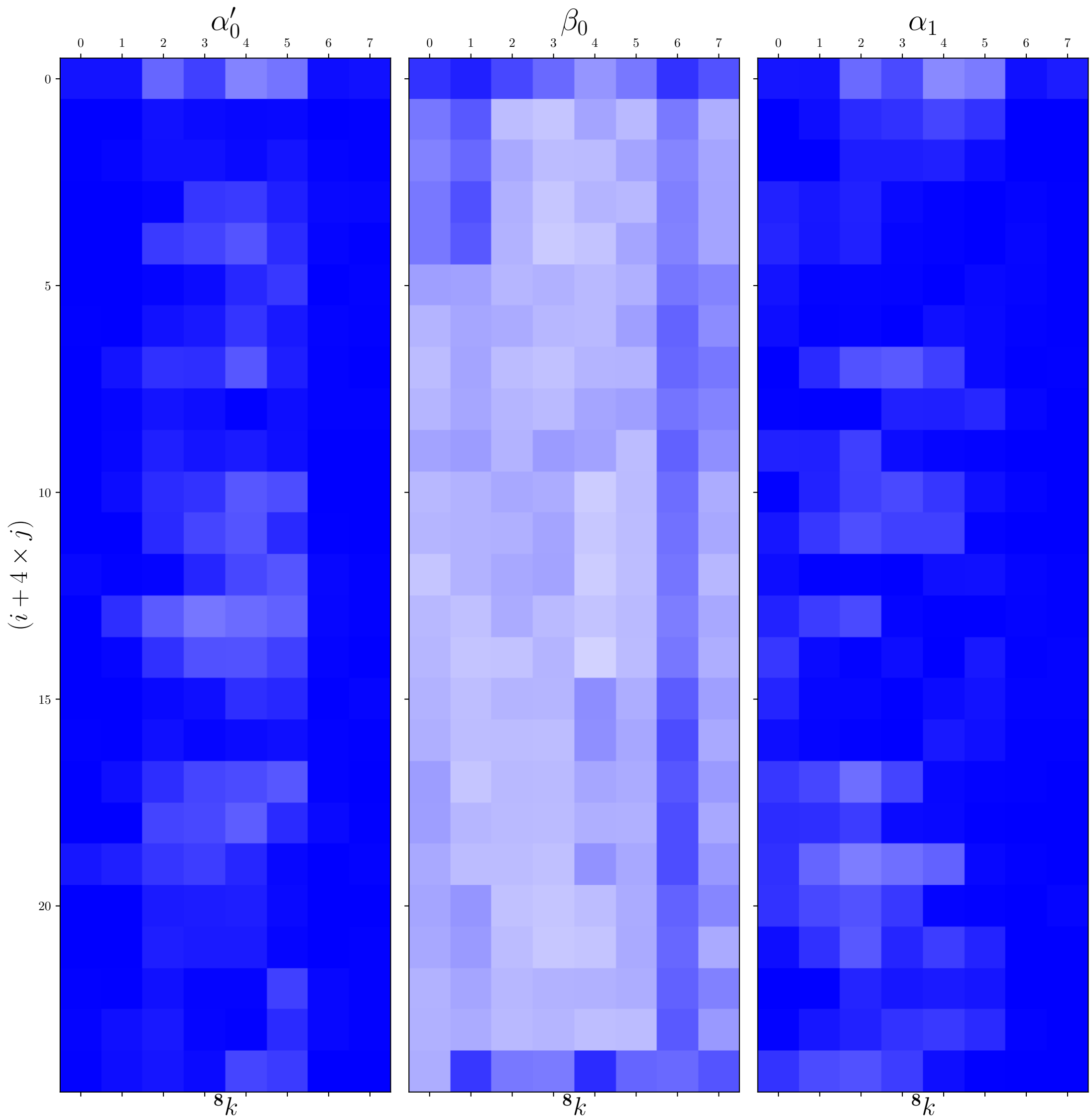


Figure 3.3: Success rates on my target states α'_0, β_0, and α_1, where the higher the value, the bluer is a block.

3.2.4 Evaluating the quality of templates

Having profiled the templates, I used the 1000 validation traces to estimate template quality, resulting in 600 rank tables for each validation trace. Table B.4 shows the resulting success rates for states α'_0 and β_0, and Table B.5 shows the guessing entropy for each byte of states α'_0 and β_0. I also plot these results in Figure 3.3 and Figure 3.4. These results show that the quality of templates for α'_0 (and the omitted α_1) were very good, as most of the corresponding success rates are higher than 0.8, while the values of guessing entropy are below 3.0. Meanwhile, the templates for β_0 were not as effective as those for the other two target intermediate states but still acceptable.

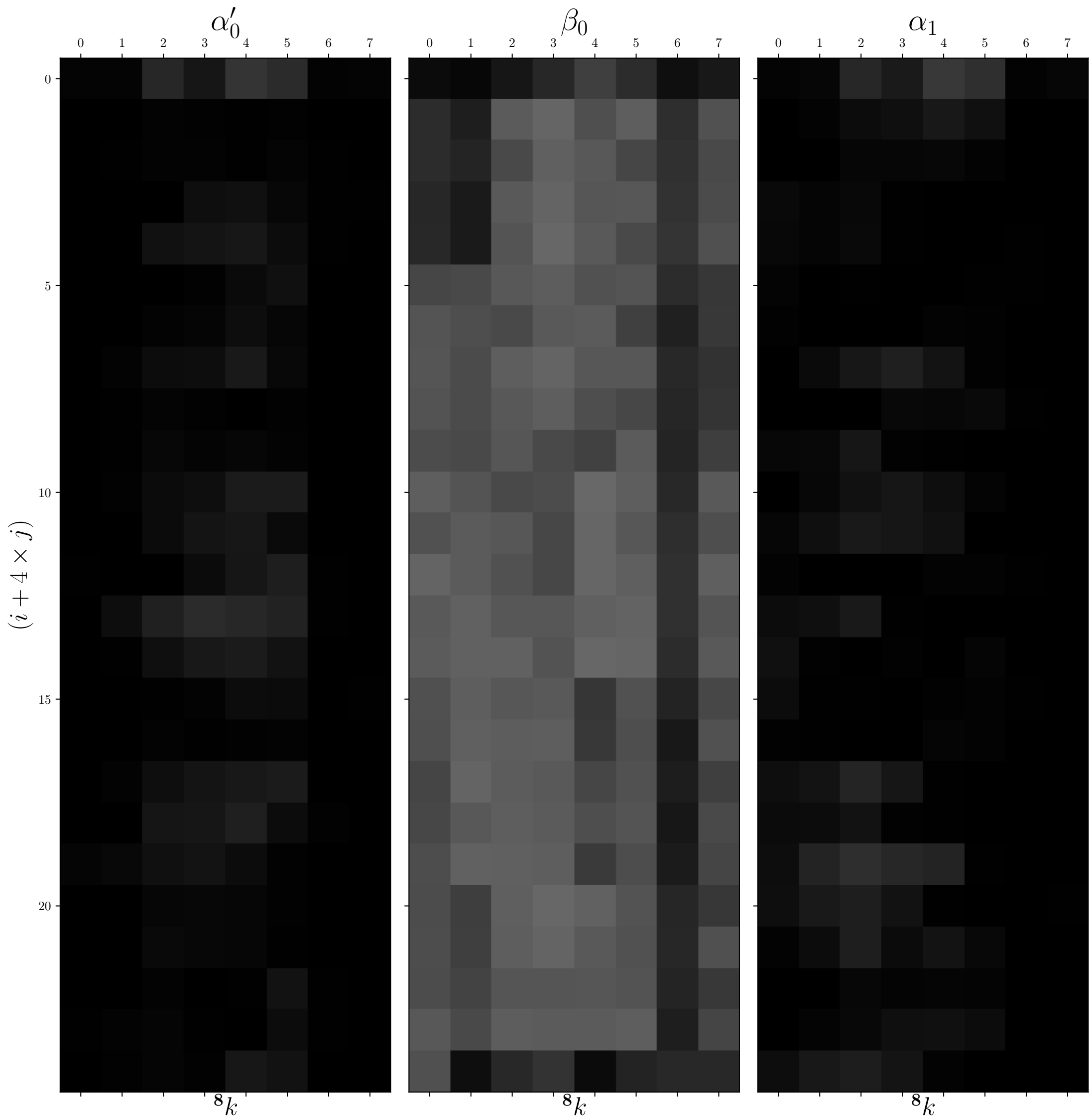


Figure 3.4: Logarithmic guessing entropy values on my target states α_0', β_0, and α_1, where the lower the value, the blacker is a block.

3.3 Searching the correct intermediate states

Now I provide a detailed description of my three-layer enumeration.

3.3.1 Layer 1: generating tables for byte rows

Between the intermediate states α_0' and β_0 is the step χ, which can be calculated within a byte row without any influence from other byte rows. This 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. I use the first byte row ($j = 0, {}^8k = 0$) here to demonstrate the enumeration procedure in this layer.

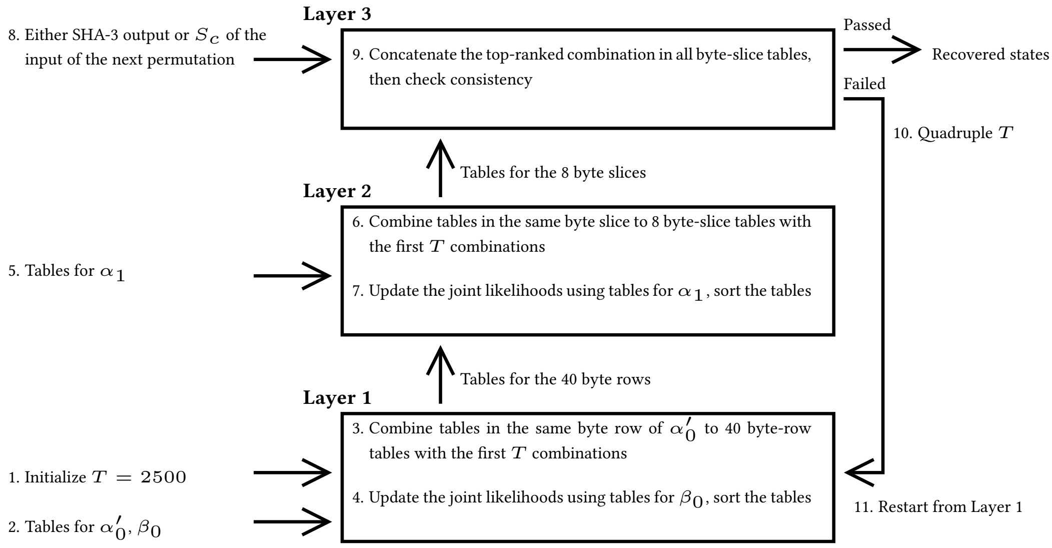


Figure 3.5: My three-layer procedure to search the full α'_0 state from 600 ranking tables.

First, we initialize $T = 2500$, which is the number of combinations that we want to collect in the resulting byte-row ranking table. For state α'_0, I use the five variables $A'_0, A'_1, A'_2, A'_3, A'_4$ to represent the values of the first byte row, i.e., $\alpha'_0[0,0,0]^{\mathbf{8}}$, $\alpha'_0[1,0,0]^{\mathbf{8}}$, $\alpha'_0[2,0,0]^{\mathbf{8}}$, $\alpha'_0[3,0,0]^{\mathbf{8}}$, $\alpha'_0[4,0,0]^{\mathbf{8}}$. As likelihood functions I use the multivariate Gaussian probability-density values provided by the template attack: $\mathcal{L}(\alpha'_0[0,0,0]^{\mathbf{8}} = A'_0) = f_{\alpha'_0[0,0,0]^{\mathbf{8}}}(\mathbf{x}_{\text{proj}}|\hat{\mathbf{x}}_{A'_0,\text{proj}}, \mathbf{S}_{\text{proj}})$, etc. With the ranking tables of these five bytes, we can use the secret enumeration procedure to search the first T combinations of a byte row, sorting with the descending order of their joint likelihood. Assuming independence, the first estimate of their joint likelihood is

$$\mathcal{L}_{\text{row}}(\alpha'_0[\cdot,0,0]^{\mathbf{8}} = (A'_0, A'_1, A'_2, A'_3, A'_4)) := \prod_{i=0}^{4} \mathcal{L}(\alpha'_0[i,0,0]^{\mathbf{8}} = A'_i).$$

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

For these T combinations, we can 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 joint 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 these values, that is

$$\mathcal{L}^{\text{new}}_{\text{row}}(\alpha'_0[\cdot,0,0]^{\mathbf{8}} = (A'_0, A'_1, A'_2, A'_3, A'_4)) := \prod_{i=0}^{4} \mathcal{L}(\alpha'_0[i,0,0]^{\mathbf{8}} = A'_i)\mathcal{L}(\beta_0[i,0,0]^{\mathbf{8}} = B_i).$$

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

3.3.2 Layer 2: generating tables for byte slices

Like Layer 1, a similar procedure can apply here to combine five byte-row ranking tables into a byte-slice ranking table. I use here the first byte slice (${}^{\mathbf{8}}k = 0$) to demonstrate this. Let R'_j represent a byte row value of state $\alpha'_0[\cdot, j, 0]^{\mathbf{8}}$ 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 can use the ranking tables of the five byte rows again for a secret enumeration to search the first T, which is the same as in Layer 1, combinations in descending order of joint likelihood of a byte slice. The initial joint likelihood estimate for a byte slice is

$$\mathcal{L}_{\text{slice}}(\alpha'_0[\cdot,\cdot,0]^{\mathbf{8}} = (R'_0, R'_1, R'_2, R'_3, R'_4)) := \prod_{j=0}^{4} \mathcal{L}_{\text{row}}^{\text{new}}(\alpha'_0[\cdot, j, 0]^{\mathbf{8}} = R'_j) = \prod_{j=0}^{4}\prod_{i=0}^{4} \mathcal{L}(\alpha'_0[i,j,0]^{\mathbf{8}} = A'_{i,j})\mathcal{L}(\beta_0[i,j,0]^{\mathbf{8}} = B_{i,j}).$$

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

$$(R_0, R_1, R_2, R_3, R_4) = \theta^*(\iota^*_{0,{}^{\mathbf{8}}k}(\chi(R'_0), \chi(R'_1), \chi(R'_2), \chi(R'_3), \chi(R'_4)), \tau).$$

where $\iota^*_{0,{}^{\mathbf{8}}k}$ represents ι in the round $\Omega = 0$ with input and output truncated to byte slice ${}^{\mathbf{8}}k$, and $\theta^*(\dots, \tau)$ is θ applied to just one byte slice, where $\tau \in \{0,1\}^5$ is 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, it lacks 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 byte slice $\alpha_1[\cdot,\cdot,0]^{\mathbf{8}}$, which is

$$\max_{\tau\in\{0,1\}^5} \prod_{j=0}^{4}\prod_{i=0}^{4} \mathcal{L}(\alpha_1[i,j,0]^{\mathbf{8}} = A_{i,j}).$$

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

$$\mathcal{L}_{\text{slice}}^{\text{new}}(\alpha'_0[\cdot,\cdot,0]^{\mathbf{8}} = (R'_0, R'_1, R'_2, R'_3, R'_4)) := \prod_{j=0}^{4}\prod_{i=0}^{4} \mathcal{L}(\alpha'_0[i,j,0]^{\mathbf{8}} = A'_{i,j})\mathcal{L}(\beta_0[i,j,0]^{\mathbf{8}} = B_{i,j})\mathcal{L}(\alpha_1[i,j,0]^{\mathbf{8}} = A_{i,j}).$$

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.

3.3.3 Layer 3: consistency checking

In Layer 3, we can again use a secret enumeration to combine the top-T entries in the eight byte-slice rank tables from Layer 2 into a single top-T ranking table for the full 200-byte state of α_0'. For each enumerated result, we then calculate the corresponding input and output of the KECCAK-$f[1600]$ permutation to check the consistency of these with any available SHA3-d data, as described in Section 3.1 and Figure 3.1. If all these T combinations fail this consistency check, we can choose a larger T and restart the search from Layer 1 every time it fails until it hits the correct combination or terminates with other pre-set conditions.

In practice, however, I found that this was not necessary for the enumeration in Layer 3 in my experiments. Here the high quality of information from templates can ensure if a byte-slice ranking table contained the correct combination, it should be already ranked top. Otherwise, most likely the correct candidate had been already missing in the tables produced by layers 1 or 2. Therefore, without an enumeration, Layer 3 in my experiments only concatenated the top-ranked combinations from all eight byte-slice tables together as the reconstruction results of intermediate state α_0'. For the failed cases, I quadrupled T each time to restart the search and gave up after still not finding a correct solution with $T = 640\,000$. This limit can of course be raised given sufficient computing resources. Figure 3.5 shows the complete procedure of my three-layer enumeration.

3.3.4 Results

SHA3-512 with only one KECCAK-$f[1600]$ invocation In my first attack trace set, each of the 1000 recorded SHA3-512 executions invokes KECCAK-$f[1600]$ only once to digest the input. In this case, it only needs to apply the template attack to obtain the 600 rank tables of intermediate bytes in that one KECCAK-$f[1600]$ invocation, apply the 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 SHA3-521 output and whether the last 1024 bits of the input are all zero. If both checks pass, the input of SHA3-512 can be reconstructed by removing the padding from the first 576 bits of the recovered KECCAK-$f[1600]$ input.

In these 1000 attack attempts, I successfully reconstructed the SHA3-512 input 999 times, while I failed to recover one remaining input even with $T = 640\,000$. The number of additional traces for which I 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 percentage	87.3%	95.0%	98.3%	99.4%	99.9%	100%
average CPU time [s]	9.38	41.14	180.59	902.95	4795.93	N/A
CPU time std. [s]	1.04	4.87	24.75	25.85	93.71	N/A

SHA3-512 with multiple Keccak-$f[1600]$ invocations When the input is over 72 bytes long, it takes multiple Keccak-$f[1600]$ invocations to absorb. There we need to use the templates to obtain the 600 rank tables of the three intermediates states in every invocation, and then start the three-layer search for each, from the last invocation to the first. We can verify the correctness and calculate the SHA3-512 input as described in Section 3.1.

The experiment on my second attack trace set demonstrated the case with four Keccak-$f[1600]$ invocations. In the 1000 attack attempts, I successfully reconstructed the SHA3-512 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 SHA3-512 with shorter input to be higher than with longer inputs[1], in these experiments the success rates were both too close to 1 to be distinguishable.

As I mentioned in Section 2.6.2, each trace covers only the first few rounds in one invocation of Keccak-$f[1600]$. Therefore, there were 4000 traces in this attack set. The following table shows the total number of successfully recovered invocations of Keccak-$f[1600]$ for each T value.

T	2500	10 000	40 000	160 000	640 000	failed
new traces recovered	3724	202	57	12	4	1
cumulative percentage	93.100%	98.150%	99.575%	99.875%	99.975%	100%
average CPU time [s]	8.42	38.67	181.55	951.05	5269.00	N/A
CPU time std. [s]	0.97	3.89	9.49	52.66	72.73	N/A

It appears that the success rate for the attack on this set was slightly higher than that with one invocation of Keccak-$f[1600]$. Recall that my profiling traces were also recorded from inputs with four invocations of Keccak-$f[1600]$, and therefore the attack on the input with four invocations would benefit from this similarity.

3.4 Belief propagation on Keccak-$f[1600]$

While I developed the attack above, a different approach for single-trace recovery on Keccak also appeared: Kannwischer et al. [89] used SASCA 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 of intermediate values in the Keccak-$f[1600]$ permutation. They suggested their SASCA approach may reach a higher success rate with a leakage model bearing more information than just Hamming weights.

[1]While expecting that the success probability for each Keccak-$f[1600]$ invocation shall be the same, the success rate of reconstructing the entire SHA3-512 input will drop with an increasing number of invocations, as the failure to recover the state of any Keccak-$f[1600]$ invocation means that two SHA3-512 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 $p^{\lceil \frac{L+1}{72} \rceil}$.

Although I had already achieved a high success rate to reconstruct SHA3-512 inputs with my strategy, which combines full-state template recovery with a three-layer enumeration, I was also curious about whether the success rate can be raised even further (or the computation can become more efficient) with a new combination of my templates and belief propagation. Therefore, I describe below how I modified Kannwischer et al.'s model such that it can apply the full-state information recovered by my templates for SHA3-512.

3.4.1 Bitwise model by Kannwischer et al.

Kannwischer et al. [89] 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, which is a common model of SHA3-$d(K\|M)$, 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 a 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

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

If we directly designed a constraint factor following this equation, it would connect to 12 variables. Instead, Kannwischer et al. [89, Fig. 1] separated it into three equations

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

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

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

where $\mathbf{C}$ and $\mathbf{D}$ are internal planes within step θ as described in Algorithm 1, Section 2.4.1. They then use these three substeps of θ to build the constraint factors in their graph.[2]

For step χ, they suggest combining 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 the backward information transmission from β to α' nodes. Fig. 3.6 shows the resulting factor graph.

[2] $\beta'[i,j,k]$, $\mathbf{C}[i,k]$, $\mathbf{D}[i,k]$, $\alpha[i,j,k]$ here are equivalent to I, P, T, O, respectively in [89, Sec. 4.3].

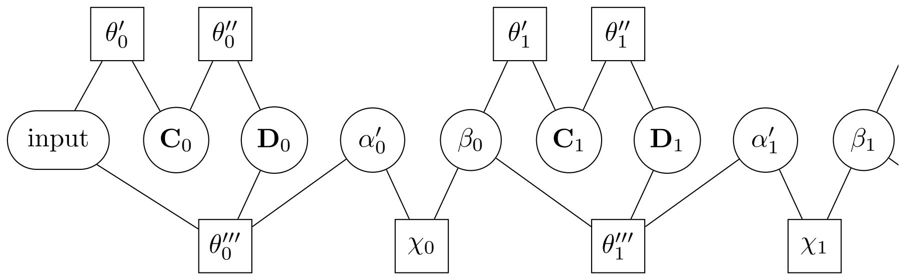


Figure 3.6: The loopy-BP graph structure for the Keccak-$f[1600]$ permutation, showing the node relations for the first two rounds. Variable nodes are in circles, and constraint factors are in squares. Observation factors are not shown here. Each state variable shown here actually represents 1600 or 320 single-bit variable nodes, respectively.

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

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., $\xi \in \mathbb{Z}_{256}$).

3.4.2 Apply the bitwise model with full-state information

Modifications of the methodology The first of my modifications is to drop the clustering technique and operate instead with single-bit variables (i.e., $\xi \in \mathbb{Z}_2$). After my templates generate the per-fragment probability tables for the selected intermediate states, I marginalize these tables to eight binary tables of their member bits and then use a bitwise loopy-BP directly as the belief propagation procedure. Kannwischer et al. [89, Sec. 4.1] stated 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 this is not necessarily the case in my experiments: these templates are based on the stochastic model $\mathcal{F}_9$ [50] (see Section 2.1.2), where bits in the target bytes are viewed as independent binary variables. With the assumption of mutual independence, this model has already, to some extent, given up exploiting information from a joint distribution across bits.

Besides that main difference, I made some other changes compared to Kannwischer et al.'s design. Firstly, instead of their *layer-after-layer* message updating, in a single iteration, my belief-propagation implementation simply updates all $r_{m \to n}$ messages in the factor graphs before updating all $q_{n \to m}$ messages, which is more consistent with the method described by MacKay [101]. Their *layer* is one full 1600-bit intermediate state, such as α'_Ω or β_Ω, which

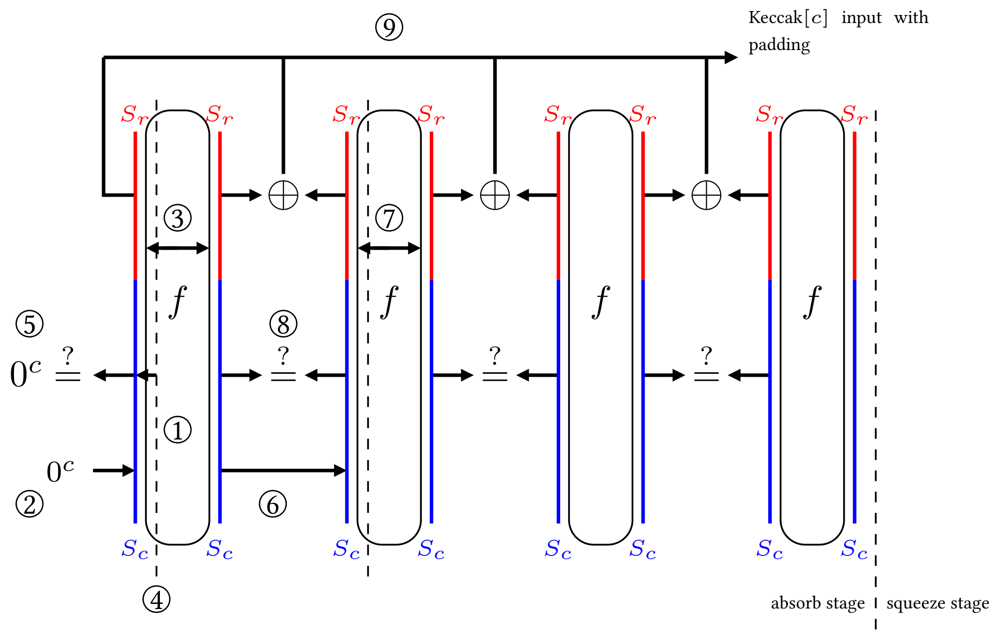


Figure 3.7: The procedure to reconstruct input (and output) of sponge function Keccak $[c]$ by template attack: ① generate the probability tables for the target intermediate states in the first Keccak-$f[1600]$ permutation and marginalize them to binary tables; ② add the observation factor for the S_c 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 the consistency of the input S_c; ⑥ add the observation-factor for the S_c of the input, where the bits match the S_c 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 S_r of consecutive invocations and concatenate these XOR results to find the padded Keccak $[c]$ input.

bears full information that can calculate the Keccak-$f[1600]$ input and output. Their layer-after-layer updating starts updating the messages from the front layers through the latter ones until it reaches the end of the factor graph, and then proceeds backward, the other way round. Secondly, I terminate the loopy-BP algorithm after either reaching a steady state or a maximum iteration count of 200. I found that checking the total entropy value helped little and dropped this termination check. Finally, they use the *damping* technique, which was introduced by Pretti [124] in 2005, to prevent possible oscillations when propagating information in their factor graph. After not finding consistent improvements when trying different damping rates (see Section 4.4.6), I present here only the results without any damping.

Recall that I did not acquire any side-channel observations for the input. Instead, its observation factors set the S_c 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. Without any prior information, variables of the input S_r connect to observation factors initialized with uniform distribution, which is mathematically equivalent to connecting to no observation factors because the uniform distribution does not provide any information.

Extend to multiple invocations Since Kannwischer et al.'s work only simulated processing short secrets that can be absorbed within one invocation of KECCAK-$f[1600]$ permutation, they did not report any attempts targeting more than one invocation. Therefore, after replacing the three-layer enumeration with the belief-propagation procedure, I slightly modify my procedure to recover the full padded input of a KECCAK sponge function as described in Fig. 3.7.

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

For a sponge function with more than one invocation, we can repeat what we have done for the first invocation, but now the S_c of the input is verified instead against the S_c of the output from 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 S_r from inputs and outputs, as described in [90] and Fig. 3.7.

3.4.3 Experiments

I use the same testing data sets as in Section 3.3.4, consisting of 1000 SHA3-512 inputs with one invocation of KECCAK-$f[1600]$ and 1000 of those with four invocations, applying three scenarios for each set:

① bitwise probability tables from the templates for states α'_0, β_0, α'_1,

② bitwise probability tables from known input capacity parts in addition to ①,

③ bitwise probability tables from the templates for states β_1 in addition to ②.

The bitwise probability tables for ① can be directly calculated by reusing the templates for states α'_0, β_0, α_1. For the first two states, I just marginalized the byte tables from their templates. For bitwise tables for α'_1, I marginalized the byte tables calculated from templates of α_1 and then change their order by the transposition steps of ρ and π. We can build the bitwise

Table 3.2: Results of recovering the SHA3-512 inputs with one and four invocations.

Experiments	#Invocations	#Recover	#Iteration*			
			Median	Mean	σ	Max
①	1	977	23	24.964	9.804	156
	4	900	22	23.327	6.071	176
②	1	1000	32	31.330	3.246	40
	4	1000	32	32.374	2.227	45
③	1	1000	31	30.952	3.239	41
	4	1000	32	32.032	2.317	45

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

tables for known input capacity parts by simply assigning the probability of the correct candidate to 1 and the other to 0. The only new templates here are those for each byte in state β_1 so that we can marginalize their predictions into bitwise tables later used in the scenario ③.

Table 3.2 shows the number of successfully recovered inputs and related statistics on the number of iterations required for each of these experiments. Given the templates we already have, we reached very high success rates, which are over 90% (①), and even 100% once taking the known capacity parts into account (②). Although the newly profiled templates for β_1 did not significantly affect the results (③), these experiments demonstrated the potential of the belief propagation considering more intermediate states.

Meanwhile, I also recorded the execution CPU time (single-threaded), where the number of each experiment for recovering SHA3-512 inputs with one invocation is: ① 1.282 ± 0.340, ② 1.392 ± 0.101, and ③ 1.499 ± 0.113 seconds, respectively. These numbers are estimated by the average of the 1000 attack attempts and their confidence intervals with two standard deviations. We can see that the belief propagation method is more efficient than my three-layer enumeration (see Section 3.3.4 for the run time).

3.5 Discussion

Recall that the previous CPA attack by Luo et al. focus on *fixed* size of the 320-bit keys used in MAC-Keccak, and their HW-based CPA approach needed about 10 000 attack traces to achieve a success rate over 90% [66]. In contrast, my experiments demonstrated that it is practical to reconstruct the arbitrary-length inputs of an unprotected Keccak software implementation on an ATxmega256A3U 8-bit microcontroller using a single-trace full-state template attack, even where the templates fail to rank some correct bytes highest. We can correct such errors by either my three-layer enumeration or the belief propagation procedure modified from Kannwischer et al.'s design [89], which was originally designed to use only HW information.

When it comes to my three-layer enumeration, 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 B.4, 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. This method could be extended by also profiling 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 would be 23 combinations of intermediate states that attackers could target using this search method.

I obtained much mathematical knowledge about Keccak when developing my three-layer enumeration. However, considering that we can benefit from using belief propagation with both the efficiency and flexibility to take information from template recovery on intermediate states in more rounds into account, as verified in my experiments, I decided to apply this technique in the later attacks instead of my three-layer enumeration.

Chapter 4

Fragment template attack on a KECCAK 32-bit implementation

Having been encouraged by the results of both the work of Kannwischer et al. [89] and my approach introduced in Section 3.3, I then decided to target a more ambitious goal, namely to reconstruct the complete arbitrary-length input of SHA-3 or SHAKE functions implemented on the 32-bit device, the CW-Lite board [109], from a single trace. To achieve this, it is crucial to figure out how to practically build templates for a 32-bit bus that can obtain more information than just the Hamming weight of a 32-bit state.

Therefore, based on Choudary and Kuhn's LDA-based template-recovery method, I introduce my *fragment template attack*, which cuts a 32-bit word into *fragments* and independently builds templates for such smaller pieces of the original 32-bit word.

4.1 Fragment template attack

If we were to directly apply an LDA-based stochastic-model template [49] on each intermediate 32-bit word, we first would use multiple 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 even on a single target 32-bit word, not to mention that we may face hundreds or even thousands of target 32-bit intermediate values for attacks on cryptographic algorithms such as KECCAK.

Therefore, I instead separate an intermediate word into fragments, here four bytes, and independently build templates for each. I expect 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 $\mathbf{W}_f$, 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 $\mathbf{W}$ would be

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

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

$$\bar{\mathbf{x}}_v = \sum_{\ell=0}^{31} (v[\ell] \cdot \mathbf{c}_\ell) + \mathbf{c}_{32}, \tag{4.1}$$

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

Instead, my LDA procedure takes the same training trace set but profiles the template with only eight bits at a time. Here we can start from splitting each word value $v \in \mathbb{Z}_{2^{32}}$ into four byte fragments $v \mapsto (F_0(v), \ldots, F_3(v))$ with $F_f(v) = \sum_{\ell=0}^{7} v[8f+\ell] \cdot 2^\ell$. Let $V_{f,b} = \{v \mid F_f(v) = b\}$ be the set of all 32-bit values where fragment number f has value b. For each f, we can apply the $\mathcal{F}_9$ stochastic model to obtain the 256 expected trace vectors

$$\bar{\mathbf{x}}_{f,b} = \sum_{\ell=0}^{7} b[\ell] \cdot \mathbf{c}_{f,\ell} + \mathbf{c}_{f,8}, \tag{4.2}$$

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 $\mathbf{W}_f$:

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

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 $\mathbf{W}_f$. In other words, they are considered to be switching noise in this model.

After projecting the profiling and attack traces to the new m'-dimensional subspace (m' is determined by the criterion introduced in Section 2.1.3) via these two matrices, 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 $\mathbf{c}_\ell$.

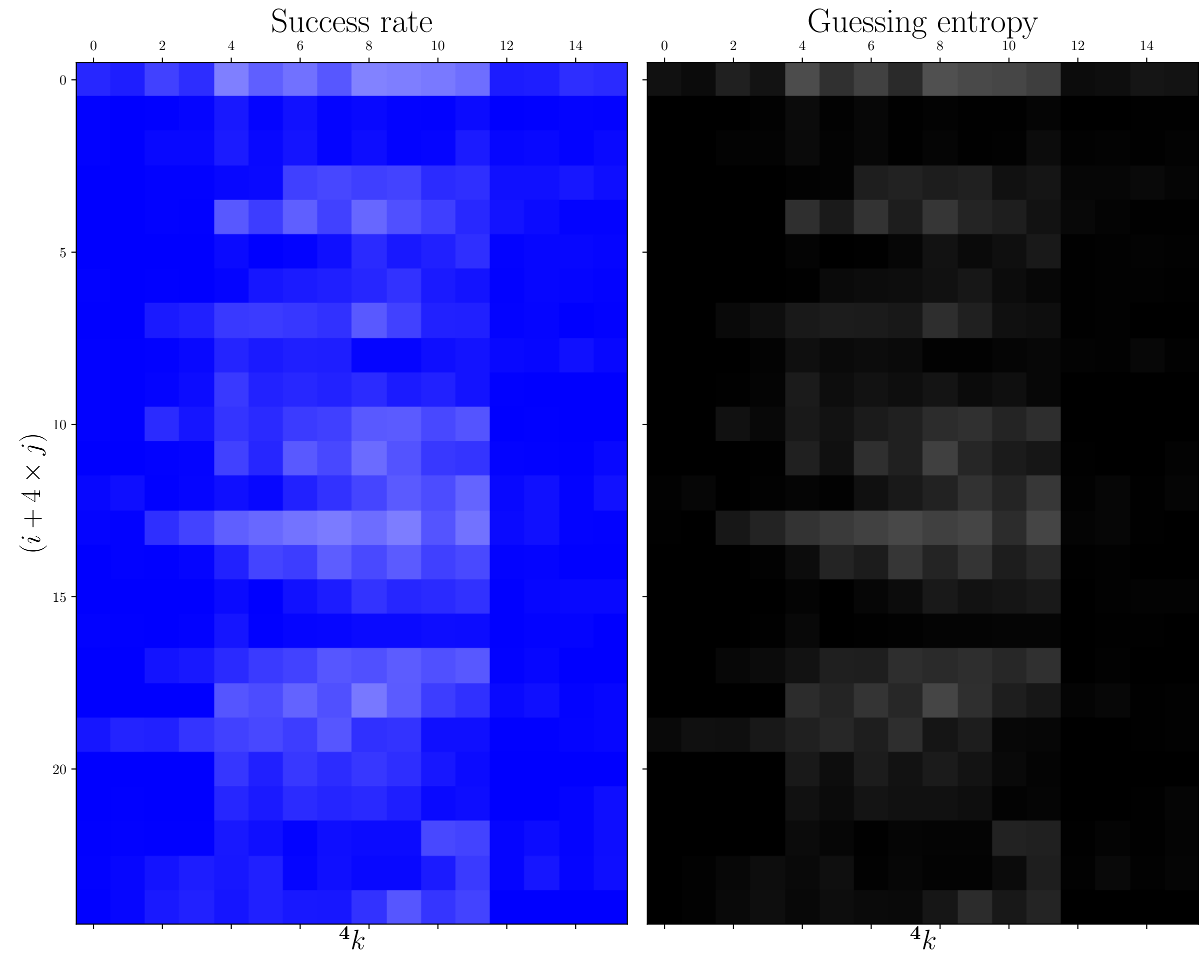


Figure 4.1: Success rates and logarithmic guessing entropy evaluated by the nibble templates on my target state α'_0. See Table B.6 and Table B.7 in Appendix B.3

4.2 Nibble templates of Keccak on the 8-bit device

Before building fragment templates for 32-bit words, I first started to build fragment templates for my old SHA-3 data set recorded from the 8-bit device as a feasibility test. Given the same profiling traces and evaluation traces, I built the 400 templates for nibbles in state α'_0, β_0, α_1 and β_1, where each byte in these states are separated into two nibbles. Figure 4.1 presents the success rates (SR) and the guessing entropy (GE) of recovering nibbles in α'_0.

Because the sizes of the targets are different, it is a little bit difficult to directly compare these results with the data in Table B.4 and Table B.5 in Chapter 3. Therefore, I applied the enumeration algorithm to calculate the ranking of the correct candidate of an intermediate byte given the ranking tables predicted by its fragment templates (the two nibble templates) and then calculated the success rate and the guessing entropy so that they can be compared with the results directly from the byte templates. Meanwhile, I also marginalized the probability table from each byte template into two probability tables for its low and high nibble, respectively, to calculate the success rate and guessing entropy, so they can be compared with the results directly calculated from the nibble (fragment) templates. I included only the results of

Table 4.1: Comparison of the templates for full bytes and the fragment templates for the two nibbles in the first lane ($i = 0$, $j = 0$) of state α'_0.

${}^{8}k$		0	1	2	3	4	5	6	7
byte templates	SR	0.924	0.924	0.598	0.749	0.485	0.542	0.946	0.931
	GE	1.095	1.109	2.336	1.616	3.215	2.592	1.074	1.096
enumerated with two nib. templates	SR	0.798	0.673	0.371	0.402	0.283	0.358	0.824	0.712
	GE	1.600	2.435	6.851	5.751	10.588	7.837	1.647	1.961
${}^{4}k = 2 \times {}^{8}k$		0	2	4	6	8	10	12	14
low nibble (marginalized)	SR	0.925	0.925	0.654	0.787	0.578	0.623	0.948	0.936
	GE	1.091	1.099	1.773	1.418	1.964	1.746	1.065	1.083
low nibble (nib. templates)	SR	0.847	0.743	0.499	0.553	0.489	0.524	0.890	0.819
	GE	1.207	1.418	2.296	2.064	2.392	2.148	1.143	1.258
${}^{4}k = 2 \times {}^{8}k + 1$		1	3	5	7	9	11	13	15
high nibble (marginalized)	SR	0.937	0.943	0.674	0.803	0.580	0.625	0.946	0.939
	GE	1.065	1.067	1.589	1.322	2.034	1.777	1.073	1.071
high nibble (nib. templates)	SR	0.882	0.820	0.626	0.658	0.505	0.568	0.881	0.833
	GE	1.129	1.238	1.687	1.587	2.237	1.964	1.165	1.237

Table 4.2: Results of recovering the SHA3-512 inputs with nibble templates.

Experiments	#Invocations	#Recover	#Iteration*			
			Median	Mean	σ	Max
①	1	914	28	32.313	15.713	188
	4	628	27	30.270	11.364	197
②	1	1000	34	33.622	2.755	44
	4	1000	34	34.248	2.337	45
③	1	1000	34	33.628	2.823	46
	4	1000	34	34.275	2.377	47

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

the first lane, where $i = 0$ and $j = 0$, in Table 4.1. The results indicate that the fragment template technique works on 8-bit words, as these templates provided us with satisfactory success rates and guessing entropy, but the quality of the fragment templates looks lower compared to those built from full intermediate bytes.

Table 4.2 shows the results from when I repeated the experiments of the bitwise belief propagation, given the bitwise probability tables being marginalized from the table predicted by the nibble templates. We can see that the results in ① (only including tables of state α'_0, β_0, and

α_1) are significantly worse than the previous results with byte templates. However, once we take more information into account, such as in cases ② and ③, the difference becomes not so significant. They can also achieve a 100% success rate, however with more iterations.

4.3 Byte templates of a stream cipher on a 32-bit device

After having shown that it is possible to apply nibble fragments to build templates for traces from the 8-bit device, I started a fragment template attack on a toy stream cipher running on a 32-bit device.

4.3.1 Target setting and trace recording

The following experiments target the processor STM32F303RCT7 on the CW-Lite 32-bit device (See Section 2.6.1). I programmed a small 64-bit stream cipher, including a 64-bit key (K), plaintext (P), and ciphertext (C), onto the CW-Lite 32-bit device. In this device, these values are stored in two 32-bit registers. In other words, the first four bytes of the key, which are referred to as $K_0\|K_1\|K_2\|K_3$, are in one register, while the last four bytes, $K_4\|K_5\|K_6\|K_7$, are in another, and the same also applies to the plaintext and the ciphertext. I 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).

I recorded traces while the device executed the encryption of the stream cipher, which is simply the XOR step $C = K \oplus P$. At 2.5 GS/s, each 20,000-sample trace recorded covers 40 clock cycles, and I categorized these traces into the sets introduced in Section 2.6.2. Since this experiment mainly focused on template profiling rather than an attack on a specific cryptographic algorithm, I did not record traces for the attack set.

4.3.2 8-bit fragment template profiling

Before any template-profiling experiments, I first used the $16\,000$ traces in the detection set to calculate the R^2 value of each fragment byte and each sample on the traces. Similar to my previous experiments, the R^2 values were evaluated with the $\mathcal{F}_9$ model. Figure 4.2 shows the R^2 values of four fragment bytes of the same words compared with the reference trace. We can see that the R^2 values for the fragments byte from the same 32-bit word are closely aligned.

Later, I recorded $64\,000$ traces for template profiling and $1\,000$ for template quality evaluation. Based on the experience from previous experiments, I would not directly use the raw data to build templates. Instead, I chose some different rates to resample the raw traces. Given a resampling rate, c, a new sample will be the summation of c consecutive samples from the raw traces. For example, if $c = 5$, there will be 100 points per clock cycle (100 PPC) in the

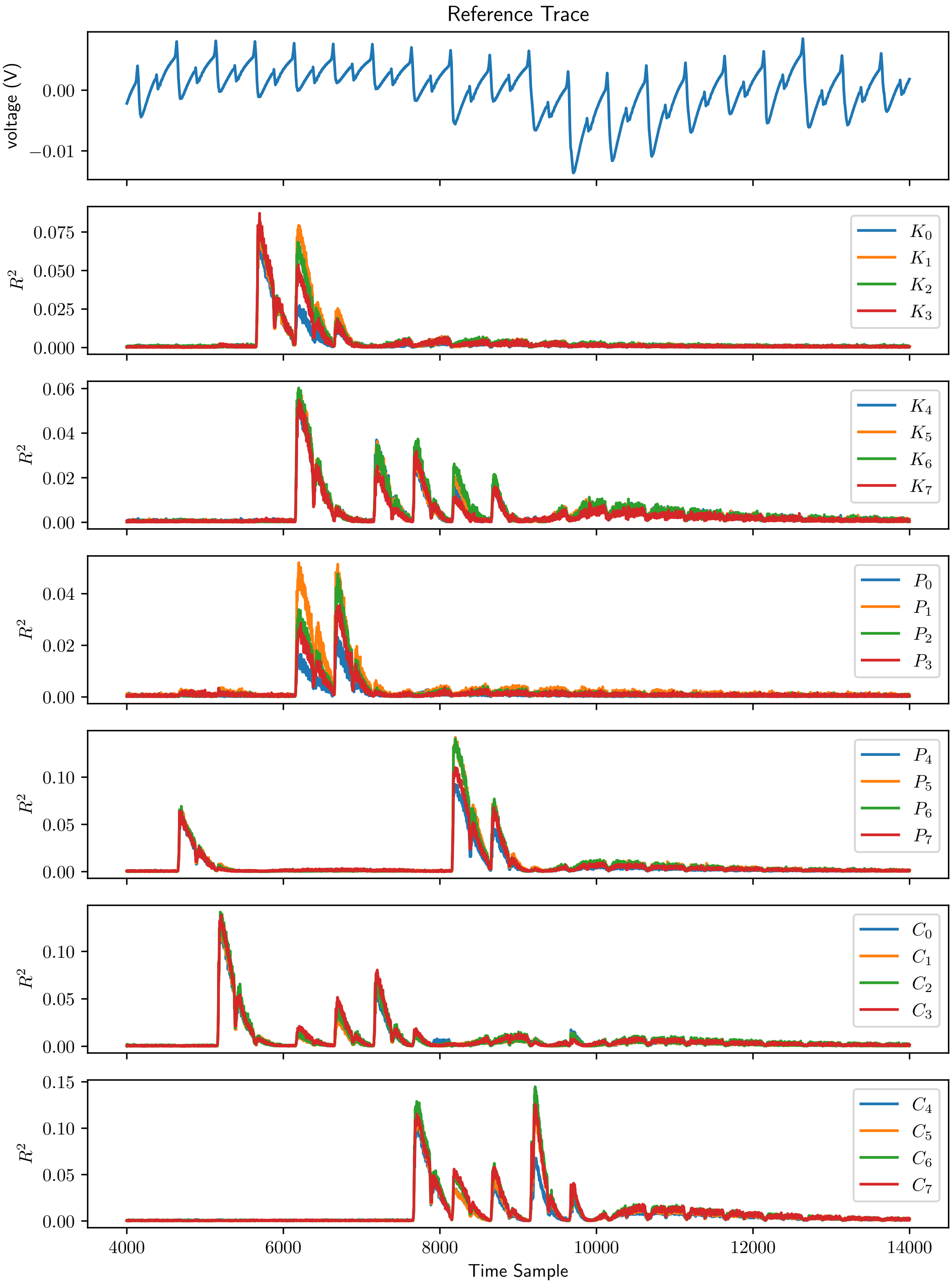


Figure 4.2: A part of the reference trace and the corresponding R^2 values for fragment bytes in target 32-bit words.

Table 4.3: Guessing entropy of the byte fragment templates

byte		K_0	K_1	K_2	K_3	K_4	K_5	K_6	K_7
PPC	c								
125	4	107.527	110.627	111.298	107.411	100.971	107.309	103.904	101.977
100	5	109.083	106.758	108.875	105.562	99.646	106.684	101.739	98.863
50	10	102.622	104.868	107.024	102.670	93.823	102.981	96.551	95.286
25	20	99.100	101.782	100.966	99.087	92.568	101.215	95.064	90.556
20	25	98.334	100.116	100.763	97.525	93.122	101.341	95.346	90.584
10	50	97.524	98.101	99.913	97.565	92.384	96.824	94.172	87.687
5	100	96.887	97.269	100.273	98.453	91.294	96.196	93.493	86.860
4	125	97.804	97.721	101.847	99.069	91.092	96.425	93.478	87.233
byte		P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
PPC	c								
125	4	109.317	80.542	104.346	108.634	96.530	77.340	100.804	103.454
100	5	107.385	79.620	106.490	106.051	95.270	77.067	102.004	100.106
50	10	105.374	75.272	98.225	101.939	91.204	73.015	97.944	97.920
25	20	102.019	71.398	94.955	97.716	89.445	68.960	92.672	93.821
20	25	100.836	72.020	96.031	97.354	86.723	68.948	93.192	95.022
10	50	99.910	72.021	95.270	96.374	85.963	69.052	90.790	93.976
5	100	98.355	72.384	94.366	97.086	86.801	68.435	90.205	92.264
4	125	98.630	73.345	93.575	97.746	87.053	69.655	91.338	93.953
byte		C_0	C_1	C_2	C_3	C_4	C_5	C_6	C_7
PPC	c								
125	4	76.350	101.259	98.211	98.298	75.595	104.709	99.194	84.177
100	5	74.545	101.250	99.668	98.386	74.174	100.704	97.952	84.241
50	10	70.832	96.781	95.174	93.587	69.867	96.933	93.711	78.399
25	20	70.215	94.208	93.197	91.269	66.674	93.385	91.350	77.473
20	25	69.024	94.274	93.849	90.064	66.651	92.493	89.845	77.175
10	50	69.444	92.169	92.241	89.680	66.088	91.138	88.402	75.850
5	100	70.184	90.064	90.344	90.112	67.143	91.176	87.663	76.317
4	125	71.387	89.390	88.696	92.167	67.168	91.087	86.970	75.998

resampled traces since there are 500 PPC in the raw traces. I selected several resampling rates from 4 to 125 for my experiments, and then used these resampled traces to build templates.

Table 4.3 shows the guessing entropy achieved with these templates, where smaller values indicate that the template quality is higher. Success rates are not provided here because so far they are not very meaningful given these levels of guessing entropy. The results suggest that it achieves slightly lower guessing entropy values with templates built from traces resampled to about 5 to 20 PPC.

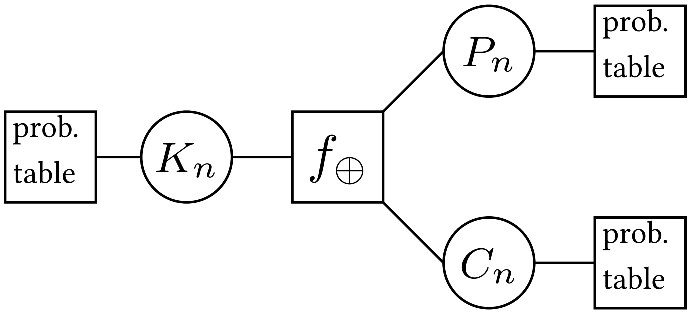


Figure 4.3: Factor graph for belief propagation of the stream cipher.

Table 4.4: Guessing entropy of the key bytes, after belief propagation with probability tables of plaintext and ciphertext bytes.

byte		K_0	K_1	K_2	K_3	K_4	K_5	K_6	K_7
PPC	c								
125	4	108.476	110.594	111.435	108.026	98.612	106.144	103.326	101.794
100	5	109.987	105.842	109.925	106.800	97.992	106.360	101.230	97.388
50	10	103.592	103.226	107.333	103.208	91.991	102.496	95.913	92.838
25	20	98.517	99.923	101.317	100.025	89.730	100.371	94.395	89.498
20	25	98.184	97.130	100.909	98.443	90.346	99.992	95.615	89.356
10	50	97.086	95.665	100.443	97.527	88.632	95.970	94.150	86.813
5	100	96.680	94.181	100.269	98.775	88.245	95.122	93.451	85.683
4	125	97.007	95.939	102.161	99.401	88.094	95.567	93.232	87.435

Table 4.5: Guessing entropy of the key bytes, after belief propagation with probability tables of plaintext bytes and known ciphertexts.

byte		K_0	K_1	K_2	K_3	K_4	K_5	K_6	K_7
PPC	c								
125	4	99.815	78.156	98.168	98.966	86.950	73.542	91.686	90.989
100	5	100.011	74.802	97.275	96.329	84.685	72.152	91.020	86.727
50	10	92.691	69.839	90.408	90.920	76.475	66.778	83.672	81.640
25	20	87.632	65.580	83.818	85.146	75.257	62.106	78.000	75.275
20	25	86.905	64.705	84.679	83.715	73.808	61.432	78.735	76.382
10	50	86.074	63.649	82.784	81.470	72.106	60.447	75.416	73.826
5	100	85.369	64.409	81.658	82.356	71.763	60.514	75.001	72.207
4	125	85.433	66.296	81.986	83.174	72.428	61.876	75.636	73.480

Apply belief propagation Since the target 32-bit words in this experiment are mathematically constrained by an XOR equation $C = P \oplus K$, I applied belief propagation according to a small factor graph covering these variables, which is shown in Figure 4.3. Because the structure of the factor graph is very simple, it is still practical to update the probability table of each key fragment directly. Therefore, I did not further marginalize these tables into bit tables for the belief propagation procedure.

Table 4.4 shows the guessing entropy of each key fragment once they are updated with the observed probability tables of plaintext fragments and ciphertext fragments through the belief propagation. Although the results are not significantly improved compared to the results before belief propagation in this case, we can also consider another more common situation, where the ciphertexts are known to the attacker. In this situation, the table of a ciphertext fragment contains only the probability of the correct candidate being equal to one and the others equal to zero. Table 4.5 shows the guessing entropy of each key fragment once they are updated with the observed probability tables of plaintext fragments and known ciphertexts through the belief propagation, where the values of guessing entropy are lower than those before belief propagation.

4.3.3 Templates for 16-bit fragments

I repeated the experiment by choosing the 16-bit fragment size to build templates. Since now these fragments are equal to two previous byte fragments being concatenated together, I use $K_0\|K_1$, $K_2\|K_3$, etc. to represent the new 16-bit fragments. Given the same 1000 testing traces, Table B.8 (See Appendix B.3 for all the tables of this subsection.) shows the guessing entropy values evaluated by the 16-bit templates. To better compare the results with the previous values evaluated by the byte templates, I marginalized the 16-bit probability tables into byte tables and then calculated their guessing entropy values, which are also provided in Table B.8. However, compared to the results evaluated from the previous byte templates, all the guessing entropy values are very similar, so we cannot tell whether these 16-bit templates are better.

For further investigating this issue, I also repeated applying belief propagation on these 16-bit tables in three different scenarios. The first one is to use the marginalized byte tables of keys, plaintexts, and ciphers, where Table B.9 shows the results of this experiment. Table B.10 shows the results of this experiment with the second scenario, which is to use the marginalized byte tables of keys and plaintexts with known ciphertexts. When it comes to the situation to use the 16-bit tables directly, I only chose the experiments using tables of keys and plaintexts with known ciphertexts as the third experiment scenario, for it can be calculated within an acceptable run time. The known values of ciphertexts ensure that only their correct candidates need to be considered in the procedure of belief propagation, while we need to consider all the 2^{16} candidates once we use the observed probability tables of ciphertexts. Table B.11 provides the guessing entropy values evaluated from the 16-bit tables after being updated by belief propagation, as well as the values evaluated from 8-bit tables marginalized from the updated

16-bit tables. It seems like the results were not significantly affected in these experiments on this small stream cipher, no matter whether we chose the 8-bit fragments or 16-bit fragments.

Considering these experiments are only based on small numbers of instructions (mostly XOR), I remained optimistic about using this technique for SHA-3 since an intermediate state will be involved in far more instructions.

4.4 Attacking a 32-bit Keccak implementation

At a high level, the attack consists of three main steps, similar to the previous procedure in Figure 3.7. Firstly, I split each 32-bit target word into a few fragments, build a set of templates targeting each fragment independently, and then use these profiled fragment templates to generate a probability table for every fragment in the words of the intermediate states that they target in an invocation of the Keccak-$f[1600]$ permutation. Secondly, I marginalize these probability tables for fragments into binary probability tables for each bit, and 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, attackers can find the complete padded input of the Keccak sponge function.

4.4.1 Keccak implementation and the target board

My experiments still target the 32-bit processor on the CW-Lite board, while the Keccak implementation is again based on the official reference C code [102]. The 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 target intermediate states as a sequence of fifty 32-bit words. I 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).

4.4.2 Trace recording

Similar to the previous setting in Section 4.3.1, the recording platform includes the NI PXIe-5160 [105] 10-bit oscilloscope and the NI PXIe-5423 [106] wave generator in the same PXIe chassis, as well as the connector for impedance matching and high-pass filtering.

I 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 covers the first four complete rounds of Keccak-$f[1600]$, and I recorded that for each invocation

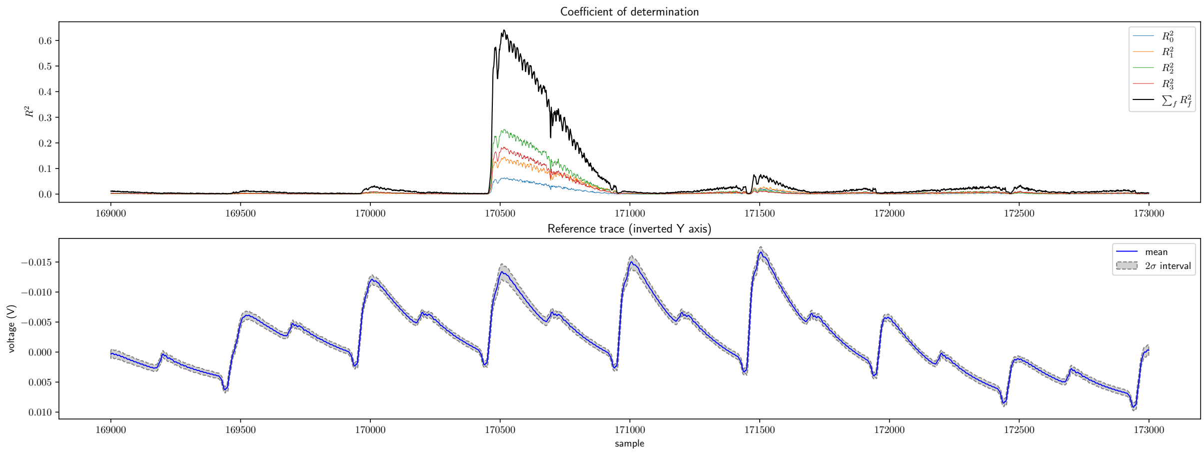


Figure 4.4: The corresponding four R_f^2 values of $(\alpha_0'[0,0,0],\dots,\alpha_0'[0,0,3])$ for each sample based on the 16 000 detection traces and their sum representing the detection results of the full 32-bit word (above), as well as the mean trace and the 2σ interval (below) at the same time samples.

of the permutation. For trigger accident detection (none were detected still), the Pearson correlation coefficient threshold here was 0.98 against an average trace of 1600 pre-recorded reference traces. 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 Section 4.4.4.

4.4.3 SASCA model building and evaluation

Interesting clock cycle detection Recall that in Section 3.2.2 (also in [90]), I used multiple linear regression to find the coefficient of determination (R^2) between the voltage-peak point in each clock cycle and the bit values of the target intermediate bytes. Using a threshold of $R^2 > 0.09$ to select the interesting clock cycle sets, I 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, I assumed that the four bytes in the same word will share the same sets. Therefore, a small change was applied to the previous 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. (4.1), which would need more traces to build, I instead estimated the correlation by adding the four R_f^2 values calculated from the independently built 8-bit model (4.2) 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. 4.4 shows a small part of the average trace for accident detection, covering the 32-bit word consisting of four member bytes $(\alpha_0'[0,0,0]^{\mathbf{8}},\dots,\alpha_0'[0,0,3]^{\mathbf{8}})$, along with the corresponding four R_f^2 values for each point, based on the 16 000 detection traces. Note that I also targeted the intermediate planes $\mathbf{C}_\Omega$ and $\mathbf{D}_\Omega$ in addition to the intermediate states α_Ω' and β_Ω compared to

Table 4.6: Numbers of interesting clock cycles selected in round $\Omega = 0$ with thresholds $\sum_f R_f^2 > 0.04$ (left) and $\sum_f R_f^2 > 0.01$ (right)

Lane$[i]$	C_0 first word	C_0 second word	D_0 first word	D_0 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 first word	α'_0 second word	β_0 first word	β_0 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$[i]$	C_0 first word	C_0 second word	D_0 first word	D_0 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 first word	α'_0 second word	β_0 first word	β_0 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

my previous experiments on the 8-bit device, so the belief-propagation procedure can also take their template-recovered information into account. Most of the data dependency is limited to one clock cycle in the time interval shown. We also can see that the R_f^2 values peak near the voltage peak, and we can use this to speed up the selection of samples from our 500 PPC data. Therefore, I summed 50 voltage samples around each voltage peak and calculated $\sum_f R_f^2$ for that to decide whether this entire clock cycle should be included. Table 4.6 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. I used the lower threshold $\sum_f R_f^2 > 0.01$. The SNR values of the points selected were in the range of 0.01 to 3.43.

Template profiling and validation As the experiments in Section 4.3 revealed, we do not need as many samples as 500 PPC to profile templates for attacks on the CW-Lite device, whereas Table 4.3 shows that using 5 to 20 PPC was good enough. I therefore decided to re-

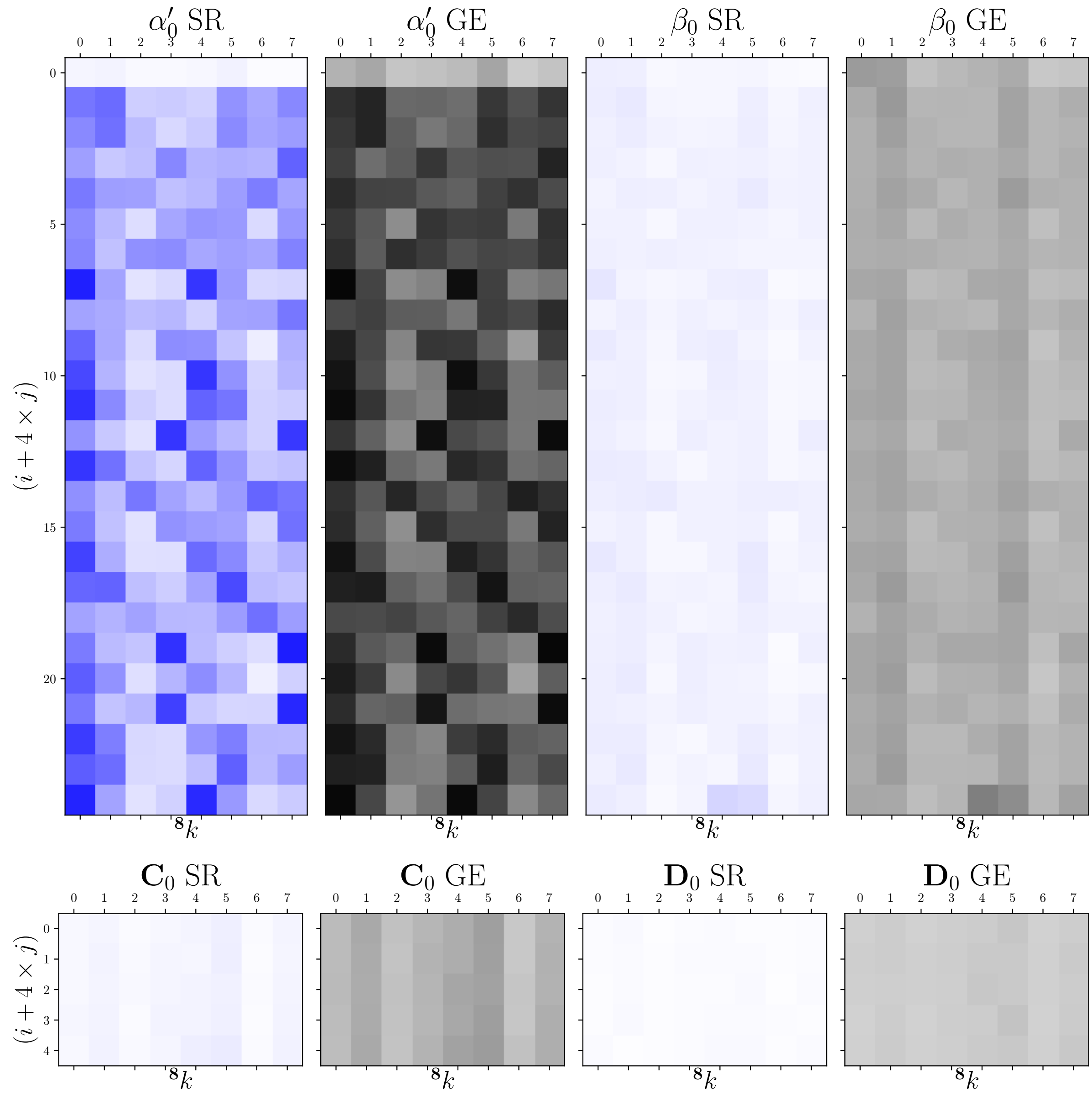


Figure 4.5: Success rates and logarithmic guessing entropy evaluated by the fragment templates on my target state α'_0, β_0, $\mathbf{C}_0$, and $\mathbf{D}_0$. See Table B.12 Table B.13, B.14, B.15 in Appendix B.4

sample 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. Given the numbers of detected interesting clock cycles shown in Table 4.6, such reduction results in profiling templates with at most 1660 samples per trace, which is still comfortably under my computing restriction introduced in Section 2.6.3.

Using the 1000 traces in the validation set, Figure 4.5 shows the resulting success rate and guessing entropy for α'_0, β_0, $\mathbf{C}_0$ and $\mathbf{D}_0$, respectively. The omitted data for other rounds look similar. The results for α'_0 and β_0 are not as good as the ones for the 8-bit processor in Sec-

Table 4.7: 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

tion 3.2.4, and possibly not good enough for the enumeration procedure there but suitable for belief propagation. Note that, similar to the results of 8-bit experiments in Table B.4 and Table B.5, 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 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 I use the marginal probabilities in the Loopy-BP network, Table 4.7 also shows the average number of correct bits in different intermediate states from the 1000 validation traces. Because the probability tables are binary after marginalization, I 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 these templates predicted the state α'_Ω more successfully in these four rounds than the other states.

Evaluation on different factor graphs We now evaluate how well the loopy-BP algorithm works when fed with marginalized binary probability tables from a single validation trace recorded from the 32-bit device, along with 1024 known bits in the capacity part of the input. Table 4.8 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. I provide results from factor graphs 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 it needs fewer iterations to reach a steady state with the four-round factor graph. Figure 4.6 (left) shows the percentage of successfully recovered traces (defined as all the bits of α'_0 being recovered correctly) out of the 1000 validation traces for these three factor graphs 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 factor graph takes more iterations to recover all validation traces correctly than the larger two.

Figure 4.6 (right) shows the percentage of successfully recovered traces out of 1000 validation traces when being provided with different numbers of known bits (not just 1024), to explore the situation when the size of the S_r (r *unknown* bits) and S_c (c *known* bits) of the permutation

Table 4.8: 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	51.730	4.374	71	1000	1000	1000	1000	1000	N/A	N/A	N/A	N/A

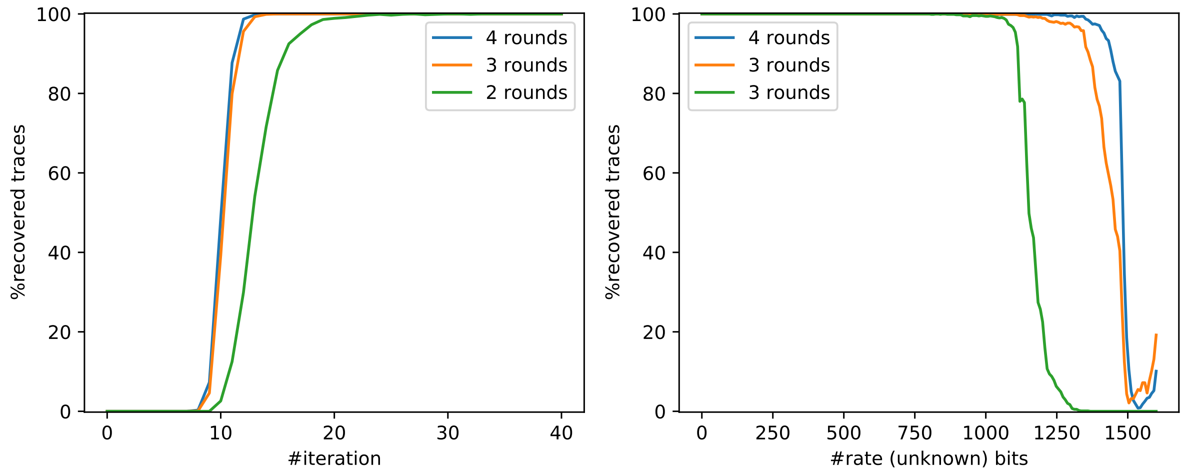


Figure 4.6: Percentage of successfully recovered traces for the different factor graphs (with different numbers of rounds observed), as a function of the number of loopy-BP iterations (left) and the number of unknown input bits (right).

input vary in different sponge functions. When the number of unknown bits increases beyond half of the full state, including up to the $1600 - 128 \times 2 = 1344$ unknown bits in SHAKE128, the four-round factor graph performs better than the others. As a result, I chose the four-round factor graph for the final version used in the belief propagation procedure.

4.4.4 Results for the SHA-3 and SHAKE functions

I 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 4.9 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. Recall that Kannwischer et al.'s results [89] 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. I 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.

Apart from SHA3-512, I 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 the belief propagation of Keccak-$f[1600]$ relies on the S_c of the

Table 4.9: Results of recovering the functions in the SHA-3 family with different numbers of invocations by the four-round factor graph.

Function	c	#Inv.	#Rec.	#Iteration*			
				Med.	Mean	σ	Max
SHA3-512	1024	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
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 the invocations successfully reaching a steady state are taken into account.

output from the previous invocation, the functions with a shorter S_c (c known bits) may encounter a lower success rate or may require more iterations to reach a steady state. Table 4.9 also shows some results of these five functions with inputs that can be absorbed by one or two invocations. We can see the results meet our expectation that the shorter the S_c, the lower the number of inputs can be successfully recovered, and the more iterations it took 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.

In addition to the four-round version, I have also tried these experiments with three-round and two-round factor graphs. Table B.17 and Table B.18 in Appendix B.4 show the results of recovering 1000 inputs with one and two invocations from the test traces of the six SHA-3 or SHAKE functions. It appears that the four-round belief propagation provides better results, suggesting that recording longer traces covering more rounds helps to push the success rate much closer to 1.

4.4.5 Experiments with 16-bit and nibble fragment templates

I also tried other choices of fragment size besides 4×8 bits: 2×16 bits, $11+11+10$ bits, 8×4 bits, 16×2 bits and 32×1 bit. As an example, the following data compare the performance of these different fragment sizes for the first bit in α'_0 after marginalization:

Fragments	2×16 bits	$11 + 11 + 10$ bits	4×8 bits	8×4 bits	16×2 bits	32×1 bit
Prob.	0.740549	0.752437	0.750506	0.752002	0.752274	0.751888
#Success	731	729	730	733	733	732
Max $\lvert\epsilon\rvert$	0.046705	0.026377	–	0.010587	0.013578	0.013906
Average $\lvert\epsilon\rvert$	0.008799	0.002809	–	0.001652	0.001872	0.002043

We can observe that fragment size had little influence on the 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.

However, recall the experiments on the 8-bit device: when I provided information from fewer templates (see the situation ① in Section 3.4.3), the results of the experiment targeting full bytes were better than the one targeting 4-bit fragments. This implies that although insignificant in single-bit prediction, the fragment size can still cause a difference in attacks to recover the full inputs of KECCAK sponge functions. As a result, I repeated the experiments on the 32-bit device with marginalized tables from 16-bit fragment templates to check whether these templates can achieve better success rates. Figure B.1 in Appendix B.4 depicts the results of recovering state α'_0 from the 1000 validation traces, as a function of the number of loopy-BP iterations (left) and the number of unknown input bits (right), while Table B.19, B.20, and B.21 in Appendix B.4 show the results for each SHA-3 or SHAKE function with templates for intermediate 16-bit fragments in the first four, three, and two rounds respectively. I found that some success rates increased especially in the case of SHA3-224 with two rounds, but there are no significant changes in the cases with three or four rounds.

Similarly, Table B.22, B.23, B.24 and Figure B.2 in Appendix B.4 demonstrate the results of using nibble fragment templates for attacks. Figure 4.7 plots the results with nibble, byte, and 16-bit fragment templates in the same subplots for the convenience of comparison, and we can see the differences are not so significant.

Therefore, although the larger size fragments may provide a (slightly) higher success rate for some cases in the experiments, I still suggest the fragment size should be chosen here to optimize computation time rather than optimize the success rates. In the attack stage, compared to templates with a smaller fragment size, a single 16-bit template recovery requires much longer run time (Table B.16). On the other hand, with 32 1-bit fragments, the profiling stage takes longer, as we need separate eigendecomposition of $\mathbf{W}_f^{-1}\mathbf{B}_f$ for each fragment in the LDA procedure, 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.

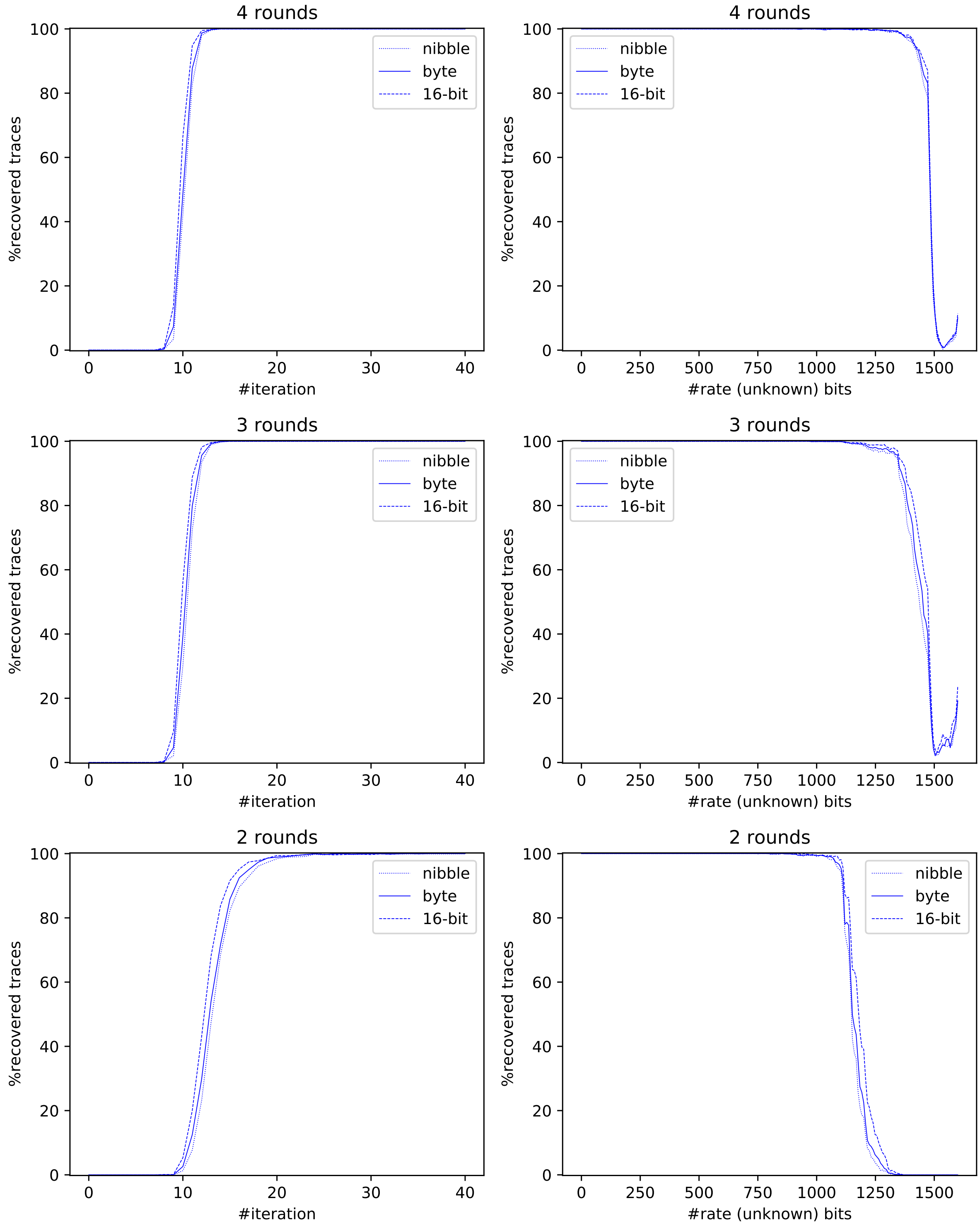


Figure 4.7: Percentage of successfully recovered traces with templates of different sizes, as a function of the number of loopy-BP iterations (left) and the number of unknown input bits (right).

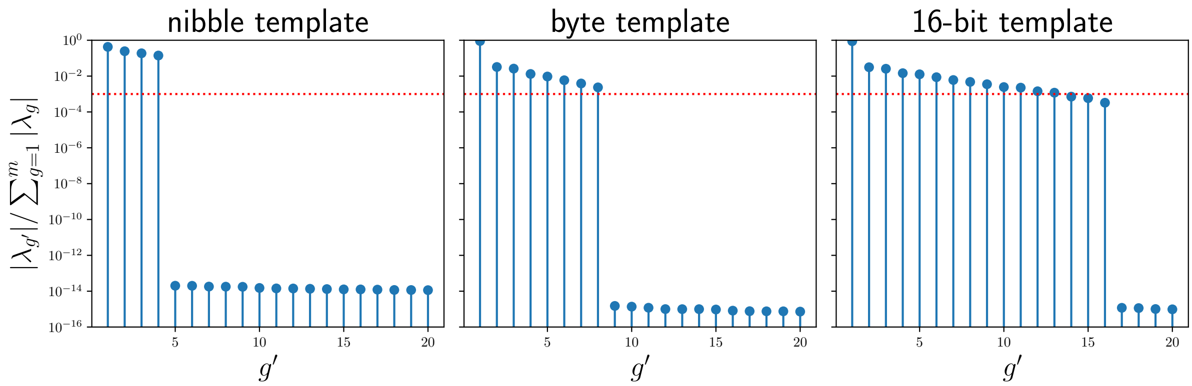


Figure 4.8: The ratio ($|\lambda_{g'}|/\sum_{g=1}^{m}|\lambda_g|$) of the largest 20 eigenvalues (for $1 \leq g' \leq 20$) versus the sum of all eigenvalues ($\sum_{g=1}^{m}|\lambda_g|$) of the $\mathbf{W}_f^{-1}\mathbf{B}_f$, for the last nibble, byte, and 16-bit fragments, respectively, of lane $L_{(4,3)}$ from α_0' (i.e., fragment $\alpha_0'[4,3,15]^{\mathbf{4}}$, $\alpha_0'[4,3,7]^{\mathbf{8}}$, and $\alpha_0'[4,3,3]^{\mathbf{16}}$).

A new eigenvector selection criterion Previously, I used the criterion introduced in Section 2.1.3 to determine the dimension m' of the projected traces after the LDA dimensionality reduction step, where the corresponding eigenvalue of a selected eigenvector needs to be larger than one-thousandth of the summation of all the eigenvalues. That criterion always selected $m' = 8$ eigenvectors with non-negligible eigenvalues when profiling templates for bytes, and $m' = 4$ for nibbles. When it came to the templates for 16-bit fragments, there were 16 eigenvectors selected by this criterion in nearly all cases. In a few exceptions, there were about 13 to 15 selected eigenvectors. This indicates that there could be the same number of independent binary variables as the number of non-negligible eigenvalues in this LDA-based multiple linear regression model, but my current *ad-hoc* criterion sometimes failed to select all the non-negligible eigenvectors when profiling templates for 16-bit fragments.

Therefore, I decided to plot the ratio ($|\lambda_{g'}|/\sum_{g=1}^{m}|\lambda_g|$) of the eigenvalues and visually check the numbers. Figure 4.8 shows the proportions of the largest 20 eigenvalues from the $\mathbf{W}_f^{-1}\mathbf{B}_f$ matrix of the last nibble, byte, and 16-bit fragments, respectively, of lane $L_{(4,3)}$ from α_0' as an example. As we can see, the fragment size matches the number of non-negligible eigenvalues when I used every single bit as an independent binary variable in the regression model. Meanwhile, the contribution of the remaining eigenvalues is below 10^{-12}. The red horizontal dotted line marks my previous threshold, 10^{-3}, for the selection criterion, and Figure 4.8 clearly shows that criterion may fail when profiling templates for 16-bit fragments.

As a result, I replaced my previous criterion with the new one, namely that we let m' be equal to the number of the independent variables, which is also the same as the fragment size, in my later experiments.

Table 4.10: Results of recovering the functions in the SHA-3 family with one invocation, with 2, 3, and 4-round factor graphs, given different damping rates.

Function	c	r	No damping ($\gamma = 1$)			$\gamma = 0.99$			$\gamma = 0.95$			$\gamma = 0.75$		
			2R	3R	4R	2R	3R	4R	2R	3R	4R	2R	3R	4R
SHA3-512	1024	576	1000	1000	1000	1000	1000	1000	1000	1000	1000	1000	1000	1000
SHA3-384	768	832	997	1000	1000	998	1000	1000	998	1000	1000	996	1000	1000
SHA3-256	512	1088	940	999	1000	947	997	1000	927	998	998	915	994	998
SHAKE256			867	999	997	945	997	998	926	997	998	913	998	998
SHA3-224	448	1152	419	992	1000	849	983	992	802	980	989	743	978	986
SHAKE128	256	1344	35	921	979	213	904	966	170	873	941	128	854	946

4.4.6 Damping in loopy belief propagation

In Kannwischer et al.'s paper, they suggest using a damping technique proposed by Pretti [124] when running the belief propagation following their factor graph for Keccak. In a loopy factor graph for belief propagation, sometimes the information flow will be stuck in an endless oscillation within a loop, which is more common when the loop is very small. In this case, the belief propagation may not terminate normally until the iteration reaches some cut-off limit. The damping technique was designed to prevent such endless oscillation. When applying it, we do not directly send the new messages over the edges connecting neighbor factor and variable nodes, but send the weighted averages of the new messages and the messages sent in the last iteration, such as

$$u = \gamma \times u_{\text{new}} + (1 - \gamma) \times u_{\text{prev.}},$$

where γ, ranging from 0 to 1, is defined as the damping rate. This is essentially low-pass filtering messages. Kannwischer et al. set $\gamma = 0.75$ in their simulations. We can set $\gamma = 1$ to disable the damping technique.

I tested this damping technique in my belief-propagation procedure for Keccak with different damping rates, and Table 4.10 shows very mixed results. For the cases where information is only collected from the first two rounds, it seems that damping rates very close to 1 will increase the success rate, while for cases with information from more rounds, the damping technique did not improve the belief propagation procedure. Therefore, I believe that the damping technique was not an essential part of my attack procedure. Perhaps the endless oscillation case did not occur frequently in my experiments.

4.5 Discussion

With the help of LDA-based dimensionality reduction, I successfully built fragment templates that generate separate probability tables for each byte in the 32-bit words of the targeted intermediate states, after the experience of separately building nibble templates for intermediate bytes from my old data recorded on the 8-bit device. In the case of the small stream cipher, it seems like the quality of templates is still not good enough for trace-single attacks since we can only collect information from a handful of instructions (or clock cycles) involved with the targets. In the case of Keccak, however, the quality of the fragment 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, the 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. My experiments 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. I 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 [125], which also use the Keccak$[256]$ or Keccak$[512]$ functions, except for different padding methods.

Here the fragment templates reconstruct full-state information stored in larger word sizes (such as 32 bits) than are practical with traditional template 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 or Keccak-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 the portability of templates across boards [96].

Chapter 5

Fragment template attack on ASCON-128 32-bit implementations

After attacking the KECCAK implementation on a ChipWhisperer-Lite 32-bit device, I started to implement a very similar attack strategy on an ASCON AEAD encryption implementation, to see if the combination of fragment template attack and belief propagation poses a more general risk than we had expected.

After a first glance at the ASCON AEAD structure, one issue I was concerned about is the use of the key four times in encryption or decryption. This may significantly enhance the chance for attackers to recover the key directly because repeated key use would lead to more interesting clock cycles for profiling templates with better quality. Therefore, my attack focused on recovering the key-related fragments with templates of their own, and with information collected from other intermediate states through belief propagation.

5.1 General experimental assumptions

As explained in the introduction of ASCON AEAD (Section 2.5.2), the input and output bitstrings involved in the encryption procedure are the key K, the nonce N, the associated data A, the plaintext P, the ciphertext C, and the tag T. I defined my attack as a profiled fixed-length known plaintext attack, only targeting the secret key K. In the profiling stage, the attacker can provide varying K, N, A, P, and can observe the corresponding C and T along with recorded power traces. In the attack stage, they can obtain values of N, A, P, C, T, and recorded power traces, to recover the secret key K.

I demonstrate the attack by targeting ASCON-128. Note that while ASCON allows arbitrary-length associated data and plaintexts, in this attack demonstration, I used empty associated data and 7-byte plaintexts, to keep the traces aligned and minimize their length when covering the entire encryption process. In other words, there will be only two invocations of permutation p^{12} involved in this case and Figure 5.1 depicts this special encryption procedure.

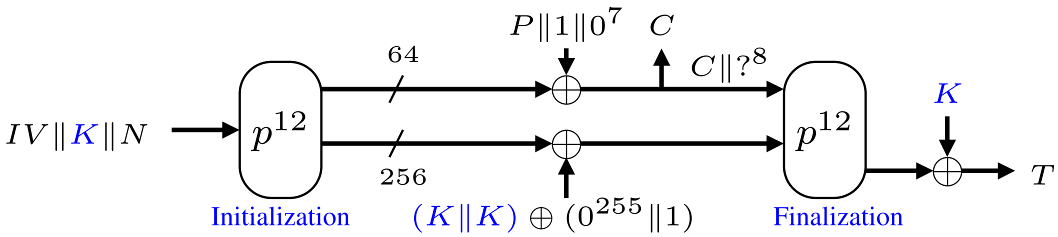


Figure 5.1: Ascon-128 with a short input.

Another good reason for the choice of such demonstration is that Ascon AEAD supports a so-called *leveled* implementation [126, 127], where we need to implement more side-channel countermeasures on Initialization and Finalization since the side-channel leakage from these two phases will pose a more serious threat to both the integrity and confidentiality of Ascon AEAD. This, however, implies that attackers can focus on these two phases for attacks.

5.2 Attack strategies

5.2.1 Attack strategy for single traces

The attack stage consists of three main steps: fragment template attack, belief propagation, and key enumeration. Some previous studies have used belief propagation and enumeration together [128] to better exploit the side-channel information, and Kannwischer et al. [89, Sec. 6.1] also indicated the possibility to integrate their attack on Keccak and key enumeration techniques to reach better results.

Fragment template attack Firstly, we need to build fragment templates for our target states. Previously, in the cases of SHA-3 and SHAKE, the traces we had recorded only covered the first four rounds of the Keccak-$f[1600]$ permutation. However, thanks to the simpler structure of Ascon, it is practical to record power traces covering the short full encryption procedure in Figure 5.1. Therefore, I built templates for target fragments of all the twelve α_Ω and thirteen β_Ω states of permutation p^{12} in both Initialization and Finalization.

In these 50 states $((12 + 13) \times 2)$, we do not need to build the templates for some special fragments. For the p^{12} in Initialization, the first 64 bits of the input are the initial vector IV, while the last 128 bits are the nonce N, and for the p^{12} in Finalization, the first 56 bits of the input is the ciphertext C. These values are public in my attacking scenario, so we only need to generate their probability tables by assigning the probability of the correct candidate to be 1 and others to be 0. Note that besides the IV and N values, the other two lanes (L_1 and L_2) of the Initialization input (β_{-1}) contain the key fragments, which are our main targets.

Similar to the previous loopy-BP procedure in the Keccak experiments, all the probability tables estimated by these templates will be marginalized into bitwise tables for later steps.

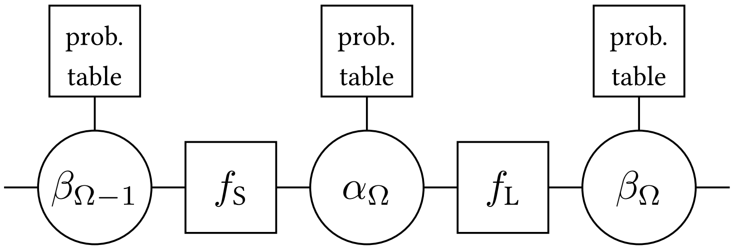


Figure 5.2: The factor graph for round Ω of the Ascon permutation. Similar to the case in Figure 3.6, state variables $\beta_{\Omega-1}$, α_Ω, and β_Ω shown here each represent 320 single-bit variable nodes, respectively.

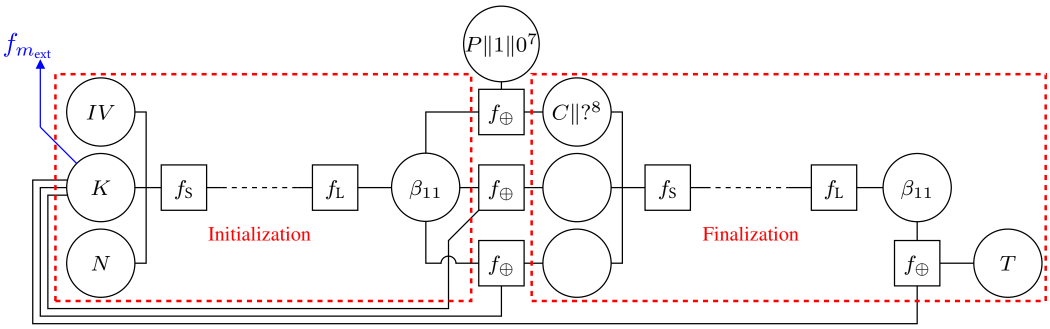


Figure 5.3: The factor graph at state level covering a full Ascon-128 encryption with null associated data and a seven-byte plaintext (observation factors omitted). The blue-part extension is for multi-trace attack, where this original single-trace graph is connected to the other single-trace graphs via the $f_{m_{\text{ext}}}$ constraint factor (See Section 5.2.2).

Belief propagation We can start by building the factor graph for a bitwise belief propagation procedure within a single round of the Ascon permutation, which is plotted in Figure 5.2. This small factor graph includes three variable states and their corresponding observed factors: $\beta_{\Omega-1}$, α_Ω, β_Ω, and two types of constraint factors, named f_{S} and f_{L}, connecting these variables.

Recall that $\alpha_\Omega = p_{\mathrm{S}} \circ p_{\mathrm{C}}(\beta_{\Omega-1})$, the constraint factors f_{S} should update the information following the mathematical relations in functions p_{C} (Constant Addition) and p_S (Substitution). Therefore, we can design these factors by connecting the five input bits and five output bits of p_{S} and use the S-box table as the mathematical constraint, just like how Kannwischer et al. designed their constraint factors for step χ in Keccak [89, Sec. 4.1]. As for p_{C}, we can just swap the probability values of the two candidates (0 and 1) when the value of the corresponding constant bit is 1, which is also like Kannwischer et al.'s design for step ι [89, Sec. 4.1]. For another type of constraint factor f_{L}, they update the information following the mathematical relations in the linear function p_{L}, which are all XOR functions with three inputs and one output in the bitwise level. For example, in the first lane, a mathematical constraint $\beta_\Omega[0,0] = \alpha_\Omega[0,0] \oplus \alpha_\Omega[0,64-19] \oplus \alpha_\Omega[0,64-28]$ holds because the linear function p_{L} updates the first lane by

$$L_0 \leftarrow L_0 \oplus \mathbf{Rot}(L_0, 19) \oplus \mathbf{Rot}(L_0, 28).$$

Once building the factor graph for the first round in the p^{12} permutation, we can simply repeat the same construction for the latter eleven rounds, only with different round constants, to cover all the states in an invocation of this permutation.

Considering the Ascon AEAD encryptions are procedures comprising a sequence of Ascon permutations with some XOR steps as well as additional input and output values, their factor graphs will be multiple single-invocation factor graphs connected by constraint factors with XOR functions and variables representing those additional inputs or outputs. Figure 5.3 shows the factor graph covering all the target states in my experiment. Here I define $f_\oplus$ as a type of constraint factor, where their only output $\mathbf{O}$ and multiple inputs $\mathbf{I}_1$ to $\mathbf{I}_N$ follow the constraint

$$\mathbf{O} = \bigoplus_{n=1}^{N} \mathbf{I}_n.$$

According to the encryption plotted in Figure 5.1, the input state (β_{-1}) of the p^{12} in Finalization will be the output state (β_{11}) of p^{12} in Initialization XORed with the following state: $P\|(0\text{x}80)\|K\|K'$, where K' is the key K with the least significant bit flipped. Therefore, via a constraint factor $f_\oplus$, the two variables respectively representing the bit in the first lane L_0 of the input state of Finalization and its counterpart in the output state of Initialization will be connected with the variable for the corresponding variable for the bit in the padded plaintext $P\|(0\text{x}80)$. Similarly, bits from the $L_1\|L_2$ of the two states will be connected to variables for the K via constraint factor $f_\oplus$, while those from $L_3\|L_4$ to variables for K as well with the probability swapping for information exchanged with the variable of the least significant bit. Likewise, we should connect the variables of the last 128 bits of the Finalization output, the key K, and the tag T together via $f_\oplus$ for the same reason.

Here the variables for key bits are connected to four different constraint factors, then forming a loopy structure. Therefore, a loopy-BP procedure applies and it will output probability tables for the bits in K once the procedure terminates.

Key enumeration Finally, we apply the key enumeration algorithm [53] to find the correct combination for the key, given the bit probability tables obtained from the belief propagation procedure. Since we know N, A, P, C, and T according to my assumptions in Section 5.1, we can simply check the correctness by encryption with these known data and the enumerated key fragment combination.

5.2.2 Attack strategy for traces from multiple encryptions

In the real world, a key for an encryption procedure may stay in use for a while. This means that we may change the other input values N, A, and P, but use the same key for several encryptions. As a result, I generalize my attack strategy to deal with such a situation. For building a factor graph covering public data and recorded power traces from multiple encryptions with the same secret key, one obvious way is to connect all the related constraint factors to the same variables for key bits. Considering each variable has already been connected to four constraint factors in the factor graph for single encryptions, it may be very messy to manage these variables if I were to build the graph this way. Instead, I introduce another external constraint factor $f_{m_{\text{ext}}}$, where the constraint is $K_1 = K_2 = \dots = K_N$, to connect each key variable from a separate factor graph for a single encryption. Figure 5.3 also demonstrates the extended factor graph when this external constraint factor is introduced to my belief propagation procedure.

Recall that for a connected variable x_n, a constraint factor f_m shall update the probability of a candidate $x_n = \xi$ in the message $r_{m \to n}$ by

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

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 the case of $f_{m_{\text{ext}}}(K_1 = K_2 = \dots = K_N)$, the only situation that makes this constraint hold is $v = \xi, \forall v \in \mathbf{v}$, which means the updated procedure will be reduced to

$$r_{m_{\text{ext}}\to n}(x_n = \xi) = \prod_{n' \in \mathcal{N}(m_{\text{ext}})\backslash n} q_{n'\to m_{\text{ext}}}(x_{n'} = \xi).$$

We can see that $f_{m_{\text{ext}}}$ updates the message more like how a variable node does in a factor graph, although it is a constraint factor by definition. After the belief propagation procedure terminates, we can use the updated message in this node instead of those in variables for keys from different encryption to evaluate the likelihood of each candidate ($Z_{m_{\text{ext}}}(x_{m_{\text{ext}}} = \xi)$) and the final probability table ($P_{m_{\text{ext}}}(x_{m_{\text{ext}}})$) by

$$Z_{m_{\text{ext}}}(x_{m_{\text{ext}}} = \xi) = \prod_{n=1}^{N} q_{n\to m_{\text{ext}}}(x_n = \xi), \quad P_{m_{\text{ext}}}(x_{m_{\text{ext}}} = \xi) = \frac{Z_{m_{\text{ext}}}(x_{m_{\text{ext}}} = \xi)}{\sum_{\xi'} Z_{m_{\text{ext}}}(x_{m_{\text{ext}}} = \xi')}.$$

Similar to the previous procedure for single encryptions, we can apply a key enumeration on these probability tables for the key fragments.

Note that this multi-trace approach is mathematically similar to the *Template-Based DPA Attack* by Oswald and Mangard [83, Sec. 2.3], but they described the part of considering the template-recovered information of the key from multiple traces with Bayes' theorem instead of the external constraint factor implemented in my factor graph.

5.2.3 Comparison against a very recent related study

Shortly before I submitted this thesis, a paper was published by Luo et al., on 17 Nov. 2022 [129], simulating a multi-trace template attack on Ascon AEAD with belief propagation. Therefore, it is noteworthy to point out the differences between their work and mine as follows.

Firstly, their attack is based on simulated noisy HW models for 8-bit devices, and therefore, their belief-propagation factor graph is designed for HWs of 8-bit values. Whereas, the side-channel information in my attack is the probability table from fragment templates for Ascon AEAD implemented on a 32-bit device, and I build a bitwise factor graph after marginalizing those probability tables.

Secondly, their attack focuses on Initialization, whereas (see Section 5.1) my attack takes both Initialization and Finalization into account, not only for the factor graph design but also for the interesting-point selection. When building templates for an 8-bit device, their approach may achieve sufficiently good templates by only considering the interesting clock cycles in Initialization, but I believe later clock cycles from Finalization also leak some information from the two XOR operations on the target key, helping attackers to build better fragment templates. Since it is more difficult to build templates for a 32-bit device, any little improvement can be critical for the success rates in the later belief propagation.

Thirdly, their attack has yet to be evaluated with key enumeration, which may increase the success rates, especially in cases with few or even single traces.

5.3 The attack with all intermediate values

5.3.1 Experiment setup

For the source code of Ascon AEAD, I first targeted Weatherley's unmasked Ascon-128 implementation [130, `ASCON/`], where they provided an optimized Ascon permutation for ARMv7-M Architecture [131], which is compatible with the Cortex-M4 processor on the CW-Lite 32-bit board. This implementation was compiled with `arm-none-eabi-gcc` (v9.2.1) compiler options `-Os` and written onto the CW-Lite 32-bit board, and again following the recording setting in Section 2.6.1. I will refer to this experiment as `U-Os`, to distinguish it from the other two experiments, `U-O3` and `M-Os`, where the former is on the unmasked implementation with the compiler option `-O3`, and the latter is on Weatherley's masked Ascon-128 implementation [130, `ASCON_masked/`] with the compiler option `-Os`.

Recall that I used empty A and seven-byte P in this attack demonstration. In the profiling stage, I recorded one trace for each encryption with varying K, N, and P, and then categorized them into the sets introduced in Section 2.6.2. I recorded 10 000 traces for the attack stage, where every 10 traces were recorded from encryptions with the same K, but varying N and P, so they can evaluate the results of our experiments for multiple-encryption belief propagation provided with up to 10 traces.

In the profiling stage, I repeated the same procedure in the Keccak experiments described in Chapter 4, which includes interesting clock cycle detection, fragment template profiling, and checking the quality of templates. In the attack stage, I limited the belief propagation to at

Table 5.1: Number of interesting clock cycles detected for lanes of intermediate states in `U-Os`. The detection for L_0, L_3, and L_4 for state β_{-1} of Initialization is not needed since these lanes are loaded with known values IV and N, and therefore we do not build templates for them.

lane		L_0	L_1	L_2	L_3	L_4	lane		L_0	L_1	L_2	L_3	L_4
	input (β_{-1})	IV	310	366	N			input (β_{-1})	143	49	56	57	91
	α_0	39	143	46	41	110		α_0	38	23	43	33	26
	β_0	29	25	33	27	40		β_0	25	27	31	35	41
	α_1	24	34	40	32	34		α_1	25	24	40	30	36
	β_1	22	23	32	24	38		β_1	15	28	29	26	41
	α_2	28	19	38	44	30		α_2	33	20	39	30	31
	β_2	20	30	33	24	42		β_2	28	33	39	36	40
	α_3	29	19	42	35	26		α_3	26	28	42	35	30
	β_3	21	23	39	32	41		β_3	19	30	31	30	39
	α_4	49	29	41	36	31		α_4	24	23	42	35	29
	β_4	20	24	34	36	38		β_4	23	25	31	34	41
	α_5	25	27	38	23	64		α_5	30	25	46	30	38
Init.	β_5	20	30	31	34	38	Fin.	β_5	27	28	30	29	41
	α_6	27	22	36	31	28		α_6	27	24	38	29	29
	β_6	25	30	30	31	38		β_6	20	23	31	31	39
	α_7	26	22	54	31	26		α_7	25	19	36	41	32
	β_7	26	26	43	31	45		β_7	27	45	32	34	41
	α_8	22	20	36	38	26		α_8	24	20	33	33	29
	β_8	20	29	33	26	38		β_8	29	29	32	26	43
	α_9	26	23	38	33	26		α_9	32	27	45	32	29
	β_9	38	28	32	29	40		β_9	23	34	32	26	40
	α_{10}	26	21	37	30	28		α_{10}	25	30	34	30	30
	β_{10}	23	32	34	34	40		β_{10}	20	25	34	29	41
	α_{11}	25	18	45	35	32		α_{11}	26	20	36	32	37
	β_{11}	116	47	86	34	58		β_{11}	79	77	74	134	163

most 1000 iterations if it did not reach a steady state. This number is higher than the previous number of 200 in Keccak since now there are more layers (full-size intermediate states, supposedly bearing all information, see Section 3.4.2), which may require more iterations to stabilize. For the key enumeration step, I limited the search to enumerating up to 100 000 candidate keys when evaluating the success rate.

5.3.2 Detecting the interesting clock cycles

At first, I had planned to directly follow the method used in Section 4.4.3 to detect interesting clock cycles for 32-bit words of intermediate states $(\beta_{-1}, \alpha_0, \ldots, \beta_{11})$ in the Ascon permutation, but Weatherley implemented a *bit-interleaved* [68, Sec. 4.1.1] version of Ascon, which affected the storage of intermediate states, so I had to modify the detection procedure.

Previously, the target Keccak implementation simply separated each 64-bit lane into its high

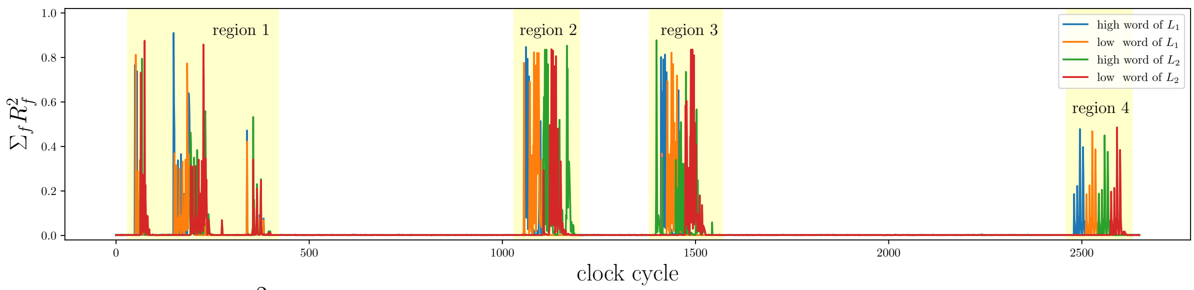


Figure 5.4: The $\Sigma_f R_f^2$ results for each 32-bit word of the 128-bit K for `U-Os`. The spikes lie in the marked regions corresponding to the four uses of K.

and low 32-bit words. However, when it comes to this bit-interleaved version of Ascon, a 64-bit lane is not just separated into a high and a low 32-bit word, but also sliced into its odd and even parts during the permutation, such that one 64-bit rotation becomes two 32-bit rotations. Therefore, data bits, especially the input and output, can be separated into high and low words (H/L words), as well as sliced into even-bit and odd-bit words (E/O words). Therefore, I decided to detect the interesting clock cycles for both the H/L and E/O words for a lane, and use their union set as the interesting clock sets for this lane, to consider both situations.

Tables B.25 and B.26 show the number of detected interesting clock cycles for each target 32-bit word of the intermediate states for the full AEAD process (H/L and E/O words, respectively). Note that I set the $\Sigma_f R_f^2$ threshold to 0.004, which was lower than the previous 0.01 in Section 4.4.3 for selecting more clock cycles into the interesting sets for better templates but not beyond the restriction introduced in Section 2.6.3. After merging them into their union set, Table 5.1 shows the number of interesting clock cycles ultimately selected for each target lane of the intermediate states. We can see that there were more interesting clock cycles detected for those words closer to input or output (i.e., β_{-1} or β_{11}), as some of their interesting clock cycles were related to operations outside of the Ascon permutation, such as loading the initial states, XORing with P or K, or calculating T.

Among all the words, we can observe the highest number of interesting clock cycles for L_1 and L_2 in β_{-1} of Initialization, since these two lanes are loaded with K, which is used four times in the full encryption. Figure 5.4 shows the $\Sigma_f R_f^2$ values for the H/L words from L_1 and L_2 in β_{-1} of Initializaion with the corresponding clock cycles. We can see that the spikes were mainly located in four separate regions, indicating the clock cycles related to the four times when Ascon AEAD uses the key K.

Similar to the previous Keccak experiment on the same device, I downsampled the selected interesting clock cycles from 500 to 10 PPC by replacing each 50 consecutive samples with their average, and then concatenated these averaged samples to form the traces $\mathbf{x}$ used for LDA-based template building.

Table 5.2: Quality evaluation of selected fragment templates for the `U-Os` experiment.

word		high				low			
byte		0	1	2	3	4	5	6	7
L_1 of Init. **input**	SR	0.859	0.809	0.852	0.733	0.804	0.684	0.751	0.619
(1st lane of K)	GE	1.244	1.395	1.276	1.695	1.457	1.803	1.514	2.224
L_2 of Init. **input**	SR	0.791	0.758	0.820	0.766	0.868	0.777	0.758	0.647
(2nd lane of K)	GE	1.399	1.515	1.274	1.421	1.214	1.462	1.492	1.900
word		even				odd			
byte		0	1	2	3	4	5	6	7
L_3 of Fin. β_{11}	SR	0.126	0.095	0.165	0.165	0.137	0.144	0.170	0.119
	GE	18.728	23.033	15.163	16.794	17.962	20.246	14.554	17.330
L_4 of Fin. β_{11}	SR	0.099	0.095	0.193	0.195	0.158	0.112	0.186	0.202
	GE	21.818	27.116	14.571	12.420	16.128	22.128	11.345	11.514
L_0 of Init. α_6	SR	0.003	0.006	0.008	0.009	0.009	0.012	0.004	0.006
	GE	108.681	103.795	90.442	112.758	101.961	107.853	108.965	106.971
L_1 of Init. α_6	SR	0.004	0.003	0.005	0.002	0.009	0.007	0.007	0.005
	GE	115.661	112.195	116.034	115.362	113.493	119.685	114.003	113.712
L_2 of Init. α_6	SR	0.014	0.010	0.011	0.014	0.007	0.011	0.016	0.003
	GE	81.931	97.490	81.405	99.522	96.865	88.722	81.449	100.995
L_3 of Init. α_6	SR	0.006	0.006	0.010	0.009	0.006	0.006	0.009	0.010
	GE	106.805	115.582	107.647	113.903	101.543	104.414	99.483	108.038
L_4 of Init. α_6	SR	0.011	0.007	0.008	0.009	0.008	0.007	0.012	0.011
	GE	101.617	107.802	103.061	110.333	110.936	111.034	99.328	106.930

Table 5.3: Quality evaluation of fragment templates for the key of Ascon AEAD with either all or only one part of the interesting clock cycles (`U-Os` experiment).

lane		L_1								L_2							
word		high				low				high				low			
byte		0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
all interesting	SR	0.859	0.809	0.852	0.733	0.804	0.684	0.751	0.619	0.791	0.758	0.820	0.766	0.868	0.777	0.758	0.647
clock cycles	GE	1.244	1.395	1.276	1.695	1.457	1.803	1.514	2.224	1.399	1.515	1.274	1.421	1.214	1.462	1.492	1. 900
region 1 only	SR	0.090	0.112	0.125	0.090	0.088	0.069	0.082	0.057	0.075	0.083	0.094	0.070	0.145	0.065	0.092	0.118
	GE	25.020	22.873	16.210	26.027	27.975	35.203	26.050	36.413	31.651	25.207	26.250	32.843	14.897	29.643	26.532	18.068
region 2 only	SR	0.100	0.085	0.093	0.051	0.166	0.112	0.141	0.089	0.150	0.166	0.203	0.158	0.175	0.177	0.149	0.098
	GE	25.797	28.291	26.319	44.628	13.469	21.747	13.740	26.549	13.986	16.174	11.067	15.628	11.194	13.307	15.471	26.583
region 3 only	SR	0.135	0.095	0.088	0.061	0.103	0.104	0.110	0.065	0.098	0.066	0.127	0.070	0.152	0.137	0.121	0.066
	GE	17.130	21.803	23.270	36.412	23.911	23.527	21.761	37.945	22.369	33.036	16.823	31.951	12.932	16.340	19.684	30.772
region 4 only	SR	0.114	0.112	0.124	0.091	0.099	0.083	0.119	0.091	0.068	0.077	0.093	0.089	0.096	0.103	0.130	0.050
	GE	19.158	15.346	15.581	16.260	20.283	18.838	15.107	20.198	25.091	20.325	17.508	18.849	18.837	16.932	13.124	28.112

5.3.3 Fragment template profiling

After building the LDA-based template parameters ($\mathbf{S}$, $\bar{\mathbf{x}}_{b,\text{proj}}$ for all 256 values b of a fragment, etc.), according to the results of the interesting-clock-cycle detection, I used the 1000 traces in the validation set to evaluate the quality of these templates. Table 5.2 shows success rate (SR) and guessing entropy (GE) from only a few example templates, while Figure 5.5 plots the results for all the target templates. Note that I built the H/L templates for the key, but E/O templates for the other intermediate values, to better match the implementation.

We can observe that templates for the key fragments had the best quality among all the templates, as K fragments were built from the highest numbers of clock cycles. The results for templates of fragments in the last two lanes in state β_{11} in Finalization were also satisfactory, considering that these two lanes are part of the permutation output and then XORed with the

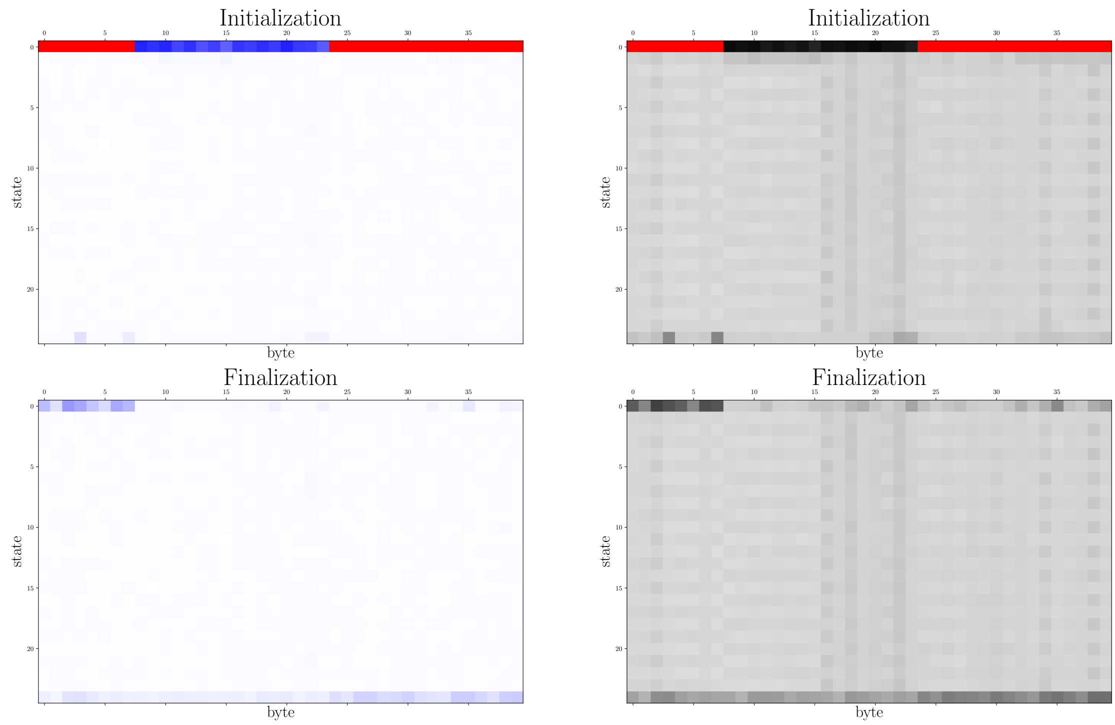


Figure 5.5: Success rate (left) and guessing entropy (right) of all target fragment templates from `U-Os`. Each row represents a 40-byte state, e.g. state 0 is β_{-1}, state 1 is α_0, state 2 is β_0, etc., in chronological order. The red blocks represent bytes of the known values IV, N, for which no templates were needed.

key for the tag T, leading to more interesting clock cycles detected. The SRs for fragments from the middle rounds, α_6 in Initialization for example, were much lower, while the corresponding GEs were much higher than those values from either K or β_{11} in Finalization, as the optimized implementation of Weatherley appears to reduce the clock cycles that operate on a single intermediate value inside the permutation, whereas the input and output of a permutation will be involved in more operations across the permutations for AEAD mode.

Table 5.3 also shows the results of quality evaluation when building the templates for the key fragments with only one of the four regions of interesting clock cycles. These results provide evidence that using the same key more than once in an Ascon AEAD significantly helps the attackers to build better templates, as the quality will be much better when considering all interesting clock cycles instead of only those from each key use.

5.3.4 Results after belief propagation and secret enumeration

With the fragment templates, I applied the previously described attack procedure to the attack data set. The loopy belief propagation was limited to 1000 iterations if it did not reach a steady state before that, while the key enumeration was limited to enumerating up to 100 000 candidate keys when evaluating the key-recovery success rate.

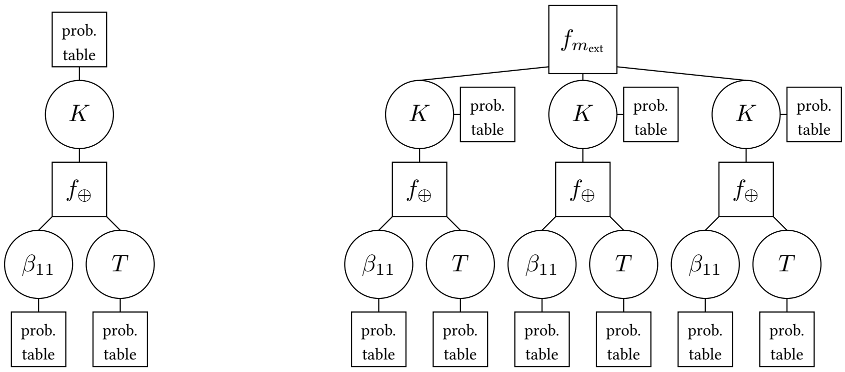


Figure 5.6: Tree-shaped factor graphs for single (left) and multiple encryptions (right).

With up to 10 encryptions for each key, Table B.27 and Figure 5.4.2 show the success rates for recovering the 1000 different attack keys after template recovery, belief propagation, and key enumeration[1]. We can see that the attack on single encryptions was not yet very successful, but the success rates exceeded 90% once the attackers obtained traces from a few encryptions.

5.4 The attack with intermediate values around the key

5.4.1 Loop-free alternative factor graph

As we can see from the results of the template evaluation in Figure 5.5, the templates for fragments in the middle states of both the permutations in Initialization and Finalization provide only a little information. Therefore, it may not be worth performing belief propagation with a large factor graph covering all the middle states. Instead, I removed the nodes for those middle states from the factor graph in my experiment, and only kept the nodes related to the XOR operation of the key K and the last 128 bits of β_{11} in Finalization to calculate the tag T, as a loop-free alternative factor graph. Figure 5.6 shows the new smaller factor graph for single encryptions and its expanded version for multiple encryptions with the same key. These smaller factor graphs will similarly output updated probability tables for key enumeration.

There are two advantages of this smaller graph design. The first one is that it is no longer a loopy structure, but a tree, so it will update the information recursively by accessing each node only once. On the other hand, thanks to the simplicity of the XOR operation, as well as the assumption that the tag T is already known by attackers, it will still be practical to perform belief propagation on byte tables or tables for even larger fragments (e.g., 16 bits), and therefore avoid the information loss caused by marginalization to bit tables. In these cases, the belief propagation procedure will output the updated probability tables for fragments instead of bits for enumeration.

[1]For the enumeration on bit tables, the implementation requires end-of-state management, see Appendix A.1.

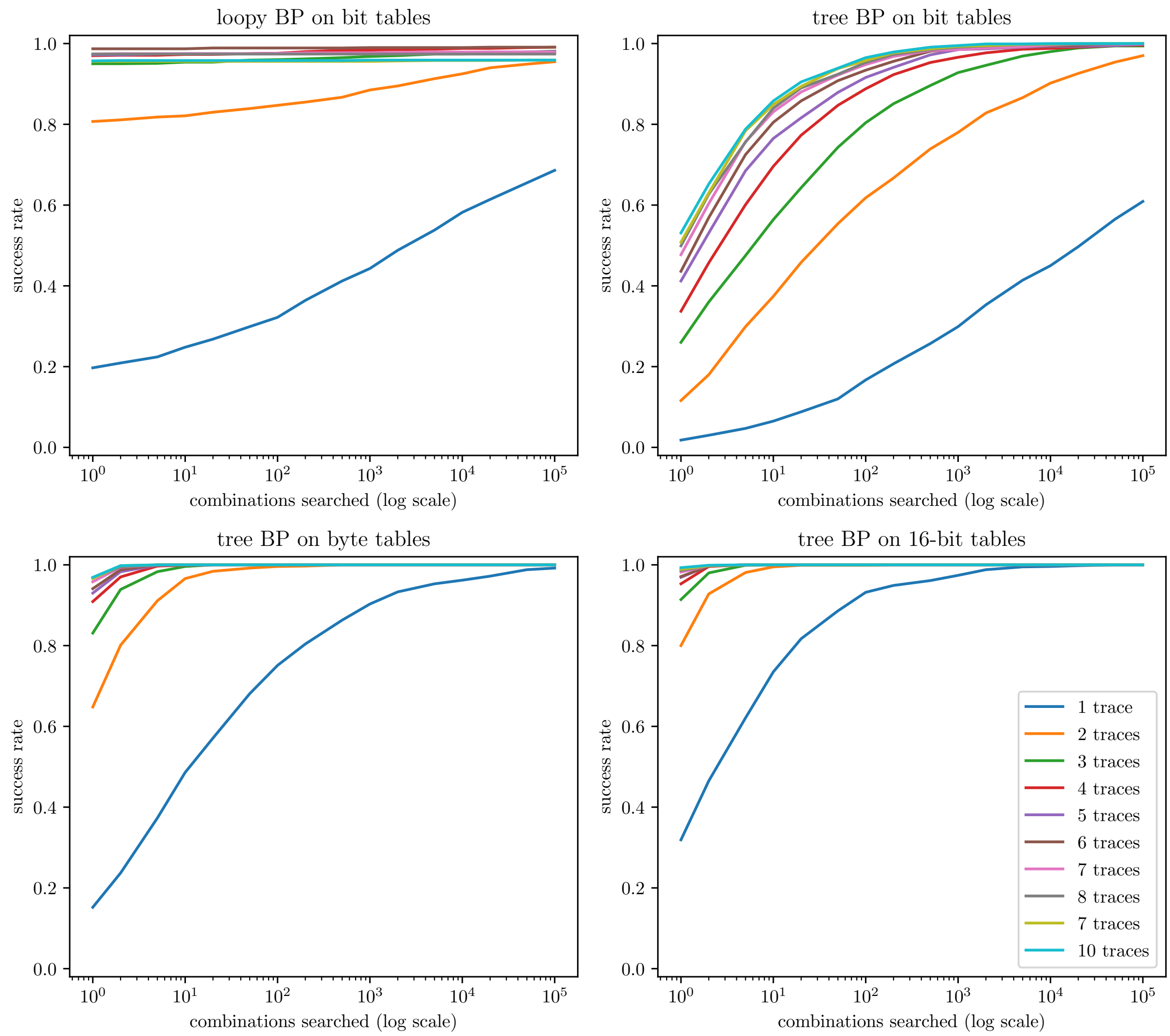


Figure 5.7: Success rates in the four experiments.

5.4.2 Results

To compare with the results of the previous experiment with the factor graph covering all the intermediate states, Table B.28 shows the success rates of recovering the 1000 attack keys when applying the smaller factor graph in the belief propagation procedure on bit tables marginalized from the predictions of byte templates. Table B.29 shows the results of the `U-Os` experiment directly on probability tables obtained from byte fragment templates. Note that without the marginalization step, it is more difficult to use probability tables from H/L templates and E/O templates within a factor graph. Therefore, we have to stick to either version in an experiment, and here I show results with H/L templates. After 100 000 combinations of key fragments enumerated, this attack achieved 100% success rates with multiple traces. Even with single attack traces, the success rate is 99.2%.

To observe whether a larger fragment size in template building helps attackers to collect more information, we decided to repeat the experiment by cutting the 32-bit words into 16-bit frag-

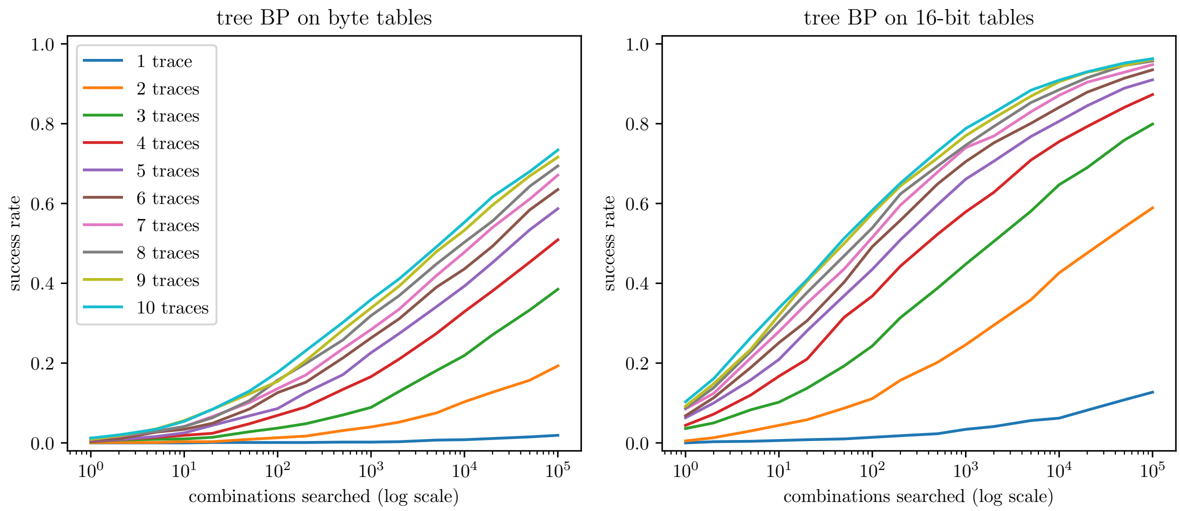


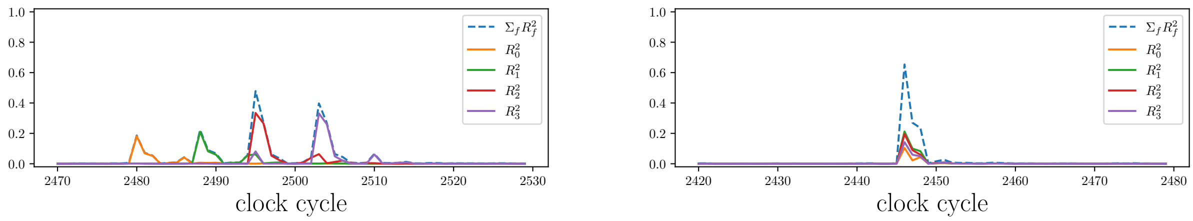
Figure 5.8: Success rates on ASCON-128 with optimization option `-O3`, for both 8 and 16-bit fragments (See Table B.32 and B.33 for the actual values). See Figure 5.7 for comparison against the `-Os` version.

ments instead of bytes for template building. Table B.30 shows the results of quality evaluation on the templates for the H/L fragments of the key and the last two lanes of β_{11} in Finalization, while Table B.31 also shows the results of the experiment directly on tables from these 16-bit fragment templates. We can see that the success rates are even higher than those with templates for bytes, given the same number of traces and the same number of searched combinations. For better comparison, Figure 5.7 depicts the results for the success rates with the four different settings (loopy BP on bit tables, tree BP on bit tables, tree BP on 8-bit tables, and tree BP on 16-bit table) in the `U-Os` experiment.

5.5 Compiler optimization levels

In my previous experiments to attack KECCAK, I had left the `gcc` optimization level to option `-Os` (optimized for space), as it was the software default setting of both the 8-bit board designed by Choudary [52, Section 2.2.2] and the ChipWhisperer platform. When I submitted the CARDIS paper about the experiments of KECCAK implemented on a 32-bit device, one of the reviewers suggested we should also look into how the optimization level and goal can affect the template profiling. They indicated that it could be more difficult if we had either compiled the C codes with level `-O3` (optimized for time) or used another manually optimized assembly implementation from the same package.

Therefore, in addition to the previous `U-Os` experiment on ASCON AEAD, I decided to repeat the experiment with `gcc` option `-O3` (optimize for time), resulting in the `U-O3` recordings, to see whether the compiler's code generation can significantly affect the performance of our attack. Note that the different optimization options will not affect the execution of Weatherley's



(a) `U-Os` experiment (b) `U-O3` experiment

Figure 5.9: The $\Sigma_f R_f^2$ results and the R_f^2 values for each byte fragment ($f = 0, 1, 2, 3$) of the high word of L_1 in K

Ascon permutation [130, `ASCON/internal-ascon-armv7m.S`], since its source code is manually optimized assembly code, which bypasses the optimizer. However, they affect the AEAD code generated around the permutation, such as XORing the key K or calculating the tag T, as these operations are written in C. Thus, here I focused only on the tree-BP experiment, as the middle rounds of the permutation will not be affected.

Figure 5.8 shows the same information as Figure 5.7, but using compiler option `-O3` instead of `-Os`. The performance of the attack is worse: it needed 10 attack traces for each key to reach near 100% success rate using 16-bit fragments (Table B.33), and achieved only 73.4% success rate with 8-bit fragments (Table B.32), rather than being able to reach nearly 100% success rate from a single attack trace in the `U-Os` experiment.

A look into both the C source code of Weatherley's unmasked implementation and the assembler listing produced by the compiler (with option `-Wa,-adhlns=file.lst`), revealed the reason. Although the handwritten assembler code for the permutation uses 32-bit registers, the surrounding C code XORs the key K with the state of the duplex construct. For example, it XORs two lanes of Finalization output (β_{11}) with K, to generate the tag T, using the macro `lw_xor_block_2_src()` in [130, `ASCON/internal-util.h`], which is a loop processing individual bytes. When compiled to optimize code space (i.e., minimize the size of the executable) with `gcc` option `-Os`, the resulting ARMv7-M assembler code looks pretty exactly like the source code suggests, i.e., a loop over 16 bytes, which loads one byte from K and one from β_{11} into the 32-bit registers, XORs them, and stores one byte of T per iteration. In contrast, if we instead ask the compiler to optimize for time (`-O3`), it not only unrolls that loop, but also converts it into a sequence of just four repetitions of the operations for loading, XORing, and storing 32-bit words. In other words, the optimizer converted here an 8-bit implementation of the key XOR operation into a 32-bit implementation.

We can also observe this difference from the recorded traces. Figure 5.9a and 5.9b show the results of the interesting clock-cycle detection for the high word of the first lane (L_1) of K during the calculation of T, when the code was compiled with options `-Os` and `-O3`, respectively. For `U-Os`, the peaks of the R_f^2 values of each 8-bit fragment are located in four different clock cycles, indicating that their operations were not executed simultaneously, whereas for `U-O3`, the peaks are located in the same clock cycle.

5.6 Attacking a masked version

After my experiments on the unprotected ASCON implementation above, I also tried to apply the combination of fragment template attack, belief propagation, and key enumeration on an implementation with masking.

5.6.1 Attack strategy

The target masked implementation of ASCON AEAD was from the same package by Weatherley [130, `ASCON_masked/`]. This offers a C implementation of the permutation and protects the inputs (key, nonce, plaintext, etc.) with first-order Boolean masking [75], separating each of these values into two shares: one is the mask, varying per encryption, provided by a pseudo-random generator based on ChaCha [132], and therefore the other share is the XOR of the input value and the mask. Throughout the encryption process, the intermediate values all remain likewise split into two shares, to randomize all the register values during execution. Besides, compared to the unmasked (naive) version, this implementation also avoids some problems that may help side-channel attacks on the former. For example, it no longer XORs 8-bit values when calculating the tag T, and the two shares of the key are only sliced once, rather than three times.

Bronchain and Standaert [88] attacked Boolean-masked implementations of AES and Clyde by extending the factor graph for the unmasked algorithm with nodes representing the original values connected to their shares in the masked version via a $f_{\oplus}$ factor. Following this idea, I introduce a multi-trace attack derived from the previous tree-shaped one, where the factor graph (Figure 5.10) will also cover the two shares of the original target states. Similar to the setting of the previous unmasked version, I use the empty associated data A and fix the size of the plaintext P to seven bytes. In the profiling stage, I assume that attackers can access all the input and output values (K, N, P, C, and T) as well as the seed of the pseudo-random generator, so they can produce fragment templates for the key, its two shares, and all the other intermediate states in the factor graph. In the attack stage, I only use the probability tables obtained from the templates, and the known T values, to perform belief propagation and key enumeration, without knowledge of the seed.

Note that Figure 5.10 reflects the mathematical relations among the original values and their shares, not the actual steps in this masked implementation to calculate T. The implementation first calculates $T^{\mathrm{A}} := K^{\mathrm{A}} \oplus \beta_{11}^{\mathrm{A}}$, $T^{\mathrm{B}} := K^{\mathrm{B}} \oplus \beta_{11}^{\mathrm{B}}$, and finally $T := T^{\mathrm{A}} \oplus T^{\mathrm{B}}$. Therefore, it is not possible to build templates for the fragments of β_{11} since this value never appears. Instead, I assign them a probability table with a uniform distribution (i.e., no information update). Besides, the assumption was that the attacker knows T, so we do not need the templates or probability tables of T^{A} and T^{B}, given that they will not affect the belief propagation following the factor graph in Figure 5.10.

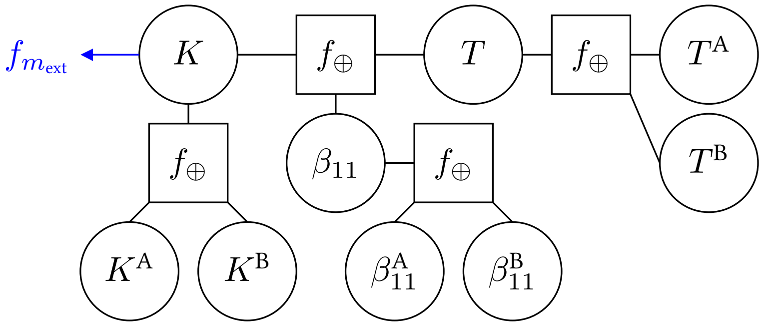


Figure 5.10: Factor graph for the `M-Os` experiment. (Each variable node also connects to an observation factor node, which is omitted in this graph.)

Table 5.4: Numbers of interesting clock cycle detected for the `M-Os` experiment.

target state		K		K^{A}		K^{B}		Fin. β_{11}^{A}		Fin. β_{11}^{B}		T^{A}		T^{B}	
Lane 1	high/low	21	26	113	109	161	157	41	42	46	40	29	26	33	33
	even/odd	16	28	113	106	213	207	36	33	28	39	26	13	26	26
	union	37		144		240		50		58		34		38	
Lane 2	high/low	20	26	107	109	203	174	36	36	44	43	35	36	30	36
	even/odd	30	33	114	116	230	227	17	41	16	39	15	37	19	30
	union	35		139		259		49		51		45		43	

Table 5.5: Quality evaluation of fragment templates for the `M-Os` experiment (10 PPC).

byte		Lane 1								Lane 2							
		even word				odd word				even word				odd word			
		0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
K	SR	0.006	0.004	0.012	0.016	0.013	0.018	0.020	0.011	0.003	0.006	0.012	0.013	0.016	0.016	0.013	0.021
	GE	94.191	93.927	95.611	96.468	86.433	89.916	80.927	89.966	98.839	98.854	92.056	89.635	95.598	96.537	88.315	77.098
K^{A}	SR	0.179	0.112	0.245	0.206	0.115	0.108	0.129	0.162	0.177	0.099	0.246	0.149	0.142	0.102	0.173	0.214
	GE	12.832	21.155	8.404	10.507	21.120	21.548	19.525	15.095	12.263	26.006	9.219	16.535	18.674	19.911	11.733	10.043
K^{B}	SR	0.308	0.400	0.440	0.511	0.354	0.535	0.340	0.598	0.330	0.390	0.419	0.509	0.313	0.431	0.307	0.622
	GE	6.428	4.157	3.978	2.619	4.847	2.563	5.265	2.155	6.159	3.344	4.002	2.688	5.710	4.301	6.615	2.186
Fin. β_{11}^{A}	SR	0.014	0.014	0.021	0.017	0.013	0.016	0.013	0.026	0.017	0.011	0.016	0.019	0.011	0.015	0.014	0.023
	GE	83.441	89.578	64.616	59.874	87.379	91.359	66.304	58.571	84.304	90.400	76.079	67.148	92.852	93.192	89.379	64.534
Fin. β_{11}^{B}	SR	0.007	0.010	0.014	0.015	0.016	0.016	0.018	0.019	0.014	0.011	0.007	0.007	0.011	0.007	0.010	0.020
	GE	90.673	95.790	73.321	75.462	85.796	88.584	69.418	57.238	95.561	103.957	101.897	88.470	80.066	83.126	70.817	55.938
T^{A}	SR	0.014	0.008	0.022	0.015	0.015	0.008	0.009	0.008	0.003	0.009	0.010	0.020	0.015	0.023	0.030	0.029
	GE	87.089	92.368	63.086	61.486	98.785	96.537	99.952	72.166	98.220	97.729	104.405	71.752	78.881	83.457	56.376	43.318
T^{B}	SR	0.007	0.012	0.011	0.016	0.011	0.014	0.014	0.013	0.004	0.011	0.007	0.012	0.018	0.008	0.027	0.017
	GE	76.986	87.305	83.276	64.351	93.276	94.675	90.775	78.822	101.118	96.928	98.279	89.296	80.710	85.440	63.543	70.187

5.6.2 Experiments

For the experimental setup, most of the environment and parameters stay the same as with the experiments on the unmasked versions (`U-Os` and `U-O3`), except for the larger number of attack traces recorded, to have 100 encryptions each for the same key. As I still use 1000 different keys, eventually I recorded 100 000 traces in total for the attack set. I will later refer to this recording and experiment as `M-Os`.

Table 5.4 shows the number of interesting clock cycles detected in the `M-Os` experiment, while Table 5.5 shows the results of the quality evaluation of the fragment templates with 10 points per clock cycle. Here the fragment templates are for sliced registers (E/O words) since that

Table 5.6: Success rates of key recovery in the experiments in the `M-Os` experiment.

#Traces	#Combinations searched																	
	1	2	5	10	20	50	100	200	500	1000	2000	5000	10000	20000	50000	10^5	2×10^5	5×10^5	10^6
1	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
2	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
5	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.001	0.003	0.005	0.011	0.018
10	0.001	0.001	0.003	0.008	0.010	0.019	0.022	0.031	0.055	0.070	0.087	0.114	0.159	0.201	0.252	0.309	0.371	0.427	0.476
20	0.016	0.024	0.053	0.076	0.104	0.148	0.199	0.248	0.310	0.360	0.419	0.504	0.562	0.614	0.698	0.741	0.780	0.835	0.867
50	0.063	0.101	0.160	0.218	0.276	0.365	0.423	0.503	0.580	0.642	0.703	0.759	0.798	0.833	0.874	0.901	0.915	0.943	0.966
100	0.092	0.142	0.216	0.283	0.369	0.456	0.525	0.595	0.661	0.709	0.755	0.816	0.845	0.876	0.902	0.931	0.951	0.970	0.976

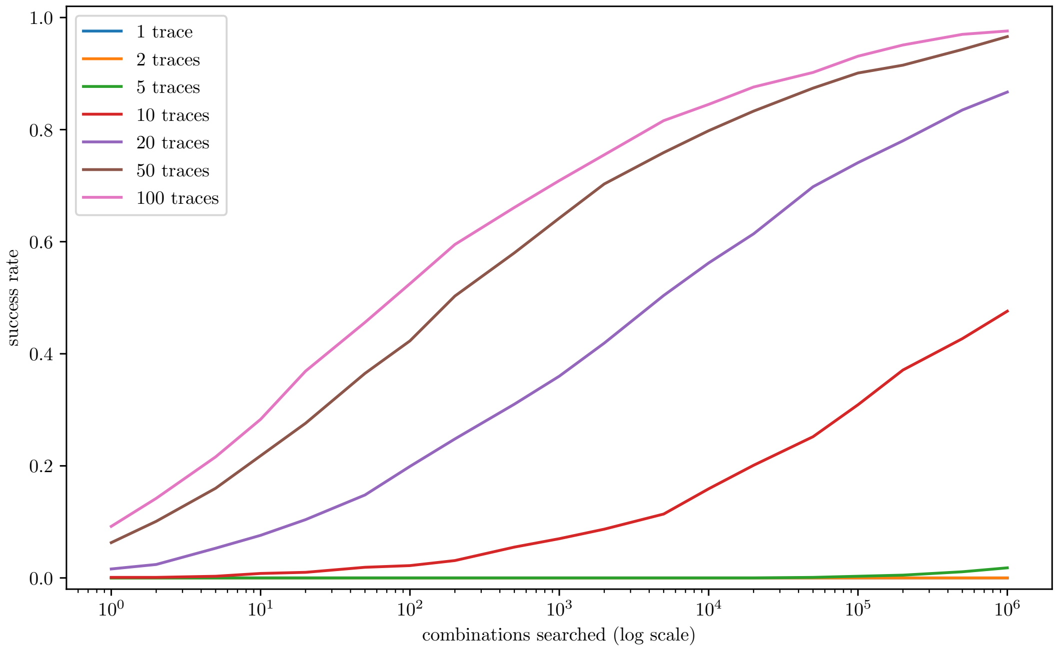


Figure 5.11: Success rates of attack on the masked implementation of Ascon-128 encryption.

is how the implementation represents most of my target states. We can see that the masking does protect the key K to some extent, as fewer interesting clock cycles (37 and 35 for the two lanes, respectively) were detected compared to the unmasked experiments (see Table 5.1, β_{-1} in Init.), leading to lower quality templates as evident from the higher guessing-entropy values for these fragments. However, for the two shares K^{A} and K^{B}, I still detected a large number of interesting clock cycles (144 and 139 for two lanes of K^{A}, 240 and 259 for K^{B}), and therefore the quality of their templates is still promising once attackers can calculate the random numbers for masking in the profiling stage. Note that there are more interesting clock cycles for K^{B}, the random mask, than for K^{A}, because for the former we can also detect leaks from where the masks are generated.

For the belief propagation and key enumeration, Figure 5.11 and Table 5.6 show the key-recovery success rates for different numbers of attack traces, and key enumeration of up to 1 000 000 combinations. With 10 traces using the same key, the success rate was 47.6%, while with 100 traces, it was 97.6%. However, single-trace attacks did not succeed in this experiment.

5.7 Size of fragments for template profiling

In the experiments on Keccak and Ascon, I provided some comparisons between the results from different sizes of fragments. From these results, it seems like we can achieve a slightly better success rate when profiling templates with larger fragments, but it is not always worth doing so since the template recovery will be slower. In my attacks on Keccak, I recorded traces long enough to cover multiple rounds in Keccak-f permutations, and then collect information from templates for hundreds of fragments. In this case, the attack procedure with 16-bit fragments became even longer.

On the other hand, given the large number of target fragments and the complicated mathematical relations between them, it is more feasible to use a bitwise belief-propagation procedure with probability table marginalization, which may further narrow down the advantage of using 16-bit or even larger fragments.

Meanwhile, in the case of attacking Ascon with tree-shaped belief propagation and key enumeration (Section 5.4), the 16-bit option is still manageable with the smaller factor graph and no marginalization applies in the procedure. Therefore, I recommend using 16-bit fragments in cases with limited numbers of target values and only simple mathematical relations between these values, such as XOR.

However, it is also possible to choose different fragment sizes in template profiling. Based on my data for experiments with Keccak implemented on the 32-bit device, there is a not yet published report about follow-up research by Spyropoulos [133] that applied 11-11-10-bit fragments and analyzed how this option could be better than using byte fragments. Therefore, I believe that the choice of fragment size in template profiling is flexible, depending on the goals of attackers and how many computing resources are available.

5.8 Discussion

On the CW-Lite 32-bit device, I have shown that we can recover the key used in an unmasked Ascon AEAD implementation by a procedure involving fragment template attack, belief propagation, and key enumeration. For the belief propagation, I first used a loopy factor graph covering all the intermediate states in the encryption procedure with marginalized tables. The results strongly indicate that the quadruple use of the key in Ascon AEAD mode increases the exposure of the key in profiled side-channel attacks, although this is cryptographically useful to strengthen the Initialization and Finalization phases. The success rate was much lower if we observed only clock cycles from any one of these four applications of the key (Table 5.3). That higher exposure of the key, which in the loopy factor graph is directly connected to four different locations (Figure 5.3), enables the belief propagation algorithm to pass messages between Initialization and Finalization. Previous attack simulations by Luo et al. [129] did not exploit this higher key exposure and used only the mathematical relations around the first use of the key, at the start of Initialization.

As Weatherley's Ascon permutation is already manually optimized for the ARMv7-M Architecture, I had only built templates with low quality for the intermediate states in the middle rounds of the permutation. Therefore, we can use the loop-free alternative factor graph only covering the single XOR operation calculating the tag, to avoid the information loss caused by table marginalization.

The successful single-trace attack (`U-Os`) benefited from some remaining 8-bit instructions in an open-source 32-bit adaption of the algorithm. Yet, even once these were fully converted to 32-bit instructions (`U-O3`), we still could recover the key used in this unmasked Ascon AEAD implementation, by belief propagation and key enumeration, with high success rates, from no more than 10 traces. From these experiments, I noticed that the optimization level of the implementation may play an important role when profiling templates.

The successful multi-trace attack on the more carefully written first-order Boolean-masked Ascon AEAD implementation demonstrates how such protection, originally designed against CPA/DPA-style attacks, can be overcome by an appropriately designed template attack. Considering the similarity between Ascon-128 and Ascon-128a, I believe that this attack procedure should also apply to both unmasked and masked Ascon-128a implementations with only minor modifications.

An additional outstanding challenge remains to recover complete Ascon hashing inputs from a single trace, as was accomplished in my previous experiments for SHA-3 (Keccak). This will likely require better templates for the internal states of the Ascon permutation. The templates for these (e.g., Init. α_6 in Table 5.2 and Figure 5.5) were less effective than those for the Keccak permutation in Figure 4.5. However, even with the very similar hardware setting I used, such direct comparisons are still complicated by the fact that the Keccak and Ascon target implementations came from different authors and had different programming styles. The former was entirely portable C code that left the 64-bit to 32-bit conversion to the compiler, whereas the latter offered a handcrafted assembler implementation of the permutation. That, but also the fact that Ascon's permutation is significantly simpler, for example, it lacks an equivalent of Keccak's complex θ step, overall appears to have resulted in less information leaking from the fewer instructions needed by Ascon to process its intermediate values.

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

Chapter 6

Conclusion

Even before I started my Ph.D. program, I believed that template attacks are the most powerful category of side-channel attacks, and Choudary et al. have shown that it is promising to use LDA-based template attacks to recover the actual values of a state (full-state recovery) rather than merely extract functions such as HW values of this state. Their research was targeting individual instructions rather than entire algorithms, relying on access to more than one attack trace.

The first main contribution of this thesis is that I have successfully extended their approach from attacking a handful of instructions to targeting a complete permutation-based cryptographic algorithm. In Chapter 3, from my experiments targeting the SHA3-512 implementation on the 8-bit device, I first built good templates for full-state recovery on target intermediate bytes in Keccak-f, and the results have shown that once being used along with algorithmic tools such as secret enumeration or belief propagation, Choudary et al.'s method of linear-regression-and-LDA-based templates can attack the newly standardized SHA-3 family. My successful single-trace attack demonstrates that LDA-based templates can be even more powerful when attacking a cryptographic algorithm compared to a single target value, given that the multiple instructions on the intermediate values in such an algorithm can leak more information that can be detected and exploited in template profiling.

Secondly, my fragment template attack experiments have demonstrated that, by cutting a 32-bit intermediate value into smaller pieces, it is possible to apply a template attack to achieve full-state recovery with independently built templates for these pieces. When this method was used to attack the Keccak-$f[1600]$ implementation on the 32-bit device, the quality of these fragment templates was good enough that their predictions could later be used in bitwise belief propagation to recover the full arbitrary-length SHA-3 or SHAKE inputs with very high success rates. This clearly shows that this LDA technique is very helpful when we perform template attacks targeting devices with registers larger than bytes.

When it comes to other possible targets of the fragment template attack, I have also shown that it can recover the key used in an unmasked Ascon-128 implementation with belief propagation, with factor graphs covering either full encryptions of Ascon-128 or solely XOR operations involving the key, tag and the output in Finalization, with multiple or even single traces. I also showed how these fragment templates can potentially be used to attack a masked Ascon-128 implementation.

Impacts of my research When it comes to the impact on future SCA techniques, I believe my fragment template attack has already encouraged other researchers to consider a full-state style recovery on 32-bit devices as a practical attack scenario. For example, recently in 2023, Cassiers et al. [134] quoted my fragment template attack method as the first successful attempt to attack a 32-bit implementation of Keccak by full-state recovery instead of by HW values and then tried to directly profile templates for 32-bit values to attack ISAP-A [135], another Ascon application for re-keying. Their accelerated algorithm makes the profiling and attack procedure more feasible for 32-bit values. However, their belief propagation procedure directly used the probability tables predicted by the 32-bit templates, leading to larger memory usage (1.13 TiB of RAM) and longer run time (2.7 hours on 2.0 GHz CPU, single-threaded). Although direct comparison is not accurate, my experiments with 8-bit or 16-bit fragments normally used no more than a few GB of RAM and completed within a few seconds or minutes, even with a Python implementation. Therefore, I expect that my fragment template attack technique will become particularly a concern for fast attacks. However, no matter which approach is applied, full-state recovery template attacks on 32-bit devices are becoming more feasible.

As for the impacts on the security of Keccak and Ascon, my research is a reminder of the threats that template attacks may pose to such newly standardized permutation-based cryptographic algorithms. Compared to the simulated attacks on Keccak by Kannwischer et al. [89] and Ascon by Luo et al. [129], my attacks were on real power traces. This provides more convincing evidence of the power and feasibility of template attacks. While their simulation work was mainly focused on 8-bit or 16-bit implementations, my fragment template attack was the first attempt to successfully attack 32-bit implementations of Keccak's sponge function and Ascon AEAD. From these attacks, we should take away how effectively template attacks can extract information from unprotected permutation-based cryptographic applications, even on 32-bit devices.

6.1 Challenges

However, real-world applications of my fragment template attack may still face challenges, such as lack of knowledge about the target source code, alignment issues outside of a laboratory environment, or the portability of template attacks. Besides, there are a few possible improvements to my experiment setup.

Knowledge of source code When applying a fragment template attack on different applications, we may need a different level of knowledge about the implementations for a successful attack. It is possible that attackers only need to identify the target algorithm, or they need the knowledge of the source code or even machine instructions, to predict intermediate values, construct factor graph, and successfully perform a template attack.

My target KECCAK implementation was entirely portable C code that closely follows the official KECCAK document [15] and leaves the 64-bit to 32-bit conversion to the compiler, where the intermediate values stay in the original order without any bit interleaving (mentioned in Section 5.3.2). This enables attackers to profile templates using only the knowledge of the KECCAK algorithm, but no lower-level implementation details, in the profiling stage: once they obtain the input state for the profiling traces, they can predict all the intermediate values they need for the $\mathcal{F}_9$ linear-regression model used in interesting-clock-cycle detection, LDA projection, and profiling templates.

On the other hand, the unmasked ASCON-128 target was based on a handcrafted assembler implementation of the ASCON permutation with bit interleaving. However, even with that, the intermediate values in this implementation are still deterministic once we provide the same K, N, A, and P for an ASCON-128 encryption. This means that attackers can still predict the intermediate values if they know whether bit interleaving was applied to their target implementations. They can observe such information from the C code or the assembly code, or may just guess it from other implementations. Meanwhile, I demonstrated how attackers can consider both situations (bit-interleaved or not) at the same time, by detecting the interesting clock cycles with both the H/L and E/O bit groupings. In this case, attackers can rely on only knowledge of the ASCON-128 algorithm, without knowing the source or assembly code (to attack, e.g., Weatherley's unmasked implementation).

However, when it comes to Weatherley's masked ASCON implementation, the attack became more complicated. For any cryptographic implementations with masking, the intermediate values will be randomized. This makes it very difficult for attackers to perform template attacks with only knowledge of the algorithm. For example, I need to access the C code to observe information on which pseudo-random generator is used to generate the masks, and whether it first bit interleaves the key and then separates them into shares, or does it the other way round. Without such information from the C code, I might have failed to predict the intermediate values used for interesting-clock-cycle detection or template profiling. Besides, it is not always possible to access the seeds used in the pseudo-random generator during the profiling phase in a real-world scenario.

For my experiments in this thesis, I did not rely on knowledge of the targeted assembly code since I located the interesting clock cycles by using statistical methods (i.e., R^2 values of from multiple linear regression with the $\mathcal{F}_9$ model). I only used it to understand the impact of using different compiler optimization options (see Section 5.5). In summary, I believe that attackers will need only the knowledge of the algorithm or pseudo code of their target applications, when using my fragment template attack on unprotected implementations with deterministic intermediate values, but they will likely benefit from the knowledge of source code to attack a masked implementation.

Improvement for measurement and trace processing Another remaining issue is that even in my laboratory-controlled environment, the setting (see Section 2.6.1) for trace recording is possibly suboptimal. In my experiments on the 32-bit device, I recorded traces with the highest sampling rate of the oscilloscope and then downsampled the traces by summing up consecutive raw samples, which is equivalent to using a digital box filter [136] for low-pass filtering. Similarly, for my experiments on Choudary's 8-bit device, I also used a box filter for trace processing. However, using an analog lower-pass filter or digital filtering alternatives with other window functions, such as Lanczos filtering [137], etc. [138], may provide a better result. It deserves a careful survey on how the choice of the low-pass filter affects my fragment template attack.

Meanwhile, the current analog high-pass filter in my measurement setup for experiments on the CW-Lite board may also affect the attacks. The time constant of this high-pass filter is 0.5 μs, equivalent to the time for 2.5 clock cycles of my target device. With this relatively long time constant, any changes in the voltage originally within a clock cycle could be prolonged to a few later clock cycles, and this may lead to some samples being affected by variation from preceding clock cycles. This may affect the template attack in a more complicated way than we expected. On one hand, this could bring additional noise to our target clock cycles, but on the other hand, this introduction of correlation could help in template profiling since both signal and noise play an important role in the LDA procedure. One phenomenon I observed from my experiments with this high-pass filter was that we may profile better templates by also selecting some neighbor clock cycles to the originally selected interesting clock cycles. Figure B.3 shows the results when I repeated the attack on the Ascon `U-Os` data set, using templates profiled with interesting clock cycle sets expanded by also including the three neighbor clock cycles, and Figure B.4 shows their single-trace results comparing against the original attack (Figure 5.7). This may indicate that some of the signals we target may be affected by the neighbor clock cycles, but this phenomenon should be carefully compared with future experiments.

To avoid this problem, it is also possible to choose a high-pass filter with a shorter time constant, but that may lead to more distortion of the shape of the recorded traces. In my opinion, if we do not want to complicate the situation by using any high-pass filter, the ultimate solution is to record the power consumption from the GND side (e.g., on Choudary's board) instead of from the V_{DD} side (e.g., on CW-Lite board) of the circuit, or use a broadband transformer or differential amplifier.

Besides, there is a problem with my interesting-clock-cycle detection. The threshold for this procedure did not stay the same in each of my experiments, as I tried to select as many PoI as possible given the computing resources available. It remains an open question whether we can set a more meaningful threshold for R^2, such as the widely-recognized 4.5 threshold for Welch's t-test used in leakage detection [139].

From laboratory environment to real-world scenarios Although my experiments were more realistic than only using simulated profiling attacks, they were still in a laboratory-controlled environment. When applying fragment template attack techniques in a more real-world scenario, we may need to overcome some challenges that previous SCA techniques also faced.

The first, and maybe the most common one, would be the alignment issue. In my measurement setting, I used phase-locked clock sources for my target device and the oscilloscope, but we cannot always supply an external clock source to the target device to synchronize both. Given the length of the recorded traces, latter samples will become less aligned. However, this problem can be fixed by a few existing algorithms (e.g., *elastic alignment* introduced by van Woudenberg et al. [140]), to realign the traces, with possibly compromised template quality. Other causes of misalignment include unexpected clock cycles that are unrelated to the target algorithms, such as interrupts from the operating system or dummy instructions implemented as a hiding countermeasure. In these situations, we may need to use preprocessing steps to remove samples from the clock cycles not related to the target algorithms before we can profile templates.

Another challenge we may face is the portability [141] of fragment templates. Templates can be very specific to the profiled target hardware and can also be sensitive to other variables (e.g., temperature). Therefore, we do not usually expect that templates will remain valid when we implement the same software on hardware devices with different specifications. However, templates can stay effective when they are applied to traces recorded from other devices with the same specification [142] or the same device but in a different environment [141]. My experiments with fragment templates were still using the same device for profiling and attack, and how well my templates can be used to extract side-channel information from the traces from other devices with the same specification is still unknown. This weakness could reduce the feasibility of my fragment template attack.

In 2018, Choudary and Kuhn [142] demonstrated using LDA-based templates profiled with traces from one device to extract information from traces from another device. They found that the most significant cause of variation from the different devices is the DC component, but LDA dimensionality reduction can to some extent eliminate this and other differences. There are also some other possible variations, such as phase shift or different amplification, and these two types of mismatches can be corrected by preprocessing attack traces with re-alignment or renormalization preprocessing, respectively [141]. On the other hand, some previous attempts [143, 144] were based on profiling templates with traces collected from more than one copy of the target device, which avoids templates overfitting a single device. These techniques can be applied to my fragment template attack, but I expect that either the success rate of the attack will be compromised, or we may need more traces for attacks on Ascon AEAD and require templates for intermediate values of more than four rounds in Keccak-f permutation.

6.2 Future research directions

In addition to the previously mentioned technical improvements on my fragment template attack, this section discusses some high-level directions that fragment templates may have an impact on.

Attacks beyond permutation-based cryptography With either a fragment template attack or Cassiers et al.'s method to directly profile templates for 32-bit values [134], it is no doubt that full-state recovery technique can pose a serious threat to permutation-based cryptographic applications. However, these attacks were based on not only the full-state information provided by templates, but also on the fact that it is not very difficult to perform belief propagation following the mathematical structures of these applications. It would be a question of whether the full-state information that templates can provide is enough once we attack cryptographic applications such as ECC or other asymmetric algorithms, where the factor graphs could be far more complicated than those for permutation-based ciphers.

Attacks on registers larger than 32 bits We may expect that the fragment template attack technique can also work on devices with registers larger than 32 bits, with some compromised quality of templates, but how much the quality will drop remains unknown. I believe that it could be a good starting point to survey the situation where Keccak and Ascon are implemented on 64-bit devices, as the basic units of both these permutations are 64-bit lanes. Then, how the fragment template attack can be used to attack applications on FPGA boards could be another interesting issue, given that these boards can use operations with even more bits changing during the same clock cycles.

Attacks on other masked implementations In this thesis, I have already analyzed the case of Weatherley's first-order Boolean masked Ascon implementation. However, it is still unclear whether my fragment template attack can easily apply to attack Ascon and Keccak implementations with higher-order masking. With Bertoni et al.'s design for the non-linear function suitable for this countermeasure (see Section 1.4), we can expect that there would be more higher-order Boolean masked implementations of these permutation-based cryptographic algorithms compared to previous algorithms such as AES. For example, the official C implementations of Ascon already provide one version supporting up to third-order masking (i.e., four shares) [145]. Therefore, further surveys on this issue would be important.

Attacks via more than one side channel Some previous surveys [146, 147] show that attackers can integrate recordings from more than one side channel to extract more information. For example, Standaert et al. [147] concatenated both traces recorded from power consumption and EM signals into one hybrid trace and then applied an LDA-based template attack. It remains an interesting question whether it will significantly help to improve the quality of templates once we use such hybrid traces in a fragment template attack. Other options could involve power traces recorded from both the GND and V_{DD} sides, or from multiple GND pins.

6.3 Review

My experiments have provided evidence of how a combination of full-state template attack, belief propagation, and enumeration can pose a threat to the implementations of permutation-based cryptographic algorithms. Even if these applications are implemented on 32-bit devices with some extent of countermeasure, we can still apply the fragment template attack to extract some full-state information rather than only the HW values of the target state. Although this technique has only been evaluated in laboratory-controlled environments and could face some challenges in real-world scenarios, I still argue that we should put more attention to the potential threats from template attacks in addition to those from CPA or DPA-style non-profiling attacks when we standardize new cryptographic applications and develop their proper hardware implementations even on 32-bit devices.

Bibliography

[1] "Keccak team figures." [Online]. Available: https://keccak.team/figures.html

[2] "National Institute of Standards and Technology (NIST)." [Online]. Available: https://www.nist.gov/

[3] "Submission requirements and evaluation criteria for the post-quantum cryptography standardization process," National Institute of Standards and Technology, December 2016. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf

[4] P. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, 1994, pp. 124–134. [Online]. Available: https://doi.org/10.1109/SFCS.1994.365700

[5] J. Proos and C. Zalka, "Shor's discrete logarithm quantum algorithm for elliptic curves," 2003. [Online]. Available: https://arxiv.org/abs/quant-ph/0301141

[6] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978. [Online]. Available: https://doi.org/10.1145/359340.359342

[7] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976. [Online]. Available: https://doi.org/10.1109/TIT.1976.1055638

[8] V. S. Miller, "Use of elliptic curves in cryptography," in *Advances in Cryptology — CRYPTO '85 Proceedings*, H. C. Williams, Ed. Springer, Berlin, Heidelberg, 1986, pp. 417–426. [Online]. Available: https://doi.org/10.1007/3-540-39799-X_31

[9] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of computation*, vol. 48, no. 177, pp. 203–209, 1987.

[10] "Submission requirements and evaluation criteria for the lightweight cryptography standardization process," National Institute of Standards and Technology. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/final-lwc-submission-requirements-august2018.pdf

[11] V. D. Hunt, A. Puglia, and M. Puglia, *RFID: a guide to radio frequency identification.* John Wiley & Sons, 2007.

[12] P. C. Kocher, "Timing attacks on implementations of Diffie–Hellman, RSA, DSS, and other systems," in *Advances in Cryptology — CRYPTO '96*, N. Koblitz, Ed. Springer, Berlin, Heidelberg, 1996, pp. 104–113. [Online]. Available: https://doi.org/10.1007/3-540-68697-5_9

[13] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Advances in Cryptology — CRYPTO' 99*, M. Wiener, Ed. Springer, Berlin, Heidelberg, 1999, pp. 388–397. [Online]. Available: https://doi.org/10.1007/3-540-48405-1_25

[14] F.-X. Standaert, *Introduction to Side-Channel Attacks.* Boston, MA: Springer US, 2010, pp. 27–42. [Online]. Available: https://doi.org/10.1007/978-0-387-71829-3_2

[15] *SHA-3 standard: permutation-based hash and extendable-output functions*, NIST, Aug. 2015, FIPS PUB 202. [Online]. Available: https://doi.org/10.6028/NIST.FIPS.202

[16] E. Biham and A. Shamir, "Differential cryptanalysis of DES-like cryptosystems," *Journal of Cryptology*, vol. 4, pp. 3–72, 1991. [Online]. Available: https://doi.org/10.1007/BF00630563

[17] M. Matsui, "Linear cryptanalysis method for DES cipher," in *Advances in Cryptology — EUROCRYPT '93*, T. Helleseth, Ed. Springer Berlin Heidelberg, 1994, pp. 386–397. [Online]. Available: https://doi.org/10.1007/3-540-48285-7_33

[18] G. Bard, *Algebraic cryptanalysis.* Springer Science & Business Media, 2009.

[19] T. Messerges, E. Dabbish, and R. Sloan, "Examining smart-card security under the threat of power analysis attacks," *IEEE Transactions on Computers*, vol. 51, no. 5, pp. 541–552, 2002. [Online]. Available: https://doi.org/10.1109/TC.2002.1004593

[20] S. Mangard, E. Oswald, and T. Popp, *Power analysis attacks: Revealing the secrets of smart cards.* Springer Science & Business Media, 2008, vol. 31.

[21] K. Gandolfi, C. Mourtel, and F. Olivier, "Electromagnetic analysis: Concrete results," in *Cryptographic Hardware and Embedded Systems – CHES 2001*, Ç. K. Koç, D. Naccache, and C. Paar, Eds. Springer, Berlin, Heidelberg, 2001, pp. 251–261. [Online]. Available: https://doi.org/10.1007/3-540-44709-1_21

[22] D. Agrawal, B. Archambeault, J. R. Rao, and P. Rohatgi, "The EM side—channel(s)," in *Cryptographic Hardware and Embedded Systems – CHES 2002*, B. S. Kaliski, Ç. K. Koç, and C. Paar, Eds. Springer, Berlin, Heidelberg, 2003, pp. 29–45. [Online]. Available: https://doi.org/10.1007/3-540-36400-5_4

[23] A. O. Bauer, "Some aspects of military line communications as deployed by the German armed forces prior to 1945," in *The History of Military Communications, Proc. 5th Annual Colloquium*, 1999.

[24] P. Wright, P. Greengrass, and G. Ladjadj-Koenig, *Spycatcher.* Heinemann Melbourne, 1987.

[25] National Bureau of Standards, "Data encryption standard," 1977, federal information processing standards publication 46 (FIPS PUB 46). [Online]. Available: https://csrc.nist.gov/files/pubs/fips/46/final/docs/nbs.fips.46.pdf

[26] S. P. Skorobogatov, "Semi-invasive attacks – A new approach to hardware security analysis," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-630, Apr. 2005. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-630.pdf

[27] J. Markoff, "Potential flaw seen in cash card security," New York Times, 1996.

[28] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the importance of checking cryptographic protocols for faults," in *Advances in Cryptology — EUROCRYPT '97*, W. Fumy, Ed. Springer Berlin Heidelberg, 1997, pp. 37–51. [Online]. Available: https://doi.org/10.1007/3-540-69053-0_4

[29] E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems," in *Advances in Cryptology — CRYPTO '97*, B. S. Kaliski, Ed. Springer Berlin Heidelberg, 1997, pp. 513–525. [Online]. Available: https://doi.org/10.1007/BFb0052259

[30] P. Dusart, G. Letourneux, and O. Vivolo, "Differential fault analysis on A.E.S," in *Applied Cryptography and Network Security*, J. Zhou, M. Yung, and Y. Han, Eds. Springer Berlin Heidelberg, 2003, pp. 293–306. [Online]. Available: https://doi.org/10.1007/978-3-540-45203-4_23

[31] S. P. Skorobogatov and R. J. Anderson, "Optical fault induction attacks," in *Cryptographic Hardware and Embedded Systems - CHES 2002*, B. S. Kaliski, ç. K. Koç, and C. Paar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 2–12. [Online]. Available: https://doi.org/10.1007/3-540-36400-5_2

[32] T. Fukunaga and J. Takahashi, "Practical fault attack on a cryptographic LSI with ISO/IEC 18033-3 block ciphers," in *2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2009, pp. 84–92. [Online]. Available: https://doi.org/10.1109/FDTC.2009.34

[33] M. Hutter and J.-M. Schmidt, "The temperature side channel and heating fault attacks," in *Smart Card Research and Advanced Applications*, A. Francillon and P. Rohatgi,

Eds. Cham: Springer International Publishing, 2014, pp. 219–235. [Online]. Available: https://doi.org/10.1007/978-3-319-08302-5_15

[34] D. Genkin, A. Shamir, and E. Tromer, "RSA key extraction via low-bandwidth acoustic cryptanalysis," in *Advances in Cryptology — CRYPTO 2014*, J. A. Garay and R. Gennaro, Eds. Springer Berlin Heidelberg, 2014, pp. 444–461. [Online]. Available: https://doi.org/10.1007/978-3-662-44371-2_25

[35] K. Gandolfi, C. Mourtel, and F. Olivier, "Electromagnetic analysis: Concrete results," in *Cryptographic Hardware and Embedded Systems — CHES 2001*, Ç. K. Koç, D. Naccache, and C. Paar, Eds. Springer Berlin Heidelberg, 2001, pp. 251–261. [Online]. Available: https://doi.org/10.1007/3-540-44709-1_21

[36] J. Heyszl, S. Mangard, B. Heinz, F. Stumpf, and G. Sigl, "Localized electromagnetic analysis of cryptographic implementations," in *Topics in Cryptology – CT-RSA 2012*, O. Dunkelman, Ed. Springer Berlin Heidelberg, 2012, pp. 231–244. [Online]. Available: https://doi.org/10.1007/978-3-642-27954-6_15

[37] J. Heyszl, D. Merli, B. Heinz, F. De Santis, and G. Sigl, "Strengths and limitations of high-resolution electromagnetic field measurements for side-channel analysis," in *Smart Card Research and Advanced Applications*, S. Mangard, Ed. Springer Berlin Heidelberg, 2013, pp. 248–262. [Online]. Available: https://doi.org/10.1007/978-3-642-37288-9_17

[38] E. Brier, C. Clavier, and F. Olivier, "Correlation power analysis with a leakage model," in *Cryptographic Hardware and Embedded Systems – CHES 2004*, M. Joye and J.-J. Quisquater, Eds. Springer, Berlin, Heidelberg, 2004, pp. 16–29. [Online]. Available: https://doi.org/10.1007/978-3-540-28632-5_2

[39] S. Chari, J. R. Rao, and P. Rohatgi, "Template attacks," in *Cryptographic Hardware and Embedded Systems – CHES 2002*, B. S. Kaliski, Ç. K. Koç, and C. Paar, Eds. Springer, Berlin, Heidelberg, 2003, pp. 13–28. [Online]. Available: https://doi.org/10.1007/3-540-36400-5_3

[40] R. McEvoy, M. Tunstall, C. C. Murphy, and W. P. Marnane, "Differential power analysis of HMAC based on SHA-2, and countermeasures," in *Information Security Applications*, S. Kim, M. Yung, and H.-W. Lee, Eds. Springer Berlin Heidelberg, 2007, pp. 317–332. [Online]. Available: https://doi.org/10.1007/978-3-540-77535-5_23

[41] E. Karabulut, E. Alkim, and A. Aysu, "Single-trace side-channel attacks on ω-small polynomial sampling: With applications to NTRU, NTRU Prime, and CRYSTALS-DILITHIUM," in *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2021, pp. 35–45. [Online]. Available: https://doi.org/10.1109/HOST49136.2021.9702284

[42] M. Joye and S.-M. Yen, "The Montgomery powering ladder," in *Cryptographic Hardware and Embedded Systems — CHES 2002*, B. S. Kaliski, ç. K. Koç, and C. Paar, Eds. Springer Berlin Heidelberg, 2003, pp. 291–302. [Online]. Available: https://doi.org/10.1007/3-540-36400-5_22

[43] P. L. Montgomery, "Speeding the Pollard and elliptic curve methods of factorization," *Mathematics of Computation*, vol. 48, no. 177, pp. 243–264, 1987. [Online]. Available: https://www.ams.org/journals/mcom/1987-48-177/S0025-5718-1987-0866113-7/

[44] M. Wu, S. Guo, P. Schaumont, and C. Wang, "Eliminating timing side-channel leaks using program repair," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 15–26. [Online]. Available: https://doi.org/10.1145/3213846.3213851

[45] D. Page, "Theoretical use of cache memory as a cryptanalytic side-channel," Cryptology ePrint Archive, Paper 2002/169, 2002. [Online]. Available: https://eprint.iacr.org/2002/169

[46] J. Benesty, J. Chen, Y. Huang, and I. Cohen, *Pearson Correlation Coefficient*. Springer, Berlin, Heidelberg, 2009, pp. 1–4. [Online]. Available: https://doi.org/10.1007/978-3-642-00296-0_5

[47] J. Daemen and V. Rijmen. (1999) AES proposal: Rijndael. Accessed 9 Nov 2022. [Online]. Available: https://www.cs.miami.edu/home/burt/learning/Csc688.012/rijndael/rijndael_doc_V2.pdf

[48] M. Dworkin, E. Barker, J. Nechvatal, J. Foti, L. Bassham, E. Roback, and J. Dray, "Advanced Encryption Standard (AES)," Nov. 2001. [Online]. Available: https://doi.org/10.6028/NIST.FIPS.197

[49] M. O. Choudary and M. G. Kuhn, "Efficient stochastic methods: profiled attacks beyond 8 bits," in *International Conference on Smart Card Research and Advanced Applications.* Springer, 2014, pp. 85–103. [Online]. Available: https://doi.org/10.1007/978-3-319-16763-3_6

[50] W. Schindler, K. Lemke, and C. Paar, "A stochastic model for differential side channel cryptanalysis," in *International Workshop on Cryptographic Hardware and Embedded Systems.* Springer, 2005, pp. 30–46. [Online]. Available: https://doi.org/10.1007/11545262_3

[51] F.-X. Standaert and C. Archambeau, "Using subspace-based template attacks to compare and combine power and electromagnetic information leakages," in *International Workshop on Cryptographic Hardware and Embedded Systems.* Springer, 2008, pp. 411–425. [Online]. Available: https://doi.org/10.1007/978-3-540-85053-3_26

[52] M. O. Choudary, "Efficient multivariate statistical techniques for extracting secrets from electronic devices," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-878, Sep. 2015. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-878.pdf

[53] N. Veyrat-Charvillon, B. Gérard, M. Renauld, and F.-X. Standaert, "An optimal key enumeration algorithm and its application to side-channel attacks," in *International Conference on Selected Areas in Cryptography.* Springer, 2012, pp. 390–406. [Online]. Available: https://doi.org/10.1007/978-3-642-35999-6_25

[54] M. Renauld and F.-X. Standaert, "Algebraic side-channel attacks," in *Information Security and Cryptology*, F. Bao, M. Yung, D. Lin, and J. Jing, Eds. Springer, Berlin, Heidelberg, 2010, pp. 393–410. [Online]. Available: https://doi.org/10.1007/978-3-642-16342-5_29

[55] Y. Oren, M. Kirschbaum, T. Popp, and A. Wool, "Algebraic side-channel analysis in the presence of errors," in *Cryptographic Hardware and Embedded Systems, CHES 2010*, S. Mangard and F.-X. Standaert, Eds. Springer, Berlin, Heidelberg, 2010, pp. 428–442. [Online]. Available: https://doi.org/10.1007/978-3-642-15031-9_29

[56] N. Veyrat-Charvillon, B. Gérard, and F.-X. Standaert, "Soft analytical side-channel attacks," in *International Conference on the Theory and Application of Cryptology and Information Security.* Springer, 2014, pp. 282–296. [Online]. Available: https://doi.org/10.1007/978-3-662-45611-8_15

[57] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "Keccak," in *Advances in Cryptology – EUROCRYPT 2013*, T. Johansson and P. Q. Nguyen, Eds. Springer, Berlin, Heidelberg, 2013, pp. 313–314. [Online]. Available: https://doi.org/10.1007/978-3-642-38348-9_19

[58] ——, "Permutation-based encryption, authentication and authenticated encryption," *Directions in Authenticated Ciphers*, pp. 159–170, 2012. [Online]. Available: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=dc8231a00d1e57008743f63044821d378cc3bceb

[59] G. Bertoni, J. Daemen, S. Hoffert, M. Peeters, G. V. Assche, and R. V. Keer, "Farfalle: parallel permutation-based cryptography," Cryptology ePrint Archive, Paper 2016/1188, 2016, https://eprint.iacr.org/2016/1188. [Online]. Available: https://eprint.iacr.org/2016/1188

[60] ——, "The keyak authenticated encryption scheme," accessed July 2023. [Online]. Available: https://keccak.team/keyak.html

[61] E. Alkim *et al.*, "NewHope: Algorithm specifications and supporting documentation," 2019. [Online]. Available: https://newhopecrypto.org/

[62] J. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehle, “CRYSTALS - Kyber: a CCA-secure module-lattice-based KEM,” in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2018, pp. 353–367. [Online]. Available: https://doi.org/10.1109/EuroSP.2018.00032

[63] P.-A. Fouque, G. Leurent, D. Réal, and F. Valette, “Practical electromagnetic template attack on HMAC,” in *International Workshop on Cryptographic Hardware and Embedded Systems.* Springer, 2009, pp. 66–80. [Online]. Available: https://doi.org/10.1007/978-3-642-04138-9_6

[64] M. Taha and P. Schaumont, “Side-channel analysis of MAC-Keccak,” in *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST).* IEEE, 2013, pp. 125–130. [Online]. Available: https://doi.org/10.1109/HST.2013.6581577

[65] ——, “Differential power analysis of MAC-Keccak at any key-length,” in *Advances in Information and Computer Security*, K. Sakiyama and M. Terada, Eds. Springer, Berlin, Heidelberg, 2013, pp. 68–82. [Online]. Available: https://doi.org/10.1007/978-3-642-41383-4_5

[66] P. Luo, Y. Fei, X. Fang, A. A. Ding, D. R. Kaeli, and M. Leeser, “Side-channel analysis of MAC-Keccak hardware implementations.” *IACR Cryptology ePrint Archive*, vol. 2015, p. 411, 2015. [Online]. Available: https://eprint.iacr.org/2015/411

[67] L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehlé, “CRYSTALS-Dilithium: a lattice-based digital signature scheme,” *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2018, no. 1, pp. 238–268, Feb. 2018. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/839

[68] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer, “Ascon v1.2,” May 2021, submission to the final round of NIST lightweight cryptography standardization process. [Online]. Available: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/ascon-spec-final.pdf

[69] M. S. Turan, K. McKay, D. Chang, Ç. Çalık, L. Bassham, J. Kang, and J. Kelsey, “Status report on the second round of the NIST lightweight cryptography standardization process,” NIST, NISTIR 8369. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/ir/2021/NIST.IR.8369.pdf

[70] M. Sönmez Turan, K. McKay, D. Chang, L. E. Bassham, J. Kang, N. D. Waller, J. M. Kelsey, and D. Hong, “Status report on the final round of the NIST lightweight cryptography standardization process,” Gaithersburg, MD, 2023, NIST Interagency or Internal Report (IR) NIST IR 8454. [Online]. Available: https://doi.org/10.6028/NIST.IR.8454

[71] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer, “Cryptanalysis of Ascon,” in *Topics in Cryptology — CT-RSA 2015*, K. Nyberg, Ed. Cham: Springer International

Publishing, 2015, pp. 371–387. [Online]. Available: https://doi.org/10.1007/978-3-319-16715-2_20

[72] H. Gross, E. Wenger, C. Dobraunig, and C. Ehrenhöfer, “Ascon hardware implementations and side-channel evaluation,” *Microprocessors and Microsystems*, vol. 52, pp. 470–479, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0141933116302721

[73] L. Batina, I. R. Buhan, Ł. M. Chmielewski, E. Gunnarsdóttir, V. Jahandideh, T. Stock, and L. J. A. Weissbart, “Side-channel evaluation report on implementations of several NIST LWC finalists,” 2022. [Online]. Available: https://repository.ubn.ru.nl/handle/2066/253567

[74] P. Sasdrich, A. Moradi, and T. Güneysu, “Hiding higher-order side-channel leakage,” in *Topics in Cryptology – CT-RSA 2017*, H. Handschuh, Ed. Cham: Springer International Publishing, 2017, pp. 131–146. [Online]. Available: https://doi.org/10.1007/978-3-319-52153-4_8

[75] S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi, “Towards sound approaches to counteract power-analysis attacks,” in *Advances in Cryptology — CRYPTO’ 99*, M. Wiener, Ed. Springer, Berlin, Heidelberg, 1999, pp. 398–412. [Online]. Available: https://doi.org/10.1007/3-540-48405-1_26

[76] E. Prouff and M. Rivain, “Masking against side-channel attacks: A formal security proof,” in *Advances in Cryptology – EUROCRYPT 2013*, T. Johansson and P. Q. Nguyen, Eds. Springer, Berlin, Heidelberg, 2013, pp. 142–159. [Online]. Available: https://doi.org/10.1007/978-3-642-38348-9_9

[77] T. S. Messerges, “Securing the AES finalists against power analysis attacks,” in *Fast Software Encryption*, G. Goos, J. Hartmanis, J. van Leeuwen, and B. Schneier, Eds. Springer, Berlin, Heidelberg, 2001, pp. 150–164. [Online]. Available: https://doi.org/10.1007/3-540-44706-7_11

[78] M.-L. Akkar and C. Giraud, “An implementation of DES and AES, secure against some attacks,” in *Cryptographic Hardware and Embedded Systems — CHES 2001*, Ç. K. Koç, D. Naccache, and C. Paar, Eds. Springer Berlin Heidelberg, 2001, pp. 309–318. [Online]. Available: https://doi.org/10.1007/3-540-44709-1_26

[79] E. Trichina, D. De Seta, and L. Germani, “Simplified adaptive multiplicative masking for AES,” in *Cryptographic Hardware and Embedded Systems — CHES 2002*, B. S. Kaliski, ç. K. Koç, and C. Paar, Eds. Springer Berlin Heidelberg, 2003, pp. 187–197. [Online]. Available: https://doi.org/10.1007/3-540-36400-5_15

[80] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, “Building power analysis resistant implementations of keccak,” in *Second SHA-3 candidate conference*, vol. 142. Citeseer,

2010. [Online]. Available: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=fe3d80a12e34d67ce14d438935302c6ef371901c

[81] T. S. Messerges, "Using second-order power analysis to attack DPA resistant software," in *Cryptographic Hardware and Embedded Systems — CHES 2000*, Ç. K. Koç and C. Paar, Eds. Springer, Berlin, Heidelberg, 2000, pp. 238–251. [Online]. Available: https://doi.org/10.1007/3-540-44499-8_19

[82] J. Waddle and D. Wagner, "Towards efficient second-order power analysis," in *Cryptographic Hardware and Embedded Systems – CHES 2004*, M. Joye and J.-J. Quisquater, Eds. Springer, Berlin, Heidelberg, 2004, pp. 1–15. [Online]. Available: https://doi.org/10.1007/978-3-540-28632-5_1

[83] E. Oswald and S. Mangard, "Template attacks on masking—resistance is futile," in *Topics in Cryptology – CT-RSA 2007*, M. Abe, Ed. Springer, Berlin, Heidelberg, 2006, pp. 243–256. [Online]. Available: https://doi.org/10.1007/11967668_16

[84] L. Lerman and O. Markowitch, "Efficient profiled attacks on masking schemes," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 6, pp. 1445–1454, 2019. [Online]. Available: https://doi.org/10.1109/TIFS.2018.2879295

[85] R. Gilmore, N. Hanley, and M. O'Neill, "Neural network based attack on a masked implementation of aes," in *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2015, pp. 106–111. [Online]. Available: https://doi.org/10.1109/HST.2015.7140247

[86] J. Kim, S. Picek, A. Heuser, S. Bhasin, and A. Hanjalic, "Make some noise. unleashing the power of convolutional neural networks for profiled side-channel analysis," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2019, no. 3, p. 148–179, May 2019. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8292

[87] V. Grosso and F.-X. Standaert, "Masking proofs are tight and how to exploit it in security evaluations," in *Advances in Cryptology – EUROCRYPT 2018*, J. B. Nielsen and V. Rijmen, Eds. Springer, Cham, 2018, pp. 385–412. [Online]. Available: https://doi.org/10.1007/978-3-319-78375-8_13

[88] O. Bronchain and F.-X. Standaert, "Breaking masked implementations with many shares on 32-bit software platforms: or when the security order does not matter," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2021, no. 3, p. 202–234, July 2021. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8973

[89] M. J. Kannwischer, P. Pessl, and R. Primas, "Single-trace attacks on Keccak," Cryptology ePrint Archive, Paper 2020/371, 2020, https://eprint.iacr.org/2020/371. [Online]. Available: https://eprint.iacr.org/2020/371

[90] S.-C. You and M. G. Kuhn, "A template attack to reconstruct the input of SHA-3 on an 8-bit device," in *Constructive Side-Channel Analysis and Secure Design*, G. M. Bertoni and F. Regazzoni, Eds. Springer, Cham, 2021, pp. 25–42. [Online]. Available: https://doi.org/10.1007/978-3-030-68773-1_2

[91] ——, "Single-trace fragment template attack on a 32-bit implementation of Keccak," in *Smart Card Research and Advanced Applications*, V. Grosso and T. Pöppelmann, Eds. Springer, Cham, 2022, pp. 3–23. [Online]. Available: https://doi.org/10.1007/978-3-030-97348-3_1

[92] S.-C. You, M. G. Kuhn, S. Sarkar, and F. Hao, "A template attack on Ascon AEAD," CHES 2022 poster 23. [Online]. Available: https://ches.iacr.org/2022/acceptedposters.php

[93] ——, "Low trace-count template attacks on 32-bit implementations of Ascon AEAD," pre-print version. [Online]. Available: https://www.cl.cam.ac.uk/~scy27/ches2023-ascon.pdf

[94] D. Oswald and C. Paar, "Breaking Mifare DESFire MF3ICD40: power analysis and templates in the real world," in *International Workshop on Cryptographic Hardware and Embedded Systems.* Springer, 2011, pp. 207–222. [Online]. Available: https://doi.org/10.1007/978-3-642-23951-9_14

[95] O. Choudary and M. G. Kuhn, "Efficient template attacks," in *International Conference on Smart Card Research and Advanced Applications.* Springer, 2013, pp. 253–270. [Online]. Available: https://doi.org/10.1007/978-3-319-08302-5_17

[96] M. O. Choudary and M. G. Kuhn, "Efficient, portable template attacks," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 2, pp. 490–501, Feb. 2018. [Online]. Available: https://doi.org/10.1109/TIFS.2017.2757440

[97] I. T. Jolliffe and J. Cadima, "Principal component analysis: a review and recent developments," *Philosophical transactions of the royal society A: Mathematical, Physical and Engineering Sciences*, vol. 374, no. 2065, p. 20150202, 2016. [Online]. Available: https://royalsocietypublishing.org/doi/full/10.1098/rsta.2015.0202

[98] S. Wold, K. Esbensen, and P. Geladi, "Principal component analysis," *Chemometrics and Intelligent Laboratory Systems*, vol. 2, no. 1, pp. 37–52, 1987, proceedings of the Multivariate Statistical Workshop for Geologists and Geochemists. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0169743987800849

[99] F.-X. Standaert, T. G. Malkin, and M. Yung, “A unified framework for the analysis of side-channel key recovery attacks,” in *Annual International Conference on the Theory and Applications of Cryptographic Techniques.* Springer, Berlin, Heidelberg, 2009, pp. 443–461. [Online]. Available: https://doi.org/10.1007/978-3-642-01001-9_26

[100] H. Maghrebi, T. Portigliatti, and E. Prouff, “Breaking cryptographic implementations using deep learning techniques,” in *Security, Privacy, and Applied Cryptography Engineering*, C. Carlet, M. A. Hasan, and V. Saraswat, Eds. Springer, Cham, 2016, pp. 3–26. [Online]. Available: https://doi.org/10.1007/978-3-319-49445-6_1

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

[102] “Extended Keccak code package,” accessed April 2019, `lib/low/KeccakP-1600/Compact64/KeccakP-1600-compact64.c`. [Online]. Available: https://github.com/XKCP/XKCP

[103] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, “The Keccak reference,” SHA-3 competition (Round 3), Jan. 2011. [Online]. Available: https://keccak.team/files/Keccak-reference-3.0.pdf

[104] “KeccakTools.” [Online]. Available: https://github.com/KeccakTeam/KeccakTools

[105] *NI PXIe-5160.* [Online]. Available: http://www.ni.com/en-gb/support/model.pxie-5160.html

[106] *NI PXIe-5423.* [Online]. Available: http://www.ni.com/en-gb/support/model.pxie-5423.html

[107] *NI PXI-4110.* [Online]. Available: http://www.ni.com/en-gb/support/model.pxi-4110.html

[108] *ATxmega256A3U*, Microchip, accessed February 2020. [Online]. Available: https://www.microchip.com/wwwproducts/en/atxmega256a3u

[109] “CW1173: ChipWhisperer-Lite,” accessed February 2018. [Online]. Available: https://media.newae.com/datasheets/NAE-CW1173_datasheet.pdf

[110] “ChipWhisperer-Lite arm edition,” schematic REV-03, commit `e2adf19` on 2 December 2019. [Online]. Available: https://github.com/newaetech/chipwhisperer/blob/develop/hardware/capture/chipwhisperer-lite-32bit/cw-lite-arm-main.pdf

[111] “Intel® Xeon® Gold 5218 processor,” accessed July 2023. [Online]. Available: https://www.intel.com/content/www/us/en/products/sku/192444/intel-xeon-gold-5218-processor-22m-cache-2-30-ghz/specifications.html

[112] "NumPy 1.21.5 project description," December 2021. [Online]. Available: https://pypi.org/project/numpy/1.21.5/

[113] "NumPy v1.21 manual," June 2021. [Online]. Available: https://numpy.org/doc/1.21/

[114] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011. [Online]. Available: https://www.jmlr.org/papers/volume12/pedregosa11a/pedregosa11a.pdf?ref=https:/

[115] "Scikit-learn – machine learning in Python." [Online]. Available: https://scikit-learn.org/stable/index.html

[116] "Scikit-learn 1.1.3 project description," Otcober 2022. [Online]. Available: https://pypi.org/project/scikit-learn/1.1.3/

[117] "Intel® oneAPIi math kernel library." [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html

[118] B. Gierlichs, K. Lemke-Rust, and C. Paar, "Templates vs. stochastic methods," in *Cryptographic Hardware and Embedded Systems — CHES 2006*, L. Goubin and M. Matsui, Eds. Springer Berlin Heidelberg, 2006, pp. 15–29. [Online]. Available: https://doi.org/10.1007/11894063_2

[119] B. J. Gilbert Goodwill, J. Jaffe, P. Rohatgi *et al.*, "A testing methodology for side-channel resistance validation," in *NIST non-invasive attack testing workshop*, vol. 7, 2011, pp. 115–136.

[120] S. Mangard, "Hardware countermeasures against DPA – a statistical analysis of their effectiveness," in *Topics in Cryptology – CT-RSA 2004.* Springer, Berlin, Heidelberg, 2004. [Online]. Available: https://doi.org/10.1007/978-3-540-24660-2_18

[121] S. Bhasin, J.-L. Danger, S. Guilley, and Z. Najm, "NICV: normalized inter-class variance for detection of side-channel leakage," in *2014 International Symposium on Electromagnetic Compatibility, Tokyo*, 2014, pp. 310–313. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6997167

[122] H. Akoglu, "User's guide to correlation coefficients," *Turkish Journal of Emergency Medicine*, vol. 18, no. 3, pp. 91–93, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2452247318302164

[123] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, Oct 2011. [Online]. Available: https://www.jmlr.org/papers/volume12/pedregosa11a/pedregosa11a.pdf

[124] M. Pretti, "A message-passing algorithm with damping," *Journal of Statistical Mechanics: Theory and Experiment*, vol. 2005, no. 11, p. P11008, nov 2005. [Online]. Available: https://dx.doi.org/10.1088/1742-5468/2005/11/P11008

[125] J. Kelsey, S. Chang, and R. Perlner, "SHA-3 derived functions: cSHAKE, KMAC, TupleHash and ParallelHash," December 2016, NIST Special Publication 800-185. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-185.pdf

[126] D. Bellizia, O. Bronchain, G. Cassiers, V. Grosso, C. Guo, C. Momin, O. Pereira, T. Peters, and F.-X. Standaert, "Mode-level vs. implementation-level physical security in symmetric cryptography," in *Advances in Cryptology – CRYPTO 2020*. Cham: Springer International Publishing, 2020, pp. 369–400. [Online]. Available: https://doi.org/10.1007/978-3-030-56784-2_13

[127] C. Verhamme, G. Cassiers, and F.-X. Standaert, "Analyzing the leakage resistance of the NIST's lightweight crypto competition's finalists," in *Smart Card Research and Advanced Applications*, I. Buhan and T. Schneider, Eds. Cham: Springer International Publishing, 2023, pp. 290–308. [Online]. Available: https://doi.org/10.1007/978-3-031-25319-5_15

[128] R. Primas, P. Pessl, and S. Mangard, "Single-trace side-channel attacks on masked lattice-based encryption," in *Cryptographic Hardware and Embedded Systems – CHES 2017*, W. Fischer and N. Homma, Eds. Springer, Cham, 2017, pp. 513–533. [Online]. Available: https://doi.org/10.1007/978-3-319-66787-4_25

[129] S. Luo, W. Wu, Y. Li, R. Zhang, and Z. Liu, "An efficient soft analytical side-channel attack on Ascon," in *Wireless Algorithms, Systems, and Applications*, L. Wang, M. Segal, J. Chen, and T. Qiu, Eds. Springer, Cham, 2022, pp. 389–400. [Online]. Available: https://doi.org/10.1007/978-3-031-19208-1_32

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

[131] "ARMv7-M architecture reference manual," Arm Limited, ARM DDI 0403E.e (ID021621). [Online]. Available: https://developer.arm.com/documentation/ddi0403/latest/

[132] D. J. Bernstein, "ChaCha, a variant of Salsa20," in *Workshop record of SASC 2008*. ECRYPT, 2008, pp. 273–278. [Online]. Available: https://cr.yp.to/chacha/chacha-20080120.pdf

[133] M. Spyropoulos, "Side-channel attacks on words longer than 8 bits," 2022.

[134] G. Cassiers, H. Devillez, F.-X. Standaert, and B. Udvarhelyi, "Efficient regression-based linear discriminant analysis for side-channel security evaluations," accepted paper for CHES—2023. [Online]. Available: https://perso.cassiersg.be/papers/lrda.pdf

[135] C. Dobraunig, M. Eichlseder, S. Mangard, F. Mendel, B. Mennink, R. Primas, and T. Unterluggauer, "Isap v2.0," *IACR Transactions on Symmetric Cryptology*, vol. 2020, no. S1, p. 390–416, Jun. 2020. [Online]. Available: https://doi.org/10.13154/tosc.v2020.iS1.390-416

[136] M. McDonnell, "Box-filtering techniques," *Computer Graphics and Image Processing*, vol. 17, no. 1, pp. 65–70, 1981. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0146664X81800093

[137] C. E. Duchon, "Lanczos filtering in one and two dimensions," *Journal of Applied Meteorology and Climatology*, vol. 18, no. 8, pp. 1016–1022, 1979. [Online]. Available: https://journals.ametsoc.org/view/journals/apme/18/8/1520-0450_1979_018_1016_lfioat_2_0_co_2.xml

[138] T. K. Roy and M. Morshed, "Performance analysis of low pass fir filters design using kaiser, gaussian and tukey window function methods," in *2013 2nd International Conference on Advances in Electrical Engineering (ICAEE)*, 2013, pp. 1–6. [Online]. Available: https:/doi.org/10.1109/ICAEE.2013.6750294

[139] T. Schneider and A. Moradi, "Leakage assessment methodology," in *Cryptographic Hardware and Embedded Systems — CHES 2015*, T. Güneysu and H. Handschuh, Eds. Springer Berlin Heidelberg, 2015, pp. 495–513. [Online]. Available: https://doi.org/10.1007/978-3-662-48324-4_25

[140] J. G. J. van Woudenberg, M. F. Witteman, and B. Bakker, "Improving differential power analysis by elastic alignment," in *Topics in Cryptology – CT-RSA 2011*, A. Kiayias, Ed. Springer Berlin Heidelberg, 2011, pp. 104–119. [Online]. Available: https://doi.org/10.1007/978-3-642-19074-2_8

[141] M. A. Elaabid and S. Guilley, "Portability of templates," *Journal of Cryptographic Engineering*, vol. 2, pp. 63–74, 2012. [Online]. Available: https://doi.org/10.1007/s13389-012-0030-6

[142] M. O. Choudary and M. G. Kuhn, "Efficient, portable template attacks," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 2, pp. 490–501, 2018. [Online]. Available: https://doi.org/10.1109/TIFS.2017.2757440

[143] O. Choudary and M. G. Kuhn, "Template attacks on different devices," in *Constructive Side-Channel Analysis and Secure Design*, E. Prouff, Ed. Cham: Springer International Publishing, 2014, pp. 179–198. [Online]. Available: https://doi.org/10.1007/978-3-319-10175-0_13

[144] U. Rioja, L. Batina, and I. Armendariz, "When similarities among devices are taken for granted: Another look at portability," in *Progress in Cryptology — AFRICACRYPT*

2020, A. Nitaj and A. Youssef, Eds. Cham: Springer International Publishing, 2020, pp. 337–357. [Online]. Available: https://doi.org/10.1007/978-3-030-51938-4_17

[145] C. Dobraunig and M. Schläffer, "Reference, highly optimized, masked C and ASM implementations of Ascon," November 2022, 32-bit masked bit-interleaved ARMv6. [Online]. Available: https://github.com/ascon/ascon-c/tree/29ef7a20a7372bd47fe7f4c92861e58e49cdce94/crypto_aead/ascon128v12/protected_bi32_armv6

[146] D. Agrawal, J. R. Rao, and P. Rohatgi, "Multi-channel attacks," in *Cryptographic Hardware and Embedded Systems - CHES 2003*, C. D. Walter, Ç. K. Koç, and C. Paar, Eds. Springer Berlin Heidelberg, 2003, pp. 2–16. [Online]. Available: https://doi.org/10.1007/978-3-540-45238-6_2

[147] F.-X. Standaert and C. Archambeau, "Using subspace-based template attacks to compare and combine power and electromagnetic information leakages," in *Cryptographic Hardware and Embedded Systems – CHES 2008*, E. Oswald and P. Rohatgi, Eds. Springer Berlin Heidelberg, 2008, pp. 411–425. [Online]. Available: https://doi.org/10.1007/978-3-540-85053-3_26

Appendix A

Implementation notes

A.1 End-of-state management in secret enumeration

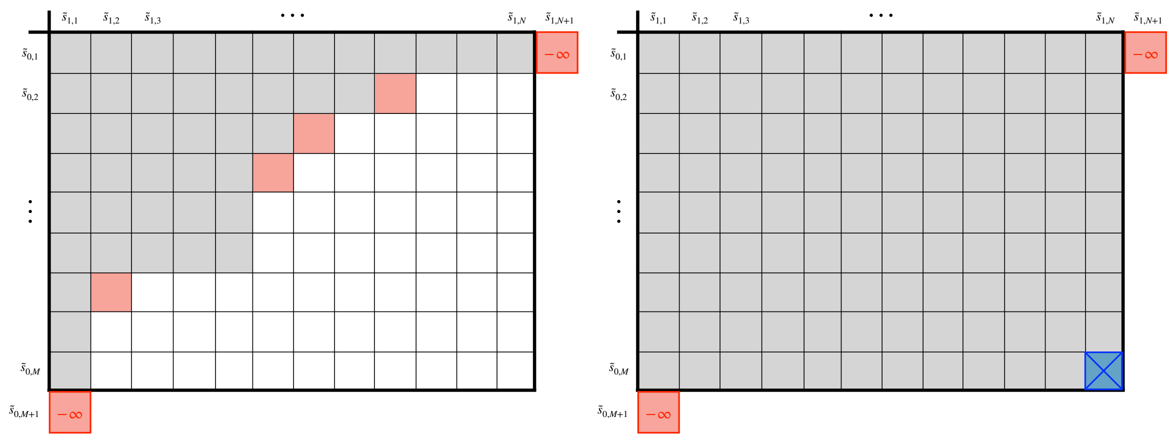


Figure A.1: The dummy combinations occupy the most top-right and the most bottom-left blocks outside the original $M \times N$ array (left), which will only be reached once the last combination, labeled in blue, is enumerated (right).

When I applied the key enumeration algorithm (see Section 2.2) in my experiments, one problem I faced was how to deal with the marginal condition when the enumeration reaches the bottom of some tables. This may happen when we have short ranking tables in table nodes, e.g. combining secrets bit-by-bit and therefore having tables with only two candidates. My solution is to add a dummy element at the end of the ranking tables. Such a dummy includes a value that is never a candidate, e.g. -1 in my case, and a logarithmic probability value $-\infty$, which is a valid IEEE floating-point number. Once reaching the bottom of its ranking table, the table node will start to return only the dummy element from this function call.

When either child table node returns a dummy element, it will naturally create a dummy element with logarithmic probability value $-\infty$, which occupies either the most top-right or most bottom-left corner as depicted in Figure A.1. These dummy elements will later never be reached before all the $M \times N$ combinations are enumerated, since no possible logarithmic probability values will be smaller than $-\infty$. Once either of them is reached, we consequently know that the enumeration in this combining node ends, and it will start to return a dummy element with a combination of $(-1, -1)$ and logarithmic probability value $-\infty$ to its parent node from this function call so that the parent node can apply similar marginal condition management.

As I always use ranking tables containing logarithmic likelihood values in this enumeration procedure, we need to convert a probability table into a logarithmic likelihood table, and then a ranking table if we obtain one from a previous procedure, such as belief propagation. Here we may meet another marginal condition when we convert those values equal to 0 into a logarithmic scale. I decide to assign their converted value to be -745.134, where the value of $\mathrm{e}^{-745.134}$ calculated by `NumPy` [113] is slightly smaller than the smallest non-zero positive number that a 64-bit floating-point number can reach. This helps me to distinguish the case of the logarithmic value of a zero from the $-\infty$ in my dummy.

Appendix B

Supporting tables and figures

B.1 Lookup tables and algorithms for Keccak and Ascon

Table B.1: **RCTable**[Ω] (in big endianness)

Ω	Constant	Ω	Constant	Ω	Constant
0	`0x0000000000000001`	8	`0x000000000000008A`	16	`0x8000000000008002`
1	`0x0000000000008082`	9	`0x0000000000000088`	17	`0x8000000000000080`
2	`0x800000000000808A`	10	`0x0000000080008009`	18	`0x000000000000800A`
3	`0x8000000080008000`	11	`0x000000008000000A`	19	`0x800000008000000A`
4	`0x000000000000808B`	12	`0x000000008000808B`	20	`0x8000000080008081`
5	`0x0000000080000001`	13	`0x800000000000008B`	21	`0x8000000000008080`
6	`0x8000000080008081`	14	`0x8000000000008089`	22	`0x0000000080000001`
7	`0x8000000000008009`	15	`0x8000000000008003`	23	`0x8000000080008008`

Table B.2: The substitution table for step χ

Input					Output				
I_0	I_1	I_2	I_3	I_4	O_0	O_1	O_2	O_3	O_4
0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	1	0	1
0	0	0	1	0	0	1	0	1	0
0	0	0	1	1	0	1	0	1	1
0	0	1	0	0	1	0	1	0	0
0	0	1	0	1	1	0	0	0	1
0	0	1	1	0	1	0	1	1	0
0	0	1	1	1	1	0	1	1	1
0	1	0	0	0	0	1	0	0	1
0	1	0	0	1	0	1	1	0	0
0	1	0	1	0	0	0	0	1	1
0	1	0	1	1	0	0	0	1	0
0	1	1	0	0	0	1	1	0	1
0	1	1	0	1	0	1	0	0	0
0	1	1	1	0	0	1	1	1	1
0	1	1	1	1	0	1	1	1	0
1	0	0	0	0	1	0	0	1	0
1	0	0	0	1	1	0	1	0	1
1	0	0	1	0	1	1	0	0	0
1	0	0	1	1	1	1	0	1	1
1	0	1	0	0	0	0	1	1	0
1	0	1	0	1	0	0	0	0	1
1	0	1	1	0	0	0	1	0	0
1	0	1	1	1	0	0	1	1	1
1	1	0	0	0	1	1	0	1	0
1	1	0	0	1	1	1	1	0	1
1	1	0	1	0	1	0	0	0	0
1	1	0	1	1	1	0	0	1	1
1	1	1	0	0	1	1	1	1	0
1	1	1	0	1	1	1	0	0	1
1	1	1	1	0	1	1	1	0	0
1	1	1	1	1	1	1	1	1	1

Table B.3: The substitution table for step p_S

Input					Output				
I_0	I_1	I_2	I_3	I_4	O_0	O_1	O_2	O_3	O_4
0	0	0	0	0	0	0	1	0	0
0	0	0	0	1	0	1	0	1	1
0	0	0	1	0	1	1	1	1	1
0	0	0	1	1	1	0	1	0	0
0	0	1	0	0	1	1	0	1	0
0	0	1	0	1	1	0	1	0	1
0	0	1	1	0	0	1	0	0	1
0	0	1	1	1	0	0	0	1	0
0	1	0	0	0	1	1	0	1	1
0	1	0	0	1	0	0	1	0	1
0	1	0	1	0	0	1	0	0	0
0	1	0	1	1	1	0	0	1	0
0	1	1	0	0	1	1	1	0	1
0	1	1	0	1	0	0	0	1	1
0	1	1	1	0	0	0	1	1	0
0	1	1	1	1	1	1	1	0	0
1	0	0	0	0	1	1	1	1	0
1	0	0	0	1	1	0	0	1	1
1	0	0	1	0	0	0	1	1	1
1	0	0	1	1	0	1	1	1	0
1	0	1	0	0	0	0	0	0	0
1	0	1	0	1	0	1	1	0	1
1	0	1	1	0	1	0	0	0	1
1	0	1	1	1	1	1	0	0	0
1	1	0	0	0	1	0	0	0	0
1	1	0	0	1	0	1	1	0	0
1	1	0	1	0	0	0	0	0	1
1	1	0	1	1	1	1	0	0	1
1	1	1	0	0	1	0	1	1	0
1	1	1	0	1	0	1	0	1	0
1	1	1	1	0	0	1	1	1	1
1	1	1	1	1	1	0	1	1	1

Algorithm 7 Ascon-128 encryption procedure

procedure Enc(K, N, A, P)
  Parameter :
  $|K| = 128$
  $r = 64$
  $a = 12$
  $b = 6$
  $IV =$ `0x80400c0600000000`
  $S_r \| S_c = S \leftarrow p^a(IV \| K \| N) \oplus (0^{320-|K|} \| K)$ ▷ Initialization
  if $|A| > 0$ **then** ▷ Processing associated data
    $A_1, \ldots, A_s \leftarrow A \| 1 \| 0^{r-1-(|A| \bmod r)}$
    for $\tau = 1 \ldots s$ **do**
      $S_r \| S_c \leftarrow S \leftarrow p^b((S_r \oplus A_\tau) \| S_c)$
    end for
  end if
  $S \leftarrow S \oplus (0^{319} \| 1)$
  $P_1, \ldots, P_t \leftarrow P \| 1 \| 0^{r-1-(|P| \bmod r)}$ ▷ Processing plaintext
  for $\tau = 1 \ldots t$ **do**
    $C_\tau \leftarrow S_r \oplus P_\tau$
    $S_r \| S_c \leftarrow S \leftarrow C_\tau \| S_c$
    if $\tau == t$ **then**
      break
    end if
    $S_r \| S_c \leftarrow S \leftarrow p^b(S_r \| S_c)$
  end for
  $C \leftarrow$ **Trunc**$(C_1 \| \ldots \| C_t, |P|)$
  $S_c \leftarrow S_c \oplus (K \| 0^{320-r-|K|})$ ▷ Finalization
  $S \leftarrow p^a(S_r \| S_c)$
  $T \leftarrow S[(320 - |K|) : 320] \oplus K$
  return C, T
end procedure

Algorithm 8 Ascon-128 decryption procedure

```
procedure Dec(K, N, A, C, T)
    Parameter :
    |K| = 128
    r = 64
    a = 12
    b = 6
    IV = 0x80400c0600000000
    S_r||S_c = S ← p^a(IV||K||N) ⊕ (0^{320−|K|}||K)          ▷ Initialization
    if |A| > 0 then                                          ▷ Processing associated data
        A_1, ..., A_s ← A||1||0^{r−1−(|A| mod r)}
        for τ = 1 ... s do
            S_r||S_c ← S ← p^b((S_r ⊕ A_τ)||S_c)
        end for
    end if
    S ← S ⊕ (0^{319}||1)
    C_1, ..., C_t ← C||1||0^{r−1−(|C| mod r)}                 ▷ Processing ciphertext
    for τ = 1 ... t do
        P_τ ← S_r ⊕ C_τ
        S_r||S_c ← S ← C_τ||S_c
        if τ == t then
            break
        end if
        S_r||S_c ← S ← p^b(S_r||S_c)
    end for
    P ← Trunc(P_1|| ... ||P_t, |C|)
    S_c ← S_c ⊕ (K||0^{320−r−|K|})                           ▷ Finalization
    S ← p^a(S_r||S_c)
    T' ← S[(320 − |K|) : 320] ⊕ K
    if T' == T then
        return P
    else
        reject decryption
    end if
end procedure
```

B.2 Data for the Keccak experiments on the 8-bit device

Table B.4: Success rates on $\alpha'_0[i,j,{}^{\mathbf{8}}k]^{\mathbf{8}}$ (left) and $\beta_0[i,j,{}^{\mathbf{8}}k]^{\mathbf{8}}$ (right). The rates for α_1 (omitted here) look similar to those for α'_0.

(i,j) \ ${}^{\mathbf{8}}k$	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) \ ${}^{\mathbf{8}}k$	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 B.5: Guessing entropy on $\alpha'_0[i,j,{}^{\mathbf{8}}k]^{\mathbf{8}}$ (left) and $\beta_0[i,j,{}^{\mathbf{8}}k]^{\mathbf{8}}$ (right). The entropy for α_1 (omitted here) look similar to those for α'_0.

(i,j) \ ${}^{\mathbf{8}}k$	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) \ ${}^{\mathbf{8}}k$	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

B.3 Data for the XOR experiments on the 32-bit device

Table B.6: Success rates on every nibble in $\hat{\alpha}'_0[i, j, {}^{\mathbf{4}}k]^{\mathbf{4}}$

(i, j) \ ${}^4k$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
(0, 0)	0.847	0.882	0.743	0.820	0.499	0.626	0.553	0.658	0.489	0.505	0.524	0.568	0.890	0.881	0.819	0.833
(1, 0)	0.990	0.997	0.995	0.979	0.903	0.982	0.931	0.981	0.968	0.985	0.990	0.953	0.998	0.994	0.984	0.986
(2, 0)	0.991	0.998	0.968	0.967	0.891	0.967	0.919	0.984	0.948	0.987	0.978	0.891	0.975	0.967	0.987	0.964
(3, 0)	0.998	0.999	0.992	0.991	0.969	0.961	0.749	0.720	0.750	0.736	0.829	0.812	0.935	0.937	0.906	0.942
(4, 0)	1.000	0.998	0.990	0.996	0.654	0.758	0.624	0.743	0.596	0.685	0.750	0.841	0.923	0.954	0.986	0.988
(0, 1)	0.998	1.000	1.000	0.999	0.960	0.998	0.985	0.939	0.834	0.905	0.872	0.815	0.986	0.973	0.971	0.979
(1, 1)	0.991	0.999	0.995	0.997	0.981	0.912	0.894	0.877	0.850	0.801	0.897	0.923	0.989	0.976	0.978	0.985
(2, 1)	0.995	0.999	0.896	0.872	0.773	0.765	0.782	0.807	0.649	0.743	0.867	0.873	0.985	0.977	0.998	0.992
(3, 1)	0.990	0.993	0.990	0.966	0.857	0.897	0.875	0.881	0.978	0.977	0.944	0.923	0.967	0.975	0.935	0.970
(4, 1)	0.992	0.996	0.984	0.951	0.775	0.865	0.847	0.863	0.830	0.891	0.868	0.922	0.994	0.998	1.000	1.000
(0, 2)	0.991	0.995	0.831	0.916	0.795	0.834	0.767	0.747	0.649	0.640	0.716	0.667	1.000	0.993	1.000	1.000
(1, 2)	0.997	1.000	0.990	0.984	0.743	0.849	0.648	0.714	0.578	0.674	0.778	0.793	0.986	0.991	0.993	0.965
(2, 2)	0.970	0.943	0.993	0.977	0.941	0.969	0.869	0.800	0.729	0.647	0.701	0.607	0.968	0.935	0.986	0.933
(3, 2)	0.981	0.990	0.815	0.737	0.627	0.591	0.548	0.517	0.571	0.508	0.668	0.550	0.959	0.937	0.987	0.990
(4, 2)	0.998	0.991	0.995	0.977	0.869	0.730	0.760	0.633	0.713	0.640	0.747	0.710	0.987	0.981	0.993	0.994
(0, 3)	1.000	1.000	0.999	0.998	0.957	0.999	0.931	0.887	0.797	0.848	0.832	0.805	0.993	0.976	0.968	0.966
(1, 3)	0.991	0.996	0.997	0.990	0.917	0.994	0.980	0.976	0.959	0.960	0.946	0.952	0.996	0.988	0.984	0.998
(2, 3)	1.000	0.998	0.924	0.903	0.834	0.771	0.740	0.659	0.684	0.638	0.684	0.654	0.991	0.974	0.999	0.999
(3, 3)	0.999	0.998	0.999	0.998	0.666	0.700	0.611	0.683	0.527	0.642	0.760	0.810	0.964	0.938	0.986	0.976
(4, 3)	0.913	0.859	0.867	0.794	0.743	0.715	0.759	0.662	0.811	0.798	0.938	0.941	0.996	0.996	0.982	0.973
(0, 4)	1.000	1.000	0.998	0.999	0.786	0.874	0.779	0.826	0.784	0.818	0.907	0.956	0.999	0.999	1.000	1.000
(1, 4)	0.999	0.996	0.997	0.997	0.849	0.896	0.830	0.853	0.839	0.882	0.964	0.946	1.000	1.000	0.981	0.945
(2, 4)	0.993	0.991	0.996	0.993	0.902	0.938	0.986	0.944	0.956	0.955	0.714	0.734	0.979	0.952	0.981	0.949
(3, 4)	0.991	0.975	0.924	0.885	0.907	0.874	0.979	0.941	0.966	0.965	0.893	0.771	0.978	0.913	0.980	0.945
(4, 4)	0.993	0.973	0.901	0.868	0.916	0.871	0.904	0.908	0.802	0.660	0.790	0.734	0.997	0.997	1.000	0.997

Table B.7: Guessing entropy of every nibble in $\hat{\alpha}'_0[i, j, {}^{\mathbf{4}}k]^{\mathbf{4}}$

(i, j) \ ${}^4k$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
(0, 0)	1.207	1.129	1.418	1.238	2.296	1.687	2.064	1.587	2.392	2.237	2.148	1.964	1.143	1.165	1.258	1.237
(1, 0)	1.010	1.003	1.005	1.022	1.135	1.022	1.081	1.019	1.036	1.017	1.013	1.057	1.002	1.007	1.019	1.015
(2, 0)	1.009	1.002	1.036	1.034	1.121	1.037	1.090	1.016	1.064	1.015	1.024	1.141	1.026	1.037	1.016	1.036
(3, 0)	1.002	1.001	1.008	1.010	1.032	1.044	1.384	1.453	1.376	1.439	1.209	1.256	1.070	1.070	1.113	1.062
(4, 0)	1.000	1.002	1.010	1.004	1.675	1.329	1.750	1.382	1.797	1.499	1.392	1.226	1.093	1.047	1.016	1.012
(0, 1)	1.002	1.000	1.000	1.001	1.052	1.002	1.018	1.074	1.221	1.126	1.181	1.312	1.016	1.030	1.035	1.025
(1, 1)	1.010	1.001	1.005	1.003	1.019	1.122	1.140	1.157	1.211	1.294	1.148	1.085	1.014	1.027	1.026	1.015
(2, 1)	1.005	1.001	1.110	1.166	1.317	1.361	1.347	1.306	1.655	1.438	1.200	1.172	1.015	1.024	1.002	1.008
(3, 1)	1.010	1.008	1.010	1.040	1.178	1.115	1.150	1.124	1.022	1.024	1.060	1.088	1.036	1.025	1.079	1.030
(4, 1)	1.008	1.004	1.016	1.050	1.345	1.162	1.221	1.170	1.253	1.137	1.182	1.081	1.006	1.002	1.000	1.000
(0, 2)	1.010	1.005	1.211	1.099	1.313	1.222	1.349	1.423	1.621	1.714	1.490	1.656	1.000	1.009	1.000	1.000
(1, 2)	1.003	1.000	1.010	1.016	1.423	1.195	1.674	1.439	2.009	1.525	1.349	1.278	1.018	1.009	1.007	1.035
(2, 2)	1.030	1.070	1.009	1.025	1.067	1.033	1.195	1.313	1.455	1.729	1.502	1.820	1.033	1.078	1.020	1.067
(3, 2)	1.021	1.010	1.283	1.475	1.756	1.879	2.050	2.184	2.001	2.113	1.633	2.117	1.045	1.074	1.013	1.010
(4, 2)	1.002	1.010	1.006	1.029	1.148	1.491	1.369	1.802	1.480	1.769	1.376	1.536	1.017	1.023	1.007	1.008
(0, 3)	1.000	1.000	1.001	1.002	1.048	1.001	1.073	1.145	1.316	1.221	1.256	1.319	1.009	1.029	1.035	1.034
(1, 3)	1.010	1.004	1.003	1.011	1.100	1.006	1.021	1.027	1.047	1.045	1.060	1.056	1.004	1.012	1.016	1.002
(2, 3)	1.000	1.002	1.082	1.132	1.227	1.407	1.389	1.653	1.590	1.663	1.533	1.694	1.009	1.027	1.001	1.001
(3, 3)	1.001	1.002	1.001	1.002	1.641	1.493	1.758	1.532	2.132	1.680	1.391	1.292	1.046	1.080	1.015	1.028
(4, 3)	1.114	1.196	1.177	1.308	1.438	1.530	1.390	1.649	1.264	1.380	1.086	1.072	1.005	1.005	1.019	1.028
(0, 4)	1.000	1.000	1.002	1.001	1.316	1.161	1.360	1.224	1.347	1.234	1.113	1.051	1.001	1.001	1.000	1.000
(1, 4)	1.001	1.004	1.003	1.003	1.210	1.129	1.251	1.209	1.211	1.165	1.040	1.056	1.000	1.000	1.021	1.062
(2, 4)	1.007	1.010	1.004	1.007	1.137	1.072	1.015	1.067	1.053	1.054	1.455	1.415	1.021	1.054	1.020	1.054
(3, 4)	1.009	1.032	1.087	1.160	1.112	1.181	1.025	1.069	1.040	1.041	1.136	1.390	1.024	1.109	1.024	1.062
(4, 4)	1.007	1.031	1.111	1.172	1.097	1.163	1.124	1.110	1.281	1.629	1.297	1.485	1.004	1.004	1.000	1.003

Table B.8: Guessing entropy of the 16-bit fragment templates

fragment		original table				marginalized table							
PPC	c	$K_0 \Vert K_1$	$K_2 \Vert K_3$	$K_4 \Vert K_5$	$K_6 \Vert K_7$	K_0	K_1	K_2	K_3	K_4	K_5	K_6	K_7
125	4	25163.901	24846.461	22400.132	22383.710	107.436	110.550	111.359	107.428	100.985	107.273	103.984	102.025
100	5	24495.540	24095.835	21879.470	21732.560	109.024	106.711	109.009	105.554	99.676	106.696	101.844	98.846
50	10	23044.614	23208.794	20341.347	20380.109	102.535	104.804	107.148	102.725	93.799	103.040	96.627	95.344
25	20	22114.698	21570.856	19882.537	19170.855	99.097	101.781	101.008	99.094	92.629	101.233	95.074	90.624
20	25	21568.866	21824.465	20083.905	19185.527	98.332	100.077	100.873	97.542	93.091	101.384	95.391	90.620
10	50	21371.623	21461.922	19290.490	18607.576	97.531	98.073	99.975	97.536	92.385	96.869	94.232	87.749
5	100	21049.649	21681.278	19117.445	18207.491	96.833	97.230	100.331	98.399	91.334	96.278	93.572	86.812
4	125	21378.706	22262.627	19085.549	18351.998	97.771	97.728	101.910	99.063	91.159	96.465	93.507	87.244

fragment		original table				marginalized table							
PPC	c	$P_0 \Vert P_1$	$P_2 \Vert P_3$	$P_4 \Vert P_5$	$P_6 \Vert P_7$	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
125	4	19162.143	23197.876	16506.523	21350.179	109.410	80.554	104.356	108.615	96.617	77.336	100.882	103.501
100	5	18485.322	22839.778	15984.318	21051.767	107.394	79.670	106.533	106.028	95.296	77.067	102.001	100.109
50	10	17116.233	20881.639	14595.119	20001.790	105.402	75.300	98.292	101.903	91.242	73.028	97.944	97.966
25	20	15984.368	19711.693	13620.423	18311.100	102.111	71.406	95.028	97.690	89.462	68.940	92.724	93.828
20	25	15771.413	19804.819	13241.689	18728.436	101.086	72.065	96.064	97.384	86.718	68.956	93.227	94.988
10	50	15691.727	19645.105	13207.999	18030.963	99.999	72.005	95.286	96.362	85.917	69.032	90.821	93.973
5	100	15753.523	19613.759	13341.622	17863.096	98.378	72.420	94.417	97.075	86.720	68.395	90.211	92.305
4	125	15997.928	19597.525	13557.455	18150.181	98.427	73.539	93.567	97.758	86.688	69.685	91.384	93.961

fragment		original table				marginalized table							
PPC	c	$C_0 \Vert C_1$	$C_2 \Vert C_3$	$C_4 \Vert C_5$	$C_6 \Vert C_7$	C_0	C_1	C_2	C_3	C_4	C_5	C_6	C_7
125	4	16738.053	20440.974	17525.669	17558.714	76.380	101.198	98.206	98.247	75.578	104.699	99.284	84.162
100	5	16206.107	20347.449	16372.716	17216.277	74.610	101.278	99.659	98.487	74.204	100.618	98.039	84.282
50	10	14953.529	18916.827	14999.383	15390.817	70.868	96.812	95.288	93.564	69.864	96.818	93.852	78.416
25	20	14439.826	18320.846	14021.992	15070.342	70.256	94.322	93.239	91.310	66.644	93.290	91.470	77.459
20	25	14000.471	18276.798	13848.165	14669.357	69.047	94.401	93.860	90.054	66.637	92.391	89.981	77.170
10	50	14006.286	18010.309	13631.111	14198.935	69.452	92.204	92.277	89.666	66.073	91.079	88.501	75.854
5	100	13911.666	17725.837	13854.991	14323.605	70.209	90.094	90.370	90.068	67.115	91.059	87.717	76.339
4	125	14347.565	17929.550	13887.252	14236.946	71.235	89.428	88.765	92.169	67.210	90.683	87.043	75.875

Table B.9: Guessing entropy of the key bytes, after belief propagation with byte probability tables of keys, plaintexts, and ciphertexts marginalized from the 16-bit template results.

fragment									
PPC	c	K_0	K_1	K_2	K_3	K_4	K_5	K_6	K_7
125	4	108.446	110.566	111.514	108.057	98.628	106.119	103.397	101.825
100	5	109.943	105.834	110.028	106.858	98.015	106.442	101.322	97.514
50	10	103.555	103.105	107.490	103.276	91.987	102.565	95.938	92.885
25	20	98.463	99.960	101.409	100.024	89.739	100.410	94.428	89.537
20	25	98.114	97.079	101.000	98.419	90.332	100.042	95.693	89.448
10	50	97.120	95.665	100.520	97.585	88.722	95.995	94.132	86.847
5	100	96.609	94.166	100.236	98.765	88.272	95.096	93.483	85.682
4	125	96.877	95.894	102.211	99.402	88.153	95.591	93.279	87.479

Table B.10: Guessing entropy of the key bytes, after belief propagation with byte probability tables of keys and plaintexts marginalized from the 16-bit template results, and known ciphertexts.

fragment		K_0	K_1	K_2	K_3	K_4	K_5	K_6	K_7
PPC	c								
125	4	99.858	78.163	98.201	98.892	86.900	73.482	91.756	90.995
100	5	99.919	74.775	97.382	96.324	84.711	72.113	91.066	86.781
50	10	92.790	69.867	90.489	90.891	76.604	66.760	83.761	81.792
25	20	87.626	65.575	83.953	85.146	75.290	62.046	78.031	75.276
20	25	87.046	64.722	84.852	83.740	73.897	61.435	78.768	76.385
10	50	86.198	63.661	82.891	81.487	72.150	60.474	75.456	73.877
5	100	85.439	64.393	81.701	82.336	71.646	60.500	75.034	72.181
4	125	85.487	66.486	82.002	83.189	72.184	61.896	75.637	73.480

Table B.11: Guessing entropy of the 16-bit fragment templates after belief propagation directly with 16-bit tables

original table						marginalized table							
fragment		$K_0 \Vert K_1$	$K_2 \Vert K_3$	$K_4 \Vert K_5$	$K_6 \Vert K_7$	K_0	K_1	K_2	K_3	K_4	K_5	K_6	K_7
PPC	c												
125	4	17281.571	19841.913	13752.300	17158.669	99.576	78.169	97.412	98.804	86.647	73.331	91.412	90.677
100	5	16407.325	19167.969	13093.923	16465.513	99.769	74.669	96.484	96.059	84.530	71.750	91.083	86.976
50	10	14666.972	17271.035	11233.157	14728.097	92.752	69.762	90.203	90.908	76.425	66.426	83.867	81.913
25	20	13239.219	15507.825	10439.856	13076.071	87.645	65.643	83.667	85.069	74.908	61.645	77.874	75.383
20	25	12802.147	15683.551	10203.931	13473.870	86.948	64.914	84.456	83.612	73.920	61.261	78.893	76.426
10	50	12572.240	15058.004	9810.055	12644.133	86.064	63.714	82.420	81.467	72.068	60.347	75.679	73.986
5	100	12538.710	15119.313	9766.814	12219.906	85.150	64.525	81.310	82.337	71.531	60.266	74.909	72.289
4	125	12970.457	15504.747	10056.026	12457.592	85.378	66.458	81.852	83.203	71.895	61.594	75.482	73.416

B.4 Data for the Keccak experiments on the 32-bit device

Table B.12: Success rates (left) and guessing entropy (right) of templates in α'_0

(i,j) \ ^8k	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) \ ^8k	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 B.13: Success rates (left) and guessing entropy (right) of templates in β_0

(i,j) \ ^8k	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) \ ^8k	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 B.14: Success rates (left) and guessing entropy (right) of templates in $\mathbf{C}_0$

(i,j) \ ${}^{8}k$	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) \ ${}^{8}k$	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 B.15: Success rates (left) and guessing entropy (right) of templates in $\mathbf{D}_0$

(i,j) \ ${}^{8}k$	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) \ ${}^{8}k$	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 B.16: The run-time estimation of template profiling and attack with different fragment size (with $m = 2000$, $N = 64000$, $m' = 8$), where the results for profiling are estimated with the average of 100 trials and the results for attack are estimated with 1000 trials.

Profiling				
fragment size		4-bit	8-bit	16-bit
Total	CPU time (s)	379.644	405.316	509.050
	Wall time (s)	45.803	46.548	61.953

Attack				
fragment size		4-bit	8-bit	16-bit
Single-core	CPU time (µs)	251.199	353.150	218269.503
	Wall time (µs)	269.781	372.640	219348.418
32-core	CPU time (µs)	18247.971	32176.628	5893249.263
	Wall time (µs)	599.665	1031.811	184540.310

Table B.17: Results of recovering the functions in the SHA-3 family with different numbers of invocations by the three-round factor graph.

Function	c	#Inv.	#Rec.	#Iteration*			
				Med.	Mean	σ	Max
SHA3-512	1024	1	1000	30	30.064	1.720	35
		2	1000	30	30.577	1.285	36
		4	1000	30	30.485	1.277	37
		5	1000	30	30.519	1.306	36
		10	1000	30	30.273	1.282	37
SHA3-384	768	1	1000	34	34.066	2.057	41
		2	1000	34	34.420	1.497	41
SHA3-256	512	1	999	38	38.023	2.924	46
		2	997	38	38.323	1.700	45
SHAKE256		1	999	39	38.789	2.727	50
		2	994	39	38.785	1.902	50
SHA3-224	448	1	992	39	39.284	2.947	52
		2	979	40	40.086	2.138	55
SHAKE128	256	1	921	43	43.511	5.021	106
		2	862	44	44.191	3.560	116

* Only the invocations successfully reaching a steady state are taken into account.

Table B.18: Results of recovering the functions in the SHA-3 family with different numbers of invocations by the two-round factor graph.

Function	c	#Inv.	#Rec.	#Iteration*			
				Med.	Mean	σ	Max
SHA3-512	1024	1	1000	51	50.909	5.362	71
		2	998	51	52.189	5.550	163
		4	999	52	52.217	4.914	104
		5	1000	52	52.266	5.082	161
		10	999	51	51.566	4.762	107
SHA3-384	768	1	997	65	66.025	10.526	154
		2	993	66	67.035	8.254	130
SHA3-256	512	1	940	89	95.240	29.147	198
		2	912	90	97.705	26.110	198
SHAKE256		1	867	95	100.993	30.085	198
		2	828	93	101.265	28.166	199
SHA3-224	448	1	419	94	98.173	30.634	195
		2	140	105	111.207	28.667	199
SHAKE128	256	1	35	59	63.943	14.485	110
		2	0	89	89.000	0.000	89

* Only the invocations successfully reaching a steady state are taken into account.

Table B.19: Results of recovering the functions in the SHA-3 family with one invocation by the 16-bit fragment templates and the four-round factor graph.

Function	c	r	#Rec.	#Iteration*			
				Median	Mean	σ	Max
SHA3-512	1024	576	1000	25	24.922	0.796	28
SHA3-384	768	832	1000	26	26.291	0.935	29
SHA3-256	512	1088	999	28	27.973	1.232	31
SHAKE256			997	28	28.362	1.237	33
SHA3-224	448	1152	999	28	28.403	1.219	33
SHAKE128	256	1344	984	30	30.052	1.488	38

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

Table B.20: Results of recovering the functions in the SHA-3 family with one invocation by the 16-bit fragment templates and the three-round factor graph.

Function	c	r	#Rec.	#Iteration*			
				Median	Mean	σ	Max
SHA3-512	1024	576	1000	29	29.417	1.686	35
SHA3-384	768	832	1000	33	33.169	2.018	40
SHA3-256	512	1088	998	37	36.787	2.806	45
SHAKE256			999	38	37.492	2.568	47
SHA3-224	448	1152	996	38	37.879	2.766	45
SHAKE128	256	1344	956	42	41.361	3.506	65

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

Table B.21: Results of recovering the functions in the SHA-3 family with one invocation by the 16-bit fragment templates and the two-round factor graph.

Function	c	r	#Rec.	#Iteration*			
				Median	Mean	σ	Max
SHA3-512	1024	576	1000	49	49.119	5.176	73
SHA3-384	768	832	1000	62	63.001	9.278	135
SHA3-256	512	1088	971	83	87.463	25.959	197
SHAKE256			944	86	92.684	26.726	199
SHA3-224	448	1152	575	89	93.790	29.063	197
SHAKE128	256	1344	47	55.5	65.729	26.573	167

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

Table B.22: Results of recovering the functions in the SHA-3 family with one invocation by the nibble templates and the four-round factor graph.

Function	c	r	#Rec.	#Iteration*			
				Median	Mean	σ	Max
SHA3-512	1024	576	1000	26	25.763	0.806	28
SHA3-384	768	832	1000	27	27.178	0.936	30
SHA3-256	512	1088	1000	29	29.049	1.264	33
SHAKE256			997	30	29.528	1.297	35
SHA3-224	448	1152	1000	30	29.545	1.275	34
SHAKE128	256	1344	976	32	31.465	1.586	41

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

Table B.23: Results of recovering the functions in the SHA-3 family with one invocation by the nibble templates and the three-round factor graph.

Function	c	r	#Rec.	#Iteration*			
				Median	Mean	σ	Max
SHA3-512	1024	576	1000	31	30.551	1.753	35
SHA3-384	768	832	1000	35	34.683	2.108	40
SHA3-256	512	1088	998	39	38.865	3.002	46
SHAKE256			998	40	39.716	2.841	53
SHA3-224	448	1152	990	40	40.257	3.026	52
SHAKE128	256	1344	908	45	44.968	5.567	128

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

Table B.24: Results of recovering the functions in the SHA-3 family with one invocation by the nibble templates and the two-round factor graph.

Function	c	r	#Rec.	#Iteration*			
				Median	Mean	σ	Max
SHA3-512	1024	576	1000	52	51.918	5.453	71
SHA3-384	768	832	997	67	67.902	11.682	184
SHA3-256	512	1088	908	94	99.515	29.847	198
SHAKE256			828	97	103.674	30.101	199
SHA3-224	448	1152	391	99	103.184	31.892	197
SHAKE128	256	1344	31	64	71.194	19.526	122

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

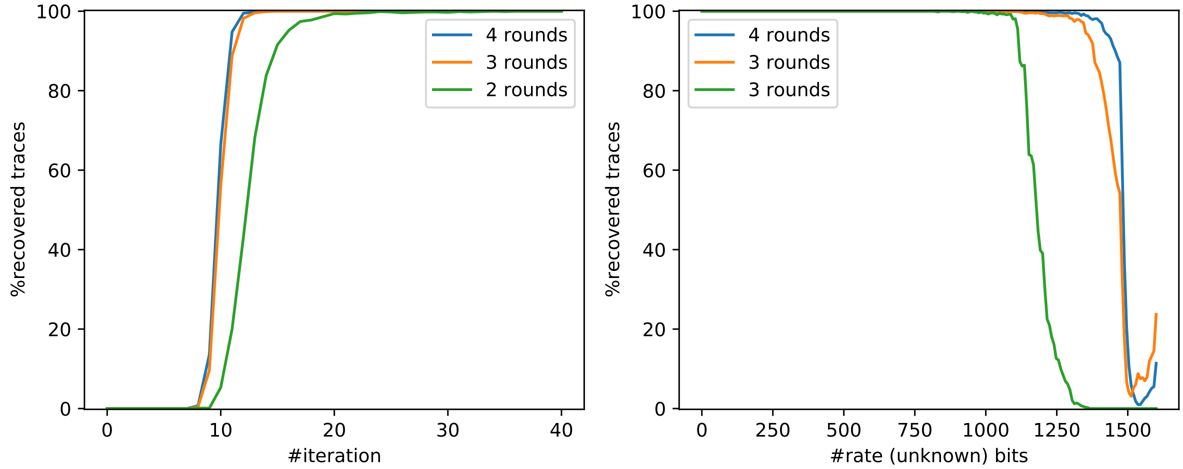


Figure B.1: Percentage of successfully recovered traces with 16-bit templates for factor graphs with different numbers of rounds observed, as a function of the number of loopy-BP iterations (left) and the number of unknown input bits (right).

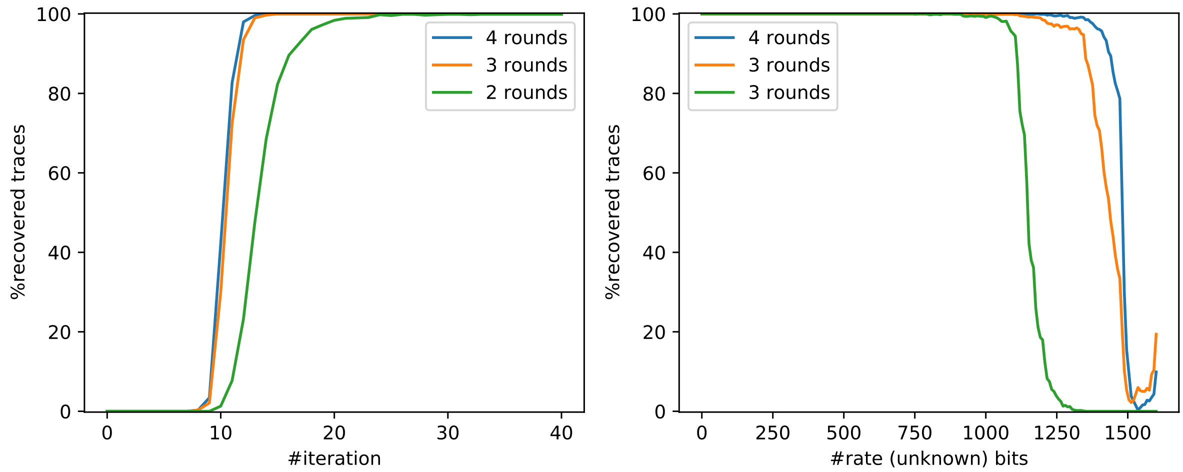


Figure B.2: Percentage of successfully recovered traces with nibble templates for factor graphs with different numbers of rounds observed, as a function of the number of loopy-BP iterations (left) and the number of unknown input bits (right).

B.5 Data for the Ascon experiments on the 32-bit device

Table B.25: Number of interesting clock cycles detected for the high and low 32-bit intermediate words in U-Os. The detection for IV and N is not needed since they are known values.

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

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

Table B.26: Number of interesting clock cycles detected for the even and odd 32-bit intermediate words in U-Os. The detection for IV and N is not needed since they are known values.

lane		L_0		L_1		L_2		L_3		L_4	
32-bit word		even	odd	even	odd	even	odd	even	odd	even	odd
Init.	input (β_{-1})	IV		257	283	315	320	N			
	α_0	22	22	115	118	28	26	26	24	79	78
	β_0	18	7	14	18	13	23	14	13	21	25
	α_1	13	14	11	22	20	25	13	20	17	17
	β_1	14	7	12	17	13	22	12	12	25	19
	α_2	15	16	9	12	23	19	25	18	18	14
	β_2	11	8	21	15	17	19	11	11	23	20
	α_3	12	16	11	11	23	25	14	19	17	13
	β_3	12	9	13	14	15	23	14	15	23	23
	α_4	19	14	16	11	20	25	17	22	16	14
	β_4	12	8	13	16	17	22	20	13	20	23
	α_5	15	13	14	16	20	24	13	17	24	21
	β_5	12	8	16	19	15	19	12	21	20	22
	α_6	16	13	11	12	19	22	14	18	16	13
	β_6	13	7	14	16	15	19	11	19	24	21
	α_7	16	15	10	11	35	20	17	20	15	14
	β_7	13	9	15	18	13	29	16	12	19	23
	α_8	16	10	9	10	21	20	16	20	15	13
	β_8	13	8	16	18	15	20	14	12	26	21
	α_9	14	16	11	10	22	23	14	19	15	14
	β_9	15	15	16	16	15	20	17	13	22	23
	α_{10}	16	12	9	13	21	20	17	18	15	17
	β_{10}	13	6	17	17	16	20	17	13	23	23
	α_{11}	16	12	9	10	25	22	17	22	18	16
	β_{11}	95	81	26	26	29	72	21	18	37	40

lane		L_0		L_1		L_2		L_3		L_4	
32-bit word		even	odd	even	odd	even	odd	even	odd	even	odd
Fin.	input (β_{-1})	132	108	31	29	41	35	29	28	75	54
	α_0	16	15	10	10	21	22	14	19	15	13
	β_0	13	10	17	15	13	15	16	15	22	25
	α_1	15	15	12	13	18	27	15	17	22	14
	β_1	10	5	15	17	16	18	13	12	28	22
	α_2	12	19	9	13	25	21	14	18	18	14
	β_2	13	10	20	16	17	23	17	11	22	22
	α_3	14	16	12	10	19	27	14	22	18	15
	β_3	12	7	14	19	16	21	14	14	21	23
	α_4	16	12	11	14	21	26	13	16	17	13
	β_4	12	8	14	19	15	18	15	14	27	20
	α_5	15	14	12	12	26	22	15	18	19	18
	β_5	10	10	14	16	18	18	17	12	23	21
	α_6	14	15	13	13	19	24	13	16	15	14
	β_6	11	9	15	14	12	20	15	12	27	21
	α_7	12	14	11	9	20	22	15	18	17	14
	β_7	13	6	22	16	16	20	14	13	24	22
	α_8	16	13	10	10	18	23	14	21	19	13
	β_8	12	13	15	17	14	21	14	14	23	25
	α_9	20	14	12	14	24	26	14	19	15	16
	β_9	13	9	14	18	18	18	14	13	21	20
	α_{10}	17	13	16	12	21	19	16	18	17	15
	β_{10}	13	8	19	14	17	18	13	12	25	22
	α_{11}	14	13	11	11	22	21	15	22	22	17
	β_{11}	72	58	62	65	62	65	117	120	141	141

Table B.27: Key-recovery success rates evaluated with 1000 testing keys, for each by up to 10 traces within our loopy factor graph (U-Os experiment).

#Traces	#Combinations searched															
	1	2	5	10	20	50	100	200	500	1000	2000	5000	10000	20000	50000	10^5
1	0.197	0.209	0.224	0.248	0.268	0.299	0.322	0.364	0.412	0.443	0.488	0.538	0.582	0.614	0.655	0.686
2	0.807	0.811	0.818	0.821	0.830	0.839	0.847	0.855	0.867	0.885	0.895	0.913	0.925	0.940	0.949	0.955
3	0.950	0.950	0.951	0.954	0.954	0.959	0.960	0.962	0.965	0.968	0.970	0.974	0.974	0.975	0.979	0.981
4	0.970	0.971	0.971	0.973	0.973	0.975	0.976	0.980	0.983	0.984	0.985	0.986	0.988	0.988	0.990	0.991
5	0.972	0.972	0.974	0.975	0.975	0.975	0.975	0.975	0.976	0.977	0.977	0.977	0.978	0.979	0.979	0.980
6	0.987	0.987	0.987	0.987	0.989	0.989	0.989	0.989	0.989	0.990	0.990	0.990	0.990	0.991	0.991	0.991
7	0.974	0.975	0.975	0.975	0.975	0.975	0.976	0.977	0.977	0.977	0.978	0.979	0.979	0.979	0.979	0.979
8	0.974	0.974	0.974	0.974	0.974	0.974	0.974	0.974	0.974	0.974	0.974	0.974	0.974	0.974	0.974	0.974
9	0.956	0.956	0.956	0.956	0.956	0.956	0.956	0.956	0.956	0.956	0.957	0.958	0.958	0.958	0.959	0.959
10	0.957	0.958	0.958	0.958	0.958	0.958	0.958	0.958	0.958	0.959	0.959	0.959	0.959	0.959	0.959	0.959

Table B.28: Success rates of recovering the 1000 testing keys by tree BP with marginalized bitwise probability tables from byte templates (U-Os experiment).

#Traces	#Combinations searched															
	1	2	5	10	20	50	100	200	500	1000	2000	5000	10000	20000	50000	10^5
1	0.018	0.030	0.047	0.065	0.088	0.120	0.167	0.207	0.257	0.299	0.353	0.414	0.450	0.497	0.565	0.609
2	0.116	0.180	0.299	0.374	0.458	0.554	0.618	0.667	0.739	0.780	0.828	0.866	0.902	0.926	0.954	0.970
3	0.260	0.360	0.475	0.564	0.643	0.743	0.804	0.851	0.896	0.928	0.946	0.969	0.980	0.989	0.994	0.994
4	0.337	0.457	0.601	0.696	0.773	0.847	0.888	0.923	0.953	0.966	0.977	0.986	0.988	0.991	0.995	0.996
5	0.412	0.531	0.685	0.765	0.816	0.879	0.916	0.941	0.972	0.985	0.987	0.991	0.993	0.993	0.995	0.998
6	0.436	0.569	0.725	0.805	0.858	0.908	0.934	0.956	0.981	0.987	0.990	0.992	0.994	0.995	0.999	0.999
7	0.477	0.604	0.757	0.831	0.880	0.922	0.948	0.968	0.983	0.985	0.989	0.992	0.996	0.998	0.998	0.999
8	0.499	0.627	0.756	0.840	0.890	0.924	0.953	0.973	0.985	0.991	0.994	0.998	0.998	0.998	0.999	1.000
9	0.508	0.630	0.784	0.849	0.894	0.938	0.958	0.977	0.987	0.993	0.995	0.998	0.999	1.000	1.000	1.000
10	0.531	0.652	0.788	0.858	0.905	0.939	0.965	0.979	0.991	0.995	0.999	0.999	1.000	1.000	1.000	1.000

Table B.29: Success rates of recovering the 1000 testing keys by tree BP with probability tables from byte templates (U-Os experiment).

#Traces	#Combinations searched															
	1	2	5	10	20	50	100	200	500	1000	2000	5000	10000	20000	50000	10^5
1	0.152	0.237	0.373	0.486	0.571	0.681	0.751	0.804	0.863	0.903	0.933	0.953	0.962	0.972	0.988	0.992
2	0.648	0.801	0.911	0.966	0.984	0.992	0.996	0.997	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
3	0.831	0.939	0.983	0.996	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
4	0.909	0.970	0.997	0.999	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
5	0.930	0.983	0.999	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
6	0.941	0.988	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
7	0.958	0.994	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
8	0.967	0.995	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
9	0.966	0.998	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
10	0.969	0.998	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000

Table B.30: Quality evaluation of templates for 16-bit fragments (H/L)

fragment		$K[0,0]^{\mathbf{16}}$	$K[0,1]^{\mathbf{16}}$	$K[0,2]^{\mathbf{16}}$	$K[0,3]^{\mathbf{16}}$	$K[1,0]^{\mathbf{16}}$	$K[1,1]^{\mathbf{16}}$	$K[1,2]^{\mathbf{16}}$	$K[1,3]^{\mathbf{16}}$
Key K	SR	0.801	0.710	0.655	0.499	0.699	0.724	0.803	0.587
	GR	1.559	2.099	2.708	4.430	2.125	1.746	1.574	3.454
fragment		$\beta_{11}[3,0]^{\mathbf{16}}$	$\beta_{11}[3,1]^{\mathbf{16}}$	$\beta_{11}[3,2]^{\mathbf{16}}$	$\beta_{11}[3,3]^{\mathbf{16}}$	$\beta_{11}[4,0]^{\mathbf{16}}$	$\beta_{11}[4,1]^{\mathbf{16}}$	$\beta_{11}[4,2]^{\mathbf{16}}$	$\beta_{11}[4,3]^{\mathbf{16}}$
Fin. β_{11}	SR	0.039	0.032	0.043	0.041	0.021	0.016	0.051	0.073
	GR	505.350	822.536	331.481	383.037	679.468	1033.571	259.831	153.293

Table B.31: Success rates of recovering the 1000 testing keys by tree BP with probability tables from 16-bit templates (`U-Os` experiment).

#Traces	#Combinations searched															
	1	2	5	10	20	50	100	200	500	1000	2000	5000	10000	20000	50000	10^5
1	0.319	0.465	0.621	0.735	0.817	0.886	0.932	0.949	0.961	0.974	0.988	0.995	0.996	0.998	1.000	1.000
2	0.800	0.928	0.981	0.995	0.999	0.999	0.999	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
3	0.914	0.980	0.999	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
4	0.953	0.996	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
5	0.969	0.998	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
6	0.971	0.997	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
7	0.982	0.998	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
8	0.986	0.998	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
9	0.989	0.998	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
10	0.993	0.999	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000

Table B.32: Success rates of recovering the 1000 testing keys by tree BP with probability tables from byte templates (`U-O3` experiment).

#Traces	#Combinations searched															
	1	2	5	10	20	50	100	200	500	1000	2000	5000	10000	20000	50000	10^5
1	0.000	0.000	0.000	0.000	0.001	0.001	0.001	0.001	0.002	0.002	0.003	0.007	0.008	0.011	0.015	0.019
2	0.001	0.001	0.001	0.003	0.003	0.009	0.013	0.017	0.031	0.040	0.052	0.075	0.103	0.127	0.157	0.193
3	0.003	0.007	0.009	0.010	0.014	0.028	0.037	0.048	0.070	0.089	0.129	0.181	0.219	0.271	0.333	0.385
4	0.004	0.008	0.014	0.019	0.024	0.048	0.069	0.090	0.134	0.166	0.210	0.274	0.329	0.381	0.453	0.509
5	0.006	0.008	0.016	0.025	0.044	0.068	0.086	0.126	0.171	0.226	0.274	0.341	0.393	0.452	0.534	0.587
6	0.008	0.010	0.027	0.034	0.049	0.085	0.126	0.152	0.213	0.263	0.311	0.390	0.436	0.493	0.584	0.635
7	0.009	0.017	0.029	0.041	0.066	0.101	0.136	0.170	0.236	0.284	0.335	0.419	0.478	0.540	0.611	0.671
8	0.012	0.018	0.029	0.041	0.063	0.106	0.157	0.199	0.258	0.319	0.369	0.448	0.503	0.556	0.643	0.694
9	0.009	0.018	0.035	0.056	0.084	0.123	0.154	0.206	0.283	0.338	0.393	0.479	0.533	0.596	0.669	0.716
10	0.012	0.020	0.034	0.054	0.084	0.130	0.177	0.231	0.302	0.359	0.411	0.491	0.553	0.617	0.680	0.734

Table B.33: Success rates of recovering the 1000 testing keys by tree BP with probability tables from 16-bit templates (`U-03` experiment).

#Traces	#Combinations searched															
	1	2	5	10	20	50	100	200	500	1000	2000	5000	10000	20000	50000	10^5
1	0.000	0.003	0.004	0.006	0.008	0.010	0.014	0.018	0.023	0.034	0.041	0.056	0.062	0.082	0.108	0.127
2	0.005	0.013	0.030	0.044	0.058	0.087	0.111	0.157	0.202	0.246	0.295	0.359	0.426	0.476	0.541	0.589
3	0.036	0.050	0.083	0.102	0.137	0.193	0.243	0.314	0.388	0.448	0.505	0.581	0.647	0.690	0.759	0.799
4	0.044	0.072	0.120	0.167	0.210	0.315	0.368	0.443	0.523	0.579	0.628	0.709	0.755	0.793	0.841	0.873
5	0.063	0.100	0.158	0.209	0.281	0.369	0.435	0.509	0.597	0.661	0.706	0.768	0.806	0.845	0.889	0.910
6	0.068	0.113	0.189	0.251	0.306	0.403	0.492	0.558	0.649	0.705	0.752	0.801	0.841	0.879	0.914	0.935
7	0.085	0.124	0.213	0.279	0.350	0.436	0.515	0.596	0.679	0.740	0.769	0.830	0.871	0.904	0.929	0.948
8	0.087	0.138	0.229	0.303	0.377	0.469	0.539	0.624	0.694	0.746	0.793	0.853	0.885	0.915	0.946	0.957
9	0.092	0.148	0.235	0.320	0.405	0.500	0.575	0.644	0.714	0.770	0.814	0.869	0.905	0.929	0.946	0.961
10	0.103	0.161	0.264	0.338	0.409	0.513	0.583	0.651	0.731	0.788	0.828	0.884	0.909	0.930	0.952	0.963

Table B.34: Quality evaluation of fragment templates for the key of Ascon AEAD with only one part of the interesting clock cycles (`U-03` data sets).

lane		L_1								L_2							
word		high				low				high				low			
byte		0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
All interesting clock cycles	SR	0.683	0.418	0.438	0.306	0.320	0.261	0.239	0.243	0.419	0.299	0.227	0.200	0.524	0.328	0.485	0.493
	GR	1.681	4.622	4.107	7.380	6.026	9.774	9.001	7.726	3.961	6.862	10.292	12.206	2.767	5.635	3.387	3.316
from region 1	SR	0.196	0.085	0.102	0.100	0.057	0.043	0.074	0.065	0.056	0.068	0.070	0.070	0.135	0.067	0.073	0.108
	GR	9.947	25.103	17.766	29.276	33.485	40.562	30.182	33.375	37.507	29.037	41.370	41.332	16.175	32.600	28.850	22.366
from region 2	SR	0.057	0.037	0.037	0.027	0.062	0.033	0.056	0.029	0.107	0.054	0.043	0.028	0.095	0.064	0.096	0.049
	GR	35.195	50.700	45.368	60.148	29.351	48.493	47.342	50.945	20.727	40.203	45.081	53.499	22.801	37.779	21.944	39.046
from region 3	SR	0.146	0.082	0.074	0.044	0.063	0.035	0.049	0.031	0.065	0.064	0.048	0.033	0.070	0.056	0.123	0.095
	GR	17.713	28.668	29.770	44.064	27.461	50.882	54.501	38.037	33.094	39.731	35.757	52.766	30.461	34.892	20.243	25.556
from region 4	SR	0.024	0.027	0.014	0.022	0.012	0.026	0.006	0.020	0.009	0.018	0.012	0.008	0.012	0.023	0.025	0.021
	GE	64.327	63.113	83.875	80.022	91.798	62.308	94.149	81.054	91.080	70.160	88.492	90.651	71.301	54.705	72.237	62.764

lane		L_3								L_4							
word		high				low				high				low			
byte		0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
β_{11} in Finalization	SR	0.089	0.042	0.048	0.046	0.111	0.077	0.091	0.045	0.110	0.079	0.062	0.040	0.118	0.069	0.122	0.063
	GR	26.965	43.649	43.980	47.691	22.775	34.171	24.764	36.879	21.568	34.633	38.302	41.534	17.675	32.054	20.596	34.329

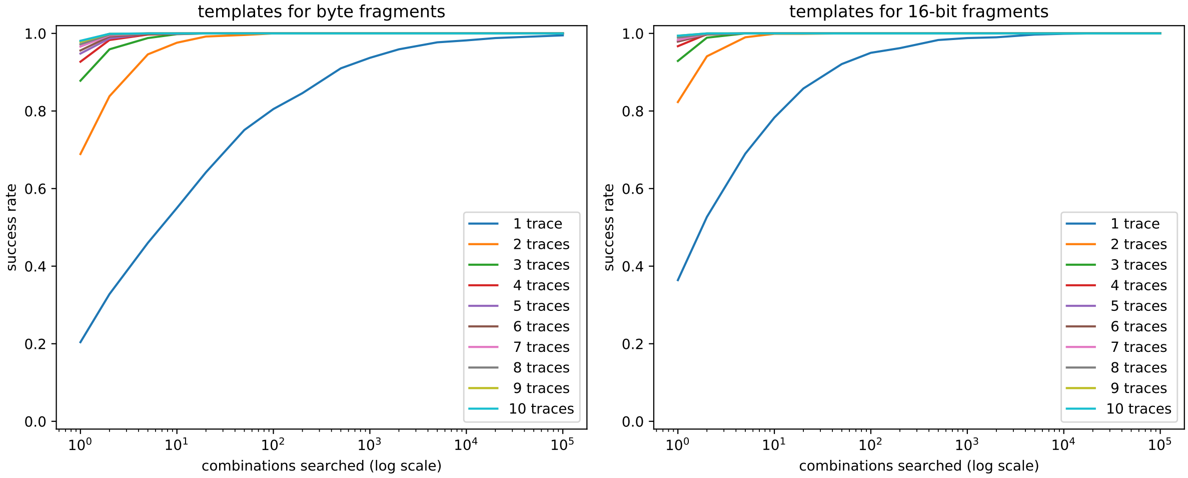


Figure B.3: Success rates on ASCON-128 with expanded interesting clock cycle sets, for both 8 and 16-bit fragments. We can compare these results with those in Figure 5.7.

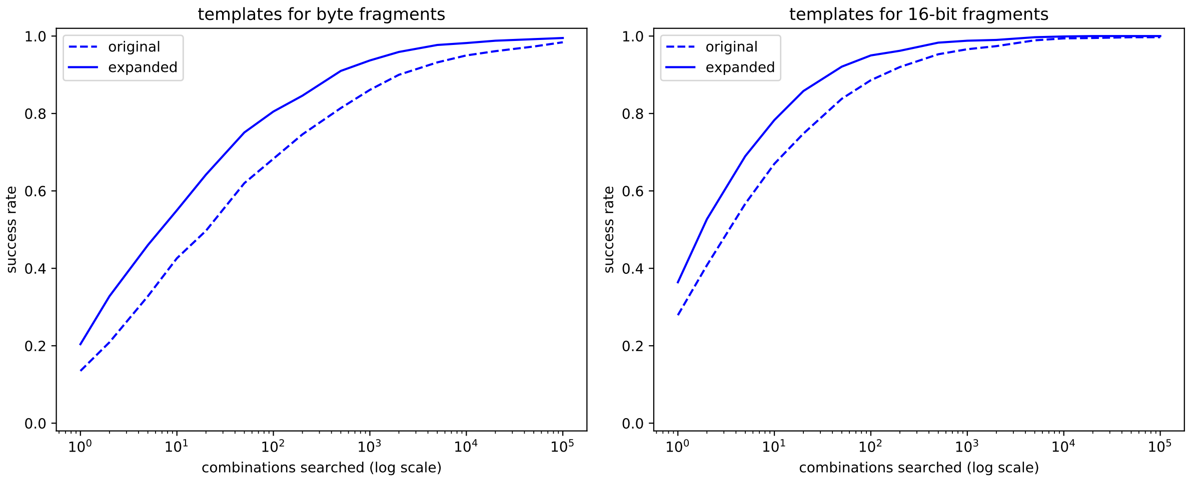


Figure B.4: Single-trace success rates on ASCON-128, with both original (Figure 5.7) and expanded (Figure B.3) interesting clock cycle sets plotted together for comparison.