Technical Report

UCAM-CL-TR-995
ISSN 1476-2986

Number 995

UNIVERSITY OF
CAMBRIDGE

Computer Laboratory

Fragment-template power-analysis attacks against microcontroller implementations of the 32-bit stream cipher ChaCha

Henry Batchelor

July 2024

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

© 2024 Henry Batchelor

This technical report is based on a dissertation submitted May 2024 by the author for the degree of Master of Engineering (Computer Science Tripos) to the University of Cambridge, Selwyn College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986
DOI *https://doi.org/10.48456/tr-995*

Abstract

ChaCha is a widely adopted stream cipher, used for both random number generation and encryption. I propose a factor graph of ChaCha to improve the success rate of side-channel attacks that provide leakages throughout the entire execution of the algorithm. I also assess (fragment) template attacks against several implementations of ChaCha to demonstrate that the factor graph is helpful when working with actual side-channel attacks.

These attacks could fully recover the correct key from an 8-bit implementation. In contrast, a 32-bit implementation, with most of the state held in registers, was significantly more challenging to attack. An adversary with access to 10 power traces and an incremented counter could achieve a success rate of 14.6%. For a 32-bit implementation, with lots of SRAM activity, an attacker could successfully recover the key in 2.6% of cases from a single trace.

Acknowledgements

I would like to thank Markus Kuhn, Shih-Chun You, and Eric Peng for support and fruitful discussions throughout this project.

I would also like to thank Richard Watts, Director of Studies in Computer Science at Selwyn College, for support throughout my degree and Mr Meakin, Head of Computer Science at Bury Grammar School, for encouraging my interest in Computer Science.

Contents

Chapter 1

Introduction

Side-channel attacks provide a method for attacking theoretically sound cryptographic functions by looking at flaws in their implementations. The most basic type of side-channel attack is a *timing attack*, where the amount of time an operation takes leaks information about a value. Many other types of side-channel can be exploited, such as the amount of power drawn and electromagnetic emissions.

ChaCha [1] is a widely used stream cipher for pseudo-random number generation, for masking other cryptographic operations, or as part of an authenticated encryption with associated data scheme (AEAD). It uses only addition (ADD), exclusive or (XOR), and rotate (ROR) instructions, which means that constant-time implementations are relatively easy to achieve, so side-channel attacks have to make use of another source of information.

It is valuable to examine how vulnerable ChaCha is to other side-channel attacks. This project attempts to infer the secret values from a small number of power traces, which naturally leads to performing a template attack [2].

Previous works [3] [4] have shown that template attacks can have their success rates significantly increased by the soft analytical side-channel attack (SASCA) [5] technique for algorithms based around permutations. The main rounds of ChaCha implement a permutation; therefore, it is natural to examine the combination of a template attack with SASCA for attacking ChaCha. SASCA provides a probabilistic model of the algorithm that we can insert the side-channel leakages into and marginalise over to maximise the chance of finding the correct values.

Initially, I focused on creating a probabilistic model of ChaCha, with particular interest paid to the design of 32-bit ADD, which had not been previously explored. Afterwards, I performed several actual attacks against ChaCha implementations, demonstrating that a normal 32-bit implementation can have its key recovered in 14.6% of cases with ten traces by an adversary. In contrast, an 8-bit implementation could have its key recovered from a single trace in all cases. I also demonstrated that SRAM activity should be minimised due to this providing more leakage.

Chapter 2 gives an overview of ChaCha's design alongside the steps typically performed in a side-channel attack. It also provides an overview of techniques for improving the success rates of attacks and assessing how successful an attack would be for a more powerful attacker, alongside a summary of related work.

Chapter 3 details designing the probabilistic model for ChaCha and experiments undertaken to allow for effective operation with actual leakages.

Chapter 4 explains different types of leakage we can extract from encryptions, including in simulations. It then covers the process I used for performing template attacks against different implementations of ChaCha.

Chapter 5 assesses the success rates of the attacks performed in various scenarios (including simulated leakages) and the template qualities.

Chapter 2

Background

2.1 ChaCha

ChaCha is a stream cipher proposed by Daniel Bernstein in 2008 [1], based on his existing Salsa20 cipher. The only operations ChaCha uses are 32-bit addition (ADD), bitwise exclusive or (XOR), and rotation (ROR). The cipher's state consists of 16 32-bit words with the initial configuration as specified in RFC 8439 [6] shown in table 2.1. The first four words are taken as their ASCII value in little-endian order, so "expa" has the hexadecimal value 0x61707865. The counter is incremented between subsequent invocations of ChaCha.

Table 2.1: The initial state configuration of ChaCha, where the words are ordered in row major order

"expa"	"nd 3"	"2-by"	"te k"
Key	Key	Key	Key
Key	Key	Key	Key
Counter	Nonce	Nonce	Nonce

According to RFC 8439, the cipher's operation consists of 20 rounds, alternating between odd and even rounds; each round comprising four quarter rounds. Figure 2.1 shows the design of the quarter round. The configurations of inputs for the odd and even rounds are shown in tables 2.2 and 2.3, with each colour corresponding to a quarter round and the top row going to a, second to b, and so on. The quarter round function is entirely reversible; this means that the main of set of rounds is reversible. Therefore, to make the cipher not a permutation, the input is added to the output of the final round for the cipher's final output, the 16 32-bit word state.

ChaCha has become widely adopted both as a cipher and for pseudo-random number generation. Some of the more notable adoptions (often including Bernstein's Poly1305 [8]

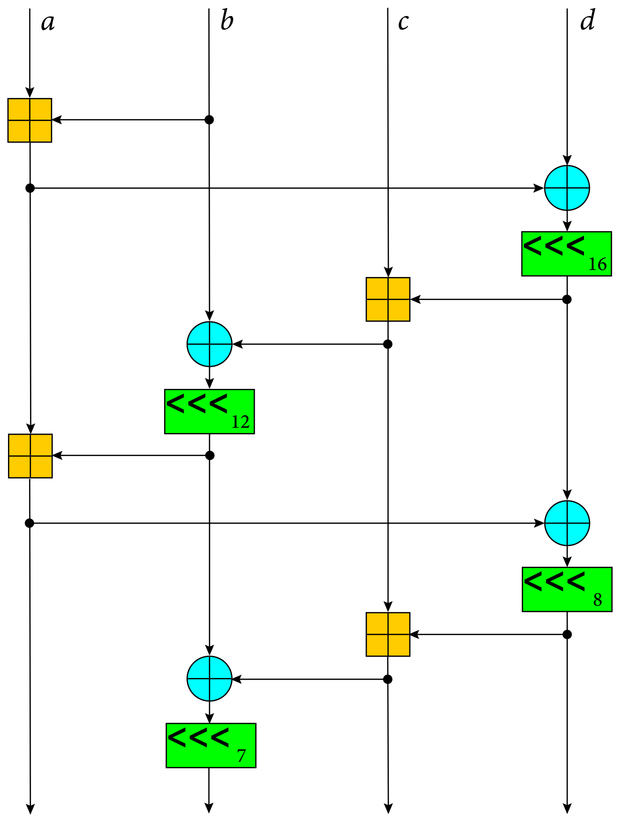


Figure 2.1: ChaCha quarter round function, diagram taken from [7]

Table 2.2: Odd round configuration of quarter rounds, each colour representing an individual quarter round and the numbers showing the position in the overall state. The inputs to each quarter round are in order according to their position in the state

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Table 2.3: Odd round configuration of quarter rounds, each colour representing an individual quarter round and the numbers showing the position in the overall state. The inputs to each quarter round are in order according to their position in the state

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

as a message authentication code) include as a cipher in TLS version 1.3 [9] and as a default cipher in OpenSSH [10]. It is also used extensively as a pseudo-random number generator; for example, in FreeBSD [11], OpenBSD [12], and NetBSD [13], it currently implements the `arc4random` generator, replacing RC4, and in the Linux kernel has implemented the `/dev/urandom` device since version 4.8 [14].

There has been some recent discussion about whether the number of rounds is too conservative, unnecessarily sacrificing performance [15], with reduced round variants becoming popular for random number generation. Go uses an 8-round version as the default PRNG [16], and Rust uses a 12-round version [17]. ChaCha is also the basis of the BLAKE hash function [18], a finalist in the NIST hash function competition [19]. There are many reasons behind its popularity, but one of the main ones is that software can efficiently implement it without specialised instructions and it allows for simple hardware accelerators.

2.2 Side-channel attacks

Side-channel attacks are a class of attacks against protocols and algorithms that use flaws in their implementations to extract information about intermediate values rather than flaws in their theoretical design. Side-channel attacks include *timing* and *power analysis* attacks. In timing attacks, the attacker measures the time of specific operations to infer the data they operate on. Timing attacks can be performed through the cache in transient execution attacks, as shown in Spectre and Meltdown [20]. Power analysis attacks instead look at the amount of power drawn by the hardware when performing certain operations to infer the values operated on.

Power analysis side-channel attacks can be broken into two categories:

- *Non-profiled attacks* – The attacker does not have detailed knowledge of the implementation. The attacker records lots of power traces with the same secret key, and statistical relationships are found between the traces to recover the secret key. Common types of non-profiled attacks are Differential Power Analysis (DPA) [21] and Correlation Power Analysis (CPA) [22].
- *Profiled attacks* – The attacker has access to the implementation of the algorithm they are attacking, and they have access to a set of input values with associated power traces. This access allows the attacker to train a classifier on the implementation to classify the value a particular variable takes. Then, during the attack, the classifier gives probabilities for all the values the variable could take. Profiled attacks can extract significantly more information per attack trace, making it possible to recover the key from a single trace.

A typical profiled attack consists of two main stages: training and attack. The training stage generally consists of the following steps:

- Trace recording – the attacker makes power supply current recordings of the system with known inputs. Ideally, they choose the inputs randomly to help avoid correlations with environmental factors such as temperature.
- Building templates for different parts of the algorithm's state. This can consist of several sub-steps:
  - Interesting point selection – analyse the traces to determine which samples contain significant amounts of information related to a targeted location. The high number of samples (for the attacks in section 4.2 over 650 000) makes it infeasible to characterise the distributions over all of them. It is often helpful to also select points around interesting points.
  - Dimensionality reduction – is an optional step to reduce the number of dimensions the templates use, improving their runtime and (hopefully) accuracy. This is commonly done by Principal Component Analysis (PCA) [23] or Linear Discriminant Analysis (LDA) [24].
  - Template profiling – find the means and covariance matrices of the different values the targeted location can take.

The attack stage consists of the following steps:

- Trace recording – the attacker collects power supply current recordings of the device when performing the encryptions they want to attack.
- Point selection and dimensionality reduction – select the same interesting points selected in the training stage and perform the same dimensionality reduction.
- Apply templates – use a multivariate Gaussian model, with the covariance matrices and means from the template, to give the likelihood that the attack trace contains each value.
- Increase success rate – if the templates are not of high quality, techniques such as SASCA and key enumeration can increase the attack's chance of success.
- Estimate the attack's performance – in experimental settings it can be helpful to find the suitability of attacks. This is often done by estimating the number of key candidates that an attacker would have to enumerate to find the correct key.

In profiled attacks, various types of leakage can be extracted, including the Hamming weight and the Hamming difference from the previous value of the register. It is also possible to get probabilities of individual values. The values can be for the entire word size as in *template attacks* [2] or for parts of the word as in *fragment template attacks* [3].

Examining the quality of templates these methods have created can also be helpful. Two standard metrics [25] used for this purpose are:

- *n-order success rate* (n-SR) – the probability that the correct value is in the n most likely. The higher the rate, the better the quality of the template. The most common version used is first-order success rate, the probability that the most likely value is correct. For a perfect template, the first-order success rate is 1; for an informationless template (one giving equal likelihood to all values), it is $\frac{1}{k}$, where k is the number of potential values.

- *Logarithmic guessing entropy* (LGE) – the base two logarithm of the expected number of guesses required to find the actual value. The lower the value, the better the quality of the template. For a perfect template, it is 0; for an informationless template, it is $\log_2(k) - 1$, where k is the number of potential values.

2.3 Soft analytical side-channel attack

A side-channel attack, as described above, gives probability distributions for each intermediate value and part of the key. We can use these probabilities directly; however, they only consider information that directly leaks their value. SASCA [5] allows for combining leakages from all intermediate values by providing a probabilistic model of the algorithm that we can marginalise over.

We can create a *factor graph* to represent the algorithm of interest, which consists of the following types of node:

- *Variable nodes* – represent the distribution of values which a particular point in the execution can take. They are the circular nodes in the graphical representation.

- *Factor nodes* – impose limitations on the values variables can take and their likelihoods and are shown as rectangular nodes in the graphical representation. We can break them down into two types:

  - *Constraint factors* – represent the operations of the algorithm, imposing which combinations of values are valid.

  - *Observation factors* – represent the leakage probabilities output from the side-channel attack.

Figure 2.2 shows a factor graph representing the code:

$$\begin{aligned} c &= a + b \\ d &= b \oplus c \\ a &= b \mathbin{\&} d \end{aligned}$$

where the subscripts on the variables correspond to their version, and f_{a_0} gives the leakage probability distribution of a_0.

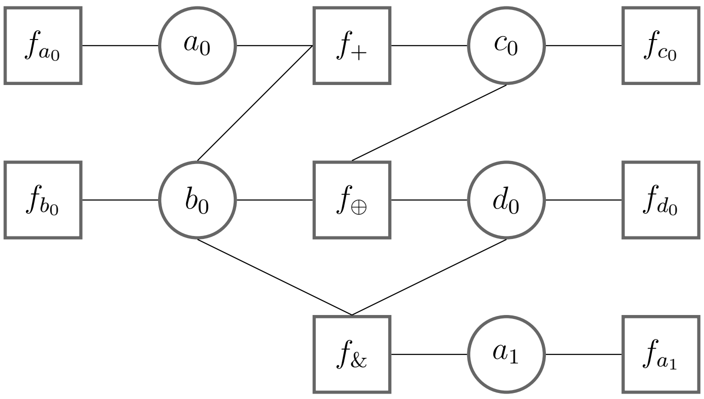


Figure 2.2: Small example factor graph

Belief propagation marginalises the output probabilities in the factor graph and has been implemented through message passing [26]. It will produce an exact marginalisation on trees, whereas its marginalisation may not be exact for graphs containing cycles.

Belief propagation has two types of messages: *factor-to-variable* messages ($r_{m\to n}$) and *variable-to-factor* messages ($q_{n\to m}$).

The factor-to-variable messages are calculated by:

$$r_{m\to n}(x_n) = \sum_{\boldsymbol{x}_m \setminus n} \Big(f_m(\boldsymbol{x}_m) \prod_{n' \in \mathrm{Neigh}(m)\setminus n} q_{n'\to m}(x_{n'}) \Big)$$

and the variable-to-factor messages are calculated by:

$$q_{n\to m}(x_n) = \prod_{m' \in \mathrm{Neigh}(n)\setminus m} r_{m'\to n}(x_n)$$

where $\mathrm{Neigh}(n)$ gives the set of neighbours of node n, f_m gives the factor's data to the graph (e.g. the outputted likelihoods from the side-channel attack or which combination of values are valid), $\boldsymbol{x}_m$ is the set of variables which the mth factor depends on, and $\boldsymbol{x}_m \setminus n$ denotes $\boldsymbol{x}_m$ with the variable x_n excluded.

The marginal distribution for a variable can then be calculated by:

$$P_n(x_n) = \frac{Z_n(x_n)}{Z} = \frac{\prod_{m\in \mathrm{Neigh}(n)} r_{m\to n}(x_n)}{Z}$$

where Z is a normalising constant (calculated by summing the numerators across all values of x_n, i.e. $Z = \sum Z_n(x_n)$).

We can order the calculation of the messages in trees so that each only needs to be calculated once. This is impossible for graphs containing loops, but we can use a similar technique where we initialise all variable-to-factor messages as 1. Then, we perform iterations of sending all factor-to-variable messages and then all variable-to-factor messages. Iteration continues until the variables' distributions converge to a stationary point or we

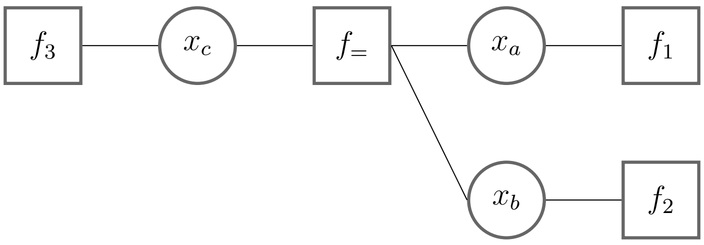


Figure 2.3: Example factor graph for equality

reach an iteration limit.

Figure 2.3 shows a small example factor graph representing equality between three variables (in this case, assumed to be one bit), each of which has an associated observation factor. The tables for the different constraints and observations are as follows:

$$f_1(x_a) = \begin{cases} 0.8 & x_a = 0 \\ 0.2 & x_a = 1 \end{cases}$$

$$f_2(x_b) = \begin{cases} 0.7 & x_b = 0 \\ 0.3 & x_b = 1 \end{cases}$$

$$f_3(x_c) = \begin{cases} 0.9 & x_c = 0 \\ 0.1 & x_c = 1 \end{cases}$$

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

The following will be performed to calculate the marginal distribution of x_a (using the tree message passing order). Calculate the numerator in the $x_a = 0$ case:

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

The following rules can then be used to update the values:

$$q_{b\to =}(x_b) = r_{2\to b}(x_b) = f_2(x_b)$$

$$q_{c\to =}(x_c) = r_{3\to c}(x_c) = f_3(x_c)$$

hence:

$$\begin{aligned} Z_a(x_a = 0) &= 0.8 \times [f_2(x_b = 0) \times f_3(x_c = 0)] \\ &= 0.8 \times [0.7 \times 0.9] \\ &= 0.504 \end{aligned}$$

Similarly:

$$\begin{aligned} Z_a(x_a = 1) &= f_1(x_a = 1) \times [f_2(x_b = 1) \times f_3(x_c = 1)] \\ &= 0.2\,[0.3 \times 0.1] \\ &= 0.006 \end{aligned}$$

Therefore the marginal distribution is as follows:

$$P_a(x_a) = \begin{cases} \frac{0.504}{0.504+0.006} = 0.988 & x_a = 0 \\ \frac{0.006}{0.504+0.006} = 0.012 & x_a = 1 \end{cases}$$

When working with loopy graphs, it is possible to use a different *schedule* for sending messages compared to sending between all nodes on every iteration. Different schedules can improve convergence speed by allowing information to flow more efficiently around the graph, but a poorly chosen schedule can significantly slow it down or prevent convergence [27]. In an extreme example of variables that are either completely unknown or have a set value, the only messages carrying information around the factor graph are those from known variables to unknown variables (clearly, they must go through a factor).

Loops in factor graphs can lead to oscillations, which we want to avoid. The introduction of oscillations makes us want to avoid introducing unnecessary loops (especially small ones) into the factor graph. We can reduce these oscillations by *damping* the message updates. Damping makes the messages sent the weighted sum of the new value calculated and the previous value of the message. A message (u) sent around the graph is calculated as follows:

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

where α is the damping rate, the smaller the rate the greater the amount of damping, u_{new} is the message calculated by the previous equations and u_{prev} is the previous value for this message.

2.4 Key enumeration

Probability tables output directly from a side-channel attack or SASCA will be subject to noise. This means that the correct values may not be the most likely. It is helpful to test potential combinations in order of their likelihood when searching for the correct key. To perform this enumeration, the attacker must have a known output and input (except for the part they are enumerating) to verify that they have found the correct key.

Charvillon et al. [28] propose an algorithm that produces the keys in likelihood order from a series of probability tables. Below, the basic ideas for combining two tables are described.

Let s and w be two sorted probability tables for two parts of a key so that s_1 (and w_1) represents the most likely value and s_n represents the least likely value (assuming there are n potential values). The algorithm maintains a *frontier* of the potential combinations of values which could be the next most likely. Figure 2.4 shows the algorithm's initial state, where each cell represents combining the two values from the likelihood tables, with white cells being completely unexplored and the red ones being in the frontier. The only value initially in the frontier is the most likely combination.

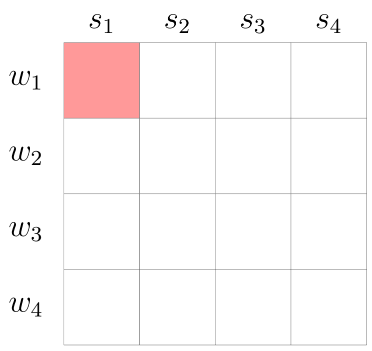


Figure 2.4: Initial unexplored state through the probability tables

The attacker picks the most likely combination from the frontier when selecting the next most likely combination. At that point, they must update the frontier to capture any new combinations that could be the next most likely. Due to working with sorted probability tables, cells in a row to the right of a cell will have a lower likelihood than that cell (and the same applies to columns going downwards). This means they only need to add newly created concave corners of the enumerated space to the frontier. Figure 2.5 shows the locations the frontier contains part-way through exploring the keys, where the grey-shaded cells have already been tested.

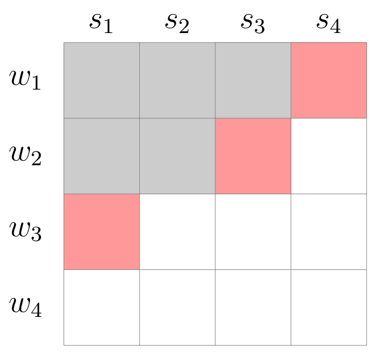


Figure 2.5: State part way through exploration of states, gray shaded cells have already been explored and the frontier is in red

The above process continues until the attacker finds the correct combination of values or reaches an iteration limit.

The above ideas can be extended to support having more than two probability tables in various ways.

One method is to work directly in a higher dimensional space, where each sorted probability table has its own dimension. The frontier is maintained in this space, and the concave corners added to the frontier as they are in the two table case.

An alternative method is to use a recursive binary tree like structure, where each node outputs the next most likely combination of the tables they are combining. The leaf nodes of the tree represent the individual probability tables and output them in order. Then the remaining nodes implement the described method for combining two probability tables outputting them in order with their probability. Each probability table (and combinations of them) is accessed in order so the entire distribution does not need to be known when expanding the frontier. This means that parent nodes only need to request the next most likely combination on the first access to a cell in a particular row or column (which will be the left most or top cell of that row or column).

2.5 Rank estimation

A powerful attacker can likely enumerate 2^{64} key candidates but such a large amount of key enumeration is often infeasible for people proposing attacks. This means it is helpful to estimate how many keys would need to be enumerated from a given set of probability tables without actually enumerating them (this problem is frequently called *rank estimation*). Rank estimation is impossible in an actual attack because it requires knowledge of the key used during the encryption. However, it is still useful when evaluating an attack in an experimental scenario.

Glowacz et al. [29] propose an algorithm for rank estimation. It works by constructing a histogram for each given probability distribution in a logarithmic scale. Then, it approximates the overall histogram of the distribution of keys by convolving all the individual histograms together. Then, to estimate the number of keys that need to be enumerated, it sums the values of all bins that are more or equally likely than the actual key.

Convolving all the histograms together is not trivial due to the large number of potential secret values (e.g. for ChaCha's key there are 2^{256}), preventing the use of standard library fast convolution functions due to not fitting in standard sized integers.

One way to perform the convolution is to slide the two histograms over each other, multiplying and summing them with arbitrary sized integers.

Better performance can be achieved by using the Chinese Remainder Theorem, because it allows the use of standard library fast convolution functions. This works by having a set of prime numbers where the convolutions are done modulo each prime number in this set, meaning that the values can fit in standard sized integers. This means that the number of convolutions will be higher but can still achieve better performance. Then the set of resulting histograms can be combined together to produce the final histogram with arbitrary sized integers. For ChaCha's key of 256 bits, the 20 largest prime numbers below 10 000 allow the unique identification of every potential value in the final histogram.

2.6 Related work

2.6.1 Efficient template and fragment template attacks

Choudary and Kuhn [30] cover several techniques which are useful when performing template attacks. These include using a pooled covariance matrix across all values, rather than each value having its own covariance matrix, because it reduces the number of traces required to get a good estimate. They also compare several methods for compressing the recorded samples, including selecting several points per clock cycle, PCA and LDA. Their practical guidance and our desired attack setting of few attack traces suggests using a pooled covariance matrix with LDA.

Choudary and Kuhn did not provide methods that would scale well for working directly with 32-bit templates. Several approaches have been proposed for working with 32-bit parallel buses, including the fragment template attack [3]. Fragment template attacks break down the 32-bit values into a series of fragments. Each fragment has a template built independently, with the other bits treated as noise. It has been shown to work well for Keccak [3], which naturally contains lots of memory accesses, allowing the building of high-quality templates, and also works for Ascon [4] although not as well.

In contrast, Cassiers et al. [31] show further optimisations that make creating 32-bit templates feasible. Allowing for better performance than fragment template attacks when run on a small number of operations, such as single XOR or an Ascon-p permutation, which fit entirely in registers. They also performs SASCA on the Ascon-p permutation with 32-bit values to maximise the use of the extracted information. Their implementation of SASCA requires storing 35 distributions, using 1.13 TiB of RAM and taking 2.7 hours. The memory and time requirements show that performing SASCA directly on 32-bit values is currently challenging for more extensive algorithms such as ChaCha.

2.6.2 Template attacks against Keccak and Ascon AEAD

Kannwischer et al. [32] propose a factor graph for Keccak, evaluated with simulated leakages, and serves as a base design for actually attacking Keccak in [3].

This shows that working with simulated leakages is useful when designing a factor graph before working with actual attacks. They also show how clustering bits together (even independently of the processor's word size) can help achieve good results. However, factor-to-variable messages will have runtimes of O(2^{cd}), with c bits per cluster and d connected variables. They also show methods which can improve the performance of certain factors (e.g. XOR) compared to the naive method of multiplying full probability distributions for larger clusters.

You et al. [4] perform a fragment template attack against ASCON [33], the winner of the NIST Lightweight Cryptography competition (2019–2023) [34], on a 32-bit microcontroller.

They attacked several implementation variations, such as changing the compiler's optimisation level and with or without *boolean masking*. Boolean masking [35] is a technique that helps prevent leakage through power side-channels by obfuscating values. The targeted masked implementation of ASCON [33] used ChaCha to generate the values used for masking. If a fragment template attack from a single trace is possible against ChaCha, then an attacker could attack this masked implementation from just the power leakage without knowing what values were used for the masking because the factor graph can also model the masking steps.

Chapter 3

Factor graph design

This chapter describes several structures that can be combined to create a factor graph for ChaCha. It also describes some initial experimentation with belief propagation on a factor graph. This experimentation helps support the choice of certain parameters for belief propagation in actual attack scenarios.

3.1 Graph structure designs

ChaCha consists of three types of operation (ADD, XOR and ROR), each of which must be modelled in the factor graph. The factor graph will naturally contain loops due to the structure of ChaCha. Most implementations of ChaCha stick closely to the description in the original paper, meaning it is sensible to have a factor graph for that version.

The 32-bit values operated on in ChaCha would be a natural size for the variables in the factor graph, but it is currently challenging to work with 32-bit values in large factor graphs. Breaking each value in the algorithm into clusters of c bits (with C clusters per 32-bit value, so $c \times C = 32$) will improve performance due to the smaller sizes. However, we may lose information by treating the clusters independently.

When operating on clusters, it is possible to implement any operation ($\odot$) as a single factor. For representing $x \odot y = z$, a factor with the following table could be used:

$$f_{\odot}(x_1, \ldots, x_C, y_1, \ldots, y_C, z_1, \ldots, z_C) = \begin{cases} 1 & \text{if } (x_C || \cdots || x_1) \odot (y_C || \cdots || y_1) = (z_C || \cdots || z_1) \\ 0 & \text{otherwise} \end{cases}$$

This does not lead to the desired performance improvements because the factor is still really operating on 32-bit values despite only working with variables of c bits. When operating on clusters, the factors cannot be connected directly to all clusters that make up a value.

The structures proposed in this section were checked by several techniques including:

- Running with a known input of the ChaCha test vectors (provided in RFC 8439 [6]) with all other values unknown until it reaches a fixed point and then taking the output. This ensures that the factor graph correctly implements the algorithm for values flowing forwards through the graph.
- Running with a known input and output of ChaCha with all other values unknown until it reaches a fixed point, and checking that the distributions of variables remain valid. This allows the checking that information can also flow effectively backwards through the graph as would be expected in ChaCha due to the main rounds implementing a permutation.
- Running with all variables set to a particular encryption and checking that it remains at the fixed point. This ensures that it will stay at a valid fixed point of an encryption.
- Running with all variables set to a particular encryption except one is perturbed. This ensures that the invalid distributions can propagate through the entire graph.
- Running with all variables set initially to uniform distributions and ensuring that a fixed point is reached with all variables having a distribution near uniform. This gives us confidence that the algorithm has been correctly implemented and that the structures do not lead to prefering certain values which are not preferred in the actual algorithm.

3.1.1 XOR

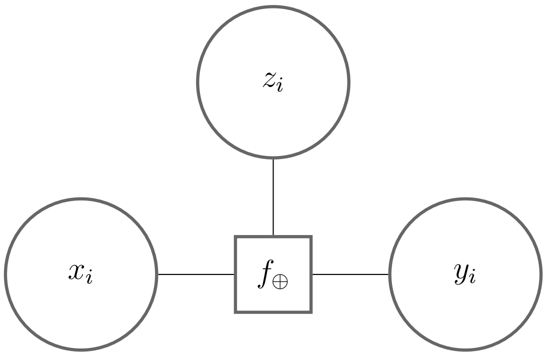


Figure 3.1: Structure for XOR in factor graph

XOR is the simplest operation to implement in a clustered factor graph because it is entirely bitwise. $x \oplus y = z$ can be implemented by the structure shown in figure 3.1 where $f_\oplus$ has a function of:

$$f_\oplus(x_i, y_i, z_i) = \begin{cases} 1 & \text{if } x_i \oplus y_i = z_i \\ 0 & \text{otherwise} \end{cases}$$

and is between x_i, y_i and z_i.

3.1.2 ROR

Rotation by amounts divisible by the cluster size can be performed by renaming other operations' input/output clusters, requiring no additional factors. However, unless we use 1-bit clusters, not all rotations can be performed this way. Therefore, we must add a structure for performing these rotations, which can be limited to less than the cluster width. This structure will introduce loops into the factor graph because it rotates a value. Several structures can perform an r-bit rotation ($x \lll r = y$) by connecting to the ith cluster and $(i-1)$th cluster (looping around to the top cluster from the bottom cluster).

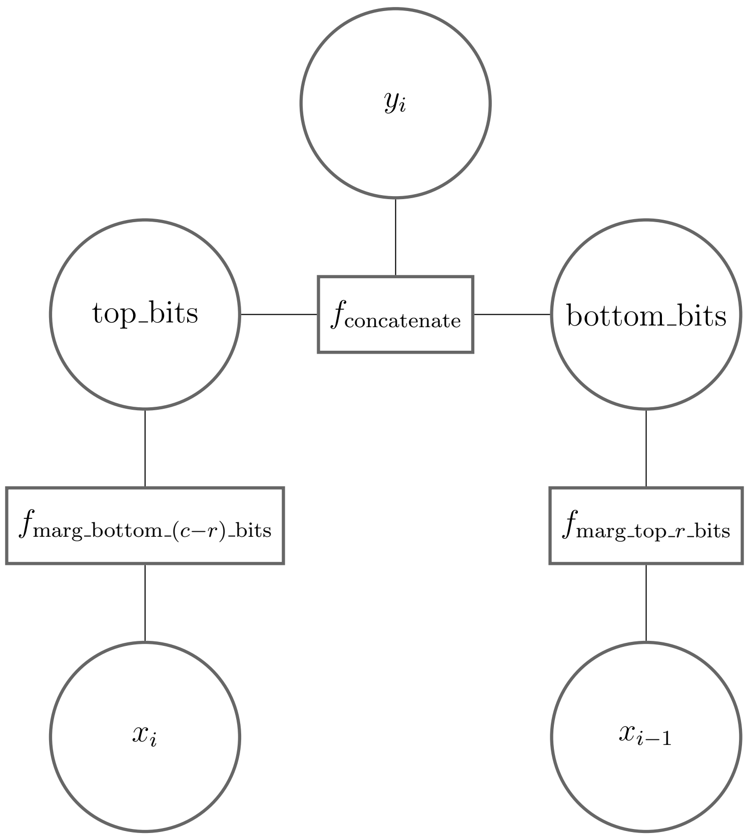

y_i
top_bits
$f_{\text{concatenate}}$
bottom_bits
$f_{\text{marg\_bottom\_}(c-r)\text{\_bits}}$
$f_{\text{marg\_top\_}r\text{\_bits}}$
x_i
x_{i-1}

Figure 3.2: Potential structure for ROR in factor graph with explicit bit selection

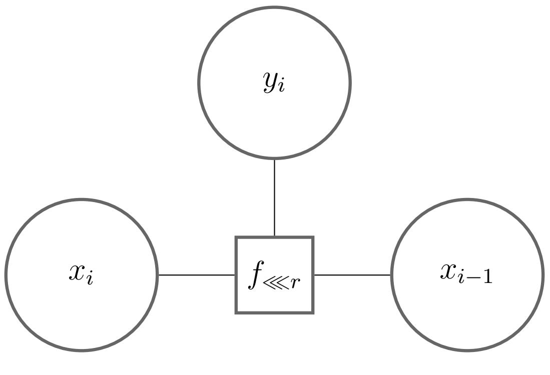


Figure 3.3: Potential structure for ROR in factor graph

Figure 3.2 shows a structure that selects the lower $(c-r)$-bits and upper r-bits of clusters combined for the output value. Figure 3.3 shows an alternative structure which does not select the bits explicitly, instead taking the full values of both clusters, and having a constraint table of:

$$f_{\lll r}(x_{i-1}, x_i, y_i) = \begin{cases} 1 & \text{if } (x_{i-1} \gg (c - r)) \vee (x_i \ll r) = y_i \\ 0 & \text{otherwise} \end{cases}$$

where $\gg$ and $\ll$ are (zero-padded) shifts right and left. The version shown in figure 3.3 was selected for use due to it introducing fewer nodes into the factor graph, not leading to much shorter loops and having a very similar structure without the explicit bit selection.

3.1.3 ADD

Addition is a more complicated operation to implement inside a clustered factor graph because the output value for each cluster depends on the values from the corresponding clusters in the input values and their other clusters. The structure of a ripple carry adder is helpful to examine when considering how to model addition in the factor graph, due to its ability to create a 32-bit adder by using only single-bit full adders with carry bits between them. It is natural to introduce carry bits between each cluster to simplify the movement of information between the clusters.

Below, I propose a couple of potential structures for c-bit addition with carry bits ($x+y+c_{\text{in}} = z+2^c \cdot c_{\text{out}}$). For the lowest cluster, an observation factor can set the carry-in bit to 0.

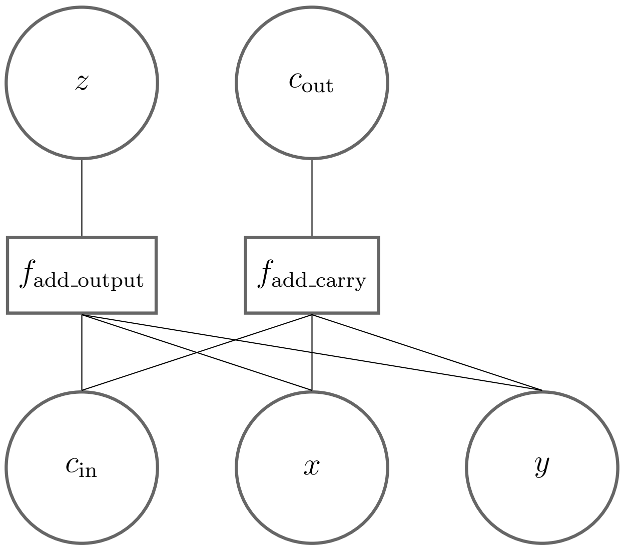


Figure 3.4: Potential structure for ADD in a factor graph introducing loops

Figure 3.4 shows the most direct structure for implementing c-bit addition with carry bits. The two constraint factors have the following tables:

$$f_{\text{add\_carry}}(c_{\text{in}}, x, y, c_{\text{out}}) = \begin{cases} 1 & \text{if } ((c_{\text{in}} + x + y) >= 2^c) = c_{\text{out}} \\ 0 & \text{otherwise} \end{cases}$$

$$f_{\text{add\_output}}(c_{\text{in}}, x, y, z) = \begin{cases} 1 & \text{if } ((c_{\text{in}} + x + y) \bmod 2^c) = z \\ 0 & \text{otherwise} \end{cases}$$

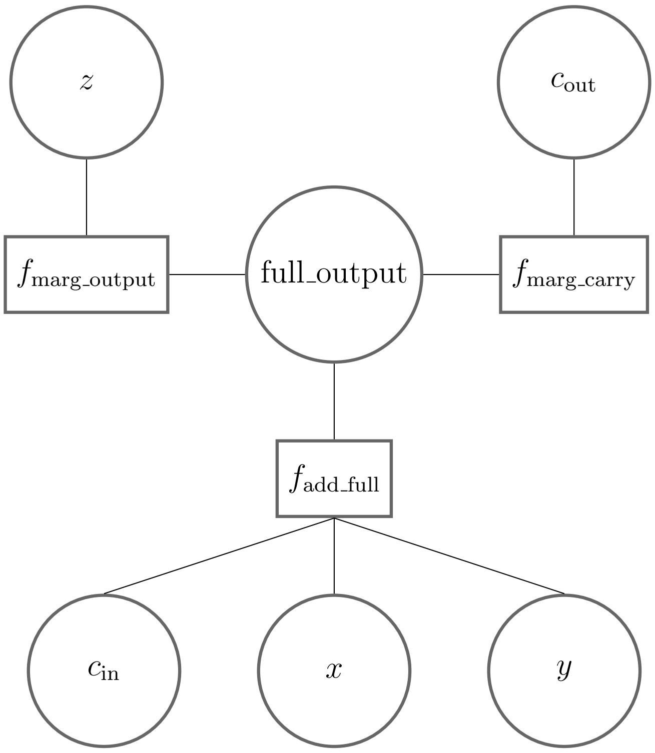


Figure 3.5: Potential tree structure for ADD in a factor graph

The design in figure 3.4 introduces many small loops into the factor graph. Generally, we want to avoid small loops in factor graphs because they can lead to oscillations, potentially preventing convergence to the correct value. Figure 3.5 shows an alternative structure which does not introduce loops into the factor graph. It adds a new variable full_output representing the $(c+1)$-bit output of the addition, which can then have its top bit selected for the carry-out bit and the remaining bits for the output of this cluster. The constraint factors have the following tables:

$$f_{\text{add\_full}}(c_{\text{in}}, x, y, \text{full\_output}) = \begin{cases} 1 & \text{if } c_{\text{in}} + x + y = \text{full\_output} \\ 0 & \text{otherwise} \end{cases}$$

$$f_{\text{marg\_output}}(\text{full\_output}, z) = \begin{cases} 1 & \text{if full\_output mod } 2^c = z \\ 0 & \text{otherwise} \end{cases}$$

$$f_{\text{marg\_carry}}(\text{full\_output}, c_{\text{out}}) = \begin{cases} 1 & \text{if } (\text{full\_output} >= 2^c) = c_{\text{out}} \\ 0 & \text{otherwise} \end{cases}$$

3.1.4 Combining small structures

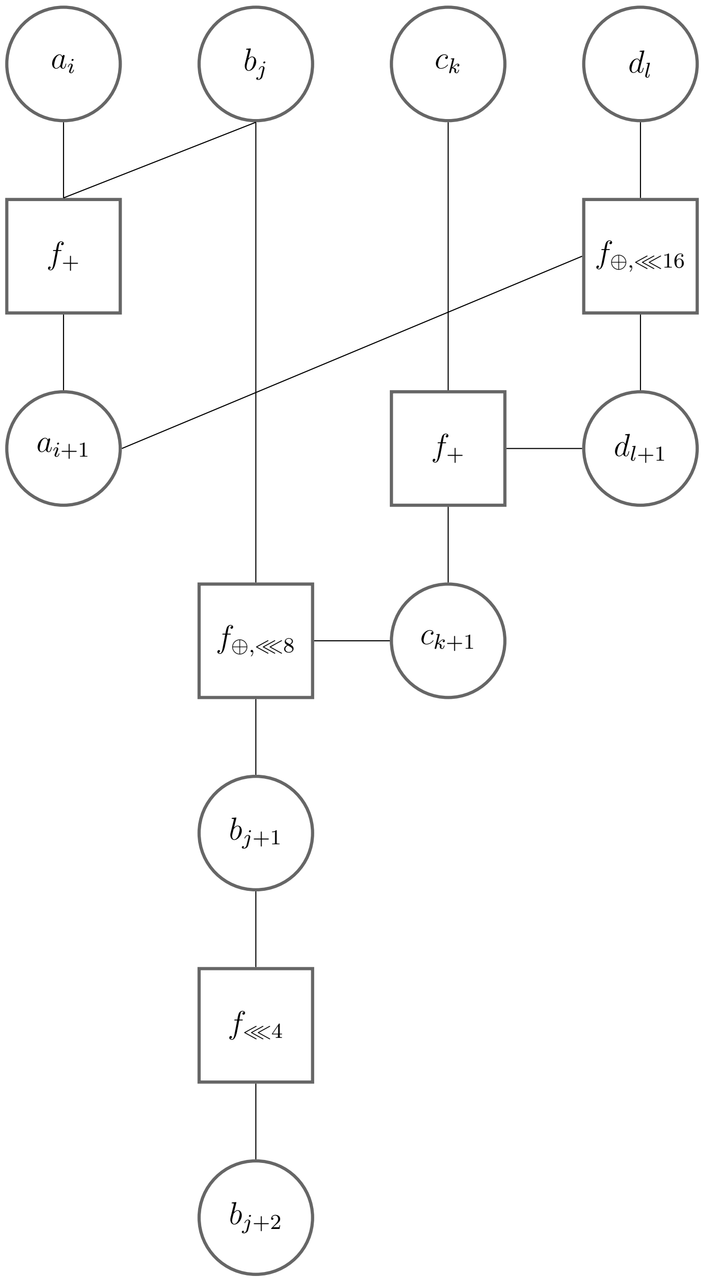


Figure 3.6: Overview of first half of a quarter round factor graph when using 8-bit clusters, the variables and factors would actually be implemented by the clustered versions so it would consist of 54 variables and 38 factors when using the tree structure for ADD

The previous structures provide ways to implement the basic operations of ChaCha. To make the factor graph for ChaCha, they can be combined to make the quarter round, which can then be combined into complete rounds and, subsequently, the entire algorithm with a final ADD at the end. Figure 3.6 shows an overview of the variables and factors needed for the first half of a quarter round. The subscripts indicate the version of the variable and XOR can perform rotations by amounts divisible by 8 bits because it is assumed to be working with 8-bit clusters. The variables and factors shown would actually consist of several variables for each cluster and be replaced with the previously discussed structures.

3.1.5 Supporting several encryptions

Using several encryptions with the same key can take advantage of more leakages. There are several methods for modelling multiple encryptions in a factor graph.

A simple method is to make an independent factor graph for each encryption, where the values that should be equal between the different encryptions (e.g. the key) have an extra constraint factor added between them, for enforcing equality. Leakages for the variables can be added as they are in single runs. This method introduces additional factors which negatively impact performance.

Another method is to have the factor graphs for each encryption share the variables which are equal; this requires no additional factors. I chose the second method due to its better runtime and additional flexibility in combining leakages.

If several traces have the same input state, then only the factor graph for a single execution is required, where the leakage probabilities for each variable have been calculated from the collection of traces.

It can be helpful to enforce simple known differences between variables in different encryptions. For example, if the counter (part of the input which is incremented between consecutive encryptions) is unknown between runs, then an ADD could be introduced between each run with a constant value of 1 for one of its inputs. ADDs could also be inserted between all pairs of encryptions with varying differences, which should allow faster information propagation between runs, but the additional ADDs would also introduce loops into the graph, which we want to avoid. There are other ways to enforce differences, such as having a shared value from which each run has a known difference, which may lead to fewer loops. I did not use these methods to introduce differences between values because I assume the counter is known for each encryption in the multi-trace scenarios and the remaining part of the input is constant between encryptions.

There are several ways to combine a set of leakages for a shared variable. The set of leakages can be added to the factor graph as a set of individual observation factors. This is not ideal because it can lead to a higher chance of floating point errors due to (potentially) very small values, and their product is taken on every iteration despite not changing. A single observation factor of the product of the individual leakage probabilities provides the same information while removing the product on every iteration. It also simplifies ensuring that values do not get too extreme. Alternative methods can be used, such as averaging the set of traces before making the probability distribution. The best technique for combining several runs will depend on the type of noise that affects the traces and how well additional traces reduce it.

3.2 Initial belief propagation experiments

We can change several parameters inside the factor graph and belief propagation, such as the structure for ADD, the size of clusters, the schedule for sending messages, and the amount of damping. The values for these were chosen to try and maximise the chance and speed of convergence for actual attacks by performing several experiments described below. These initial experiments also highlighted a need for optimising belief propagation and revealed interesting findings about information flow.

3.2.1 ADD design

Table 3.1: Statistics for known inputs and outputs with unknown intermediate values with the loopy add structure

Bits per cluster	Number of iterations	Time per iteration	Total time
1	189	0.627 s	119 s
2	117	0.389 s	46 s
4	72	1.383 s	100 s

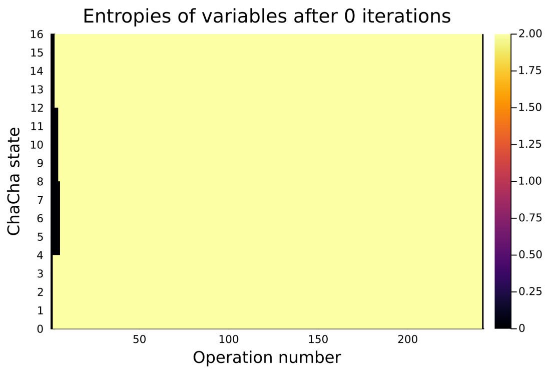
Table 3.2: Statistics for known inputs and outputs with unknown intermediate values with the tree add structure

Bits per cluster	Number of iterations	Time per iteration	Total time
1	181	0.668 s	121 s
2	114	0.426 s	49 s
4	79	2.168 s	171 s

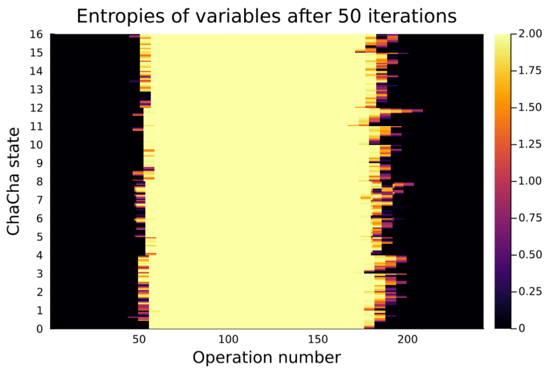
Tables 3.1 and 3.2 show the number of iterations (of the simple schedule, with no damping) taken to converge when run with known input and output (and uniformly distributed intermediate values), for the loopy and tree structures respectively. They also show the time taken which has been timed by the Julia macro `@benchmark` on an AMD Ryzen 9 8945H.

They clearly show that reducing the size of the clusters increases the number of iterations required to converge but can reduce the time each iteration takes. The 2-bit clusters take less time per iteration than the 1-bit clusters due to having fewer nodes in the graph, so fewer messages are sent, while the time taken to compute each message has not increased significantly.

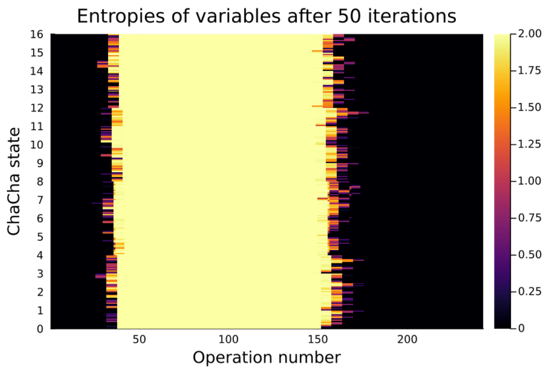
They also show that the tree structure takes fewer iterations to converge for smaller clusters than the loopy structure, although this inverts for larger clusters. Each iteration for a particular cluster size takes longer for the tree version due to having more nodes and introducing a larger variable. This shows how the loopy structure for ADD can lead to faster information flow in cases of known values propagating through the graph.



(a) Initial Shannon entropies of variables with known input and output



(b) Shannon entropies of variables after 50 iterations with loopy adds when run with known input and output



(c) Shannon entropies of variables after 50 iterations with tree adds when run with known input and output

Figure 3.7: Comparison of the state after 50 iterations of belief propagation in the case of known input and output, but no observation factors for intermediate variables, with 2-bit clusters with also the initial state shown

Figure 3.7 shows the state of the factor graph initially and also after fifty iterations of belief propagation when using the loopy and tree structures. Each heatmap shows the Shannon entropy of each intermediate value (which is present in ChaCha and the factor graph). The black pixels have completely known values while the light yellow ones have a uniform distribution. The y-axis represents the cipher's state, and the x-axis represents moving an operation through ChaCha. Each variable can occupy several pixels due to operations not affecting the whole state.

It shows that the information in these cases flows from the outside (the input and output of ChaCha), where the known values start, into the centre. It also shows that the information flows significantly faster backwards through the graph when using the tree structure for ADD than the loopy structure, there is a larger area of no entropy near the output. However, information flows slightly slower forwards through the factor graph with the tree structure than the loopy structure.

Faster information flow is helpful when working with leakages because we want the information to flow as quickly as possible from the different parts of the graph to each other. Section 3.2.3 contains a comparison with XOR and explains the overall reason for information flowing faster backwards than forwards.

Table 3.3: First-order success rates with different structures for adds with the leakages taken from previous KECCAK experiments

Cluster size	**Tree adds**	**Loopy adds**
1	0.84	0.35
2	0.95	0.88
4	0.99	0.97

It is essential to consider the success rates of the different ADD structures with more realistic leakages rather than just with known inputs and outputs. Table 3.3 shows the first-order success rate of the key after 200 iterations (or fewer if converged beforehand) for both the loopy and tree versions with different cluster sizes. The leakages used are from the KECCAK template set described in section 4.1.2 with the keys using set A, ADD set C, ROR set B and a damping value of 0.8. The tree structure obtains higher success rates than the loopy structure, particularly with smaller cluster sizes. Larger cluster sizes help improve the chance of converging to the correct values in both cases because they allow bits inside the cluster to not be independent.

Going forward I will use the tree structure for ADD to maximise the chance of converging to the correct value with as large a cluster as feasible.

3.2.2 Factor optimisations

We want to use large clusters because they do not enforce independence between bits inside them, potentially increasing the amount of information used. The problem with larger clusters is that calculating a variable-to-factor message takes $O(d2^c)$ steps, and a factor-to-variable message takes $O(2^{cd})$ steps, where c is the number of bits per cluster and d is the number of connected nodes.

The optimisations described below focus on improving the performance of factors, which can be done by using information about their purpose, and dramatically reducing their memory use.

The performance of marginalisation and rotation factors can be improved by just working with the distributions of each value. For example, if a factor wants to marginalise the top bit from a cluster, the message to the top bit can be calculated by summing the first half (when the top bit is zero) and second half (when the top bit is one) of the message from the cluster. We can calculate other messages sent by these clusters in similar ways.

Improving the performance of XOR factors is more complex because it is not obvious how to avoid the general multiplication. However, previous work [32] has shown how they can be efficiently implemented by first applying the Hadamard transform to all input messages, multiplying the transformed versions together before transforming back.

Previous work [36] has shown that factors can efficiently implement modular addition by using the Fourier transform due to the addition really performing a circular convolution. I extended these ideas to support adding three values and variables of different sizes.

The optimisations for XOR and ADD introduce small values (lower bits of the mantissa) which are not present when just performing multiplications. These small values can be negative, which are invalid inside messages, and larger than values that are actually of interest. We can resolve these problems by replacing all values less than a threshold with that threshold. The larger the threshold, the smaller the chance of an invalid distribution, but information cannot flow as far through the factor graph. The messages sent around the factor graph are also normalised to prevent them from getting increasingly large or small.

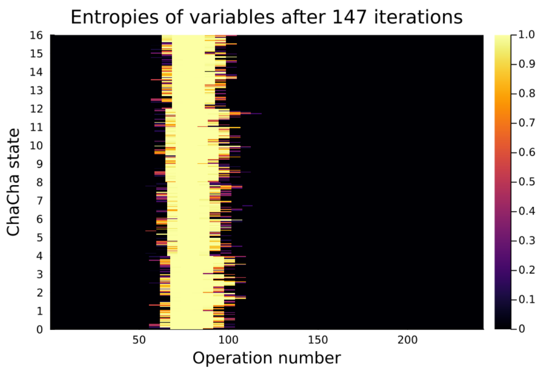
Table 3.4: Execution time of a full iteration before and after optimisations (both single threaded), timing is done via the Julia macro `@benchmark` on an AMD Ryzen 9 8945H

Cluster size	**Original implementation**	**Optimised version**
1	0.668 s	0.675 s
2	0.426 s	0.481 s
4	2.168 s	0.549 s
8	1.352 h	0.847 s

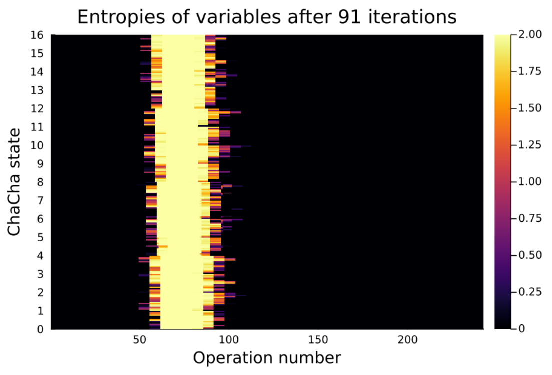
The results of the optimisations to the factors computations' are shown in table 3.4, where the original implementation represents all factors by multi-dimensional probability distributions, and the optimised version uses the previously discussed specialisations per factor type. It shows that optimisations do not have a large effect for smaller cluster sizes but that the improvement is significant for larger clusters.

We can apply other optimisations to belief propagation, such as ensuring that factors that will never change their message (e.g. observation factors) only calculate it on the first iteration. It is also simple to use multi-threading to improve performance because the calculation of all factor-to-variable and variable-to-factor messages are independent of each other.

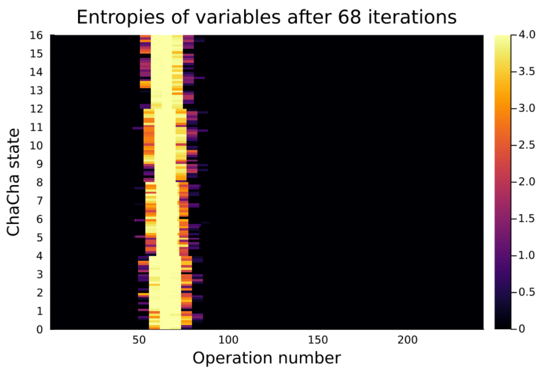
3.2.3 Information flow



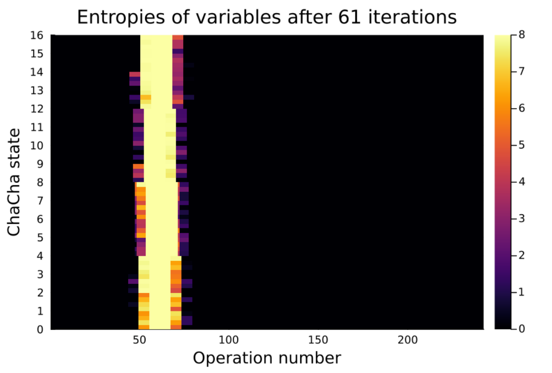
(a) 1-bit clusters factor graph entropies before finishing



(b) 2-bit clusters factor graph entropies before finishing



(c) 4-bit clusters factor graph entropies before finishing



(d) 8-bit clusters factor graph entropies before finishing

Figure 3.8: Entropies of variables in the factor graph with varying cluster sizes when run with known input and output, and no observation factors for intermediate variables

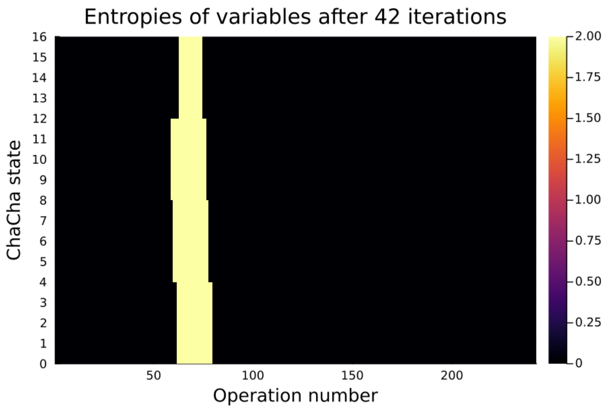


Figure 3.9: ADD replaced by XOR in the ChaCha factor graph with a known output and input, and no observation factors for intermediate variables

The design of ChaCha means that if the output of a quarter round is known, then the input can be calculated by performing the operations of the quarter round in reverse, which in turn means that the main twenty rounds are reversible. Figure 3.8 shows several runs of belief propagation slightly before it has converged to a steady state, with different cluster sizes. Figure 3.9 shows a modified version of ChaCha where XOR has replaced ADD.

In every case, information flows faster from the output than the input (the remaining entropy is nearer the input than the output). The faster flow of information is caused by the backwards version of the quarter round function having more instruction-level parallelism than the forward version, which belief propagation can take advantage of.

The ADDs introduce more uncertainty in the top bits of each value, which is introduced by the carry bits. This is most evident in the bit-wise factor graph, where there are visible triangles of entropy.

3.2.4 Scheduling

The schedule in which we send messages can dramatically affect the execution time of belief propagation by changing the number of messages required to converge or potentially stopping convergence altogether [27].

There are lots of potential schedules including:

- *Simple schedule* – every iteration sends all factor-to-variable messages and then all variable-to-factor messages. This guarantees that information can flow any way it wants inside the factor graph but potentially wastes a significant amount of time.
- *Forwards backwards* – every iteration runs through the factor graph of ChaCha forwards and then backwards. This is effective for other algorithms [32] and was helpful for unit tests with known information flowing through the graph.
- *End rounds* – every iteration performs the basic belief propagation schedule in the first and last n rounds of ChaCha, allowing it to take advantage of higher levels of information present at the end and start of ChaCha.

The best schedule to use depends on the quality of the templates and their distribution through the algorithm. For high quality templates throughout the entire algorithm, using a simple schedule is helpful because it allows the templates to provide information throughout the factor graph. The end rounds schedule is helpful with lower quality templates for intermediate values but much higher quality for the input and output (including known counter, nonce and output) because it still makes good use of the information provided while sending significantly fewer messages. The forwards backwards schedule was not very helpful when run with realistic leakages.

Animations showing the entropies of variables for an actual 8-bit attack are shown at `https://tbeakl.github.io/Part-III-Project-Visualisations/`, which clearly show how the information flows around the graph with different schedules.

The schedule selected in the evaluation depends on the scenario. Mainly the end rounds schedule was used (with a few iterations across the entire factor graph). The simple schedule was used when leakages were provided for every value (including the counter, nonce, and output) and for the simulations.

I briefly experimented with a greedy schedule, which attempted to remove as much entropy as possible. I found it to be useful for very high-quality templates and unit tests. However, for more realistic leakages, I found that it reduced the entropy quickly but brought the graph towards an incorrect solution. Another dynamic schedule would likely lead to better performance, such as dynamically selecting the rounds of ChaCha to pass messages in.

3.2.5 Damping

Different amounts of damping were experimented with; in cases of very high-quality or low-quality templates, it made little difference. However, damping significantly affects the results from borderline templates. Table 3.5 shows the first-order success rate for different damping rates. It shows that a small amount of damping can significantly improve performance but that larger amounts slow down the convergence to good values. For evaluating attacks, I used a damping rate of 0.95 due to it striking a good balance between finding solutions and good performance.

Table 3.5: The first-order success rate for different damping rates from 100 key combinations after 200 iterations of belief propagation with the simple schedule. The templates are in four dimensions where each bit has a **1** vector plus an offset picked randomly from a Gaussian with a standard deviation of 0.1. The noise level added was a Gaussian with a standard deviation of 0.4 with 2-bit clusters in the factor graph

Damping rate	**1-SR**
1.00	0.25
0.99	0.46
0.95	0.49
0.90	0.48
0.75	0.41
0.50	0.33

Chapter 4

Side-channel information generation

There are several leakage models for information that can be extracted from traces. These leakage models include the Hamming weight of values being operated on, the Hamming difference between the current value and the previous value of a register, and the actual value operated on. I first tested my factor graph on simulated leakage data where I have full control over the SNR, before attempting attacks on real traces. This chapter first describes some leakage models used in simulations, before describing the process I used for performing actual template attacks against ChaCha.

4.1 Simulated information

To simulate a side-channel attack, we need to extract the algorithm's intermediate values along with its input and output, which we can then convert into leakage values and probability distributions of potential values.

For extracting the intermediate values from ChaCha, the Julia package `CryptoSideChannel.jl` [37] was used, which introduces specialised logging types that emit their value on every write. These types allow a regular implementation of ChaCha in Julia to output intermediate values. It is essential to use only values that are likely to be present in an implementation of ChaCha. For example, if two shifts and an OR implement rotation, only the final value and input should be used because of rotate instructions in instruction sets, or for more complex ALUs which potentially combine together several operations.

I found the use of simulated information to be helpful when designing the factor graph because it allows fast experimentation with new structures with varying the amount of information given by the templates. However it is important to also work with real attack data because it is quite hard to judge the quality of templates which would be achievable and especially when combining multiple traces simulations can give a much more optimistic view than in reality due to assumptions about independent noise.

4.1.1 Noisy Hamming weights

The power drawn by a processor is often correlated to the Hamming weight of the values being processed. Therefore, the processor's word size significantly affects the values which leak out. When measuring the power supply current drawn by real processors, many noise sources affect the measurement, which means that the estimated Hamming weight is subject to noise, which we can model as Gaussian noise.

The simulated version breaks each 32-bit intermediate value into w bit words. Then, each of these has its Hamming weight calculated with one-dimensional Gaussian noise (with standard deviation σ) added to it to produce the noisy Hamming weight.

The noisy Hamming weights (generated through simulation or from a real trace) must be converted into a probability distribution for each cluster of c bits. Assuming that c divides w exactly, the clusters that make up a particular word will have the same distribution due to only having access to the Hamming weight of the word, and each value in a cluster with a particular Hamming weight will have the same likelihood.

The following formula gives the likelihood of getting a value $v \in [0, 2^c)$ in a cluster when the leakage value is l (to get the probability, it can be normalised once calculated for all values):

$$\Pr(x = v|l) = \sum_{n=0}^{w} \left(\Pr_{\text{Normal}(l,\sigma)}(n) \times \prod_{i=0}^{\text{HW}(v)} \frac{n-i}{w-i} \times \prod_{i=0}^{c-\text{HW}(v)} \frac{w-n-i}{w-\text{HW}(v)-i-1} \right)$$

The basic idea behind the calculation is to calculate the likelihood of the processor word having a Hamming weight n given the leakage l. Then, with that particular word Hamming weight, what is the probability of drawing the Hamming weight of v in c tries. Then, sum across all possible Hamming weights for the processor word, giving the overall likelihood that the cluster has a particular value.

It is also possible to have a classifier give probabilities for each potential Hamming weight the processor word can take. These probabilities can replace $\Pr_{\text{Normal}(l,\sigma)}(n)$ in the above equation.

4.1.2 Value information

The leakage can give more information about the particular value than its Hamming weight. This is possible because of manufacturing and design differences between different bits, so building a signal for each bit is possible. The Hamming weight will still be a significant component of the signal, although subsequent transformations may make this component hard to see.

Below, several methods are described for finding mean vectors and a pooled noise distribution (every value has the same distribution) to provide probabilities for each value.

Table 4.1: Average first-order success rates and logarithmic guessing entropies of templates taken from different sets in the Keccak paper [3]

Template set	**Actual performance**		**Simulated performance**	
	1-SR	**LGE**	**1-SR**	**LGE**
A	0.356	3.20	0.369	2.89
B	0.052	5.51	0.053	5.37
C	0.037	5.67	0.049	5.58
D	0.013	6.43	0.013	6.38

Other methods can also produce probability distributions of the values.

One method is to have a defined signal-to-noise ratio (SNR) where the mean vectors are sampled from the same distribution as the noise and multiplied by that SNR.

A second method is to have a vector associated with every bit of the word, which are then added together based on the set bits for the value to produce the mean vector for that value. This model allows the Hamming weight to significantly affect the emitted value compared to having a defined SNR.

A third method for selecting the mean vectors and noise distribution is to use a set of templates from a previous experiment, with different templates selected for different intermediate locations in the algorithm. This method allows the leakage to more closely resemble an actual attack. The main set of templates used come from [3], a fragment template attack on an implementation of Keccak, recorded using the same setup as in section 4.2.1. The average first-order success rates and logarithmic guessing entropies are shown in table 4.1; the randomly sampled versions achieve slightly better performance than with real traces in the original paper.

These leakage models provide a set of mean vectors and a pooled noise distribution. We can create a leakage for a particular intermediate value by adding the mean vector for that value to a random vector sampled from the noise. We can then apply the standard procedure for finding the likelihoods of each potential value by calculating the likelihoods of the mean vectors in a Gaussian distribution (with the pooled covariance matrix) centred on the leakage vector, which can then be normalised to produce probabilities.

We can insert leakages into the factor graph at most intermediate values. It is not possible to insert leakages directly on the output of the XORs because they will be highly related to the results of their rotations (and the variable might not be present in the factor graph). The outputs of ADD and ROR can have their templates selected from different sources, allowing for different qualities. We may want to do this because we expect that ROR will have higher quality leakage given their similarity to the outputs of XOR. The key (and nonce/counter/output) also allows different sets of templates to reflect their wider use.

4.2 Actual template attack

I then attacked several implementations of ChaCha on a 32-bit microcontroller. The steps required for performing the attack are detailed below.

4.2.1 Recording setup

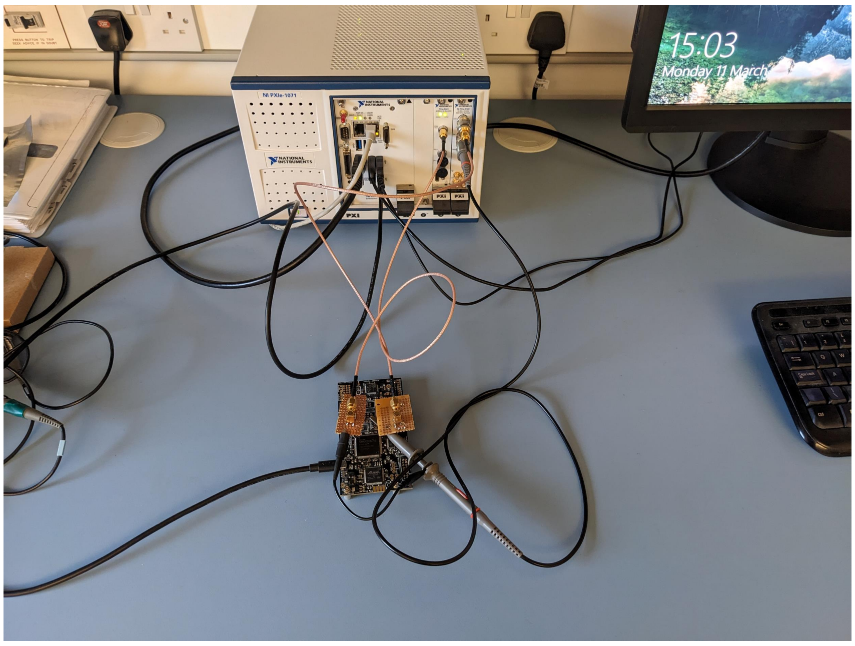


Figure 4.1: The recording setup used

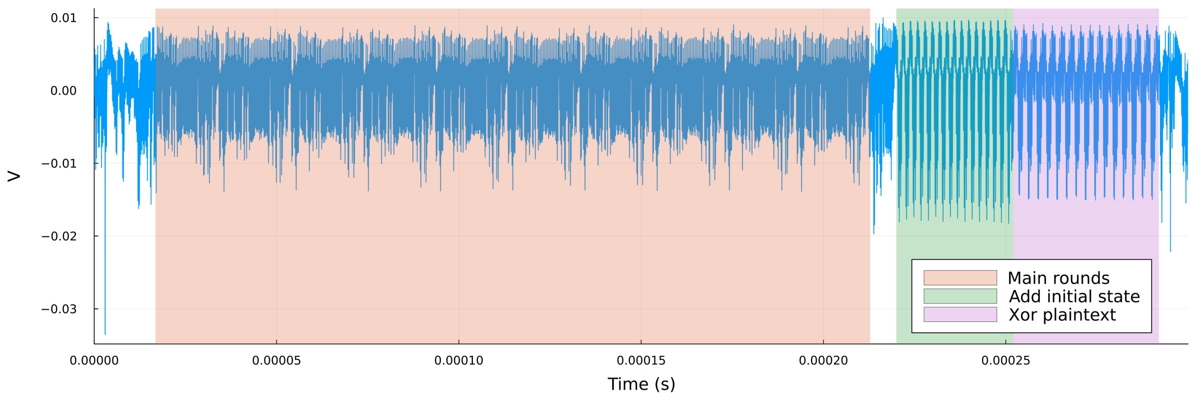
Figure 4.1 shows the recording setup used: a ChipWhisperer-Lite [38] board with an STM32F303 32-bit microcontroller using a Cortex-M4 as the target. The ChipWhisperer has an integrated clock and oscilloscope. I did not use these for recording traces due to their inability to acquire lots of samples per clock cycle and limitations with the number of samples they can record. An NI PXIe-5160 10-bit oscilloscope and an NI PXIe-5423 function generator (generating a square wave) were used, with a synchronised clock input so clock drift does not need to be considered. The oscilloscope is connected to a high-side resistor through a high pass filter with a time constant of 0.5 µs. The target raises an IO pin to high for the duration of the execution to trigger the recording.

All implementations of ChaCha that I attacked were compiled by `gcc` with the option `-Os` and run with a target frequency of 5 MHz. The three implementations were:

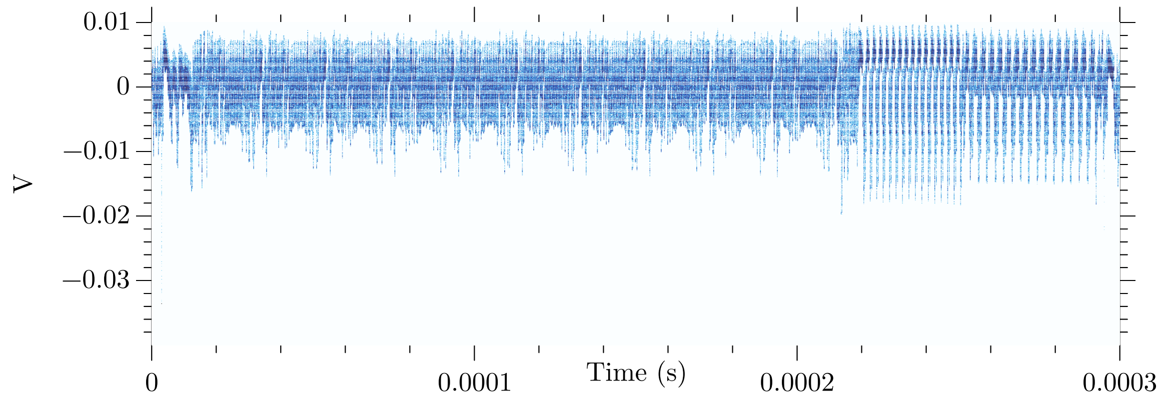
- Normal 32-bit implementation – Taken from [39], which keeps most of the state in registers during the execution. The oscilloscope had a sampling rate of 2.5 GHz.
- Custom 8-bit implementation – A custom ChaCha implementation that exclusively uses 8-bit values so that fewer bits were operated on simultaneously and more operations would need to spill to SRAM. The oscilloscope had a sampling rate of 250 MHz.

- Volatile 32-bit implementation – A modified 32-bit implementation so that the state is marked as volatile, meaning every operation required loading from and storing to SRAM. The oscilloscope had a sampling rate of 500 MHz.

4.2.2 Traces recorded and trace validation



(a) An example raw power trace from the normal 32-bit implementation with the different stages of the algorithm approximately shaded with different colours



(b) An example raw power trace from the normal 32-bit implementation, where it has been shaded based on time spent in each location (the clear horizontal lines are plotting artefacts)

Figure 4.2: Example power traces from the normal 32-bit implementation with different methods of shading

Table 4.2: Number of traces collected for different purposes for the different implementations (the attack number represents the number of keys used, each of which can have several traces recorded with it)

Purpose	**32-bit implementations**	**8-bit implementation**
Trace validation	2 000	2 000
Detection	16 000	8 000
Profiling (Training)	64 000	48 000
Validation	1 000	1 000
Attack	1 000	1 000

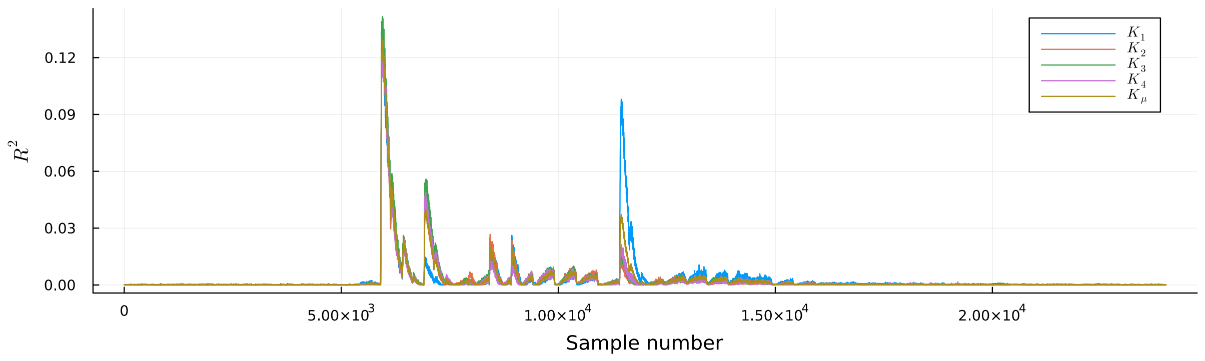
Figure 4.2 shows an example power trace from the normal 32-bit implementation, with the different stages of the execution highlighted. Table 4.2 shows the number of traces recorded and their purpose.

We must validate that all recorded traces were of correct executions and that other sources of error do not make them unusable. This was done by creating a mean trace out of the validation traces against which all traces can then have their correlation calculated and traces below a certain threshold (e.g. 0.98) are highlighted for potential removal from the set of usable traces. This highlighted several traces for potential removal. Upon inspection, it was clear that they were correct executions but had either been triggered slightly early or late. Their correlations can be improved by shifting them to better align with each other. After aligning the traces, they all had correlations above the threshold, so were not removed from the set of usable traces.

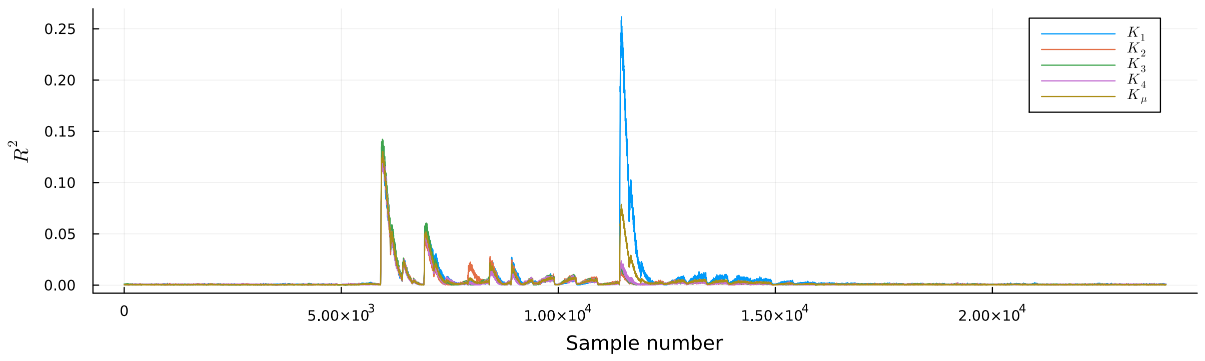
4.2.3 Clock signal detection

The high number of samples in each trace makes it infeasible to build templates directly from the entire trace for each location in the execution. Most samples in the trace will contain little or no information about a particular intermediate value due to it not being operated on. It is helpful to consider only points which contain significant information about a particular value. Detecting interesting points on a per-sample basis across all traces is impractical due to the large number of samples and intermediate values. Specific clock cycles will likely contain all the interesting points because their instruction modifies the value. Hence, finding the clock edge allows the samples in each cycle to be combined to calculate the interesting clock cycles.

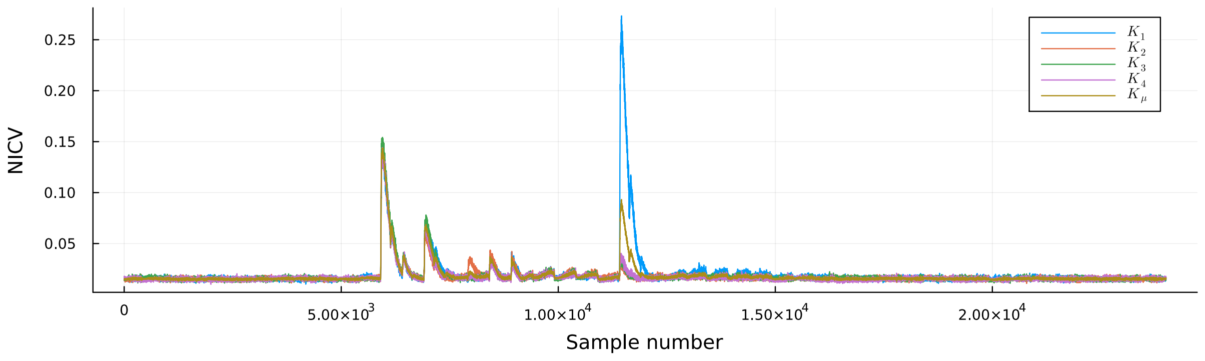
Several different statistics have been proposed for detecting interesting clock cycles, such as correlation with a linear model [3], SNR [40], t-test [41], and normalised inter-class variance (NICV) [42].



(a) Correlation of samples with the Hamming weight



(b) Correlation of samples with a bitwise linear model



(c) NICV of samples

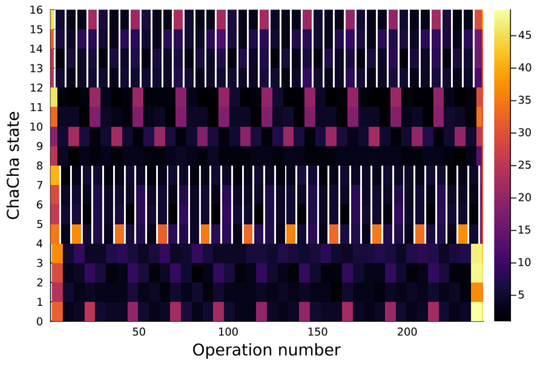
Figure 4.3: Statistics reported for the first word of the key, where K_1 refers to the first byte and K_μ the average over each byte in the normal 32-bit implementation

Figure 4.3 shows several ways of detecting where the clock cycles fall in terms of samples. It shows that the different metrics highlight the same samples as being interesting. It is interesting to observe that the Hamming weight model leads to quite different results than the others, showing that there is more information in the samples than just the Hamming weights. The bitwise linear model and NICV show very similar trends.

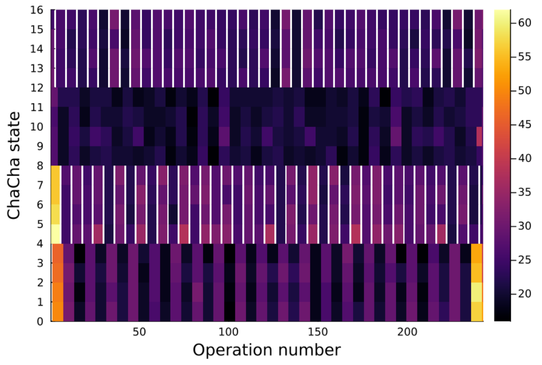
4.2.4 Interesting cycle detection

I used the mean of the samples in each clock cycle to detect the interesting clock cycles for each intermediate value because it provides enough information about whether a particular value is operated on. Each intermediate value was broken into bytes for calculating if a clock cycle was interesting. For the 32-bit implementations, the metrics were combined across the bytes because they only operate on 32-bit values, whereas for the 8-bit implementation, the bytes were kept separate.

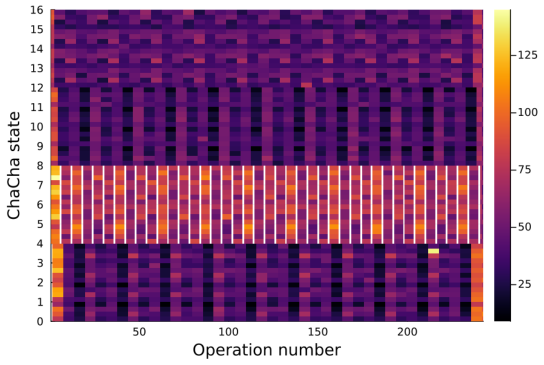
The metric I selected for determining interesting clock cycles was correlation with a bitwise linear model (each bit is a separate variable in the model). This metric was selected because it is simple to combine the different bytes of an intermediate value, and it provides a clean result for whether a cycle is of interest compared to other metrics, such as NICV. A clock cycle is interesting if its summed R^2 is greater than 0.04, as was done in [3].



(a) Number of interesting cycles of every intermediate value in the normal 32-bit implementation



(b) Number of interesting cycles of every intermediate value in the volatile 32-bit implementation



(c) Number of interesting cycles of every intermediate value in the 8-bit implementation

Figure 4.4: Number of interesting cycles detected for different intermediate values, the white pixels are where a template is not created (the outputs of XOR before the complete rotation)

Figure 4.4 shows the number of interesting clock cycles for each intermediate value in the different implementations. As expected, the input has the most interesting clock cycles because it is not only used inside a regular round but also in the final addition and is copied at the start of the execution. The 8-bit implementation has many more interesting clock cycles due to the need for more SRAM accesses and operations potentially taking more than a single cycle.

4.2.5 Template profiling

A vector of the values being targeted (either the fragment of the word or the whole word) and the cycle level bitmask of interesting cycles for the corresponding intermediate value are required to create a template for an intermediate value. The cycle level bitmask can then be dilated around the interesting clock cycles to ensure we have captured all helpful information for a particular value. Dilating in front of the interesting samples helps give more characteristics about the noise, such as the previous input power to the high pass filter, which allows for LDA to more successfully deal with the noise.

I created the templates using downsampled versions of the traces. For the normal 32-bit implementation I used 10 samples per clock cycle, for the 8-bit implementation 25 samples per clock cycle, and for the volatile 32-bit implementation 20 samples per clock cycle. I found that changing the number of clock cycles did not significantly change the templates' performance.

To create the templates, I used a similar procedure to what is described in [3]. I fitted a bitwise linear model for every sample independently. These models can then create expected mean vectors for every potential value. I then calculated the intra-class and inter-class covariance matrices from the original samples and these mean vectors. The inter-class covariance matrix ($\boldsymbol{B}$) is calculated by:

$$\boldsymbol{B} = \frac{1}{\sum_b n_b} \sum_b n_b (\bar{\boldsymbol{x}}_b - \bar{\boldsymbol{x}})(\bar{\boldsymbol{x}}_b - \bar{\boldsymbol{x}})^\top$$

with the total intra-class covariance matrix ($\boldsymbol{W}$) calculated by:

$$\boldsymbol{W} = \frac{1}{\sum_b n_b} \sum_b \sum_{t=1}^{n_b} (\boldsymbol{x}_{b,t} - \bar{\boldsymbol{x}}_b)(\boldsymbol{x}_{b,t} - \bar{\boldsymbol{x}}_b)^\top$$

where n_b is the number of traces with value b, $\bar{\boldsymbol{x}}_b$ is the mean vector for value b, $\bar{\boldsymbol{x}}$ is the overall mean vector and $\boldsymbol{x}_{b,t}$ is the t-th vector with value b.

These matrices are then input into LDA, which projects the data into the space which maximises the SNR. The first d dimensions are selected (when ordered by eigenvalue size), where d is the number of bits in the linear model. Previous works [43] have found that the number of bits corresponds well to the number of significant eigenvalues returned.

Many parameters can be changed when designing templates, including the amount of dilation before and after, the number of samples per clock cycle, the use of an NICV weighted mean for downsampling, and the number of samples per cycle between dilated regions and interesting regions. None of them had a large effect on the quality of the templates. For example, table 4.3 shows how the average first-order success rate of templates changes as the number of cycles dilated in front of interesting cycles varies. I used four cycles of dilation in front and two cycles behind, with all selected cycles having the same number of samples, which have been downsampled by an unweighted mean.

Table 4.3: Average first-order success rate for 8-bit fragment templates on the normal 32-bit implementation with differing amounts of dilation in front

Dilation amount (clock cycles)	**1-SR**
0	0.0096
2	0.0100
4	0.0100
6	0.0100
8	0.0100

I considered breaking down the leakages into separate leakage events (the different regions that exceeded the threshold) and then taking the product of their distributions rather than just putting all the values into a single LDA. This was found to lead to significantly worse performance.

Chapter 5

Evaluation

This chapter describes the results of simulated and actual attacks against ChaCha as well as the quality of templates in the different cases.

Unless stated otherwise for attacks, I assumed that the encryption's counter, nonce and output were known rather than just having leakage information. In all cases, I assumed that only the key section of the input must be enumerated.

When the number of keys required for enumeration was less than 2^{20}, the number of key candidates was calculated by actually performing key enumeration. For larger values, the number of key candidates was estimated by the rank estimation algorithm described in the section 2.5. A powerful attacker can likely enumerate between 2^{50} and 2^{70} key candidates, going forward I use 2^{64} key candidates when calculating success rates for an attacker.

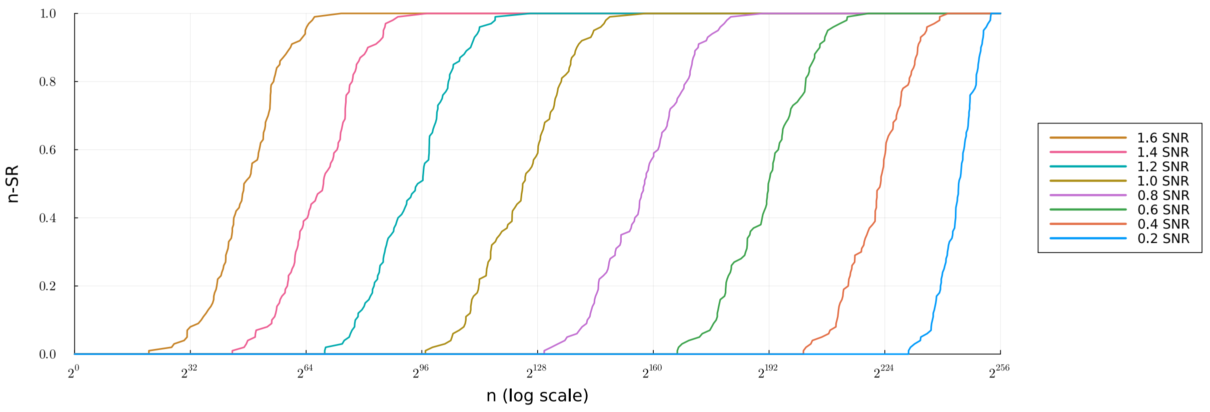
5.1 Simulation results

The simulated results of the attack help show the SNR required to reach certain performance levels in different scenarios. They are not necessarily representative of the performance of actual attacks due to the varying quality of templates and potential correlation between the different leakages. For this reason, I only considered a single trace because it is likely that the noise would not be independent between multiple traces, so the simulation would give an unrepresentatively high chance of success. The templates evaluated are all for byte values with eight dimensions (which have equal and uncorrelated variance), using the first method described in section 4.1.2. The simulations also validate that the factor graph correctly represents ChaCha.

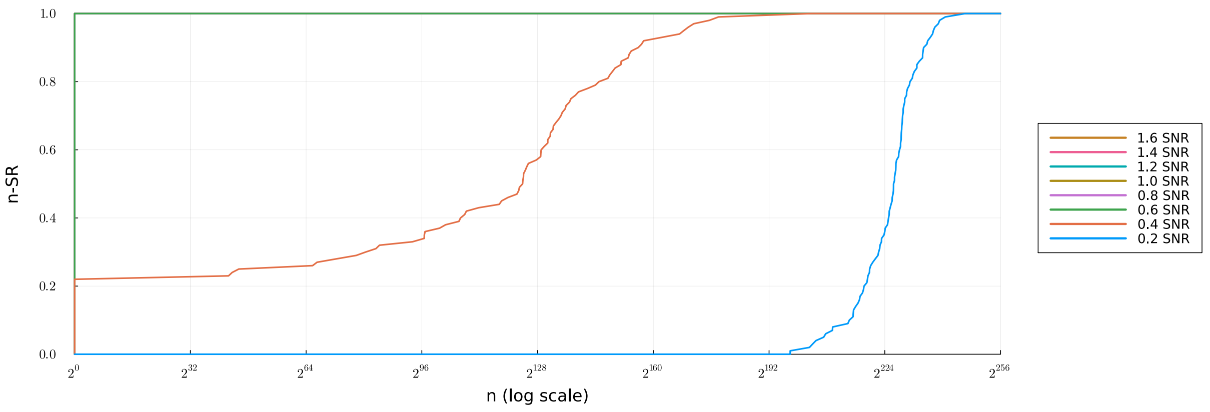
Table 5.1 shows the average first-order success rates and logarithmic guessing entropies of templates with differing SNRs. As expected, the lower the SNR, the worse the quality of the templates.

Table 5.1: Average first-order success rates and logarithmic guessing entropies of random templates generated with different success rates

SNR	1-SR	LGE
0.2	0.016	6.48
0.4	0.045	5.82
0.6	0.103	5.05
0.8	0.189	4.23
1.0	0.300	3.39
1.2	0.423	2.59
1.4	0.542	1.87
1.6	0.653	1.28



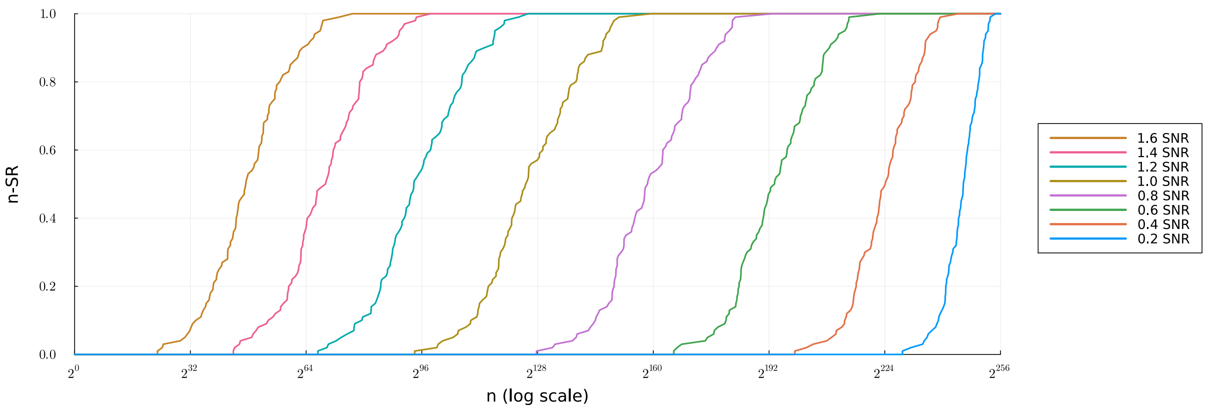
(a) The n-SR of the key with varying the signal-to-noise ratio with simulated value templates without performing belief propagation with a known counter, nonce and output



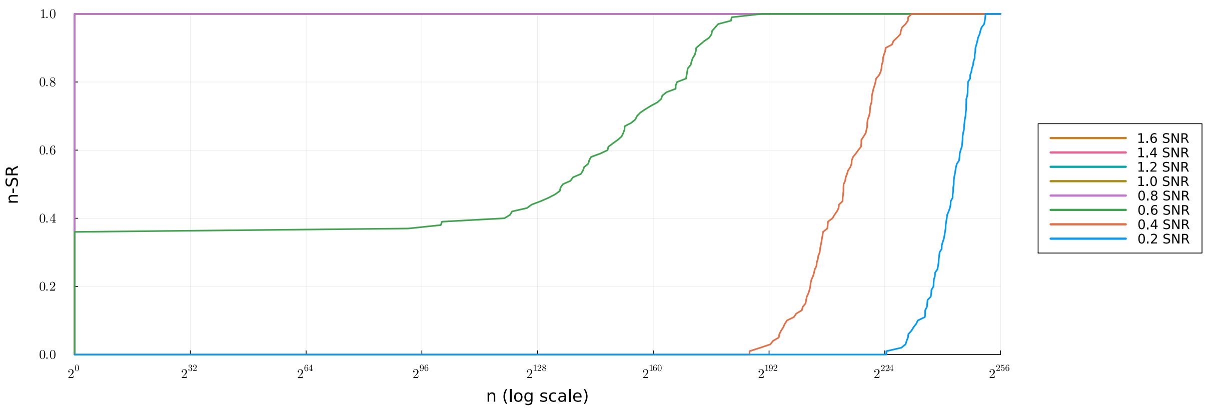
(b) The n-SR of the key with varying the signal-to-noise ratio with simulated value templates after performing belief propagation with a known counter, nonce and output. No key enumeration was required for SNRs greater than or equal to 0.6

Figure 5.1: The amount of key enumeration required for differing SNRs to find the correct key with a known counter, nonce and output

Figure 5.1 shows the proportion of keys successfully found after differing amounts of key enumeration for different SNRs in the standard attack scenario. Belief propagation significantly reduces the number of key candidates that must be enumerated to find the correct result. Without enumeration, the correct solution is always found when the SNR is greater than or equal to 0.6.



(a) The n-SR of the key with varying the signal-to-noise ratio with simulated value templates without performing belief propagation with only leakages for the counter, nonce and output



(b) The n-SR of the key with varying the signal-to-noise ratio with simulated value templates after performing belief propagation with only leakages for the counter, nonce and output. No key enumeration was required for SNRs greater than or equal to 0.8

Figure 5.2: The amount of key enumeration required for differing SNRs to find the correct key with only leakages for the counter, nonce and output

Figure 5.2 shows the same information as figure 5.1, but with leakages for the output, nonce and counter. The different scenario does not affect the results with belief propagation compared to known information (because it captures the same information as before). The post belief propagation results are worse at lower SNRs than in the standard scenario. The correct solution is always found without enumeration, when the SNR is greater than or equal to 0.8.

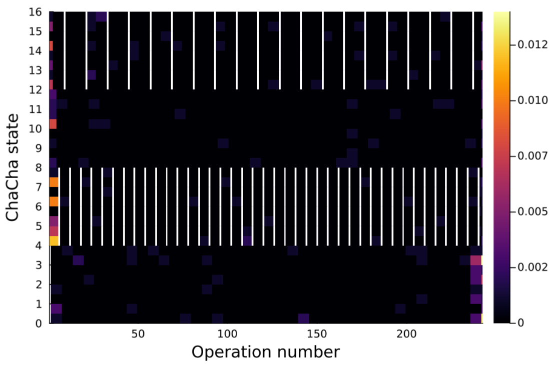
5.2 Actual attack results

5.2.1 Template quality and combining them without belief propagation

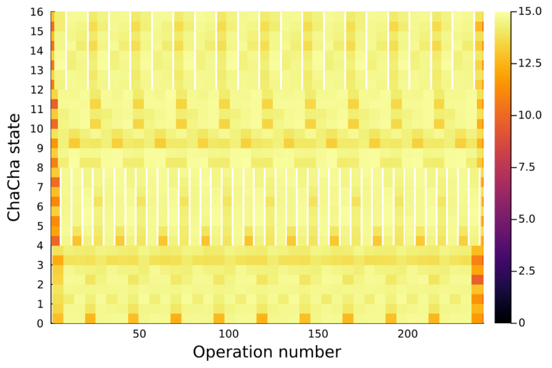
Table 5.2 shows the average first-order success rates and logarithmic guessing entropies for several cases with figure 5.3 showing their distribution across the algorithm. The 8-bit implementation has significantly higher success rates and lower guessing entropies. Interestingly, the lowest bits of the 32-bit words are where the lowest logarithmic guessing entropy and highest first-order success rates occur. The volatile implementation has much more consistent quality across the algorithm due to the SRAM activity providing much of the signal. That activity is very consistent between intermediate values.

Table 5.2: Average first-order success rates and logarithmic guessing entropies of different types of (fragment) templates before SASCA. Details of the implementations are described in section 4.2.1. The maximum LGE is one less than the fragment size.

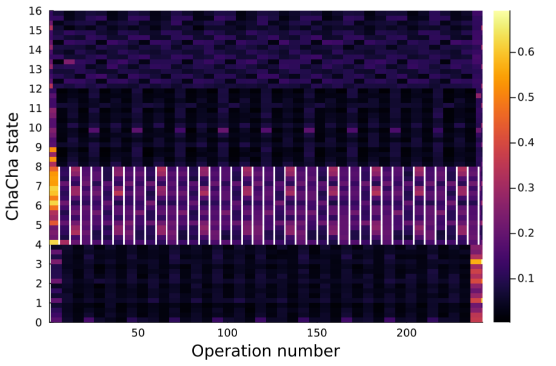
Template type		**1-SR**	**LGE**
Fragment size (bits)	**Implementation**		
8	Normal 32-bit	0.0119	6.62
16	Normal 32-bit	0.0003	14.39
8	8-bit	0.0914	4.95
8	Volatile 32-bit	0.0189	6.15



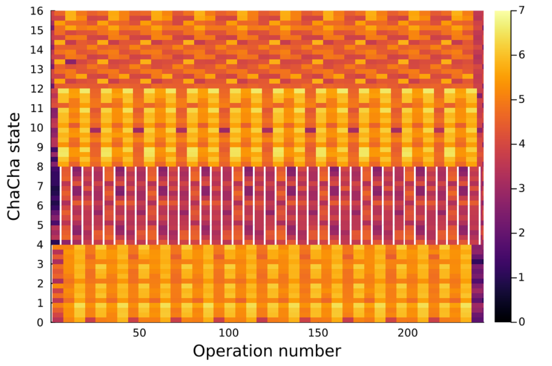
(a) First-order success rates of 16-bit fragment templates in normal 32-bit implementation



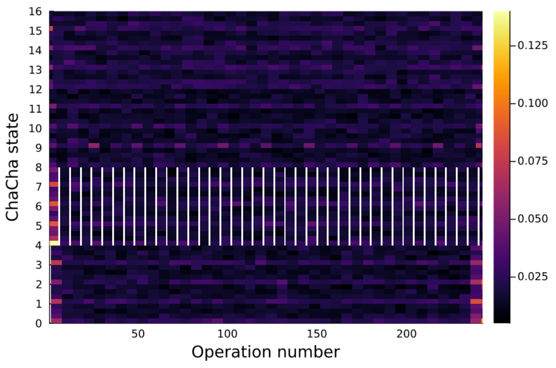
(b) Logarithmic guessing entropies of 16-bit fragment templates in normal 32-bit implementation



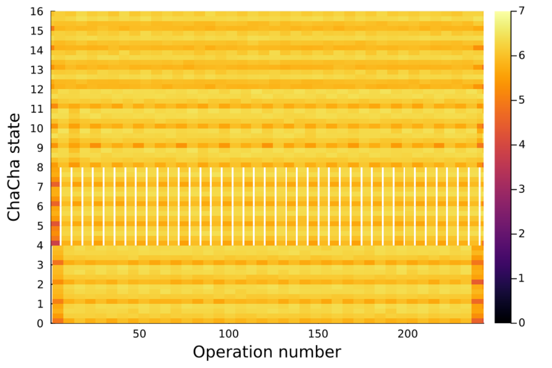
(c) First-order success rates of byte templates on 8-bit implementation



(d) Logarithmic guessing entropies of byte templates on 8-bit implementation



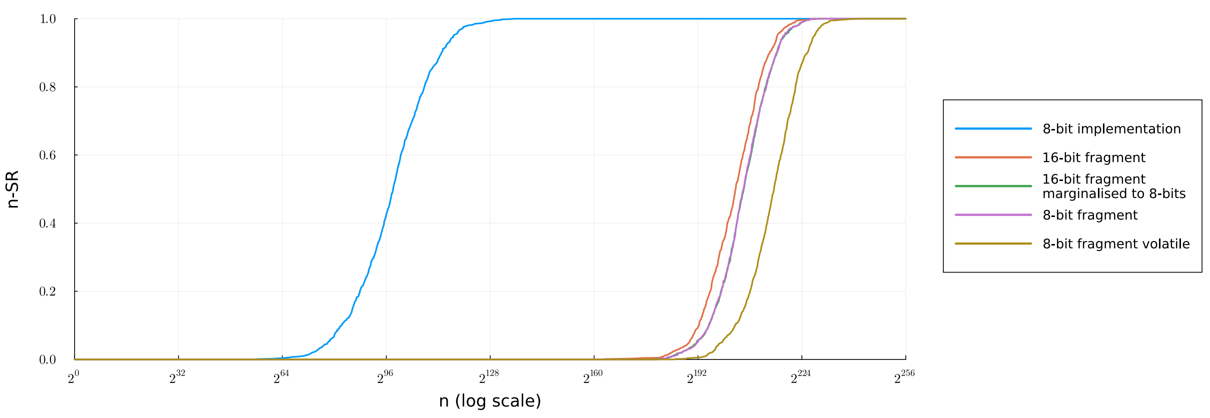
(e) First-order success rates of 8-bit fragment templates in volatile 32-bit implementation



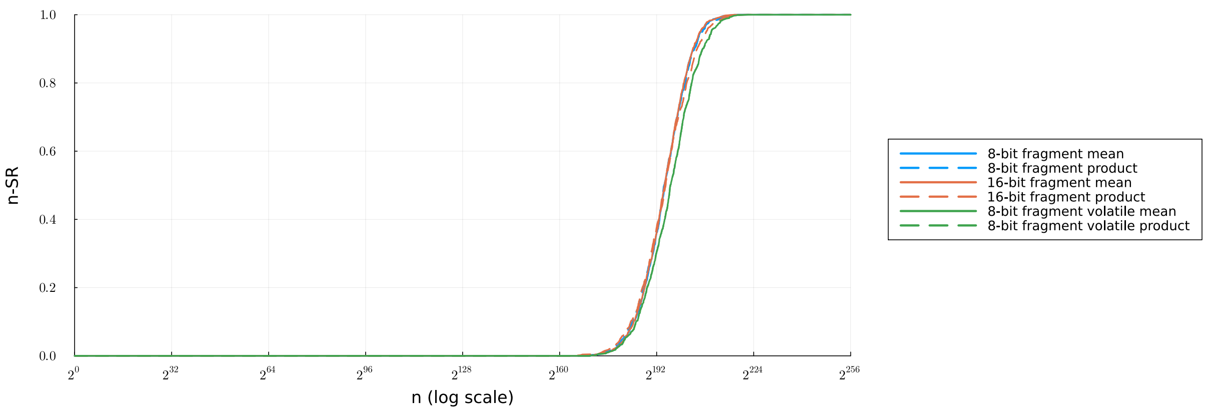
(f) Logarithmic guessing entropies of 8-bit fragment templates in volatile 32-bit implementation

Figure 5.3: First-order success rates and logarithmic guessing entropies of templates across the algorithm before SASCA

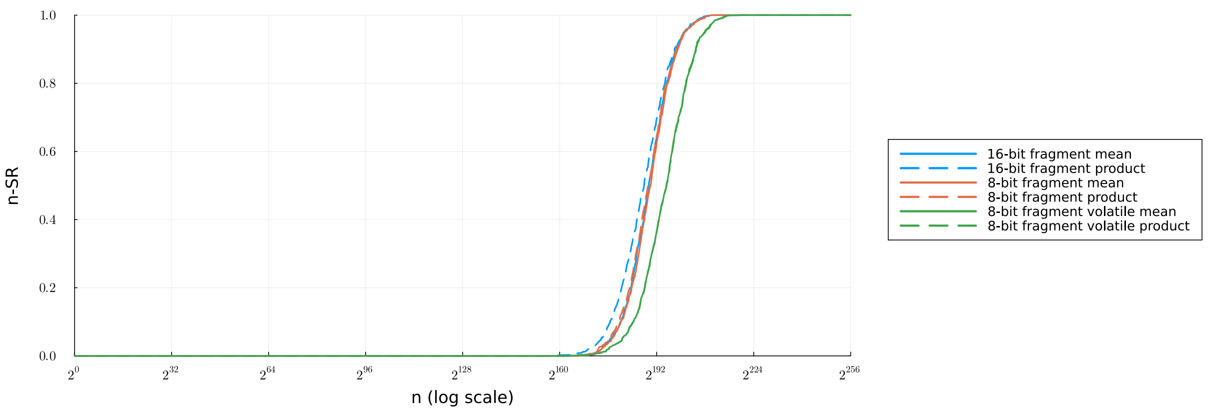
Figure 5.4 shows the proportion of keys correctly obtained after differing amounts of key enumeration in different scenarios. It clearly shows that for a single trace without belief propagation, only the 8-bit implementation can succeed with just key enumeration, and that 8-bit and 16-bit fragments perform similarly once marginalised to 8-bits. It also shows that simply combining more traces is not a very effective method for reducing the amount of enumeration required.



(a) Amount of key enumeration required for single traces with different types of template



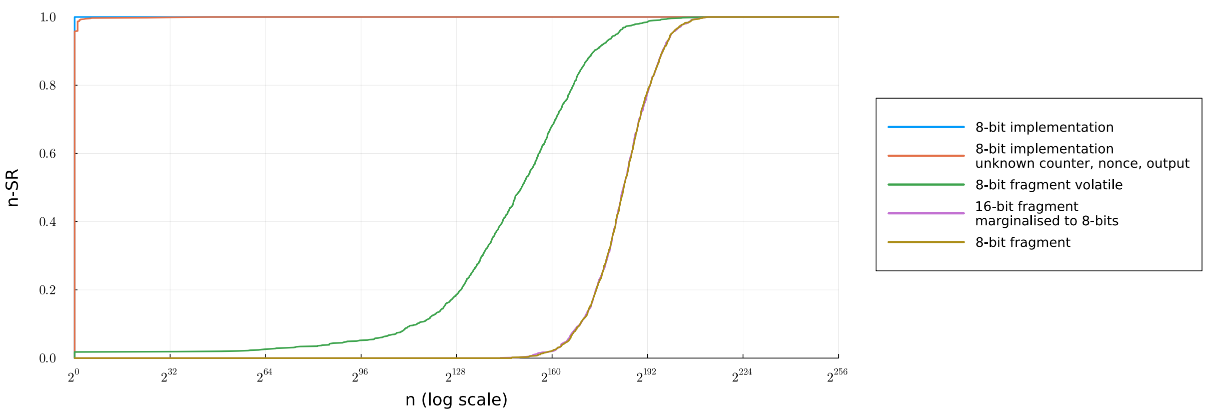
(b) Amount of key enumeration required when combining ten traces with a constant counter



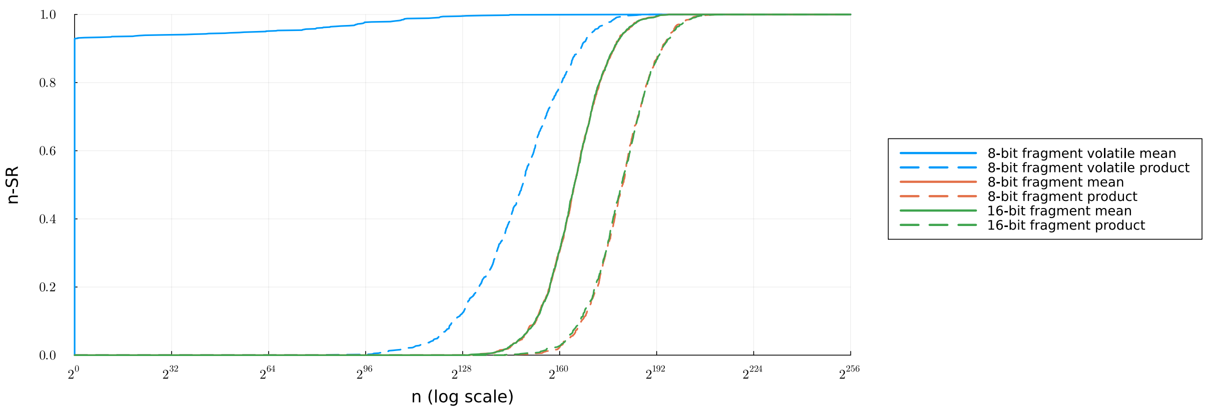
(c) Amount of key enumeration required when combining ten traces with an incremented counter

Figure 5.4: Amount of key enumeration required for different scenarios without performing belief propagation

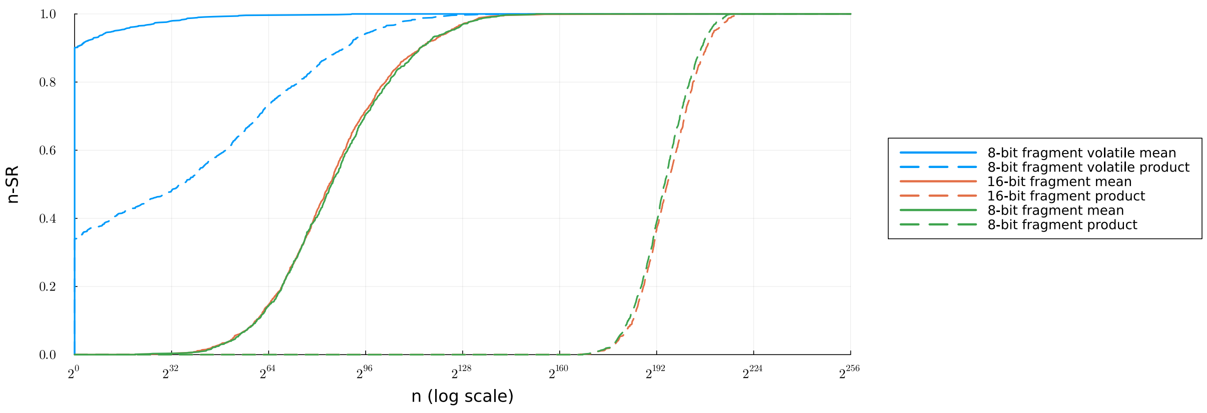
5.2.2 Results after belief propagation



(a) Amount of key enumeration required for single traces with different types of template



(b) Amount of key enumeration required when combining ten traces with a constant counter



(c) Amount of key enumeration required when combining ten traces with an incremented counter

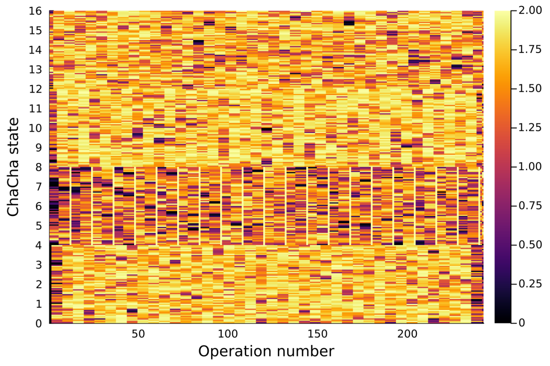
Figure 5.5: Amount of key enumeration required for different scenarios after performing belief propagation

Figure 5.5 shows the results of the running belief propagation with different types of template and ways of combining several traces. It shows that the fragment templates for the normal 32-bit implementation do not provide enough information from a single trace to make the results enumerable. The 8-bit implementation managed to achieve a 100% success rate without key enumeration. The performance for the 8-bit implementation is worse when only having leakage information for the output, nonce and counter. However, in all cases for the 8-bit implementation, the correct key is within an enumeration range of 2^{64}. The volatile 32-bit implementation achieves a first-order success rate of 1.8% and that 2.6% of keys are within an enumeration range of 2^{64} on a single trace. The increased success rate shows that more SRAM activity significantly improves the chance of attacks succeeding and means that SRAM activity should be avoided as much as possible in the inner parts of cryptographic algorithms. However, this additional leakage can only be exploited when combined across the algorithm.

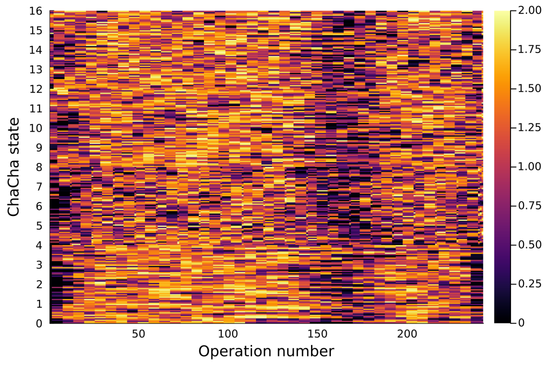
When combining several traces run with identical parameters in the normal 32-bit implementation, the results are not significantly improved compared to a single trace. In contrast, for the volatile implementation, 95.1% of keys can be recovered with fewer than 2^{64} key candidates enumerated. This difference in behaviour shows how the noise affecting the two implementations is distinct and can be removed more effectively with additional traces for the volatile version. When combining traces with an incremented counter, the normal 32-bit implementation becomes more vulnerable, with 14.6% of keys recoverable within 2^{64} candidate keys. In the volatile 32-bit implementation, 90.1% of keys require no enumeration, and 99.6% of keys are in the first 2^{64} candidate keys.

Calculating the mean of leakages for shared values before calculating their likelihoods in all cases provided better results (after belief propagation) than taking the product of their individual likelihoods. We would not expect this with independent samples, showing how the noise affecting the traces is not independent. The lack of independence is probably caused by factors such as deterministic pipeline effects and the unmodelled bits in the word. The lack of independence means other techniques for combining several traces may be helpful with fragment template attacks (or complete algorithm executions) compared to those discussed in [30] for regular template attacks, which recommends using a product.

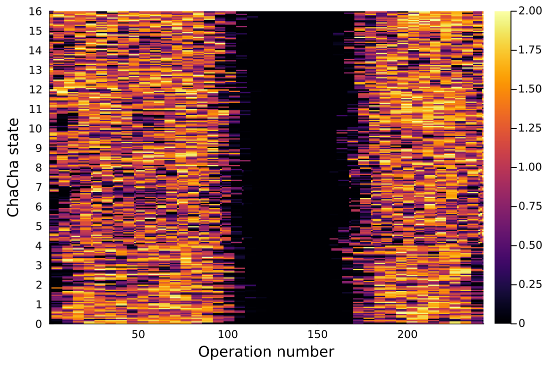
Figure 5.6 shows the entropies of variables in the factor graph after different number of iterations of belief propagation in the case that only leakages are provided from the 8-bit implementation. It shows that SASCA is able to successfully calculate part of the intermediate state despite not having access to any known values at that point, and then this known information can flow backwards through the graph to find the values for the input state. This shows that it is able to make use of leakage information provided throughout the entire trace compared to just on the key directly. It is also interesting to see how initially the entropy drops rapidly due to enforcing local constraints before taking a longer time to find a solution in one part of the factor graph.



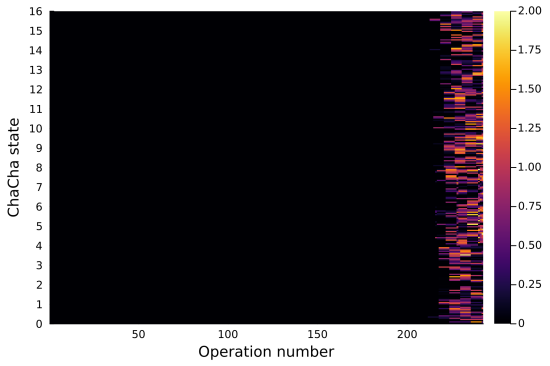
(a) Entropies of variables after 0 iterations of belief propagation



(b) Entropies of variables after 60 iterations of belief propagation



(c) Entropies of variables after 130 iterations of belief propagation



(d) Entropies of variables after 180 iterations of belief propagation

Figure 5.6: Entropies of variables for an attack against the 8-bit implementation, using 2-bit clusters, with only access to leakages for all values. It is highly recommended to see the corresponding animation at `https://tbeakl.github.io/Part-III-Project-Visualisations/`

Chapter 6

Summary and conclusions

This project has accomplished its initial aim of creating a factor graph for ChaCha, which allows for effective belief propagation with simulated leakages. The proposed tree structure for ADD led to a better chance of convergence than the loopy structure. A potentially valuable piece of future work would be to design a compiler for automatically converting an implementation of an algorithm into a factor graph.

I then performed template attacks against several implementations of ChaCha on an ARM Cortex-M4. The different implementations (described in section 4.2.1) led to very different levels of performance, with the following proving to be most successful scenarios:

- 8-bit implementation with known or unknown counter and nonce from a single trace – it was possible to achieve a 100% success rate with minimal amounts of key enumeration.
- Volatile 32-bit implementation with a known counter and nonce from 10 traces – it was possible to recover the key over 95% of the time when enumerating the first 2^{64} most likely key candidates.

The following scenarios are more borderline in being successful:

- Volatile 32-bit implementation with a known counter and nonce from a single trace – it was possible to recover the key 2.6% of the time within the first 2^{64} key candidates.
- Normal 32-bit implementation with a known counter and nonce from 10 traces with an incremented counter – it was possible to recover the key 14.6% of the time within the first 2^{64} key candidates.

The following scenarios were not able to successfully recover the key within a reasonable amount of key enumeration:

- Normal 32-bit implementation with a known counter and nonce from a single trace.
- Normal 32-bit implementation with a known counter and nonce from multiple traces

with a constant counter.

The different success rates achieved show that the number of SRAM accesses should be minimised when implementing and designing cryptographic functions similar to ChaCha, due to their much higher leakage.

A natural piece of future work (for someone with more computational capacity than me) would be to make 32-bit templates as was shown in [31] to see how effective they are at attacking a 32-bit algorithm on a 32-bit machine.

It would also be helpful to look at more methods for combining several traces due to correlated noise between runs. This could involve creating several noise distributions, for example, one for deterministic sources of noise, which is not reduced by more traces, and a second for noise, which is reduced by additional traces.

A repository containing the code used for both the implementations of ChaCha and performing the attacks can be found at `https://github.com/Tbeakl/PartIIIProject`.

Bibliography

[1] D. Bernstein, "ChaCha, a variant of Salsa20," 01 2008. `https://cr.yp.to/chacha/chacha-20080128.pdf`.

[2] 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.), (Berlin, Heidelberg), pp. 13–28, Springer Berlin Heidelberg, 2003.

[3] S.-C. You and M. G. Kuhn, "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.), (Cham), pp. 3–23, Springer International Publishing, 2022.

[4] S.-C. You, M. G. Kuhn, S. Sarkar, and F. Hao, "Low trace-count template attacks on 32-bit implementations of ASCON AEAD," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2023, Issue 4, pp. 344–366, 2023.

[5] N. Veyrat-Charvillon, B. Gérard, and F.-X. Standaert, "Soft analytical side-channel attacks," in *Advances in Cryptology – ASIACRYPT 2014* (P. Sarkar and T. Iwata, eds.), (Berlin, Heidelberg), pp. 282–296, Springer Berlin Heidelberg, 2014.

[6] Y. Nir and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols." RFC 8439, June 2018.

[7] T. Arcieri, "ChaCha cipher quarter round function," 2020. `https://commons.wikimedia.org/wiki/File:ChaCha_Cipher_Quarter_Round_Function.svg` File:ChaCha Cipher Quarter Round Function.svg.

[8] D. J. Bernstein, "The Poly1305-AES message-authentication code," in *Fast Software Encryption* (H. Gilbert and H. Handschuh, eds.), (Berlin, Heidelberg), pp. 32–49, Springer Berlin Heidelberg, 2005.

[9] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3." RFC 8446, Aug. 2018.

[10] M. Damien, "/openbsd/usr.bin/ssh/protocol.chacha20poly1305," Feb 2020. `http://bxr.su/OpenBSD/usr.bin/ssh/PROTOCOL.chacha20poly1305`.

[11] markm, “Revision 317015,” Apr 2017. `https://svnweb.freebsd.org/base?view=revision&revision=r317015`.

[12] “/openbsd/lib/libc/crypt/arc4random.c,” Jul 2022. `http://bxr.su/OpenBSD/lib/libc/crypt/arc4random.c`.

[13] “/netbsd/lib/libc/gen/arc4random.c,” May 2024. `http://bxr.su/NetBSD/lib/libc/gen/arc4random.c`.

[14] T. Ts’o, “[git pull] /dev/random driver changes for 4.8.” `https://lkml.iu.edu/hypermail/linux/kernel/1607.3/00275.html`.

[15] J.-P. Aumasson, “Too much crypto.” Cryptology ePrint Archive, Paper 2019/1492, 2019. `https://eprint.iacr.org/2019/1492`.

[16] R. Cox and F. Valsorda, “Secure randomness in Go 1.22,” May 2024. `https://go.dev/blog/chacha8rand`.

[17] “Rust Docs rand::rngs::StdRng.” `https://docs.rs/rand/latest/rand/rngs/struct.StdRng.html`.

[18] J.-P. Aumasson, W. Meier, R. Phan, and L. Henzen, *The Hash Function BLAKE*. Springer Berlin, 2016.

[19] NIST, “SHA-3 project,” 2012. `https://csrc.nist.gov/projects/hash-functions/sha-3-project`.

[20] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre attacks: Exploiting speculative execution,” in *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 1–19, 2019.

[21] P. Kocher, J. Jaffe, and B. Jun, “Differential power analysis,” in *Advances in Cryptology — CRYPTO’ 99* (M. Wiener, ed.), (Berlin, Heidelberg), pp. 388–397, Springer Berlin Heidelberg, 1999.

[22] 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.), (Berlin, Heidelberg), pp. 16–29, Springer Berlin Heidelberg, 2004.

[23] K. Pearson, “LIII. on lines and planes of closest fit to systems of points in space,” *Philosophical Magazine Series 1*, vol. 2, pp. 559–572, 1901.

[24] R. A. Fisher, “The statistical utilization of multiple measurements,” *Annals of Eugenics*, vol. 8, no. 4, pp. 376–386, 1938.

[25] F.-X. Standaert, T. G. Malkin, and M. Yung, “A unified framework for the analysis of side-channel key recovery attacks,” in *Advances in Cryptology - EUROCRYPT*

2009 (A. Joux, ed.), (Berlin, Heidelberg), pp. 443–461, Springer Berlin Heidelberg, 2009.

[26] D. MacKay, *Information Theory, Inference and Learning Algorithms.* Cambridge University Press, 2003.

[27] D. Koller and N. Friedman, *Probabilistic Graphical Models: Principles and Techniques.* Adaptive Computation and Machine Learning series, MIT Press, 2009.

[28] N. Veyrat-Charvillon, B. Gérard, M. Renauld, and F.-X. Standaert, "An optimal key enumeration algorithm and its application to side-channel attacks." Cryptology ePrint Archive, Paper 2011/610, 2011. `https://eprint.iacr.org/2011/610`.

[29] C. Glowacz, V. Grosso, R. Poussier, J. Schueth, and F.-X. Standaert, "Simpler and more efficient rank estimation for side-channel security assessment." Cryptology ePrint Archive, Paper 2014/920, 2014. `https://eprint.iacr.org/2014/920`.

[30] O. Choudary and M. G. Kuhn, "Efficient template attacks," in *Smart Card Research and Advanced Applications* (A. Francillon and P. Rohatgi, eds.), (Cham), pp. 253–270, Springer International Publishing, 2014.

[31] G. Cassiers, H. Devillez, F.-X. Standaert, and B. Udvarhelyi, "Efficient regression-based linear discriminant analysis for side-channel security evaluations: Towards analytical attacks against 32-bit implementations," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2023, p. 270–293, Jun. 2023.

[32] 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`.

[33] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer, "Ascon v1.2: Lightweight authenticated encryption and hashing," *J. Cryptol.*, vol. 34, no. 3, p. 33, 2021.

[34] NIST, "Lightweight cryptography," 2023. `https://csrc.nist.gov/projects/lightweight-cryptography`.

[35] 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.), (Berlin, Heidelberg), pp. 398–412, Springer Berlin Heidelberg, 1999.

[36] 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.), (Cham), pp. 513–533, Springer International Publishing, 2017.

[37] S. Schwarz, "CryptoSideChannel.jl: A customizable side-channel modelling and analysis framework in Julia," Jul 2021. `https://parablack.github.io/CryptoSideChannel.jl/dev/`.

[38] C. O'Flynn and Z. D. Chen, "ChipWhisperer: An open-source platform for hardware embedded security research," in *Constructive Side-Channel Analysis and Secure Design* (E. Prouff, ed.), (Cham), pp. 243–260, Springer International Publishing, 2014.

[39] R. Weatherley, "Lightweight cryptography primitives documentation," Apr 2021. `https://rweather.github.io/lightweight-crypto/index.html`.

[40] S. Mangard, "Hardware countermeasures against DPA – a statistical analysis of their effectiveness," in *Topics in Cryptology – CT-RSA 2004* (T. Okamoto, ed.), (Berlin, Heidelberg), pp. 222–235, Springer Berlin Heidelberg, 2004.

[41] G. Goodwill, B. Jun, J. Jaffe, and P. Rohatgi, "A testing methodology for side channel resistance validation," 2011. `https://api.semanticscholar.org/CorpusID:16852899`.

[42] 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*, pp. 310–313, 2014.

[43] S.-C. You, *Single-trace template attacks on permutation-based cryptography.* PhD thesis, Apollo – University of Cambridge Repository, 2022.