KECCAK sponge function family main document

Guido BERTONI[1]
Joan DAEMEN[1]
Michaël PEETERS[2]
Gilles VAN ASSCHE[1]

http://keccak.noekeon.org/

Version **1.2**
April 23, 2009

[1]STMicroelectronics
[2]NXP Semiconductors

Contents

Chapter 1

Introduction

Keccak [8] is a family of cryptographic hash functions [68] or, more accurately, sponge functions [7]. This document describes the properties of the Keccak family and presents its members as candidates to NIST's request for a new cryptographic hash algorithm family called SHA-3 [51].

This introduction offers in Section 1.1 a summary of the Keccak specifications using pseudocode, sufficient to understand its structure and building blocks. In no way should this introductory text be considered as a formal and reference description of Keccak. For the formal definition of the Keccak family, we refer to the separate document [8], to which we assume the reader has access. While the Keccak definition is fixed, this present document is likely to evolve over time, so we suggest the reader to obtain the latest version from our website `http://keccak.noekeon.org/`.

The document is organized as follows. The design choices behind the Keccak sponge functions are summarized in Chapter 2. Chapters 3–5 provide a security analysis and a rationale for our design choices. Each of these three chapters looks at a particular level, from top to bottom.

- Chapter 3 looks at the use of the sponge construction in our submission.
- Chapter 4 gives more insight on the use of an iterated permutation in the sponge construction.
- Chapter 5 looks more particularly at Keccak-f, the chosen permutation.

Examples of modes of use, as well as other details regarding the use of the Keccak sponge functions, are provided in Chapter 6. Finally, Chapter 7 takes a look at the software and hardware implementation aspects.

1.1 Specifications summary

Any instance of the KECCAK sponge function family makes use of one of the seven KECCAK-f permutations, denoted KECCAK-$f[b]$, where $b \in \{25, 50, 100, 200, 400, 800, 1600\}$ is the width of the permutation. These KECCAK-f permutations are iterated constructions consisting of a sequence of almost identical rounds. The number of rounds n_r depends on the permutation width, and is given by $n_r = 12+\ell$, where $2^\ell = b/25$. This gives 18 rounds for KECCAK-$f[1600]$.

```
KECCAK-$f[b](A)$
  for $i$ in $0 \ldots n_r - 1$
    $A = \mathrm{Round}[b](A, \mathrm{RC}[i])$
  return $A$
```

A KECCAK-f round consists of a sequence of invertible steps each operating on the state, organized as an array of 5×5 *lanes*, each of length $w \in \{1, 2, 4, 8, 16, 32, 64\}$ $(b = 25w)$. When implemented on a 64-bit processor, a lane of KECCAK-$f[1600]$ can be represented as a 64-bit CPU word.

```
$\mathrm{Round}[b](A, \mathrm{RC})$
  $\theta$ STEP
    $C[x] = A[x,0] \oplus A[x,1] \oplus A[x,2] \oplus A[x,3] \oplus A[x,4],$        $\forall x$ in $0 \ldots 4$
    $D[x] = C[x-1] \oplus \mathrm{ROT}(C[x+1], 1),$                                 $\forall x$ in $0 \ldots 4$
    $A[x,y] = A[x,y] \oplus D[x],$                                                   $\forall (x,y)$ in $(0 \ldots 4, 0 \ldots 4)$

  $\rho$ AND $\pi$ STEPS
    $B[y, 2x+3y] = \mathrm{ROT}(A[x,y], r[x,y]),$                                    $\forall (x,y)$ in $(0 \ldots 4, 0 \ldots 4)$

  $\chi$ STEP
    $A[x,y] = B[x,y] \oplus ((\mathrm{NOT}\, B[x+1,y])\, \mathrm{AND}\, B[x+2,y]),$  $\forall (x,y)$ in $(0 \ldots 4, 0 \ldots 4)$

  $\iota$ STEP
    $A[0,0] = A[0,0] \oplus \mathrm{RC}$

  return $A$
```

Here the following conventions are in use. All the operations on the indices are done modulo 5. A denotes the complete permutation state array and $A[x,y]$ denotes a particular lane in that state. $B[x,y]$, $C[x]$ and $D[x]$ are intermediate variables. The symbol $\oplus$ denotes the bitwise exclusive OR, NOT the bitwise complement and AND the bitwise AND operation. Finally, $\mathrm{ROT}(W, r)$ denotes the bitwise cyclic shift operation, moving bit at position i into position $i + r$ (modulo the lane size).

The constants $r[x,y]$ are the cyclic shift offsets and are specified in the following table.

	$x=3$	$x=4$	$x=0$	$x=1$	$x=2$
$y=2$	25	39	3	10	43
$y=1$	55	20	36	44	6
$y=0$	28	27	0	1	62
$y=4$	56	14	18	2	61
$y=3$	21	8	41	45	15

The constants RC[i] are the round constants. The following table specifies their values in hexadecimal notation for lane size 64. For smaller sizes they must be truncated.

RC[0]	`0x0000000000000001`	RC[9]	`0x0000000000000088`
RC[1]	`0x0000000000008082`	RC[10]	`0x0000000080008009`
RC[2]	`0x800000000000808A`	RC[11]	`0x000000008000000A`
RC[3]	`0x8000000080008000`	RC[12]	`0x000000008000808B`
RC[4]	`0x000000000000808B`	RC[13]	`0x800000000000008B`
RC[5]	`0x0000000080000001`	RC[14]	`0x8000000000008089`
RC[6]	`0x8000000080008081`	RC[15]	`0x8000000000008003`
RC[7]	`0x8000000000008009`	RC[16]	`0x8000000000008002`
RC[8]	`0x000000000000008A`	RC[17]	`0x8000000000000080`

We obtain the KECCAK[r, c, d] sponge function, with parameters capacity c, bitrate r and diversifier d, if we apply the sponge construction to KECCAK-$f[r+c]$ and perform specific padding on the message input. The following pseudocode is restricted to the case of messages that span a whole number of bytes and where the bitrate r is a multiple of the lane size.

```
KECCAK[$r, c, d$]($M$)
  INITIALIZATION AND PADDING
  $S[x, y] = 0$,                                        $\forall(x, y)$ in $(0 \dots 4, 0 \dots 4)$
  $P = M || \texttt{0x01} || \text{byte}(d) || \text{byte}(r/8) || \texttt{0x01} || \texttt{0x00} || \dots || \texttt{0x00}$

  ABSORBING PHASE
  for every block $P_i$ in $P$
    $S[x, y] = S[x, y] \oplus P_i[x + 5y]$,               $\forall(x, y)$ such that $x + 5y < r/w$
    $S = $ KECCAK-$f[r + c](S)$

  SQUEEZING PHASE
  $Z = $ empty string
  while output is requested
    $Z = Z || S[x, y]$,                                  $\forall(x, y)$ such that $x + 5y < r/w$
    $S = $ KECCAK-$f[r + c](S)$

  return $Z$
```

Here S denotes the state as an array of lanes. The padded message P is organised as an array of blocks P_i, themselves organized as arrays of lanes. The || operator denotes byte string concatenation.

1.2 NIST requirements

In this section, we provide a mapping from the items required by NIST to the appropriate sections in this document.

- Requirements in [51, Section 2.B.1]
  - The complete specifications can be found in [8].
  - Design rationale: a summary is provided in Chapter 2, with pointers to sections with more details.
  - Any security argument and a preliminary analysis: this is the purpose of the complete Chapters 3–5.
  - Tunable parameters: a summary is provided in Section 2.4, with pointers to sections with more details.
  - Recommended value for each digest size: see [8, Section 1] for the number of rounds and [8, Section 4] for the other parameters.
  - Bounds below which we expect cryptanalysis to become practical: this can be found in Sections 3.3.2 and 5.4.
- Requirements in [51, Section 2.B.2]
  - The estimated computational efficiency can be found in Chapter 7.
  - A description of the platforms used to generate the estimates can be found in Section 7.3.1.
  - The speed estimate on the reference platform can also be found in Section 7.3.1.
- Requirements in [51, Section 2.B.3]
  - The known answer and Monte Carlo results can be found on the optical media.
- Requirements in [51, Section 2.B.4]
  - The expected strength of Keccak is stated in [8, Section 3].
  - The link between the security claim and the expected strength criteria listed in [51, Section 4.A] can be found in Section 6.1.2. More details can be found in Sections 3.1.3, 3.1.4, 3.1.5 and 4.7.
  - For HMAC specifically, see also Section 6.2.3.
  - Other pseudo random functions (PRF) constructions: some modes of use are proposed in Section 6.1.
- Requirements in [51, Section 2.B.5]
  - We formally state that we have not inserted any trapdoor or any hidden weakness in Keccak. Moreover, we believe that the structure of the Keccak-f permutation does not offer enough degrees of freedom to hide a trapdoor or any other weakness.
- Requirements in [51, Section 2.B.6]
  - Advantages and limitations: a summary is provided in Chapter 2, with pointers to sections with more details.

1.3 Acknowledgments

We wish to thank (in no particular order) Charles Bouillaguet and Pierre-Alain Fouque for discussing their results later published in [12] with us, Dmitry Khovratovich for discussing not yet published results [40] with us, Joachim Strömbergson for useful comments on the FPGA implementation, Yves Moulart, Bernard Kasser and all our colleagues at STMicroelectronics and NXP Semiconductors for creating the working environment in which we could work on this, and especially Joris Delclef and Jean-Louis Modave for kindly lending us fast hardware.

Chapter 2

Design rationale summary

The purpose of this chapter is to list the design choices and to briefly motivate them, although further analysis is provided in the subsequent chapters.

2.1 Choosing the sponge construction

We start with defining a generic attack:

Definition 1. *A shortcut attack [8] on a sponge function is a* generic attack *if it does not exploit specific properties of the underlying permutation or transformation.*

The KECCAK hash function makes use of the sponge construction, following the definition of [7, 9][1]. This results in the following property:

Provability It has a proven upper bound for the success probability, and hence also a lower bound for the expected workload, of generic attacks. We refer to Chapter 3 for a more in-depth discussion.

The design philosophy underlying KECCAK is the *hermetic sponge strategy.* This consists of using the sponge construction for having provable security against all generic attacks and calling a permutation (or transformation) that should not have structural properties with the exception of a compact description (see Section 4.1).

Additionally, the sponge construction has the following advantages over constructions that make use of a compression function:

Simplicity Compared to the other constructions for which upper bounds have been proven for the success of generic attacks, the sponge construction is very simple, and it also provides a bound that can be expressed in a simple way.

Variable-length output It can generate outputs of any length and hence a single function can be used for different output lengths.

Flexibility Security level can be incremented at the cost of speed by trading in bitrate for capacity, using the same permutation (or transformation).

[1]Note that RADIOGATÚN [6] and GRINDAHL [41] are not sponge functions.

Functionality Thanks to its long outputs and proven security bounds with respect to generic attacks, a sponge function can be used in a straightforward way as a MAC function, stream cipher, deterministic pseudorandom bit generator and a mask generating function (see Section 6.1).

To support arbitrary bit strings as input, the sponge construction requires a padding function. We refer to Section 3.2 for a rationale for the specific padding function we have used.

2.2 Choosing an iterated permutation

The sponge construction requires an underlying function f, either a transformation or a permutation. Informally speaking, f should be such that *it does not have properties that can be exploited in shortcut attacks.* We have chosen a permutation, constructed as a sequence of (almost) identical rounds because of the following advantages:

Block cipher experience An iterated permutation is an iterated block cipher with a fixed key. In its design one can build on knowledge obtained from block cipher design and cryptanalysis (see Chapter 5).

Memory efficiency Often a transformation is built by taking a permutation and adding a feedforward loop. This implies that (at least part of) the input must be kept during the complete computation. This is not the case for a permutation, leading to a relatively small RAM footprint.

Compactness Iteration of a single round leads to a compact specification and potentially compact code and hardware circuits.

2.3 Designing the Keccak-f permutations

The design criterion for the Keccak-f permutations is to have no properties that can be exploited in a shortcut attack when being used in the sponge construction. It is constructed as an iterated block cipher similar to Noekeon [26] and Rijndael [27], with the key schedule replaced by some simple round constants. Here we give a rationale for its features:

Bit-oriented structure Attacks where the bits are grouped (e.g., in bytes), such as integral cryptanalysis and truncated trails or differentials, are unsuitable against the Keccak-f structure.

Bitwise logical operations and fixed rotations Dependence on CPU word length is only due to rotations, leading to an efficient use of CPU resources on a wide range of processors. Implementation requires no large tables, removing the risk of table-lookup based cache miss attacks. They can be programmed as a fixed sequence of instructions, providing protection against timing attacks.

Symmetry This allows to have very compact code in software (see Section 7.3) and a very compact co-processor circuit (see Section 7.4.3) suitable for constrained environments.

Parallelism Thanks to its symmetry and the chosen operations, the design is well-suited for ultra-fast hardware implementations and the exploitation of SIMD instructions and pipelining in CPUs.

Round degree 2 This makes the analysis with respect to differential and linear cryptanalysis easier, leads to relatively simple (albeit large) systems of algebraic equations and allows the usage of very powerful protection measures against differential power analysis (DPA) both in software (see Section 7.3.4) and hardware (see Section 7.4.4).

Matryoshka structure The analysis of small versions is relevant for larger versions (see Section 5.2).

Eggs in another basket The choice of operations is very different from that in SHA-1 and the members of the SHA-2 family on the one hand and from AES on the other.

2.4 Choosing the parameter values

In KECCAK, there are basically three security-relevant parameters that can be varied:

- b: width of KECCAK-f,
- c: capacity, limited by $c < b$,
- n_r: number of rounds in KECCAK-f.

The parameters of the candidate sponge functions have been chosen for the following reasons.

- $c = 576$: The value of the capacity is the smallest such that $c \geq 512$ and r is a power of two. It is chosen to be at least 512 so as to meet the required security level of the SHA3-224 and SHA3-256 candidates. See Section 3.3.1.
- $c = 1088$: The value of the capacity is the smallest such that $c \geq 1024$ and r is a power of two. It is chosen to be at least 1024 so as to meet the required security level of the SHA3-384 and SHA3-512 candidates. See Section 3.3.1.
- $b = 1600$: The width of the KECCAK-f permutation is chosen to favor 64-bit architectures while supporting both $b > c = 576$ and $b > c = 1088$ using the same implementation. See Section 3.3.2.
- $n_\mathrm{r} = 18$: The value of n_r has been chosen to have both a good security margin and good performances. See Section 5.4.
- r a power of two: It may be convenient in some applications to have a block size which is a power of two, e.g., for a real-time application to align its data source (assumed to be organized in blocks of size a power of two) to the block size without the need of an extra buffer.

Chapter 3

The sponge construction

In this chapter, we treat the implications of the use of the sponge construction on Keccak.

3.1 Security of the sponge construction

The Keccak hash function makes use of the sponge construction, as depicted in Figure 3.1. We have introduced and analyzed this construction in [7] and proven that it is indifferentiable from a random oracle in [9].

3.1.1 Indifferentiability from a random oracle

In [9] we have proven that given capacity c, the success probability of any generic attack is upper bounded by $1 - \exp\left(-N(N+1)2^{-(c+1)}\right)$ with N the number of calls to the underlying permutation or its inverse. If $1 \ll N \ll 2^{c/2}$ this bound simplifies to $2^{-(c+1)}N^2$, resulting in a lower bound for the expected complexity of differentiating the sponge construction calling a random permutation or transformation from a random oracle of $\sqrt{\pi}2^{c/2}$. Note that this is true independently of the output length. For example, finding collisions for output lengths shorter than c has for a random sponge the same expected complexity as for a random oracle.

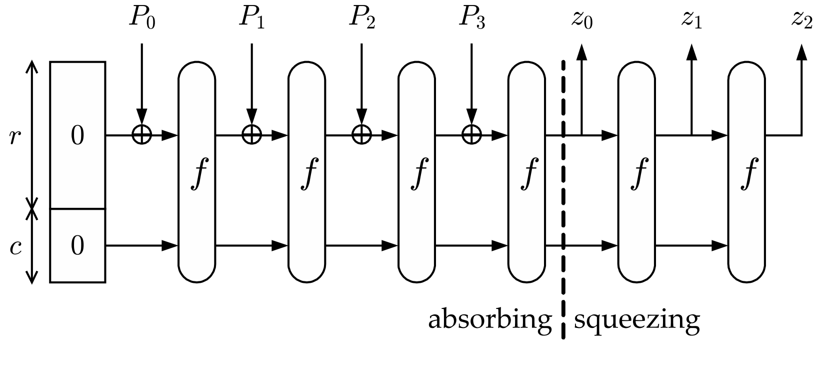

Figure 3.1: The sponge construction

3.1.2 Indifferentiability of multiple sponge functions

In our SHA-3 proposal we have multiple sponge functions that make use of the same f. The indifferentiability proof of [9] actually only covers the indifferentiability of a single sponge function instance from a random oracle. In this section we extend this proof to indifferentiability from a set of random oracles of any set of sponge functions with different capacity and/or diversifier parameters calling the same f.

Clearly, the best one can achieve is bounded by the strength of the sponge construction instance with the smallest capacity, as an adversary can always just try to differentiate the weakest construction from a random oracle. The next theorem states that we achieve this bound.

Theorem 1. *Differentiating an array of padded sponge constructions (S_i) according to [8, Algorithm 1] and calling the same random function (resp. permutation) f of width b with different (c_i, d_i) from an array of independent random oracles ($\mathcal{RO}_i$) has the same success probability as differentiating a padded sponge construction with capacity $\min_i c_i$ calling a random function (resp. permutation) f of width b from a random oracle.*

Proof: An array $(\mathcal{RO}_i)$ of independent random oracles can be alternatively implemented by having a single central random oracle $\mathcal{RO}$ and simple algorithms I_i that pre-process the input strings, so that $\mathcal{RO}_i(M) = \mathcal{RO}(I_i(M))$. To simulate independent random oracles, each I_i must produce a different range of output strings, i.e., provide domain separation. In other words, the mapping from the couple (i, M) to $x = I_i(M)$ must be injective. This reasoning is also valid if the output of the random oracles is processed by some algorithms O_i that extracts bits at predefined positions, so that $\mathcal{RO}_i(M) = O_i(\mathcal{RO}(I_i(M)))$.

In this proof, we will do similarly for the array of padded sponge constructions, by simulating them via a single sponge construction $S_{\min}$ that calls the common random function (or permutation) f. We will then rely on the indifferentiability proof in [9] for the indifferentiability between $S_{\min}$ and $\mathcal{RO}$.

For $S_{\min}$, consider the padded sponge construction where the padding simply consists of the function pad [8]. This padding satisfies the conditions imposed by the indifferentiability proof in [9]. The capacity of $S_{\min}$ is chosen to be $c_{\min} = \min_i c_i$. In the proof we make use of the bitrates r_i that are fully determined by the width b of f and the capacities c_i: we have $r_i = b - c_i$ and denote $b - c_{\min}$ by $r_{\max}$.

The function I_i is built as follows. The input message M is padded with a single 1 followed by the minimum number of zeroes such that its length becomes a multiple of 8. Then it is followed by the binary coding of d_i and that of $r_i/8$. Subsequently, if $r_i < r_{\max}$, the following processing is performed. The result is split into blocks of r_i bits and to each complete block $r_{\max} - r_i$ zeroes are appended. Note that zeroes are appended to the last block only if it is *complete*, i.e., if it has length r_i. Finally the blocks are concatenated together again and the result is $x = I_i(M)$. Due to the fact that the only allowed bitrate values are those multiple of 8, the length of x is a multiple of 8.

The function O_i is built as follows. The output $z = O_i(y)$ is obtained by splitting y in $r_{\max}$-bit blocks and truncating each block to r_i bits.

It follows that each of the functions S_i can be simulated as $S_i(M) = O_i(S_{\min}(I_i(M)))$. Furthermore, the mapping $x = I_i(M)$ from a triplet (M, r_i, d_i) to x is injective. We demonstrate this by giving an algorithm for reconstructing (M, r_i, d_i) from x. We start by extracting r_i from x. If the length of x is not a multiple of $r_{\max}$, no zeroes were added to the last block and the binary encoding of $r_i/8$ is in the last byte of x. Otherwise, it is in the last non-zero byte of x. Now we split x in $r_{\max}$ blocks, truncate each block to its first r_i bits, concatenate them again and call the resulting string m. We can now find the binary encoding of d_i in the second to last byte of m. Finally, we obtain M by removing the two last bytes from m and subsequently removing the trailing 10^* bit string.

Differentiating the array (S_i) from the array $(\mathcal{RO}_i)$ comes down to differentiating $S_{\min}$ from $\mathcal{RO}$, where $S_{\min}$ has capacity $c_{\min} = \min_i c_i$.

□

Note that for the proof to work it is crucial that the inner part (i.e., the c bits unaffected by the input or hidden from the output, see Section 3.4.1) of the sponge function instance with the smallest capacity is inside the inner parts of all other sponge function instances. This is realized in our sponge construction by systematically taking as inner part of the state its last c bits.

So if several sponge construction instances are considered together, only the smallest capacity counts. When considering a sponge construction instance, one may wonder whether the mere existence of a sponge function instance with a smaller capacity has an impact on the security of that sponge construction. This is naturally not the case, as an adversary has access to f and can simulate any construction imaginable on top of f. What matters is that the value N used in the expression for the workload shall include all calls to f and f^{-1} of which results are used.

3.1.3 Immunity to generic attacks

The indifferentiability result gives us a provable upper bound for the success probability, and hence a provable lower bound for the expected workload of any generic attack. More particularly, for a sponge construction with given c there can be no generic attacks with expected workload below $\sqrt{\pi}2^{c/2}$.

In the last few years a number of generic attacks against iterated hash functions have been published that demonstrated unexpected weaknesses:

- multicollisions [37],
- second preimages on n-bit hash functions for much less than 2^n work [39],
- herding hash functions and the Nostradamus attack [45].

Clearly these attacks are covered by the indifferentiability proof and for a sponge function the workload of generic versions of these attacks cannot be below $\sqrt{\pi}2^{c/2}$. As a matter of fact, all these attacks imply the generation of inner collisions and hence they pose no threat if generating inner collisions is difficult. We will discuss non-generic methods for the generation of inner collisions applicable to KECCAK in Section 4.3.

3.1.4 Randomized hashing

Interesting in this context is the application of randomized hashing [51]. Here a signing device randomizes the message prior to hashing with a random value that is unpredictable by the adversary. This increases the expected workload of generating a signature that is valid for two different messages from generating two colliding messages to that of generating a second pre-image for a message already signed. Now, if we keep in mind that for the sponge construction there are no generic attacks with expected workload of order below $2^{c/2}$, we can conclude the following. A lower bound for the expected complexity for generating a collision is $\min(2^{n/2}, 2^{c/2})$ and for generating a second preimage $\min(2^n, 2^{c/2})$. Hence, if $c > 2n$, randomization increases the strength against signature forgery due to generic attacks against the hash function from $2^{n/2}$ to 2^n. If the capacity is between n and $2n$, the increase is from $2^{n/2}$ to $2^{c/2}$. If $c < n$, randomized hashing does not significantly increase the security level.

3.1.5 Keyed modes

With a random oracle, one can construct a pseudo-random function (PRF) $F_k(m)$ by prepending the message m with a key k, i.e., $F_k(m) = \mathcal{RO}(k||m)$. In such a case, the function behaves as a random function to anyone not knowing the key k but having access to the same random oracle. Note that the same reasoning is valid if k is appended to the message.

More specifically, let us consider the following differentiating experiment. In a first world, let the adversary have access to the PRF $F_k(m) = \mathcal{RO}_1(k||m)$ and to the random oracle instance $\mathcal{RO}_1$ used by the PRF. In a second world, the adversary has access to two independent random oracle instances $\mathcal{RO}_2$ and $\mathcal{RO}_3$. The adversary has to differentiate the couple $(F_k, \mathcal{RO}_1)$ from the couple $(\mathcal{RO}_2, \mathcal{RO}_3)$. The only statistical difference between the two pairs comes from the identity between $F_k(m)$ and $\mathcal{RO}_1(k||m)$, whereas $\mathcal{RO}_2(m)$ and $\mathcal{RO}_3(k||m)$ give independent results. Therefore, being able to detect such statistical difference means that the key k has been recovered. For a key k containing n independent and uniform random bits, the workload to generically recover it is about 2^{n-1}.

As a consequence of the indifferentiability result, the same construction can be used with a sponge function and the same security can be expected when the adversary does not have access to a complexity of order higher than $2^{c/2}$.

Note that two options are possible, namely, prepending or appending the key. Prepending the key prevents the adversary from performing offline computations without access to F_k. If the key is appended, the adversary can for instance generate a state collision (see Section 3.4.1) before querying F_k. The difference between the two options does not make a difference below the $2^{c/2}$ complexity order bound, though.

3.2 Rationale for the padding

The padding we apply has three purposes:

- sponge input preparation,
- multi-capacity property,
- digest-length dependent digest.

We explain these three purposes in the following subsections.

3.2.1 Sponge input preparation

The padding converts the input string M in an injective way into a string P that satisfies the requirements for the input to an unpadded sponge [7, 9]: the length is a non-zero multiple of r and the last block is different from 0^r. This way, the indifferentiability proof is applicable.

3.2.2 Multi-capacity property

In the padding the value of the bitrate divided by 8 is binary coded and appended to the message. This allows to apply the sponge construction to the same permutation with different capacity values. Using this padding, the fact that the same f is used for different capacities does not jeopardize the security of the sponge construction. We have proven in Section 3.1.2 that given a random permutation (or transformation) f, for any set of allowed capacity values $\{c_1, c_2, \ldots\}$, differentiating the resulting set of sponge functions from a set of random oracles is not easier than differentiating the sponge function with capacity $\min_i c_i$ from a random oracle. We have limited the allowed bitrate values to multiples of 8 to limit splitting of input strings at byte boundaries. Note that this does not impose restrictions on possible input and output lengths.

3.2.3 Digest-length dependent digest

One may have the requirement that a hash function with the same input but requesting a different number of output bits shall behave as different hash functions. More particularly, the SHA-3 requirements specify a range of fixed digest lengths while our KECCAK sponge functions in principle have an output with arbitrary length. To achieve this we set the value of the diversifier d to the digest length expressed in bytes. We have proven in Section 3.1.2 that given a random permutation (or transformation) f, for any set of diversifier values $\{d_1, d_2, \ldots\}$ and given the capacity c, differentiating the resulting set of sponge functions from a set of random oracles is not easier than differentiating a single sponge function with capacity c from a random oracle.

3.3 Parameter choices

3.3.1 Capacity

In fixed digest-length hash functions, the required resistance against attacks is expressed relative to the digest length. Until recently one has always found it reasonable to expect a hash function to be as strong as a random oracle with respect to the classical attacks: collisions and (second) preimage. This changed after the publication of the generic attacks listed in Section 3.1.3.

For variable output-length hash functions expressing the required resistance with respect to the output length makes little sense as this would imply that it should be possible to increase the security level indefinitely by just taking longer digests. In our papers [7, 9], we have shown that for iterated variable output-length hash functions it is natural to express the resistance against attacks with respect to a single parameter called the *capacity*. Given a flat sponge claim with a specific capacity c, the claim implies that with respect to any attack with expected complexity below $\sqrt{\pi}2^{c/2}$ the hash function is as strong as a random oracle.

Choosing $c \geq 512$ for SHA3-256 (and SHA3-224) makes the sponge construction as strong as a random oracle since $c \geq 2n$. This strictly complies with the security requirements as specified in [51].

Choosing $c \geq 1024$ for SHA3-512 (and SHA3-384) follows the same reasoning. This strictly complies with the security requirements as specified in [51]. In particular, $c \geq 1024$ is needed by the requirement that (second) preimage resistance should be at least 2^{512}. Note that a capacity of $c \geq 512$ would otherwise be sufficient, especially if it is to be used in applications that require the same security level as AES-256, i.e., in the worst case, the expected workload of an attack should be 2^{256}. Also, requiring a resistance of 2^{256} is by itself already quite strong, as according to laws of thermodynamics the energy needed by an irreversible computer to perform that many operations is unreachable [61, pages 157–158].

3.3.2 Width

The width b of the permutation has been chosen as a trade-off between bitrate and memory footprint.

In a straightforward implementation, the RAM footprint is limited to the state and some working memory. For the 1600-bit version, is still limited to slightly above 200 bytes. Moreover, it allows to have a bitrate of 1024 bit and still have a high capacity.

KECCAK-f is oriented towards 64-bit CPUs. In applications that are expected to run mainly on 32-bit CPUs, one may consider using KECCAK-$f[800]$ in KECCAK$[r = 256, c = 544]$ or KECCAK$[r = 512, c = 288]$. The former has a small bitrate, hence impacting its performance. The latter is twice as fast and has a claimed security level of 2^{144} with respect to all shortcut attacks. Note that this is higher than the collision resistance claimed today for SHA-224 and SHA-256.

The smallest value of b for which a reasonable level of security can be obtained is $b = 200$. In our opinion the value of c below which attacks become practical is somewhere between 110 and 140, depending on the resources of the adversary.

3.3.3 The default sponge function KECCAK[]

One may ask the question: if we can construct arbitrary output-length hash functions, why not just have a single function and truncate at required length instead of trying to have a different hash function per supported output length? This is why we propose KECCAK[] as a fifth candidate. As said, it has a capacity of 576 bits, a bitrate of 1024 bits and its diversifier is fixed to 0. The capacity and bitrate sum up to 1600, the width of the KECCAK-f variant with lane length of 64 bits, the dominant word length in modern CPUs. For the bitrate we have chosen the largest power of 2 such that the capacity is not smaller than 512 bits. Note that the capacity of 576 bits precludes any generic attacks with expected workload below the (astronomical) number 2^{288}.

The default value of the diversifier is 0 as we believe differentiation between versions with different output lengths is in general not a requirement. Still, we are aware that there may be schemes in which different hash (or sponge) functions are used that must behave as different functions, possibly even if they have equal output length. In the latter case, setting the diversifier to the output length does not solve the issue. Rather, in such a case, the requirement that the different instances behave as different functions can be satisfied by applying domain separation. This can be done by appending or prepending different constants to the input for each of the function instances: $f_i(M) = \text{KECCAK}[](M||C_i)$ or $f_i(M) = \text{KECCAK}[](C_i||M)$. Note that for an appropriate set of constants C_i, Theorem 1 can be extended to this mode of use.

3.4 The four critical operations of a sponge

In this section we consider four critical operations that generic attacks on a sponge functions seem to imply.

3.4.1 Definitions

We call the last c bits of a state S the inner part and we denote it by $\widehat{S}$.

In the sequel, we make use of the $S_f[]$ function. For a given input string P (after padding), $S_f[P]$ denotes the value of the state obtained after absorbing P. If $s = S_f[P]$, we call P a *path* to state s (under f). Similarly, if $\widehat{s} = \widehat{S_f[P]}$ we call P a path to the inner state $\widehat{s}$. The $S_f[]$ function is defined by the following recursion:

$$\begin{aligned} S_f[\text{empty string}] =& 0^r||0^c, \\ S_f[P||a] =& f\left(S_f[P] \oplus (a||0^c)\right) \text{ for any string } P \text{ of length multiple of } r \\ & \text{and any } r\text{-bit block } a\ . \end{aligned}$$

In general, the j-th r-bit block of a sponge output is

$$z_j = S_f[P||0^{jr}],\ j \geq 0.$$

The $S_f[]$ function can be used to express the states that the sponge traverses both as it absorbs an input P and as it is being squeezed. The traversed states are $S_f[P']$ for any P' prefix of $P|0^\infty$ with $|P'| = kr$, including the empty string.

Definition 2. *A* state collision *is a pair of different paths $P \neq Q$ to the same state: $S_f[P] = S_f[Q]$.*

Definition 3. *An* inner collision *is a pair of two different paths $P \neq Q$ to the same inner state: $\widehat{S_f[P]} = \widehat{S_f[Q]}$.*

Clearly, a state collision on $P \neq Q$ implies an inner collision on $P \neq Q$. The converse is not true. However, in the absorbing phase it is very easy to produce a state collision from an inner collision. Given $P \neq Q$ such that $\widehat{S_f[P]} = \widehat{S_f[Q]}$, the pair $P||a$ and $Q||(a \oplus \lfloor S_f[P] \oplus S_f[Q] \rfloor_r)$ forms a state collision for any r-block a.

3.4.2 The operations

The four critical operations are:

- finding an inner collision;
- finding a path to a given inner state;
- finding a cycle in the output: finding an input string P and an integer $d > 0$ such that $S_f[P] = S_f[P||0^{dr}]$;
- binding an output string to a state: given a string z with length $|z|$, finding a state value s such that the sponge generates z as output. Here we can distinguish two cases:

- Short output string ($z \leq b$): the number of possible output strings z is below the number of possible states. It is likely that an inner state value can be found, and the expected number of solutions is $\approx 2^{b-z}$.
- Long output string ($z > b$): the number of possible output strings z is above the number of possible states. For a randomly chosen z, the probability that a state value may be found is 2^{b-z}. If one is found, it is likely that the inner state value is unique.

As explained in [7], the classical attacks can be executed as a combination of these operations. In [7] we have discussed generic approaches to these four operations and the corresponding success probabilities.

The optimum algorithm to find an inner collision is to build a rooted tree [7] until a collision is found. The success probability of this algorithm coincides with the success probability of differentiating the sponge construction calling a random permutation or transformation from a random oracle.

The optimum algorithm to find a path to an inner state for a permutation is to build two trees: a rooted tree and a tree ending in the final inner state. The path is found when a new node in one of the trees is also a node in the other tree. The success probability of this algorithm is slightly below that of generating an inner collision. For a transformation the tree ending in the final inner state cannot be built and the success probability is much lower. However, both for a transformation as for a permutation, the success probability for finding a path to an inner state is below that of differentiating the sponge construction from a random oracle.

For a discussion on how to find a cycle or bind an output string to a state, we refer to [7].

Chapter 4

Sponge functions with an iterated permutation

The purpose of this chapter is to discuss a number of properties of an iterated permutation that are particularly relevant when being used in a sponge construction.

4.1 The philosophy

4.1.1 The hermetic sponge strategy

For our KECCAK functions we make a flat sponge claim with the same capacity as used in the sponge construction. This implies that for the claim to stand, the underlying function (permutation or transformation) must be constructed such that it does not allow mounting shortcut attacks that have a higher success probability than generic attacks for the same workload. We call the design philosophy of adopting a sponge construction using a permutation that should not have exploitable properties the *hermetic sponge strategy.*

Thanks to the indifferentiability proof a shortcut attack on a concrete sponge function implies a distinguisher for the function (permutation or transformation) it calls. However, a distinguisher for that function does not necessarily imply an exploitable weakness in a sponge function calling it.

4.1.2 The impossibility of implementing a random oracle

Informally, a distinguisher for a function (permutation or transformation) is the demonstration of any property that sets it significantly apart from a randomly chosen function (permutation or transformation). Unfortunately, it is impossible to construct such a function that is efficient and has a reasonably sized description or code. It is not hard to see why: any practical b-bit transformation (permutation) has a compact description and implementation not shared by a randomly chosen transformation (or permutation) with its $b2^b$ (or $\log_2 2^b! \approx (b-1)2^b$) bits of entropy.

This is better known as the random oracle implementation impossibility and a formal proof for it was first given in [13] and later an alternative proof was given in [48]. In their proofs, the authors construct a signature scheme that is secure when calling a random oracle but is insecure when calling a function f taking the place of the random oracle, where the function f has a limited (polynomial) running time and can be expressed as a Turing program of limited size. This argument is valid for any cryptographic function, and so includes KECCAK-f. Now, looking more closely at the signature schemes used in [13] and [48], it turns out that they are especially designed to fail in the case of a concrete function. We find it hard to see how this property in a protocol designed to be robust may lead to its collapse of security. The proofs certainly have their importance in the more philosophical approach to cryptography, but we don't believe they prevent the design of cryptographic primitives that provide excellent security in well-engineered examples. Therefore, we address the random oracle implementation impossibility by just making an exception in our security claim.

4.1.3 The choice between a permutation and a transformation

As can be read in [7], the expected workload of the best generic attack for finding a second preimage of a message of length $|m|$ when using a transformation is of the order $2^c/|m|$. When using a permutation this is only of order $2^{c/2}$. In that respect, a transformation has preference over a permutation. This argument makes sense when developing a hash function dedicated to offering resistance against second preimage attacks. Indeed, using a transformation allows going for a smaller value of c providing the same level of security against generic attacks.

When developing a general-purpose hash function however, the choice of c is governed by the security level against the *most powerful* attack the function must resist, namely collision attacks. The resistance against output collisions that a sponge function can offer is determined by their resistance against generating inner collisions. For high values of r, the resistance against generating inner collisions is the same for a transformation or a permutation and of the order $2^{c/2}$.

4.1.4 The choice of an iterated permutation

Clearly, using a random transformation instead of a random permutation does not offer less resistance against the four critical operations, with the exception of detecting cycles [7] and the latter is only relevant if very long outputs are generated. Hence, why choose for a permutation rather than a transformation?

We believe a suitable permutation can be constructed as a fixed-key block cipher: as a sequence of simple and similar rounds. A suitable transformation can also be constructed as a block cipher, but here the input of the transformation would correspond with the key input of the block cipher. This would involve the definition of a key schedule and in our opinion results in less computational and memory usage efficiency and a more difficult analysis.

Our KECCAK functions apply the sponge construction to iterated permutations that are designed in the same way as modern block ciphers: iterate a simple nonlinear round function enough times until the resulting permutation has no properties that can be exploited in attacks. The remainder of this chapter deals with such properties and attacks. First, as an iterated permutation can be seen a block cipher with a fixed and known key, it should be impossible to construct for the full-round versions distinguishers like the known-key distinguishers for reduced-round versions of DES and AES given in [43]. This includes differentials with high differential probability (DP), high input-output correlations, distinguishers based on integral cryptanalysis or deviations in algebraic expressions of the output in terms of the input. We call this kind of distinguishers *structural*, to set them apart from trivial distinguishers that are of no use in attacks such as checking that $f(a) = b$ for some known input-output couple (a, b) or the observation that f has a compact description.

In the remainder of this chapter we will discuss some important structural distinguishers for iterated permutations, identify the properties that are relevant in the critical sponge operations and finally those for providing resistance to the classical hash function attacks.

4.2 Some structural distinguishers

In this section we discuss structural ways to distinguish an iterated permutation from a random permutation: differentials with high differential probability (DP), high input-output correlation, non-random properties in the algebraic expressions of the input in terms of the output (or vice versa) and the difficulty of solving a particular problem: the constrained-input constrained-output problem.

4.2.1 Differential cryptanalysis

A (XOR) *differential* over a function α consists of an input difference pattern a' and an output difference pattern b' and is denoted by a couple (a', b'). A *right pair* of a differential is a pair $\{a, a \oplus a'\}$ such that $\alpha(a \oplus a') \oplus \alpha(a) = b'$. In general, one can define differentials and (ordered) pairs for any Abelian group operation of the domain and codomain of α. A right (ordered) pair is then defined as $\{a + a', a\}$ such that $\alpha(a + a') = \alpha(a) \odot b'$, where $+$ corresponds to the group operation of the domain of α and $\odot$ of its codomain. In the following we will however assume that both group operations are the bitwise XOR, or equivalently, addition in $\mathbb{Z}_2^b$.

The cardinality of (a', b') is the number of right pairs and its differential probability (DP) is the cardinality divided by the total number of pairs with given input difference. We define the weight of a differential $\mathrm{w}(a', b')$ as minus the binary logarithm of its DP, hence we have $\mathrm{DP}(a', b') = 2^{-\mathrm{w}(a',b')}$. The set of values a with a a member of a right pair of a differential (a', b') can be expressed by a number of conditions on the bits of a. Hence a differential imposes a number of conditions on the absolute value at its input. In many cases these conditions can be expressed as $\mathrm{w}(a', b')$ independent binary equations.

It is well known (see, e.g., [28]) that the cardinality of non-trivial (i.e., with $a' \neq 0 \neq b'$) differentials in a random permutation operating on $\mathbb{Z}_2^n$ with n not very small has a Poisson distribution with $\lambda = 1/2$ [28]. Hence the cardinality of non-trivial differentials of an iterated permutation used in a sponge construction shall obey this distribution.

Let us now have a look at how differentials over iterated mappings are structured. A *differential trail* Q over an iterated mapping f of n_r rounds R_i consists of a sequence of $n_\mathrm{r} + 1$ difference patterns $(q_0, q_1, \ldots, q_{n_\mathrm{r}})$. Now let $f_i = \mathrm{R}_{i-1} \circ \mathrm{R}_{i-2} \circ \ldots \mathrm{R}_0$, i.e., f_i consists of the first i rounds of α. A *right pair* of a trail is a couple $\{a, a \oplus a'_0\}$ such that for all i with $0 < i \leq n_\mathrm{r}$:

$$f_i(a \oplus q_0) \oplus f_i(a) = q_i \ .$$

Note that a trail can be considered as a sequence of n_r round differentials (q_{i-1}, q_i) over each R_i. The cardinality of a trail is the number of right pairs and its DP is the cardinality divided by the total number of pairs with given input difference. We define the weight of a trail $\mathrm{w}(Q)$ as the sum of the weights of its round differentials.

The cardinality of a differential (a', b') over f is the sum of the cardinalities of all trails Q within that differential, i.e., with $q_0 = a'$ and $q_{n_\mathrm{r}} = b'$. From this, the condition on the values of the cardinality of differentials of f implies that there shall be no trails with *high* cardinality and there shall not be differentials containing *many* trails with non-zero cardinality.

Let us take a look at the cardinality of trails. First of all, note that $\mathrm{DP}(Q) = 2^{-\mathrm{w}(Q)}$ is not necessarily true, although in many cases it may be a good approximation, e.g., when $\mathrm{w}(Q) < b - 4$. The cardinality of the trail is then given by $2^{b-1} \times \mathrm{DP}(Q)$. Now, when $\mathrm{w}(Q) > b-1$, we cannot have $\mathrm{DP}(Q) = 2^{-\mathrm{w}(Q)}$ as the number of pairs is an integer. Typically, a trail with $\mathrm{w}(Q) > b-1$ has no pairs, maybe one pair and very maybe a few pairs. If all trails over an iterated permutation have weight significantly above b, most pairs will be in a trail that has only 1 right pair. In other words, trails with more than a single right pair will be rare. In those circumstances, finding a trail with non-zero cardinality is practically equivalent to finding a right pair for it. This makes such trails of very small value in cryptanalysis.

If there are no trails with low weight, it remains to be verified that there are no systematic clustering of non-zero cardinality trails in differentials. A similar phenomenon is that of *truncated differentials*. These are differentials where the input and output difference patterns are not fully determined. A first type of truncated differentials are especially a concern in ciphers where the round function treats the state bits in sets, e.g., bytes. In that case, a typical truncated differential only specifies which bytes in the input and/or output difference patterns are passive (equal to zero) and which ones are active (different from zero). The central point of these truncated differentials is that they also consist of truncated trails and that it may be possible to construct truncated trails with high cardinality. Similar to ordinary differential trails, truncated trails also impose conditions on the bits of the intermediate computation values of a, and the number of such conditions can again be quantified by defining a weight function.

A second type of truncated differentials are those where part of the output is truncated. Instead of considering the output difference pattern over the complete output of f, one considers it over a subset of (say, n of) its output bits (e.g., the inner part $\widehat{f}$). For a random b-bit to n-bit function, the cardinality of non-trivial differentials has a normal distribution with mean 2^{b-n-1} and variance 2^{b-n-1} [28]. Again, this implies that there shall be no trails of the truncated function f with low weight and there shall be no clustering of trails.

Given a trail for f, one can construct a corresponding trail for the truncated version of f. This requires exploiting the properties of the round function of f. In general, the trail for the truncated version will have a weight that is equal to or lower than the original trail. How much lower depends on the round function of f. Typically, the trail in f determines the full difference patterns up to the last few rounds. In the last few rounds the difference values in some bit positions may become unconstrained resulting in a decrease of the number of conditions.

4.2.2 Linear cryptanalysis

A (XOR) *correlation* over a function α consists of an input selection pattern v and an output selection pattern u and is denoted by a couple (v, u). It has a correlation value denoted by $C(v, u)$ equal to the correlation between the Boolean functions $v^{\mathrm{T}}a$ and $u^{\mathrm{T}}\alpha(a)$. This correlation is a real number in the interval $[-1, 1]$. We define the weight of a correlation by:

$$\mathrm{w}(v, u) = -\log_2(C^2(v, u))/2 \ .$$

In general, one can define correlations for any Abelian group operation of the domain and codomain of α, where $C(v, u)$ is a complex number in the closed unit disk [2]. In the following we will however assume that both group operations are the bitwise XOR, or equivalently, addition in $\mathbb{Z}_2^b$. We only give an introduction here, for more background, we refer to [22].

Correlations in a permutation operating on $\mathbb{Z}_2^b$ are integer multiples of 2^{2-b}. The distribution of non-trivial correlations (i.e., with $u \neq 0 \neq v$) in a random permutation operating on $\mathbb{Z}_2^b$ with b not very small has as envelope a normal distribution with mean 0 and variance 2^{-b} [28]. Hence non-trivial correlations of an iterated permutation used in a sponge construction shall obey this distribution.

Let us now have a look at how correlations over iterated mappings can be decomposed into *linear trails*. A *linear trail* Q over an iterated mapping f of n_r rounds R_i consists of a sequence of $n_\mathrm{r}+1$ selection patterns $(q_0, q_1, \ldots, q_{n_\mathrm{r}})$. A linear trail can be considered as a sequence of n_r round correlations (q_i, q_{i+1}) over each R_i and its *correlation contribution* $C(Q)$ consists of the product of the correlations of its round correlations: $C(Q) = \prod_i C(q_i, q_{i+1})$. It follows that $C(Q)$ is a real number in the interval $[-1, 1]$. We define the weight of a linear trail by

$$\mathrm{w}(Q) = -\log_2(C^2(Q))/2 = \sum_i \mathrm{w}(q_i, q_{i+1}) \ .$$

A correlation $C(v, u)$ over f is now given by the sum of the correlation contributions of all linear trails Q within that correlation, i.e., with $q_0 = v$ and $q_{n_\mathrm{r}} = u$. From this, the condition on the values of the correlations of f implies that there shall be no trails with *high* correlation contribution (so low weight) and there shall not be correlations containing *many* trails with high correlation contributions.

4.2.3 Algebraic expressions

In this section we discuss the algebraic normal form (ANF) considered over GF(2). In a mapping operating on b bits, one may define a grouping of bits in d-bit blocks for any d dividing b and consider the ANF over GF(2^d). The derivations are very similar, the only difference is that the coefficients are in GF(2^d) rather than GF(2) and that the maximum degree of individual variables is $2^d - 1$ rather than 1.

Let $g : \mathrm{GF}(2)^b \to \mathrm{GF}(2)$ be a mapping from b input bits to one output bit. The ANF is the polynomial

$$g(x_0, \ldots, x_{b-1}) = \sum_{e \in \mathrm{GF}(2)^b} G(e) x^e, \text{ with } x^e = \prod_{i=0}^{b-1} x_i^{e_i} \text{ and } G(e) \in \mathrm{GF}(2).$$

Given the truth table of $g(x)$, one can compute the ANF of g with complexity of $O(b2^b)$ as in Algorithm 1.

When g is a (uniformly-chosen) random function, each monomial x^e is present with probability one half, or equivalently, $G(e)$ behaves as a uniform random variable over $\{0, 1\}$ [30]. A transformation $f : \mathrm{GF}(2)^b \to \mathrm{GF}(2)^b$ can be seen as a tuple of b binary functions $f = (f_i)$. For a (uniformly-chosen) random transformation, each $F_i(e)$ behaves as a uniform and independent random variable over $\{0, 1\}$.

If f is a random permutation over b bits, each $F_i(e)$ is not necessarily an independent uniform variable. For instance, the monomial of maximal degree $x_0 x_1 \ldots x_{b-1}$ cannot appear since the bits of a permutation are balanced when x is varied over the whole range $\mathrm{GF}(2)^b$.

If b is small, the ANF of the permutation f can be computed explicitly by varying the b bits of input and applying Algorithm 1. A statistical test on the ANF of the output bit

Algorithm 1 Computation of the ANF of $g(x)$

```
Input $g(x)$ for all $x \in \mathrm{GF}(2)^b$
Output $G(e)$ for all $e \in \mathrm{GF}(2)^b$
Define $G[t] = G(e)$, for $t \in \mathbb{N}$, when $t = \sum_i e_i 2^i$
Start with $G(e) \leftarrow g(e)$ for all $e \in \mathrm{GF}(2)^b$
for $i = 0$ to $b - 1$ do
  for $j = 0$ to $2^{b-i-1} - 1$ do
    for $k = 0$ to $2^i - 1$ do
      $G[2^{i+1}j + 2^i + k] \leftarrow G[2^{i+1}j + 2^i + k] + G[2^{i+1}j + k]$
    end for
  end for
end for
```

functions is performed and if an abnormal deviation is found, the permutation f can be distinguished from a random permutation. Examples of statistical tests on the ANF can be found in [30].

If b is large, only a fraction of the input bits can be varied, the others being set to some fixed value. All the output bits can be statistically tested, though. This can be seen as a sampling from the actual, full b-bit, ANF. For instance, let $\tilde{f}$ be obtained by varying only the first $n < b$ inputs of f and fixing the others to zero:

$$\tilde{f}(x_0, \dots, x_{n-1}) = f(x_0, \dots, x_{n-1}, 0, \dots, 0).$$

Then, it is easy to see that any monomial x^e in the ANF of $\tilde{f}$ also appears in the ANF of f, and vice-versa, whenever $i \geq n \Rightarrow e_i = 0$.

4.2.4 The constrained-input constrained-output (CICO) problem

In this section we define and discuss a problem related to f whose difficulty is crucial if it is used in a sponge construction: the constrained-input constrained-output (CICO) problem. Let:

- $\mathcal{X} \subseteq \mathbb{Z}_2^b$: a set of possible inputs.
- $\mathcal{Y} \subseteq \mathbb{Z}_2^b$: a set of possible outputs.

Solving the CICO problem consists in finding a couple (x, y) with $y = f(x)$, $x \in \mathcal{X}$ and $y \in \mathcal{Y}$.

The sets $\mathcal{X}$ and $\mathcal{Y}$ can be expressed by a number of equations in the bits of x and y respectively. In the simplest variant, the value of a subset of the bits of x (or y) are fixed. A similarly simple case is when they are determined by a set of linear conditions on the bits of x (or y).

We define the weight of $\mathcal{X}$ as

$$\mathrm{w}(\mathcal{X}) = b - \log_2 |\mathcal{X}|,$$

and $\mathrm{w}(\mathcal{Y})$ likewise. When the conditions $y = f(x)$, $x \in \mathcal{X}$ and $y \in \mathcal{Y}$ are considered as independent, the expected number of solutions is $2^{b-(\mathrm{w}(\mathcal{X})+\mathrm{w}(\mathcal{Y}))}$. Note that there may be no solutions, and this is even likely if $\mathrm{w}(\mathcal{X}) + \mathrm{w}(\mathcal{Y}) > b$.

The expected workload of solving a CICO problem depends on b, $\mathrm{w}(\mathcal{X})$ and $\mathrm{w}(\mathcal{Y})$ but also on the nature of the constraints and the nature of f. If we make abstraction of the difficulty of finding members of $\mathcal{X}$ or $\mathcal{Y}$, generic attacks impose upper bounds to the expected complexity of solving the CICO problem:

- If finding x values in $\mathcal{X}$ is easy,
  - Trying values $x \in \mathcal{X}$ until one is found with $f(x) \in \mathcal{Y}$ is expected to take $2^{\mathrm{w}(\mathcal{Y})}$ calls to f.
  - Trying all values $x \in \mathcal{X}$ takes $2^{b-\mathrm{w}(\mathcal{X})}$ calls to f. If there is a solution, it will be found.
- If finding y values in $\mathcal{Y}$ is easy,
  - Trying values $y \in \mathcal{Y}$ until one is found with $f^{-1}(y) \in \mathcal{X}$ is expected to take $2^{\mathrm{w}(\mathcal{X})}$ calls to f^{-1}.
  - Trying all values $y \in \mathcal{Y}$ takes $2^{b-\mathrm{w}(\mathcal{Y})}$ calls to f^{-1}. If there is a solution, it will be found.

When $\mathrm{w}(\mathcal{X})$ or $\mathrm{w}(\mathcal{Y})$ is small or close to b, this problem may be generically easy, provided there is a solution.

In many cases, a CICO problem can be easily expressed as a set of algebraic equations in a set of unknowns and one may apply algebraic techniques for solving these equations such as Gröbner bases [21].

4.2.5 Multi-block CICO problems

The CICO problem can be extended from a single iteration of f to multiple iterations in a natural way. We distinguish two cases: one for the absorbing phase and another one for the squeezing phase.

An e-block absorbing CICO problem for a function f is defined by two sets $\mathcal{X}$ and $\mathcal{Y}$ and consists of finding a solution $(x_0, x_1, x_2, \ldots x_e)$ such that

$$\begin{aligned}
& x_0 \in \mathcal{X} \ , \\
& x_e \in \mathcal{Y} \ , \\
\text{for } 0 < i < e : \quad & \widehat{x_i} = 0^c \ , \\
& y_1 = f(x_0) \ , \\
\text{for } 1 < i < e : \quad & y_i = f(y_{i-1} \oplus x_{i-1}) \ , \\
& x_e = f(y_{e-1} \oplus x_{e-1}) \ .
\end{aligned}$$

A priori, this problem is expected to have solutions if $\mathrm{w}(\mathcal{X}) + \mathrm{w}(\mathcal{Y}) \leq c + er$.

An e-block squeezing CICO problem for a function f is defined by $e+1$ sets $\mathcal{X}_0$ to $\mathcal{X}_e$ and consists of finding a solution x_0 such that:

$$\begin{aligned}
\text{for } 0 \leq i \leq e : \quad & x_i \in \mathcal{X}_i \ , \\
\text{for } 0 < i \leq e : \quad & x_i = f(x_{i-1}) \ .
\end{aligned}$$

A priori, this problem is expected to have solutions if $\sum_i \mathrm{w}(\mathcal{X}_i) < b$. If it is known that there is a solution, it is likely that this solution is unique if $\sum_i \mathrm{w}(\mathcal{X}_i) > b$.

Note that if $e = 1$ both problems reduce to the simple CICO problem.

4.2.6 Cycle structure

Consider the infinite sequence $a, f(a), f(f(a)), ...$ with f a permutation over a finite domain and a an element of that set. This sequence is periodic and the set of different elements in this sequence is called a *cycle* of f. In this way, a permutation partitions its domain into a number of cycles.

Statistics of random permutations have been well studied, see [70] for an introduction and references. The cycle partition of a permutation used in a sponge construction shall again respect the distributions. For example, in a random permutation over $\mathbb{Z}_2^b$:

- The expected number of cycles is $b \ln 2$.
- The expected number of fixed points (cycles of length 1) is 1.
- The number of cycles of length at most m is about $\ln m$.
- The expected length of the longest cycle is about $G \times 2^b$, where G is the Golomb-Dickman constant ($G \approx 0.624$).

4.3 Inner collision

Assume we want to generate an inner collision with two single-block inputs. This requires finding states a and a^* such that

$$\widehat{f(a)} \oplus \widehat{f(a^*)} = 0^c \text{ with } \widehat{a} = \widehat{a^*} = 0^c\ .$$

This can be rephrased as finding a right pair $\{a, a^*\}$ with $\widehat{a} = \widehat{a^*} = 0^c$ for the differential $(a \oplus a^*, 0^c)$ of $\widehat{f}$. Requiring $\widehat{a} = \widehat{a^*} = 0^c$ is needed to obtain valid paths from the root state to iteration of f where the differential occurs. In general, it is required to know a path to the inner state $\widehat{a} = \widehat{a^*} = \widehat{S_f[P]}$; the case $\widehat{a} = \widehat{a^*} = 0^c$ is just a special case of that as $0^c = \widehat{S_f[\text{empty string}]}$.

4.3.1 Exploiting a differential trail

Assume f is an iterated function and we have a trail Q in $\widehat{f}$ with initial difference a' and final difference b' such that $\widehat{a'} = \widehat{b'} = 0^c$. This implies that for a right pair (a, a^*) of this trail, the intermediate values of a satisfy $\mathrm{w}(Q)$ conditions. If $\mathrm{w}(Q)$ is smaller than b, the expected number of pairs of such a trail is 2^{b-w}. Let us now assume that given a trail and the value of $\widehat{a}$, it is easy to find right pairs $\{a, a \oplus a'\}$ with given $\widehat{a}$. We consider two cases:

- $\mathrm{w}(Q) < r$: it is likely that right pairs exist with $\widehat{a} = 0^c$ and an inner collision can be found readily. The paths are made of the first r bits of the members of the found pair, $a \neq a^*$.
- $\mathrm{w}(Q) \geq r$: the probability that a right pair exists with $\widehat{a} = 0^c$ is $2^{r-\mathrm{w}(Q)}$.

If several trails are available, one can extend this attack by trying it for different trails until a right pair with $\widehat{a} = 0^c$ is found. If the weight of trails over f is lower bounded by $w_{\min}$, the expected workload of this method is higher than $2^{w_{\min}-r}$. With this method, differential trails do not lead to a shortcut attack if $w_{\min} > c/2 + r = b - c/2$.

One can extend this attack by allowing more than a single block in the input. In a first variant, an initial block in the input is used to vary the inner part of the state and are equal for both members of the pair that will be found. Given a trail in the second block, the problem is now to find an initial block that, once absorbed, leads to an inner state at the input of the trail, for which trail in the second block has a right pair. In other words, that leads to an inner state that satisfies a number of equations due to the trail in the second block. The equations in the second block define a set $\mathcal{Y}$ for the output of the first block with $\mathrm{w}(\mathcal{Y}) \approx \mathrm{w}(Q) - r$: the conditions imposed by the trail in the second block on the inner part of the state at its input. Moreover, the fact that the inner part of the input to f in the first iteration is fixed to zero defines a set $\mathcal{X}$ with $\mathrm{w}(\mathcal{X}) = c$. Hence, even if a right pair for the trail can be found, a CICO problem must be solved with $\mathrm{w}(\mathcal{X}) = c$ and $\mathrm{w}(\mathcal{Y}) \approx \mathrm{w}(Q) - r$ for determining the first block of the inputs.

Note that if there are no trails with weight below b, the expected number of right pairs per trail is smaller than 1 and trails with more than a single right pair will be rare. In this case, even if a trail with a right pair can be found, the generation of an inner collision implies solving a CICO problem for the first block with $\mathrm{w}(\mathcal{X}) = \mathrm{w}(\mathcal{Y}) = c$.

One can input pairs that consist of multiple input blocks where there is a difference in more than a single input block. Here, chained trails may be exploited in subsequent iterations of f. However, even assuming that the transfer of equations through f due to a trail and conditions at the output is easy, one ends up in the same situation with a number of conditions on the bits of the inner part of the state at the beginning of the first input differential. And again, if there are no trails with weight below b, the generation of an inner collision implies solving a CICO problem with $\mathrm{w}(\mathcal{X}) = \mathrm{w}(\mathcal{Y}) = c$.

If $c > b/2$, typically a CICO problem with $\mathrm{w}(\mathcal{X}) = \mathrm{w}(\mathcal{Y}) = c$ will have no solution. In that case one must consider multiple blocks and the problem to solve becomes a multi-block absorbing CICO problem. The required number of rounds e for there to be a solution is $\lceil c/r \rceil$.

4.3.2 Exploiting a differential

In the search for inner collisions, all right pairs $(a, a \oplus a')$ with $\widehat{a} = 0^c$ of a differential $(a', 0^c)$ with $\widehat{a'} = 0^c$ over $\widehat{f}$ are useful, and not only the pairs of a single trail. So it seems like a good idea to consider differentials instead of trails. However, where for a given trail it may be easy to determine the pairs that follow it, this is not true in general for a differential. Still, an $\widehat{f}$-differential may give an advantage with respect to a trail if it contains more than a single trail with low weight. On the other hand, the conditions to be right pairs of one of a set of trails tend to become more complicated as the number of trails grows. This makes algebraic manipulation more and more difficult as the number of trails to consider grows.

If there are no trails over $\widehat{f}$ with weight below b, the set of right pairs of a differential is expected to be a set that has no simple algebraic characterization and we expect the most efficient way to determine right pairs is to try different outputs of f with the required difference and computing the corresponding inputs.

4.3.3 Truncated trails and differentials

As for ordinary differential trails, the conditions imposed by a truncated trail can be transferred to the input and for finding a collision a CICO problem needs to be solved. Here the factor $\mathrm{w}(\mathcal{Y})$ is determined by the weight of the truncated trail. Similarly, truncated trails can be combined to truncated differentials and here the same difficulties can be expected as when combining ordinary trails

4.4 Path to an inner state

If $c \geq b/2$, this is simply a CICO problem with $\mathrm{w}(\mathcal{X}) = \mathrm{w}(\mathcal{Y}) = c$ and solving it results in a single-block path to an inner state. If $c < b/2$, an e-block path to the inner state can be found by solving a multi-block absorbing CICO problem with $e = \lceil r/c \rceil$.

4.5 Detecting a cycle

This is strongly linked to the cycle structure of f. If f is assumed to behave as a random permutation, the overwhelming majority of states will generate very long cycles. Short cycles do exist, but due to the sheer number of states, the probability that this will be observed is extremely low.

4.6 Binding an output to a state

We consider here only the case where the output must fully determine the state. If the capacity is smaller than the bitrate, it is highly probable that a sequence of two output blocks fully determines the inner state. In that case, finding the inner state is a CICO problem with $\mathrm{w}(\mathcal{X}) = \mathrm{w}(\mathcal{Y}) = r$.

If the capacity is larger than the bitrate, one needs more than two output blocks to uniquely determine the inner state. Finding the state consists in solving a multi-block squeezing CICO problem with $\mathrm{w}(\mathcal{X}_i) = r$. The required number of rounds e to uniquely determine the state is $\lceil b/r \rceil$.

4.7 Classical hash function criteria

In this section we discuss the properties of an iterated permutation that are relevant in the classical hash function criteria.

4.7.1 Collision resistance

We assume that the sponge function output is truncated to its first n bits and we try to generate two outputs that are the same for two different inputs. We can distinguish two ways to achieve this: with or without an inner collision. While the effort for generating an inner collision is independent of the length of the output to consider, this is not the case in general for generating output collisions. If n is smaller than the capacity, the generic attack to generate an output collision directly has a smaller workload than generating an inner collision. Otherwise, generating an inner collision and using this to construct a state collision is expected to be more efficient.

We refer to Section 4.3 for a treatment on inner collisions. With some small adaptations, that explanation also applies to the case of directly generating output collisions. The only difference is that for the last iteration of the trail, instead of considering differentials $(a', 0^c)$ over $\widehat{f}$, one needs to consider differentials $(a', 0^n)$ over $\lfloor f \rfloor_n$. When exploiting a trail, and in the absence of high-probability trails, this reduces to solving a CICO problem with $\mathrm{w}(\mathcal{X}) = \mathrm{w}(\mathcal{Y}) = c$ to find a suitable first block.

4.7.2 Preimage resistance

We distinguish three cases:

- $n > b$: in this case the output fully determines the state just prior to squeezing. Generating a preimage implies binding a state to an output and subsequently finding a path to that state. As explained in Sections 4.4 and 4.6, this comes down to solving two CICO problems.
- $r < n \leq b$: Here a sequence of input block can in theory be found by solving a problem that can be seen as a combination of a multi-round squeezing CICO problem and a multi-round absorbing CICO problem.
- $n \leq r$: A single-block preimage can be found by solving a single-block CICO problem with $\mathrm{w}(\mathcal{X}) = c$ and $\mathrm{w}(\mathcal{Y}) = n$.

4.7.3 Second preimage resistance

There are two possible strategies for producing a second preimage. In a first strategy, the adversary can try to find a second path to one of the inner states traversed when absorbing the first message. Finding a second preimage then reduces to finding a path to a given inner state [7], which is discussed in Section 4.4. As a by-product, this strategy exhibits an inner collision.

In a second strategy, the adversary can ignore the inner states traversed when absorbing the first message and instead take into account only the given output. In this case, the first preimage is of no use and the problem is equivalent to finding a (first) preimage as discussed in the two last bullets of Section 4.7.2.

4.7.4 Length extension

Length extension consists in, given $h(M)$ for an unknown input M, being able to predict the value of $h(M||x)$ for some string x. For a sponge function, length extension is successful if one can find the inner state at the end of the squeezing of M. This comes down to binding the output to a state, discussed in Section 4.6. Note that the state is probably only uniquely determined if $n \geq b$. Otherwise, the expected number of state values the output can be bound to is 2^{b-n}. In that case, the probability of success of length extension is $\max(2^{n-b}, 2^{-n})$.

In principle, if the permutation f has high input-output correlations (v, u) with $\widehat{v} = \widehat{u} = 0^c$, this could be exploited to improve the probability of guessing right when doing length extension by a single block.

4.7.5 Pseudo-random function

One can use a sponge function with an iterated permutation as a pseudorandom function (PRF) by pre-pending the input by a secret key: $\mathrm{PRF}[k](M) = \mathrm{sponge}(k||M)$. As explained in Section 6.1, this can be used to construct MAC functions and stream ciphers. A similar application is randomized hashing where an unpredictable value takes the place of the key.

Distinguishing the resulting PRF from a random oracle can be done by finding the key, or by detecting properties in the output that would not be present for a random oracle. Examples of such properties are the detection of large DP values or high correlations over f. If the key is shorter than the bitrate, finding it given the output corresponding to a single input is a CICO problem. If the key is longer, this becomes a multi-round absorbing CICO problem. If more than a single input-output pair is available, this is no longer the case. In general, an adversary can even request outputs corresponding with adaptively chosen inputs.

When we use a PRF for MAC computation, the length of the key is typically smaller than the bitrate and the output is limited to (less than) a single output block. For this case, breaking the MAC function can be considered as solving the following generic problem for f.

An adversary can query f for inputs P with $P = k||x||0^c$ and

- k: an n_k-bit secret key,
- x: an $r - n_k$-bit value chosen by the adversary,

and is given the first n bits of $f(P)$, with $n \leq r$. The goal of the adversary is predict the output of $\lfloor f(P) \rfloor_n$ for non-queried values of x with a success probability higher than 2^{-n}.

4.7.6 Output subset properties

One can define an m-bit hash function based on a sponge function by, instead of taking the m first bits of its output, just specify m bit positions in the output and consider the corresponding m bits as the output. Such a hash function shall not be weaker than a hash function where the m bits are just taken as the first m bits of the sponge output stream. If the m bits are from the same output block, there is little difference between the two functions. If the m bits are taken from different output blocks, the CICO problems implied by attacking the function tend to become more complicated and are expected to be harder to solve.

Chapter 5

The Keccak-f permutations

This chapter discusses the properties of the Keccak-f permutations that are relevant for the security of Keccak. After discussing some structural properties, we treat the different mappings that make up the round function. This is followed by a discussion of differential and linear cryptanalysis. Finally we briefly discuss the applicability of a number of cryptanalytic techniques to Keccak-f and the results of some experiments we conducted, mainly on Keccak-$f[25]$.

As a reminder, the seven Keccak-f permutations are parameterized by their width $b = 25w = 25 \times 2^\ell$, for $0 \le \ell \le 6$.

5.1 Translation invariance

Let $b = \tau(a)$ with τ a mapping that translates the state by 1 bit in the direction of the z axis. For $0 < z < w$ we have $b[x][y][z] = a[x][y][z-1]$ and for $z = 0$ we have $b[x][y][0] = a[x][y][w-1]$. Translating over t bits gives $b[x][y][z] = a[x][y][(z-t) \bmod w]$. In general, a translation $\tau[t_x][t_y][t_z]$ can be characterized by a vector with three components (t_x, t_y, t_z) and this gives:

$$b[x][y][z] = a[(x - t_x) \bmod 5][(y - t_y) \bmod 5][(z - t_z) \bmod w] \ .$$

Now we can define *translation-invariance.*

Definition 4. *A mapping α is* translation-invariant *in direction (t_x, t_y, t_z) if*

$$\tau[t_x][t_y][t_z] \circ \alpha = \alpha \circ \tau[t_x][t_y][t_z] \ .$$

Let us now define the z-period of a state.

Definition 5. *The z-period of a state a is the smallest integer $d > 0$ such that:*

$$\forall x, y \in \mathbb{Z}_5 \textit{ and } z \in \mathbb{Z}_w : a[x][y][(z+d) \bmod w] = a[x][y][z] \ .$$

It is easy to prove the following properties:

- The z-period of a state divides w.
- A state a with z-period d can be represented by w, its z-period d, and its d first slices $a[.][.][z]$ with $z < d$. We call this the *z-reduced representation* of a.

- The number of states with z-period d is zero if d does not divide w and fully determined by d only, otherwise.
- There is a one-to-one mapping between the states a' with z-period d for any lane length w that is a multiple of d and the states a with z-period d of lane length d: $a'[.][.][z] = a[.][.][z \bmod d]$.
- If α is translation-invariant in the direction of the z axis, the z-period of $\alpha(a)$ divides the z-period of a. Moreover, the z-reduced state of $\alpha(a)$ is independent of w.
- If α is injective and translation-invariant, α preserves the z-period.
- For a given w, the z-period defines a partition on the states.
- For w values that are a power of two (the only ones allowed in KECCAK), the state space consists of the states with z-period 1, 2, 2^2 up to $2^\ell = w$.
- The number of states with z-period 1 is 2^{25}. The number of states with z-period 2^d for $d \geq 1$ is $2^{2^d 25} - 2^{2^{d-1} 25}$.

5.2 The Matryoshka structure

With the exception of ι, all step mappings of the KECCAK-f round function are translation-invariant in the direction of the z axis. This allows the introduction of a size parameter that can easily be varied without having to re-specify the step mappings. As in several types of analysis abstraction can be made of the addition of constants, this allows the re-use of structures for small width versions to symmetric structures for large width versions. We refer to Section 5.5.2 for an example. As the allowed lane lengths are all powers of two, every smaller lane length divides a larger lane length. So, as the propagation structures for smaller width version are embedded as symmetric structure in larger width versions, we call it Matryoshka, after the well-known Russian dolls.

5.3 The step mappings of KECCAK-f

A round is composed from a sequence of dedicated mappings, each one with its particular task. The steps have a simple description leading to a specification that is compact and in which no trapdoor can be hidden.

Mapping the *lanes* of the state, i.e., the one-dimensional sub-arrays in the direction of the x axis, onto CPU words, results in simple and efficient software implementation for the step mappings. We start the discussion of each of the step mappings by pseudocode where the variables $a[x, y]$ represent the old values of lanes and $A[x, y]$ the new values. The operations on the lanes are limited to bitwise Boolean operations and rotations. In our pseudocode we denote by $\mathrm{ROT}(a, d)$ a translation of a over d bits where bit in position z is mapped to position $z + d \bmod w$. If the CPU word length equals the lane length, the latter can be implemented with rotate instructions. Otherwise a number of shift and bitwise Boolean instructions must be combined or bit-interleaving can be applied (see Section 7.2.2).

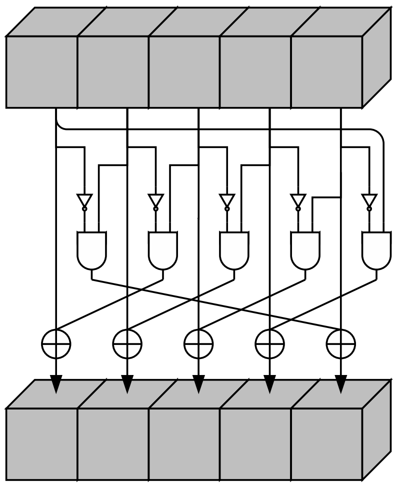

Figure 5.1: χ applied to a single row

5.3.1 Properties of χ

Figure 5.1 contains a schematic representation of χ and Algorithm 2 its pseudocode.

Algorithm 2 χ

```
for $y = 0$ to 4 do
  for $x = 0$ to 4 do
    $A[x, y] = a[x, y] \oplus ((\text{NOT}\, a[x + 1, y])\, \text{AND}\, a[x + 2, y])$
  end for
end for
```

χ is the only nonlinear mapping in KECCAK-f. Without it, the KECCAK-f round function would be linear. It can be seen as the parallel application of $5w$ S-boxes operating on 5-bit rows. χ is translation-invariant in all directions and has algebraic degree two. This has consequences for its differential propagation and correlation properties. We discuss these in short in Sections 5.3.1.1 and Section 5.3.1.2 and refer to [22, Section 6.9] for an in-depth treatment of these aspects.

χ is invertible but its inverse is of a different nature than χ itself. For example, it does not have algebraic degree 2. We refer to [22, Section 6.6.2] for an algorithm for computing the inverse of χ.

χ is simply the complement of the nonlinear function called γ used in RADIOGATÚN [6], PANAMA [23] and several other ciphers [22]. We have chosen it for its simple nonlinear propagation properties, its simple algebraic expression and its low gate count: one XOR, one AND and one NOT operation per state bit.

5.3.1.1 Differential propagation properties

Thanks to the fact that χ has algebraic degree 2, for a given input difference a', the space of possible output differences forms a linear affine variety [21] with $2^{\mathrm{w}(a',b')}$ elements. Moreover, the cardinality of a differential (a', b') over χ is either zero or a power of two. The corresponding (restriction) weight $\mathrm{w}(a', b') = \mathrm{w}(a')$ is an integer that only depends on the input difference pattern a'. A possible differential imposes $\mathrm{w}(a')$ linear conditions on the bits of input a.

We now provide a recipe for constructing the affine variety of output difference patterns corresponding to an input difference pattern, applied to a single row. Indices shall be taken modulo 5 (or in general, the length of the register). We denote by $\delta(i)$ a pattern with a single nonzero bit in position i and $\delta(i, j)$ a pattern with only non-zero bits in positions i and j.

We can characterize the linear affine variety of the possible output differences by an offset A' and a basis $\langle c_j \rangle$. The offset is $A' = \chi(a')$. We construct the basis $\langle c_j \rangle$ by adding vectors to it while running over the all bit positions i:

- If $a'_i a'_{i+1} a'_{i+2} a'_{i+3} \in \{\cdot 100, \cdot 11\cdot, 001\cdot\}$, extend the basis with $\delta(i)$.
- If $a'_i a'_{i+1} a'_{i+2} a'_{i+3} = \cdot 101$, extend the basis with $\delta(i, i+1)$.

The (restriction) weight of a difference pattern is equal to its Hamming weight plus the number of patterns 001. The all-1 input difference pattern results in the affine variety of odd-parity patterns and has weight 4 (or in general the length of the register minus 1). Among the 31 non-zero difference patterns, 5 have weight 2, 15 weight 3 and 11 weight 4.

A differential (a', b') leads to a number of conditions on the bits of the absolute value a. Let $B = A' \oplus b' = \chi(a') \oplus b'$, then we can construct the conditions on a by running over each bit position i:

- $a'_{i+1} a'_{i+2} = 10$ imposes the condition $a_{i+2} = B_i$.
- $a'_{i+1} a'_{i+2} = 11$ imposes the condition $a_{i+1} \oplus a_{i+2} = B_i$.
- $a'_{i+1} a'_{i+2} = 01$ imposes the condition $a_{i+1} = B_i$.

5.3.1.2 Correlation properties

Thanks to the fact that χ has algebraic degree 2, for a given output selection vector u, the space of input selection vectors v whose parities have a non-zero correlation with the parity determined by u form a linear affine variety. This variety has $2^{\mathrm{w}(v,u)}$ elements, with $\mathrm{w}(v, u) = \mathrm{w}(u)$ the (correlation) weight function, which is an even integer that only depends on the output selection pattern u. Moreover, the magnitude of a correlation over χ is either zero or equal to $2^{\mathrm{w}(u)}$.

We now provide a recipe for constructing the affine variety of input selection patterns corresponding to an output selection pattern, applied to a single row. Indices shall again be taken modulo 5 (or in general, the length of the register). We use the term *1-run of length ℓ* to denote a sequence of ℓ 1-bits preceded and followed by a 0-bit.

We characterize the linear affine variety with an offset U' and a basis $\langle c_j \rangle$ and build the offset and basis by running over the output selection pattern. First initialize the offset to 0 and the basis to the empty set. Then for each of the 1-runs $a_s a_{s+1} \ldots a_{s+\ell-1}$ do the following:

- Add a 1 in position s of the offset U'.
- Set $i = s$, the starting position of the 1-run.
- As long as $a_i a_{i+1} = 11$ extend the basis with $\delta(i + 1, i + 3)$ and $\delta(i + 2)$, add 2 to i and continue.

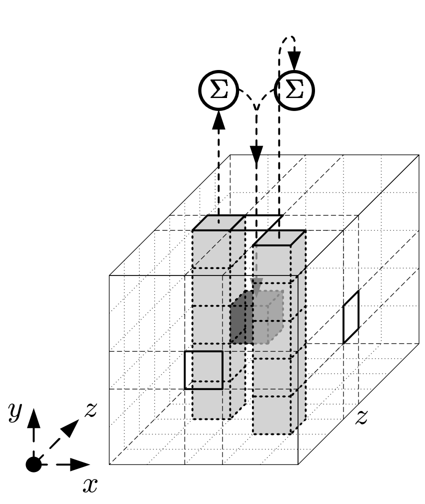

Figure 5.2: θ applied to a single bit

- If $a_i a_{i+1} = 10$ extend the basis with $\delta(i+1)$ and $\delta(i+2)$.

The (correlation) weight of a selection pattern is equal to its Hamming weight plus the number of 1-runs of odd length. The all-1 output selection pattern results in the affine variety of odd-parity patterns and has weight 4 (or in general the length of the register minus 1). Of the 31 non-zero selection patterns, 10 have weight 2 and 21 have weight 4.

5.3.2 Properties of θ

Figure 5.2 contains a schematic representation of θ and Algorithm 3 its pseudocode.

Algorithm 3 θ

```
for $x = 0$ to 4 do
  $C[x] = a[x, 0]$
  for $y = 1$ to 4 do
    $C[x] = C[x] \oplus a[x, y]$
  end for
end for
for $x = 0$ to 4 do
  $D[x] = C[x-1] \oplus \mathrm{ROT}(C[x+1], 1)$
  for $y = 0$ to 4 do
    $A[x, y] = a[x, y] \oplus D[x]$
  end for
end for
```

The θ mapping is linear and aimed at diffusion and is translation-invariant in all directions. Its effect can be described as follows: it adds to each bit $a[x][y][z]$ the bitwise sum of the parities of two columns: that of $a[x-1][\cdot][z]$ and that of $a[x+1][\cdot][z-1]$. Without θ, the KECCAK-f round function would not provide diffusion of any significance. The θ mapping has a branch number as low as 4 but provides a high level of diffusion on the average. We refer to Section 5.5.3 for a more detailed treatment of this.

The state can be represented by a polynomial in the three variables x, y and z with binary coefficients. Here the coefficient of the monomial $x^i y^j z^k$ denotes the value of bit $a[i][j][k]$. The exponents i and j range from 0 to 4 and the exponent k ranges from 0 to $w-1$. In this representation a translation $\tau[t_x][t_y][t_z]$ corresponds with the multiplication by the monomial $x^{t_x} y^{t_y} z^{t_z}$ modulo the three polynomials $1+x^5$, $1+y^5$ and $1+z^w$. More exactly, the polynomial representing the state is an element of a polynomial quotient ring defined by the polynomial ring over $\mathrm{GF}(2)[x, y, z]$ modulo the ideal generated by $\langle 1+x^5, 1+y^5, 1+z^w \rangle$. A translation corresponds with multiplication by $x^{t_x} y^{t_y} z^{t_z}$ in this quotient ring. The z-period of a state a is d if d is the smallest nonzero integer such that $1+z^d$ divides a. Let a' be the polynomial corresponding to the z-reduced state of a, then a can be written as

$$a = (1 + z^d + z^{2d} + \ldots + z^{w-d}) \times a' = \frac{1+z^w}{1+z^d} \times a' \ .$$

When the state is represented by a polynomial, the mapping θ can be expressed as the multiplication (in the quotient ring defined above) by the following polynomial :

$$1 + \bar{y}\left(x + x^4 z\right) \ \text{ with } \bar{y} = \sum_{i=0}^{4} y^i = \frac{1+y^5}{1+y} \ . \tag{5.1}$$

The inverse of θ corresponds with the multiplication by the polynomial that is the inverse of polynomial (5.1). For $w = 64$, we have computed this with the open source mathematics software SAGE [1] after doing a number of manipulations. First, we assume it is of the form $1 + \bar{y}Q$ with Q a polynomial in x and z only:

$$\left(1 + \bar{y}(x + x^4 z)\right) \times (1 + \bar{y}Q) = 1 \bmod \left\langle 1+x^5, 1+y^5, 1+z^{64} \right\rangle \ .$$

Working this out and using $\bar{y}^2 = \bar{y}$ yields

$$Q = 1 + (1 + x + x^4 z)^{-1} \bmod \left\langle 1+x^5, 1+z^{64} \right\rangle \ .$$

The inverse of $1+x+x^4z$ can be computed with a variant of the extended Euclidian algorithm for polynomials in multiple variables. At the time of writing this was unfortunately not supported by SAGE. Therefore, we reduced the number of variables to one by using the change of variables $t = x^{-2}z$. We have $x = t^{192}$ and $x^4 z = t^{193}$, yielding:

$$Q = 1 + (1 + t^{192} + t^{193})^{-1} \bmod (1 + t^{320}) \ .$$

By performing a change in variables from t to x and z again, Q is obtained.

For $w < 64$, the inverse can simply be found by reducing Q modulo $1+z^w$. For $w = 1$, the inverse of θ reduces to $1 + \bar{y}(x^2 + x^3)$.

For all values of $w = 2^\ell$, the Hamming weight of the polynomial of θ^{-1} is of the order $b/2$. This implies that applying θ^{-1} to a difference pattern with a single active bit results in a difference pattern with about half of the bits active. Similarly, a selection pattern at the output of θ^{-1} determines a selection pattern at its input with about half of the bits active.

We have chosen θ for its high average diffusion and low gate count: two XORs per bit. Thanks to the interaction with χ each bit at the input of a round potentially affects 31 bits at its output and each bit at the output of a round depends on 31 bits at its input. Note that without the translation of one of the two sheet parities this would only be 25 bits.

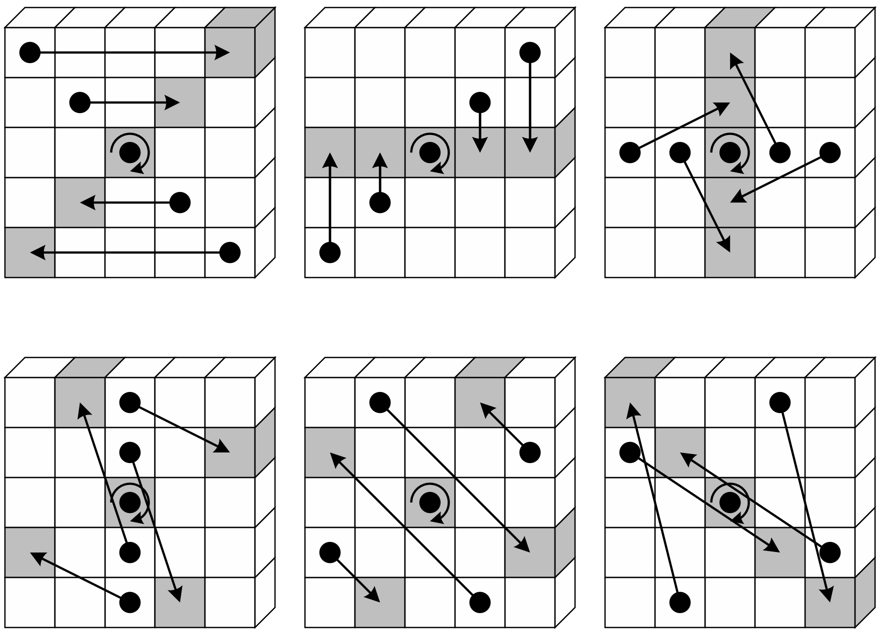

Figure 5.3: π applied to a slice. Note that $x = y = 0$ is depicted at the center of the slice.

5.3.3 Properties of π

Figure 5.3 contains a schematic representation of π and Algorithm 4 its pseudocode.

Algorithm 4 π

```
for $x = 0$ to 4 do
  for $y = 0$ to 4 do
    $\begin{pmatrix} X \\ Y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$
    $A[X, Y] = a[x, y]$
  end for
end for
```

Note that in an efficient program π can be implemented implicitly by addressing.

The mapping π is a transposition of the lanes that provides dispersion aimed at long-term diffusion. Without it, Keccak-f would exhibit periodic trails of low weight. π operates in a linear way on the coordinates (x, y): the lane in position (x, y) goes to position $(x, y)M^{\mathrm{T}}$, with $M = \left(\begin{smallmatrix} 0 & 1 \\ 2 & 3 \end{smallmatrix}\right)$ a 2 by 2 matrix with elements in GF(5). It follows that the lane in the origin $(0, 0)$ does not change position. As π operates on the slices independently, it is translation-invariant in the z-direction. The inverse of π is defined by M^{-1}.

Within a slice, we can define 6 axes, where each axis defines a *direction* that partitions the 25 positions of a slice in 5 sets:

- x axis: rows or planes;
- y axis: columns or sheets;

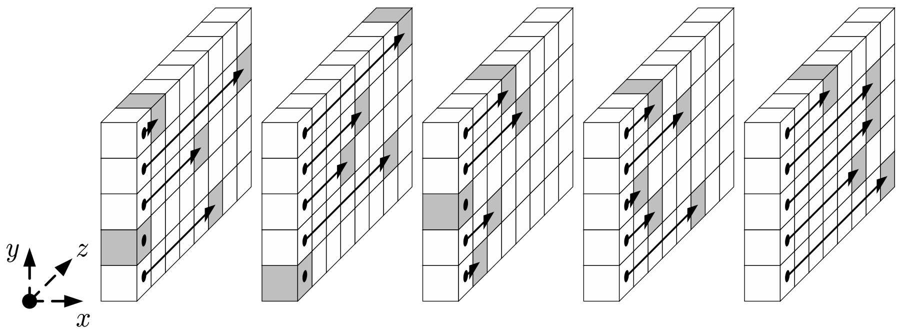

Figure 5.4: ρ applied to the lanes. Note that $x = y = 0$ is depicted at the center of the slices.

- $y = x$ axis: rising 1-slope;
- $y = -x$ axis: falling 1-slope;
- $y = 2x$ axis: rising 2-slope;
- $y = -2x$ axis: falling 2-slope;

The x axis is just the row through the origin, the y axis is the column through the origin, etc.

There are many matrices that could be used for π. In fact, the invertible 2 by 2 matrices with elements in GF(5) with the matrix multiplication form a group with 480 elements containing elements of order 1, 2, 3, 4, 5, 6, 8, 10, 12, 20 and 24. Each of these matrices defines a permutation on the 6 axes, and equivalently, on the 6 directions. Thanks to its linearity, the 5 positions on an axis are mapped to 5 positions on an axis (not necessarily the same). Similarly, the 5 positions that are on a line parallel to an axis, are mapped to 5 positions on a line parallel to an axis.

For π we have chosen a matrix that defines a permutation of the axes where they are in a single cycle of length 6 for reasons explained in Section 5.5.6. Implementing π in hardware requires no gates but results in wiring.

5.3.4 Properties of ρ

Figure 5.4 contains a schematic representation of ρ, while Table 5.1 lists its translation offsets. Algorithm 5 gives pseudocode for ρ.

Algorithm 5 ρ

```
  $A[0,0] = a[0,0]$
  $\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$
  **for** $t = 0$ to 23 **do**
    $A[x,y] = \mathrm{ROT}(a[x,y], (t+1)(t+2)/2)$
    $\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$
  **end for**
```

	$x = 3$	$x = 4$	$x = 0$	$x = 1$	$x = 2$
$y = 2$	153	231	3	10	171
$y = 1$	55	276	36	300	6
$y = 0$	28	91	0	1	190
$y = 4$	120	78	210	66	253
$y = 3$	21	136	105	45	15

Table 5.1: The offsets of ρ

The mapping ρ consists of translations within the lanes aimed at providing inter-slice dispersion. Without it, diffusion between the slices would be very slow. It is translation-invariant in the z-direction. The inverse of ρ is the set of lane translations where the constants are the same but the direction is reversed.

The 25 translation constants are the values defined by $i(i+1)/2$ modulo the lane length. It can be proven that for any ℓ, the sequence $i(i+1)/2 \bmod 2^\ell$ has period $2^{\ell+1}$ and that any sub-sequence with $n2^\ell \leq i < (n+1)2^\ell$ runs through all values of $\mathbb{Z}_{2^\ell}$. From this it follows that for lane lengths 64 and 32, all translation constants are different. For lane length 16, 9 translation constants occur twice and 7 once. For lane lengths 8, 4 and 2, all translation constants occur equally often except the translation constant 0, that occurs one time more often. For the mapping of the (one-dimensional) sequence of translation constants to the lanes arranged in two dimensions x and y we make use of the matrix of π. This groups the lanes in a cycle of length 24 on the one hand and the origin on the other. The non-zero translation constants are allocated to the lanes in the cycle, starting from $(1, 0)$.

ρ is very similar to the transpositions used in RadioGatún[6], Panama [23] and Step-RightUp [22]. In hardware its computational cost corresponds to wiring.

5.3.5 Properties of ι

The mapping ι consists of the addition of round constants and is aimed at disrupting symmetry. Without it, the round function would be translation-invariant in the z direction and all rounds of Keccak-f would be equal making it subject to attacks exploiting symmetry such as slide attacks. The number of *active bit positions* of the round constants, i.e., the bit positions in which the round constant can differ from 0, is $\ell + 1$. As ℓ increases, the round constants add more and more asymmetry.

The bits of the round constants are different from round to round and are taken as the output of a maximum-length LFSR. The constants are only added in a single lane of the state. Because of this, the disruption diffuses through θ and χ to all lanes of the state after a single round.

In hardware, the computational cost of ι is a few XORs and some circuitry for the generating LFSR. In software, it is a single bitwise XOR instruction.

5.3.6 The order of steps within a round

The reason why the round function starts with θ is due to the usage of Keccak-f in the sponge construction. It provides a mixing between the inner and outer parts of the state. Typically, the inner part is the part that is unknown to, or not under the control of the adversary. The order of the other step mappings is arbitrary.

5.4 Choice of parameters: the number of rounds

We here provide our estimate for our SHA-3 candidates specified in [8, Section 4] of how many rounds in Keccak-f[1600] are sufficient to provide resistance against three types of attacks:

- Construction of structural distinguisher for Keccak-f[1600]: 13 rounds.
- Shortcut attack for collision or (second) preimage for any of the five candidates: 11 rounds.
- Practical generation of an actual collision or (second) preimage for any of the five candidates (where the output of Keccak[] is truncated to not less than 256 bits): 9 rounds.

These estimates are based on the results of our preliminary analysis that is treated in the remainder of this chapter. By having 18 rounds in Keccak-f[1600], we take a considerable security margin with respect to all three types of attack.

5.5 Differential and linear cryptanalysis

5.5.1 Trail propagation

Keccak-f can be seen as the alternation of the nonlinear mapping χ, the addition of the round constant ι, and a linear mapping $\pi \circ \rho \circ \theta$, that we will denote by λ.

Differential trails propagate in the following way. A difference pattern a' at the input of χ determines a linear affine variety of possible difference patterns b' at its output. This can be expressed by an offset and a basis with $\mathrm{w}(a')$ elements. The mapping ι has no impact on a difference pattern and can be ignored in differential trails. A difference pattern a' at the input of λ determines the difference pattern b' at its output by $b' = \lambda(a')$. It suffices to specify one difference pattern per round to fully determine a differential trail. We choose to specify the difference pattern at the input of χ. This has the advantage that the weight of the trail is given by the sum of the weights of its difference patterns. An n-round differential trail Q defined by the difference patterns $(q_0, q_1, \ldots, q_n)$ has weight:

$$\mathrm{w}(Q) = \sum_{i=0}^{n-1} \mathrm{w}(q_i) \ .$$

Linear trails propagate in the following way. A selection pattern u at the output of χ determines a linear affine variety of possible selection patterns v at its input. Again, this can be expressed by a basis with $\mathrm{w}(u)$ elements and an offset. The mapping ι has no impact on a selection pattern. A selection pattern u at the output of λ determines the selection pattern v at its input. If we express the linear mapping λ by the multiplication by a matrix M_λ, we have $v = M_\lambda^{\mathrm{T}} u$. It suffices to specify one selection pattern per round to fully determine a linear trail. Here we choose to specify the selection pattern at the output of χ. This has the advantage that the weight of the trail is given by the sum of the weights of its selection patterns. An n-round linear trail Q defined by the selection patterns $(q_0, q_1, \ldots, q_n)$ has weight:

$$\mathrm{w}(Q) = \sum_{i=1}^{n} \mathrm{w}(q_i) \ .$$

The magnitude of the correlation contribution of a trail is given by $2^{-\mathrm{w}(Q)}$. The sign is the product of the correlations over the χ and ι steps in the trail. The sign of the correlation contribution of a linear trail hence depends on the round constants.

5.5.2 The Matryoshka consequence

The weight and existence of trails (both differential and linear) is independent of ι. The fact that all other step mappings of the round function are translation-invariant in the direction of the z axis, makes that a trail Q implies $w - 1$ other trails: those obtained by translating the patterns of Q over any non-zero offset in the z direction. If all patterns in a trail have a z-period below or equal to d, this implies only $d - 1$ other trails.

Moreover, a trail for a given width b implies a trail for all larger widths b'. The patterns are just defined by their z-reduced representations and the weight must be multiplied by b'/b. Note that this is not true for the cardinality of differential trails and the sign of the correlation contribution of linear trails, as these do depend on the round constants.

5.5.3 The column parity kernel

The mapping θ is there to provide diffusion. As said, it can be expressed as follows: add to each bit $a[x][y][z]$ the bitwise sum of the parities of two columns: that of $a[x-1][\cdot][z]$ and that of $a[x+1][\cdot][z-1]$. From this we can see that for states in which all columns have even parity, θ is the identity. We call this set of states the *column parity kernel* or *CP-kernel* for short.

The size of the CP-kernel is 2^{20w} as there are in total $2^b = 2^{25w}$ states and there are 2^{5w} independent parity conditions. The kernel contains states with Hamming weight values as low as 2: those with two active bits in a single column. Due to these states, θ only has a branch number (expressed in Hamming weight) of 4.

The low branch number is a consequence of the fact that only the column parities propagate. One could consider changing θ to improve the worst-case diffusion, but this would significantly increase the computational cost of θ as well. Instead, we have chosen to address the CP-kernel issue by carefully choosing the mapping π.

We can compute from a $25w$-bit state its $5w$-bit *column parity pattern*. These patterns partition the state space in 2^{5w} subsets, called the *parity classes*, with each 2^{20w} elements. We can now consider the branch number restricted to the states in a given parity class. As said, the minimum branch number that can occur is 4 for the CP-kernel, the parity class with the all-zero column parity pattern. Over all other parity classes, the branch number is at least 12.

Note that for states where *all* columns have odd parity, θ adds 0 to every bit and also acts as the identity. However, the Hamming weight of states in the corresponding parity class is at least $5w$ resulting in a branch number of $10w$.

5.5.4 One and two-round trails

Now we will have a look at minimum weights for trails with one and two rounds. The minimum weight for a one-round differential trail (q_0, q_1) is obtained by taking a difference pattern q_0 with a single active bit and has weight 2. For a linear trail this is obtained by a selection pattern q_1 with a single active bit or two neighboring active bits in the same row, and the weight is also 2. This is independent of the width of KECCAK-f.

For the minimum weight of two-round trails we use the following property of χ: if a difference pattern at the input of χ restricted to a row has a single active bit, the same difference pattern is a possible difference pattern at its output. Hence for difference patterns with zero or one active bits per row, χ can behave as the identity. Similarly, for selection patterns with zero or one active bits per row, χ can behave as the identity. We call such trails in which the patterns at the input and output of χ are the same, *χ-zero trails*. Note that all patterns in a χ-zero trail are fully determined by the first pattern q_0.

For all widths, the two-round trails with minimum weight are χ-zero trails. For a differential trail, we choose for q_0 a difference pattern with two active bits that are in the same column. After χ the difference pattern has not changed and as it is in the CP-kernel, it goes unchanged through θ as well. The mappings π and ρ move the two active bits to different columns, but in no case to the same row. This results in a value of q_1 with two active bits in different rows. As the weight of both q_0 and q_1 is 4, the resulting trail has weight 8. For linear trails, a similar reasoning applies and the minimum trail weight is also 8. Note that the low weight of these trails is due to the fact that the difference pattern at the input of θ in round 0 is in the CP-kernel.

5.5.5 Three-round trails: kernel vortices

From here on, we concentrate on differential trails as the explanation is very similar for linear trails. We can construct a three-round χ-zero trail where both difference patterns q_0 and q_1 are in the CP-kernel. As in a χ-zero trail χ behaves as the identity and q_0 is in the CP-kernel, we have $q_1 = \pi(\rho(q_0))$. Hence, we can transfer the conditions that q_0 is in the kernel to conditions on q_1, or vice versa.

We will now look for patterns q_0 where both q_0 and $\pi(\rho(q_0))$ are in the CP-kernel. q_0 cannot be a pattern with only two active bits in one column since $\pi \circ \rho$ maps these bits to two different columns in q_1.

The minimum number of active bits in q_0 is four, where both q_0 and q_1 have two active columns with each two active bits. We will denote these four active bits as *points* 0, 1, 2 and 3. Without loss of generality, we assume these points are grouped two by two in columns in q_0: $\{0, 1\}$ in one column and $\{2, 3\}$ in another one. In q_1 we assume they are grouped in columns as $\{1, 2\}$ and $\{3, 0\}$.

The mapping π maps sheets (containing the columns) to falling 2-slopes and maps planes to sheets. Hence the points $\{0, 1\}$ and $\{2, 3\}$ are in falling 2-slopes in q_1 and the points $\{1, 2\}$ and $\{3, 0\}$ are in planes in q_0. This implies that projected on the (x, y) plane, the four points of q_0 form a rectangle with horizontal and vertical sides. Similarly, in q_1 they form a parallelogram with vertical sides and sides that are falling 2-slopes.

The (x, y) coordinates of the four points in q_0 are completely determined by those of the two opposite corner points (x_0, y_0) and (x_2, y_2). The four points have coordinates: (x_0, y_0), (x_0, y_2), (x_2, y_2) and (x_2, y_0). The number of possible choices is $\binom{2}{5}^2 = 100$. Now let us have a look at their z coordinates. Points 0 and 1 should be in the same column and points 2 and 3 too. Hence $z_1 = z_0$ and $z_3 = z_2$. Moreover, ρ shall map points 1 and 2 to the same slice and bits 3 and 0 too. This results in the following conditions for their z-coordinates:

$$\begin{aligned} z_0 + r[x_0][y_2] &= z_2 + r[x_2][y_2] \mod w \ , \\ z_2 + r[x_2][y_0] &= z_0 + r[x_0][y_0] \mod w \ , \end{aligned} \tag{5.2}$$

with $r[x][y]$ denoting the translation offset of ρ in position (x, y). They can be converted to the following two conditions:

$$\begin{aligned} z_2 &= z_0 + r[x_0][y_2] - r[x_2][y_2] \mod w \ , \\ z_2 &= z_0 + r[x_0][y_0] - r[x_2][y_0] \mod w \ . \end{aligned}$$

In any case z_0 can be freely chosen, and this determines z_2. Subtracting these two equations eliminates z_0 and z_2 and results in:

$$r[x_0][y_0] - r[x_0][y_2] + r[x_2][y_2] - r[x_2][y_0] = 0 \bmod w \ . \tag{5.3}$$

If this equation is not satisfied, the equations (5.2) have no solution.

Consider now $w = 1$. In that case Equation (5.3) is always satisfied. However, in order to be χ-zero, the points must be in different rows, and hence in different planes, both in q_0 and q_1, and this is not possible for a rectangle.

If $\ell \geq 1$, Equation (5.3) has a priori a probability of $2^{-\ell}$ of being satisfied. Hence, we can expect about $2^{-\ell}100$ rectangles to define a state q_0 with both q_0 and $\pi(\rho(q_0))$ in the CP-kernel. So it is not inconceivable that such patterns exists for $w = 64$. This would result in a 3-round trail with weight of 8 per round and hence a total weight of 24. However, for our choice of π and ρ, there are no such trails for $w > 16$.

Note that here also the Matryoshka principle plays. First, the z-coordinate of one of the points can be freely chosen and determines all others. So, given a rectangle that has a solution for Equation (5.3), there are 2^ℓ patterns q_0, one for each choice of z_0. Second, if Equation (5.3) is not satisfied for ℓ but it is for some $\ell' < \ell$, it implies a pattern q_0 with $2^{\ell-\ell'}4$ points rather than 4 for which both q_0 and $\pi(\rho(q_0))$ are in the kernel.

These patterns can be generalized by extending the number of active bits: a pattern q_0 with both q_0 and $\pi(\rho(q_0))$ in the kernel can be constructed by arranging $2e$ points in a cycle in the (x, y) plane and giving the appropriate z-coordinates. In such a cycle each combination of points $\{2i, 2i+1\}$ are in the same sheet and each combination of points $\{2i+1, 2i+2\}$ are in the same plane. We call such a cycle of $2e$ (x, y) positions a *kernel vortex* V.

For the z coordinates, the conditions that the points $\{2i, 2i + 1\}$ are in the same column in q_0 and the points $\{2i + 1, 2i + 2\}$ are in the same column in q_1 results in $2e$ conditions. Similar to the rectangle case, these conditions only have a solution if the ρ rotation constants in the lanes of the cycle satisfy a condition. For a given kernel vortex V, we define its depth $d(V)$ as:

$$d(V) = \sum_{i=0}^{2e-1} (-1)^i r[\text{point } i]. \tag{5.4}$$

Now, the vortex results in a valid pattern q_0 if $d(V) \bmod w = 0$. We call the largest power of 2 dividing $d(V)$ the *character* of the vortex $c(V)$. If $d(V) = 0$, we say its character is $c(V) = \infty$. Summarizing, a vortex V defines a valid pattern q_0 with $2e$ active bits for lane length $w \leq c(V)$. For constructing low-weight 3-round trails, it suffices to find vortices with small e and large character: given a vortex V it results in a 3-round trail with weight $12e$ for all values of $2^\ell \leq c(V)$ and with weight $12e2^\ell/c(V)$ for all values of $2^\ell > c(V)$ (using symmetric trails of period $c(V)$).

As the length of vortices grows, so does their number. There are 600 vortices of length 6, 8400 of length 8 and 104040 of length 10. The character $c(V)$ over these vortices has an exponential distribution: about half of them has character 1, 1/4 have character 2, 1/8 have character 4 and so on. It follows that as their length $2e$ grows, there are more and more vortices that result in valid pattern q_0 with $2e$ active bits, even for lane length 64.

Moreover, one can construct patterns q_0 containing two or more vortices, provided that they do not result in a row with two active bits in either q_0 or q_1. The character of such a combination is just the minimum of the characters of its component vortices. Clearly, due the large number of kernel vortices, it is likely that there are three-round trails with low weight for any choice of ρ and π. For our choice of π and ρ, the vortex that leads to the 3-round trail with the smallest weight for Keccak-f is one of length 6 and character 64. It results in a 3-round trail with weight 36.

5.5.6 Beyond three-round trails: choice of π

We will now try to extend this to four-round trails: we try to find patterns q_0 such that q_0, q_1 and q_2 are in the CP-kernel.

A vortex of length 4, i.e., with $e = 2$ cannot do the job with our choice of π: a rectangle in q_0 with sheets and planes as sides results in a parallelogram in q_1 with falling 2-slopes and columns as sides and in a parallelogram in q_2 with rising 2-slopes and falling 2-slopes as sides. Hence the four points in q_2 cannot grouped in columns 2 by 2 and therefore it cannot be in the kernel.

Consider now a vortex of length 6. We choose the points such that the grouping in columns is $\{0,1\},\{2,3\},\{4,5\}$ in q_0, it is $\{1,2\},\{3,4\},\{5,0\}$ in q_1 and $\{1,4\},\{2,5\},\{3,0\}$ in q_2. The grouping in q_1 simply implies that $\{1,2\},\{3,4\},\{5,0\}$ are grouped in planes in q_0. Actually, the first two groupings are similar to the three-round trail case: they determine a character $c(V)$ and fix the z coordinates of all points but one. We will now study the implications of the grouping in q_2 on the (x, y) coordinates. Grouping in columns (sheets) in q_2 implies grouping in planes in q_1 and subsequently grouping in rising 1-slopes in q_0.

For the z-coordinates this results in 3 additional conditions: points 1 and 4, points 2 and 5 and points 3 and 0 must have the same z-coordinate in q_2. Similar to Equation (5.2) these conditions are equalities modulo 2^ℓ. For each of the equations, the a priori probability that it is satisfied for a given value of 2^ℓ is $2^{-\ell}$. With each of these equations we can again associate a character: the largest value w that is a power of two for which the equation is satisfied. The 4-round character (i.e. leading to q_0, q_1 and q_2 all three in the kernel) of the vortex in this context is now the minimum of the 3-round character (i.e. leading to both q_0 and q_1 in the kernel) of the vortex and the characters of the three additional equations. The probability that the 4-round character is larger than 2^ℓ is approximately $2^{-4(\ell+1)}$. It turns out that for our choice of π and ρ, 8 of the 50 candidate vortices have 4-round character 2 and the others have all 4-round character 1.

The conditions on the (x, y) coordinates imply that only vortices are suited that have an even number of active points in each sheet, each plane and each rising 1-slope. This limits the number of suitable vortices of length 6 to 50, of length 8 to 300, of length 10 to 4180 and of length 12 to 53750. To illustrate this, let us now study the number of activity patterns in the

(x, y) coordinates of q_0 assuming there is only a single active bit in each lane. In total there are $2^{25} - 1$ nonzero patterns. If we impose the pattern to be in the CP-kernel, the parity of each sheet must be even, resulting in 5 independent linear equations. Hence there are $2^{20} - 1$ patterns in the kernel. Additionally requiring q_1 to be in the kernel imposes that the number of points in each plane of q_0 must be even. This adds 5 parity conditions. However, one is redundant with the ones due to q_0 as the total parity of the activity pattern over the state is even. Hence there are $2^{16} - 1$ such patterns. Additionally requiring q_2 to be in the kernel imposes that the number of points in each rising 1-slope of q_0 must be even. This adds again 5 new parity condition, with one of them redundant and reduces the number of possible patterns to $2^{12} - 1$. Since π runs through all directions, adding more rounds results in $2^8 - 1$, and $2^4 - 1$ and finally 0 patterns. It follows that the range of possible activity patterns shrinks exponentially as the number of rounds grows.

This is the main reason for choosing a π that runs through all axes in a single cycle. Consider a π that would map sheets to rising 1-slopes and rising 1-slopes back to sheets. For such a π there would be $2^{16} - 1$ activity patterns with q_0, q_1 and q_2 in the kernel. Moreover, this number would not decrease for more rounds and periodic χ-zero trails of low weight might appear.

When trying vortices with length above 6, the conditions on the z coordinates can be more involved. If in a particular sheet of q_2 the number of active points is 2, the condition is the same as for the case described above: their z coordinates should match. However, if there are 4, 6 or any even number of active points, there are several ways for them to be grouped in different columns. In general a character can be computed per sheet and the character of the complete structure is the minimum of all these characters. The character for a given sheet can be computed in a recursive way. The probability that an active sheet has character 1 is 1/2. For larger characters, the probability decreases faster with growing number of active bits in the character.

We have done tests for vortex lengths up to 14 and for constructions making use of two vortices totaling to about 1 million valid q_0 patterns. The vast majority have character 1, less than 13000 have character 2, 103 have character 4 and one has character 8. This last one is based on vortex of length 8 and it results in a 4-round trail with weight 512 in KECCAK-f[1600].

5.5.7 Truncated trails and differentials

Truncated trails deal with the propagation of activity patterns rather than difference patterns [42]. A partition of the state in sub-blocks is defined where the activity patterns describe whether a sub-block has no active bits (passive or 0) or has at least one active bit (active or 1). The structure of the state in KECCAK-f suggests several bundlings. In a first order, one may take rows, columns, lanes, sheets, planes or slices as sub-blocks. We have gone through an exercise of attempting this but got stuck very soon for each of the choices. The problem is that for every choice, at least one of the step mappings completely tears apart the sub-blocks. We have also considered hybrid state definitions, such as the combination of row activities with column parities. However, in the cases that could be interesting, i.e., states with low weight (with respect to the truncation considered), this soon lead to the full specification of the difference.

In [40] truncated cryptanalysis was applied to RADIOGATÚN [6], where the truncation was defined by a linear subspaces of the word vectors. In the attack it made sense as part of the RADIOGATÚN round function is linear. In KECCAK-f the round function is uniformly non-linear and we do not believe that this approach can work.

5.5.8 Other group operations

We have considered differential and linear cryptanalysis while assuming the bitwise addition as the group operation. One may equivalently consider differential and linear properties with respect to a wide range of other group operations that can be defined on the state. However, for any other choice than the bitwise addition, θ becomes a nonlinear function and for most choices also ι, π and ρ become nonlinear. We do not expect this to lead to better results.

5.5.9 Differential and linear cryptanalysis variants

There are many attacks that use elements from differential cryptanalysis and/or linear cryptanalysis. Most are applied to block ciphers to extract the key. We have considered a number of techniques:

- Higher-order differentials [42]: we believe that due to the high average diffusion it is very difficult to construct higher-order differentials of any significance for KECCAK-f.
- Impossible differentials [69]: we expect the KECCAK-f permutations to behave as random permutations. If so, the cardinality of differentials has a Poisson distribution with $\lambda = 1/2$ [28] and hence about 60 % of the differentials in KECCAK-f will have cardinality 0, and so are impossible. However, given a differential (a, b), it is a priori hard to predict whether it is impossible. Settings in which one could exploit impossible differentials are keyed modes, where part of the input is fixed and unknown. In this case one would need truncated impossible differentials. If the number of rounds is sufficient to avoid low-weight differential trails, we believe this can pose no problem.
- Differential-linear attacks [47]: in these attacks one concatenates a differential over a number of rounds and a correlation over a number of subsequent rounds. We think that for reduced-round versions of KECCAK-f differential-linear distinguishers are a candidate for the most powerful structural distinguisher. The required number of pairs is of the order $\mathrm{DP}^{-2}\mathrm{LP}^{-2}$ with DP the differential probability of the distinguisher's differential and LP the square of the distinguisher's correlation. If we assume the differentials is dominated by a single low-weight differential trail, we have $\mathrm{DP} \approx 2^{-\mathrm{w}(Q_d)}$. Additionally, if we assume the correlation is dominated by a single low-weight linear trail, we have $\mathrm{DP} \approx 2^{-\mathrm{w}(Q_l)}$. This gives for the number of required pairs: $2^{2(\mathrm{w}(Q_d)+\mathrm{w}(Q_l))}$. The number of required pairs to exploit a trail in a simple differential or linear attack is of the order $2^{\mathrm{w}(Q)}$. Hence, over a number of rounds, the differential-linear distinguisher is more powerful than a simple differential or linear distinguisher if $\mathrm{w}(Q_d) + \mathrm{w}(Q_l) < \mathrm{w}(Q)/2$. Where Q is a trail over all rounds, Q_d a trail of the first n rounds and Q_l a trail over the remaining rounds. As we expect in the KECCAK-f variants with large width and a low number of rounds, the minimum trail weight tends to grow exponentially, and the chaining of two half-length trails is favored over a single full-length trail.
- (Amplified) Boomerang [66, 38] and rectangle attacks [11]: These attacks chain (sets of) differentials over a small number of rounds to construct distinguishers over a larger

	DC		**LC**	
Number of rounds	$w = 1$	$w = 2$	$w = 1$	$w = 2$
2	8	8	8	8
3	16	18	16	16
4	23	29	24	30
5	30	42	30	?
6	37	54	38	?

Table 5.2: Lower bounds for trail weights

number of rounds. These are also likely candidates for good structural distinguishers, for the same reason as differential-linear ones.

- Integral cryptanalysis (Square attacks) [24]: this type of cryptanalysis lends itself very well to ciphers that treat the state in blocks. It was applied to bit-oriented ciphers in [71]. Based on the findings of that paper we estimate that it will only work on reduced-round versions of KECCAK-f with three to four rounds.

In this section we have limited ourselves to the construction of structural distinguishers. We have not discussed how these distinguishers can be used to attack the sponge function making use of the permutation.

5.5.10 Bounds for symmetric trails

We have performed a pruned tree search for low-weight trails for widths 25 and 50 that result in the lower bounds in Table 5.2. For $b = 25$, five rounds are sufficient to have no trails with weight below the width. For $b = 50$, six rounds are sufficient to have no differential trails with weight below the width. It can be observed that as the number of rounds grows, the difference between width 25 and width 50 grows. We expect this effect to be similar for larger widths.

Note that thanks to the Matryoshka structure, these bounds imply lower bounds for symmetric trails for all larger widths. More specifically, a trail for $w = 1$ with weight W corresponds with a trail for $w = 2^\ell$ with weight $wW = 2^\ell W$.

5.6 Solving CICO problems

There are several approaches to solving a CICO problem for KECCAK-f. The most straightforward way is to use the KECCAK-f specification to construct a set of algebraic equations in a number of unknowns that represents the CICO problem and try to solve it. Thanks to the simple algebraic structure of KECCAK-f, constructing the algebraic equations is straightforward. A single instance of KECCAK-f results in $(n_\mathrm{r} - 1)b$ intermediate variables and about as many equations. Each equation has algebraic degree 2 and involves about 31 variables. Solving these sets of equations is however not an easy task. This is even the case for the toy version KECCAK-$f[25]$ with lane length $w = 1$.

5.7 Strength in keyed mode

In keyed modes we must consider attack scenario's such as explained in Section 4.7.5. Here we see two main approaches to cryptanalysis. The first one is the exploitation of structural distinguishers and the second one is an algebraic approach, similar to the one presented in Section 5.6. A possible third approach is the intelligent combination of exploiting a structural distinguisher and algebraic techniques. In our opinion, the strength in keyed modes depends on the absence of good structural distinguishers and the difficulty of algebraically solving sets of equations.

5.8 Symmetry weaknesses

Symmetry in the state could lead to properties similar to the complementation property. Symmetry between the rounds could lead to slide attacks. We believe that the asymmetry introduced by ι is sufficient to remove all exploitable symmetry from KECCAK-f.

5.9 Experimental data

The KECCAK-f permutations should have no propagation properties significantly different from that of a random permutation. For the smallest KECCAK-f version, KECCAK-$f[25]$, it is possible to experimentally reconstruct significant parts of the distribution of certain properties. In this section, we report on the results of such experiments.

As a reference, we have generated a pseudorandom permutation operating on 25 bits using a simple algorithm from [44] taking input from a pseudorandom number generator based on a cipher that is remote from KECCAK and its inventors: RC6 [58]. We denote this permutation by the term Perm-R.

5.9.1 Differential probability distributions

We have investigated the distribution of the cardinality of differentials for KECCAK-$f[25]$, several reduced-round versions of KECCAK-$f[25]$ and of Perm-R. For these permutations, we have computed the cardinalities of 2^{41} differentials of type (a', b') where a' ranges over 2^{16} different non-zero input patterns and b' over all 2^{25} patterns. For Perm-R we just tested the first (when considered as an integer) 2^{16} non-zero input patterns. For the KECCAK-$f[25]$ variants we tested as input patterns the first 2^{16} non-zero entries in the lookup table of Perm-R.

In a random permutation the cardinality of differentials has a Poisson distribution with $\lambda = 1/2$. This is studied and described among others in [28]. Moreover, [28] also determines the distribution of the maximum cardinality of a large number of differentials over a random permutation. According to [28, Section 5.2], the expected value of the maximum cardinality over the 2^{41} samples is 12 and the expected value of the maximum cardinality over all $2^{50} - 1$ non-trivial differentials (a', b') is 14.

We provide in a sequence of diagrams the histograms obtained from these samplings, indicating the envelope of the theoretical Poisson distribution for a random permutation as a continuous line and the results of the measurements as diamond-shaped dots. We have adopted a logarithmic scale in the y axis to make the deviations stand out as much as possible.

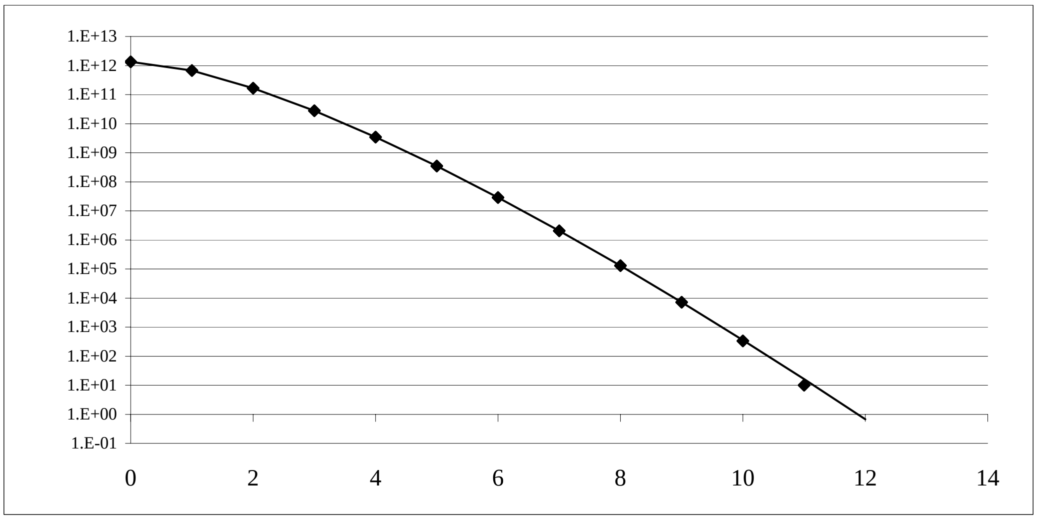

Figure 5.5: Cardinality histogram of sampling of Perm-R

Figure 5.5 shows that Perm-R exhibits a distribution that follows quite closely the theoretically predicted one. The maximum observed cardinality is 11.

Figure 5.6 shows the distribution for the two-round version of KECCAK-f[25]: the distribution deviates significantly from the theoretical Poisson distribution. Note that here also the x axis has a logarithmic scale. The largest cardinality encountered is 32768. It turns out that the pairs of this differential are all in a single trail with weight 9. The number of pairs is equal to the number of pairs predicted by the weight: $2^{24-9} = 2^{15}$. Note that there are 2-round trails with weight 8 (see Table 5.2) but apparently no such trail was encountered in our sampling.

Figure 5.7 shows the distribution for the three-round version of KECCAK-f[25]. The deviation from the theoretical Poisson distribution is smaller. The largest cardinality encountered is now 146. The pairs of this differential are all in a single trail with weight 17. The number of pairs is slightly higher than the number of pairs predicted by the weight: $2^{24-17} = 2^7$. The 3-round trails with weight 16 (see Table 5.2) were not encountered in our sampling.

Figure 5.8 shows the distribution for the four-round version of KECCAK-f[25]. The sampling does no longer allow to distinguish the distribution from that of a random permutation. The largest cardinality encountered is now 12. The pairs of this differential are in 12 different trails with weight ranging from 56 to 64. For the 4-round trails with weight 23 (see Table 5.2) it is not clear whether they were encountered in our sampling: the expected number of pairs is only 2 and this may have gone unnoticed.

Finally, Figure 5.9 shows the distribution for the 12-round version of KECCAK-f[25]. As expected, the distribution is typical of a random permutation. The maximum cardinality observed is 12.

5.9.2 Correlation distributions

We have investigated the distribution of the correlations for KECCAK-f[25], several reduced-round versions of KECCAK-f[25] and Perm-R. For these permutations, we have computed the

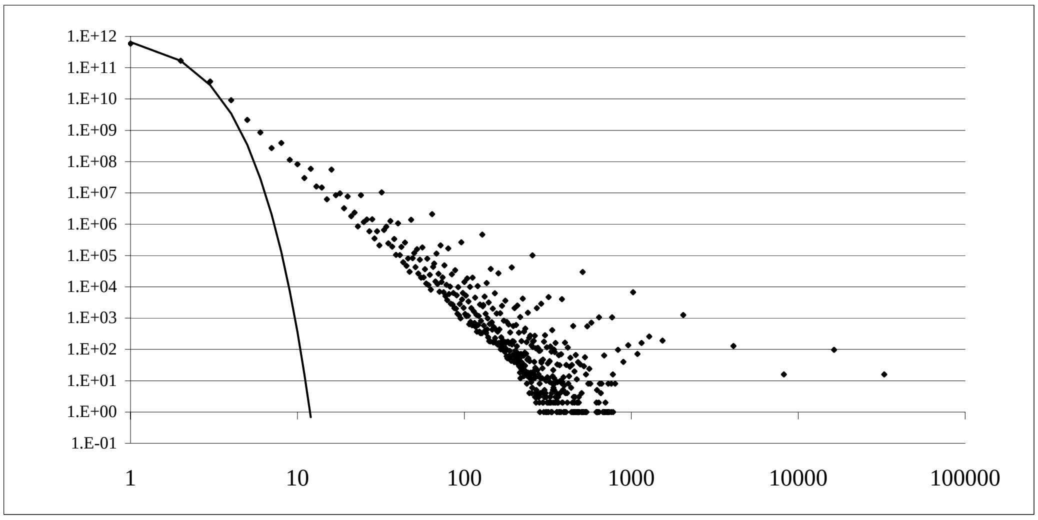

Figure 5.6: Cardinality histogram of sampling of 2-round version of KECCAK-$f[25]$

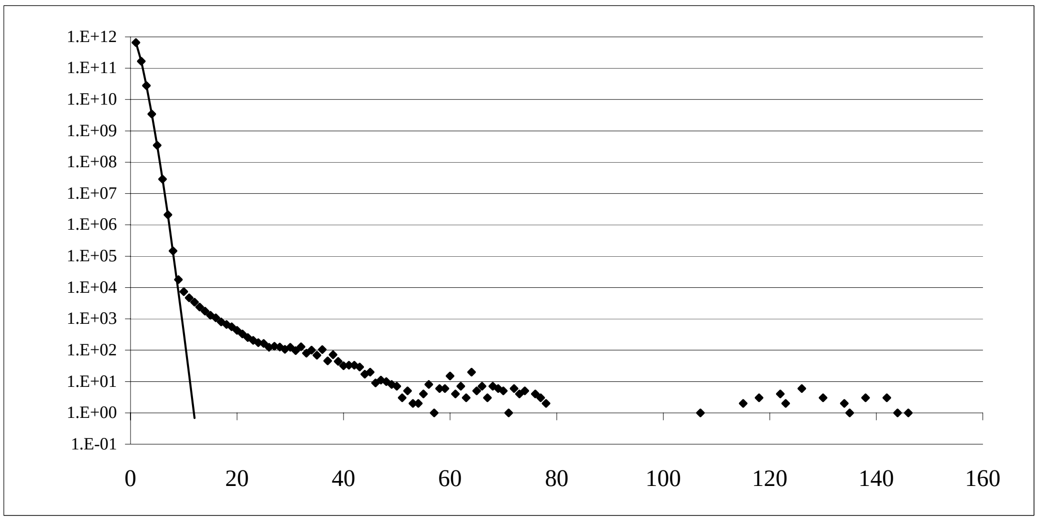

Figure 5.7: Cardinality histogram of sampling of 3-round version of KECCAK-$f[25]$

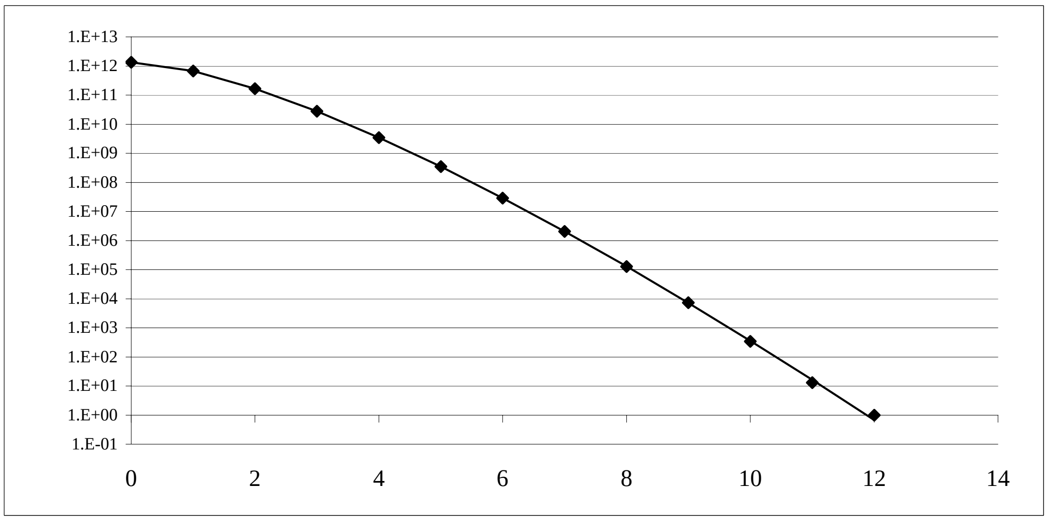

Figure 5.8: Cardinality histogram of sampling of 4-round version of KECCAK-$f[25]$

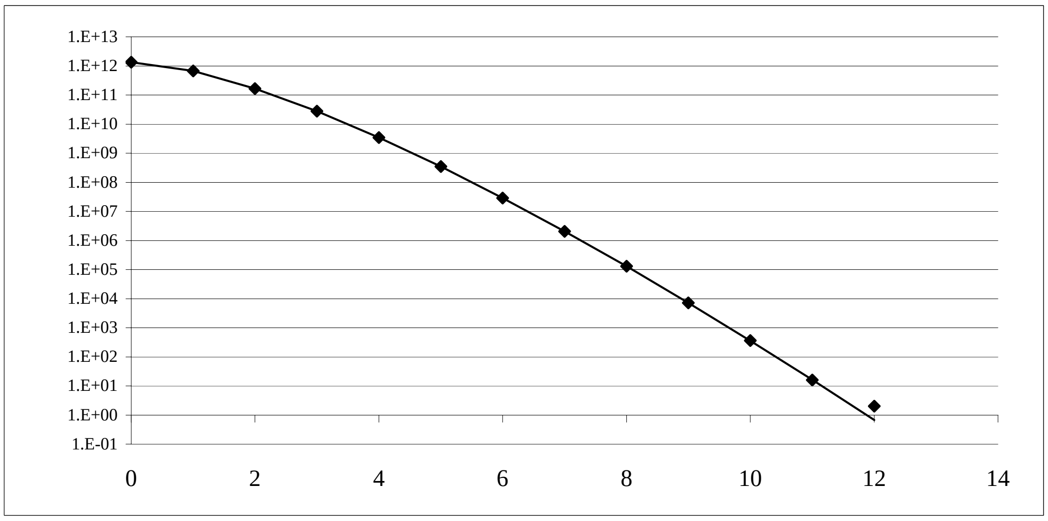

Figure 5.9: Cardinality histogram of sampling of KECCAK-$f[25]$

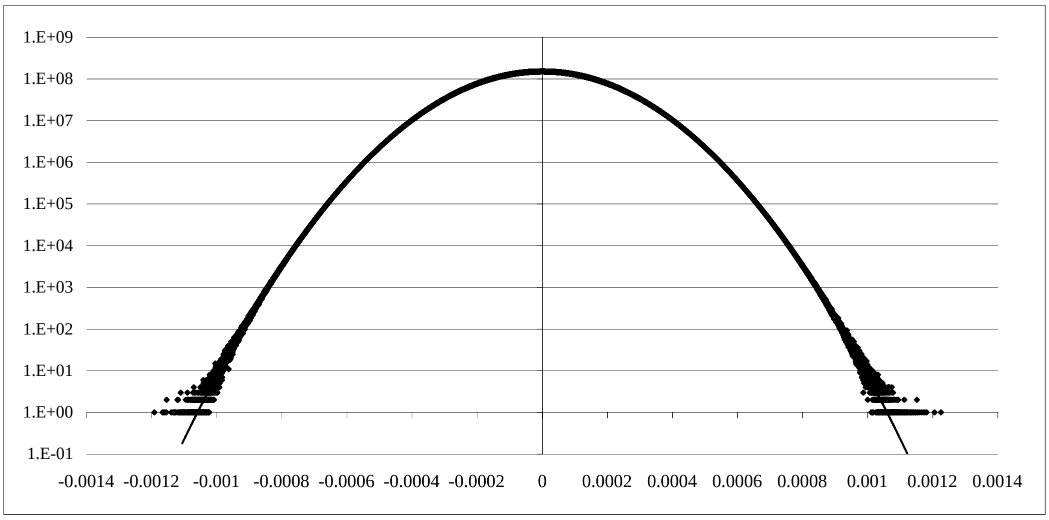

Figure 5.10: Correlation histogram of sampling of Perm-R

correlations of 2^{39} couples (v, u) where u ranges over 2^{14} different non-zero output selection patterns and v over all 2^{25} patterns. For Perm-R we just tested the first (when considered as an integer) 2^{14} non-zero output selection patterns. For the Keccak-$f[25]$ variants we tested as output selection patterns the first 2^{14} non-zero entries in the lookup table of Perm-R.

In a random permutation with width b the input-output correlations have a discrete distribution enveloped by a normal distribution with $\sigma^2 = 2^{-b}$. This is studied and described in [28]. Moreover, [28] also determines the distribution of the maximum correlation magnitude of a large number of couples (v, u) over a random permutation. According to [28, Section 5.4], the expected value of the maximum correlation magnitude over the 2^{39} samples is 0.00123 and the expected value of the maximum correlation magnitude over all $2^{50} - 1$ non-trivial correlations (v, u) is 0.0017.

We provide in a sequence of diagrams the histograms obtained from these samplings, indicating the envelope of the theoretical normal distribution for a random permutation as a continuous line and the results of the measurements as diamond-shaped dots. We have adopted a logarithmic scale in the y axis to make the deviations stand out as much as possible.

Figure 5.10 shows that Perm-R exhibits a distribution that follows quite closely the normal envelope. At its tails the experimental distribution exhibits its discrete nature. Because it is a permutation, the correlation can only be non-zero in values that are a multiple of 2^{2-b}. For a given correlation value c that is a multiple of 2^{2-b}, the a priori distribution of the corresponding value in the cryptogram is a Poisson distribution with λ given by the value of the normal envelope in that point. The largest correlation magnitude observed is 0.001226, quite close to the theoretically predicted value.

Figure 5.11 shows the distribution for the two-round version of Keccak-$f[25]$: the distribution deviates significantly from the theoretical normal envelope. Additionally, it is zero for all values that are not a multiple of 2^{-15} (rather than 2^{-23}). This is due to the fact that the Boolean component functions of Keccak-$f[25]$ have only reached degree 4 after two rounds (see Table 5.3), rather than full degree 24. The largest correlation magnitude encountered

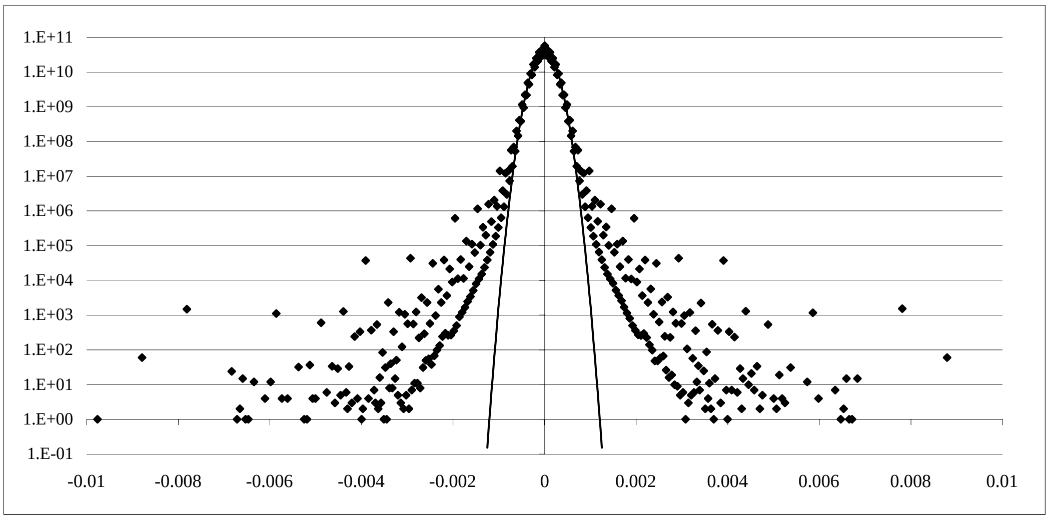

Figure 5.11: Correlation histogram of sampling of 2-round version of Keccak-f[25]

is 0.03125 (outside the scale of the figure). This is the correlation magnitude 2^{-5} one would obtain by a single linear trail with weight 10. The 2-round linear trails with weight 8 (see Table 5.2) were apparently not encountered in our sampling.

Figure 5.12 shows the distribution for the three-round version of Keccak-f[25]: the deviation from the theoretical normal envelope becomes smaller. This distribution is zero for all values that are not a multiple of 2^{-18} due to the fact that the Boolean component functions of Keccak-f[25] have only reached degree 8 after three rounds (see Table 5.3). The largest correlation magnitude encountered is 0.003479. This is a correlation magnitude that cannot be obtained by a single linear trail. The 3-round linear trails with weight 16 (see Table 5.2) would give correlation magnitude $2^{-8} \approx 0.0039$. It is quite possible that the observed correlation value is the sum of the (signed) correlation contributions of some trails, including one with weight 16 and some with smaller weight.

Figure 5.13 shows the distribution for the four-round version of Keccak-f[25]. The shape of the distribution and the maximum values do no longer allow to distinguish the distribution from that of a random permutation. The largest correlation magnitude encountered is 0.001196. However, this distribution differs from that of a random permutation because it is zero for all values that are not a multiple of 2^{-20} due to the fact that the Boolean component functions of Keccak-f[25] have only reached degree 16 after four rounds (see Table 5.3). After 5 rounds the distribution is zero for values that are not a multiple of 2^{-22} and only after 6 rounds this becomes 2^{-23}. This is consistent with Table 5.3.

Finally, Figure 5.14 shows the distribution for the 12-round version of Keccak-f[25]. As expected, the distribution is typical of a random permutation. The maximum correlation magnitude observed is 0.001226.

5.9.3 Algebraic normal form experiments

There are several ways to describe Keccak-f algebraically. One could compute the algebraic normal form (ANF, see Section 4.2.3) with elements in GF(2), GF(2^5), GF(2^{25}) or GF(2^w),

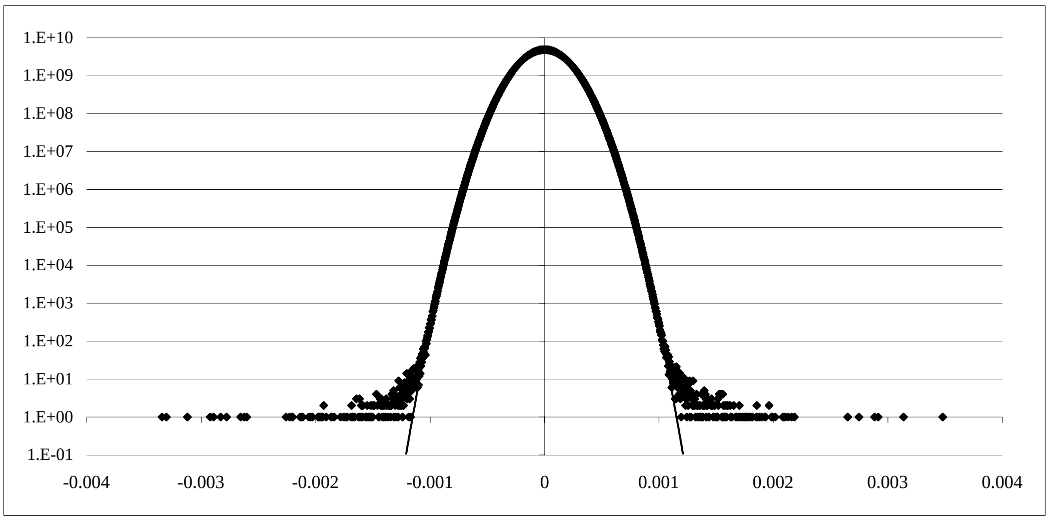

Figure 5.12: Correlation histogram of sampling of 3-round version of KECCAK-f[25]

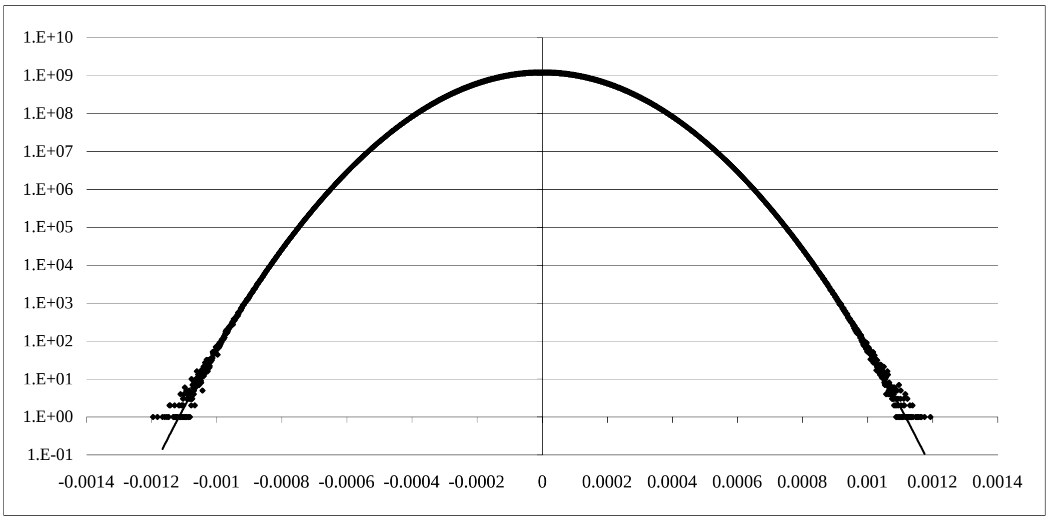

Figure 5.13: Correlation histogram of sampling of 4-round version of KECCAK-f[25]

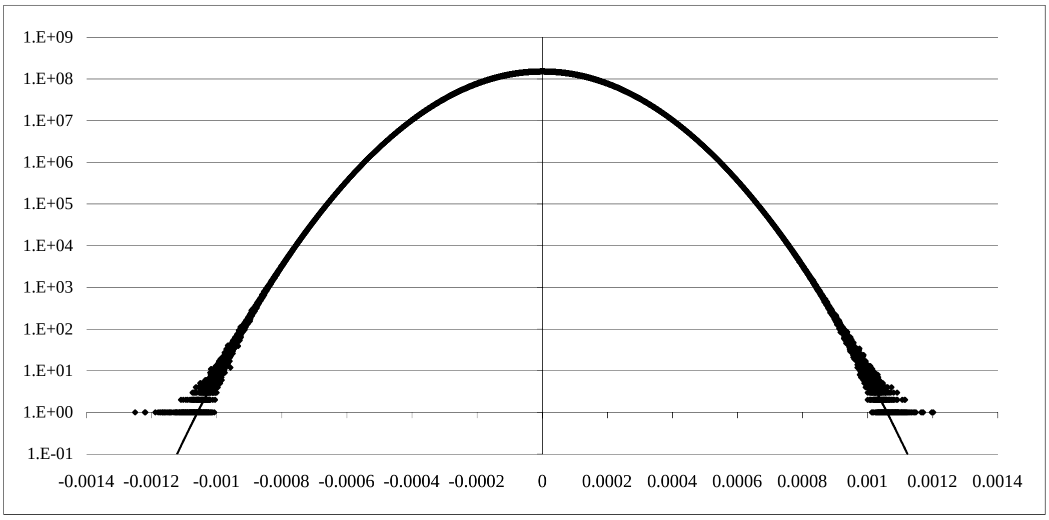

Figure 5.14: Correlation histogram of sampling of KECCAK-$f[25]$

but given the bit-oriented structure and matching θ, ρ, π, ι and χ as operations in GF(2), the ANF in GF(2) seems like a natural way to represent the KECCAK-f permutation.

For instance, one could take the rows as variables in GF(2^5). This way, the χ operation applies independently per variable. However, the other operations will have a complex expression.

5.9.3.1 Algebraic normal form of KECCAK-$f[25]$

We performed a statistical test based on the ANF of KECCAK-$f[25]$ and its inverse in GF(2), varying the number of rounds from 1 to 12. The test consists in varying the 25 bits of input and counting the number of monomials of degree d of all 25 output bits. The statistical test is performed per degree independently. The number of monomials of degree d should be present in a ratio of about one half. The test fails when the observed number of monomials is more than two standard deviations away from the theoretical average. We look for the highest degree that passes the test.

Note that the (inverse) round function has only degree two (degree three) and thus no monomial of degree higher than 2^i (3^i) can appear after after i rounds.

The results are displayed in Table 5.3. Starting from 7 rounds, all monomials up to order 24 exist and appear with a fraction close to one half. Since KECCAK-$f[25]$ is a permutation, the monomial of order 25 does not appear.

5.9.3.2 Algebraic normal form of KECCAK-$f[1600]$

We performed two statistical tests based on the ANF of KECCAK-$f[1600]$ and its inverse in GF(2), varying the number of rounds. Since we obviously cannot vary all the 1600 input bits, we varied a subset of them. In a first test, we varied the 25 bits of slice $z = 0$. In a second test, we varied the 25 bits of coordinates $(0, 0, z)$ for $0 \leq z < 25$. In both cases, the other 1575 bits are set to zero.

Rounds	**Maximum degree to pass test**		**Monomials exist up to degree**	
	f	f^{-1}	f	f^{-1}
1	(none)	(none)	2	3
2	1	2	4	9
3	6	10	8	17
4	14	18	16	21
5	22	23	22	23
6	23	23	24	24
7-12	24	24	24	24

Table 5.3: The ANF statistical test on KECCAK-f[25] and its inverse

Rounds	**Maximum degree to pass test**		**Monomials exist up to degree**	
	f	f^{-1}	f	f^{-1}
1	(none)	(none)	2	3
2	(none)	3	3	9
3	(none)	10	5	17
4	5	25	9	25
5	16	25	17	25
6-18	25	25	25	25

Table 5.4: The ANF statistical test on KECCAK-f[1600] and its inverse, varying slice $z = 0$

We counted the total number of monomials of degree d that appear in all 1600 output bits. The statistical test is performed per degree independently, in the same was as for KECCAK-f[25].

The results can be found in Tables 5.4 and 5.5. Extrapolating this to the full 1600-input-bit ANF and assuming that it starts with a good set of monomials up to degree 16 at round 5, like here, and then doubles for each additional round, we need at least 12 rounds to populate a good set of monomials up to degree 1599.

In all cases, increasing the number of rounds by one every time the width of the permutation doubles, as $n_{\mathrm{r}} = 12 + \ell$, compensates for the need to populate twice the monomial degrees while keeping extra rounds as a security margin.

5.9.4 Solving CICO problems algebraically

We did some attempts to solve a CICO problem algebraically using SAGE [1]. We wrote a program that generates the round equations of KECCAK-f for any width in a form that can be used in SAGE. We defined a CICO problem for KECCAK-f[25] where we fix the value of 16 bits at its input and 9 bits at its output and tried to solve that with the tools available in SAGE. We were able to solve instances of this problem for a reduced-round version KECCAK-f[25] with 2 rounds in a matter of minutes. Similar trials with a 3-round version caused SAGE to run out of memory. We have made available KECCAKTOOLS [10], a public software for

Rounds	**Maximum degree to pass test**		**Monomials exist up to degree**	
	f	f^{-1}	f	f^{-1}
1	(none)	(none)	2	1
2	(none)	3	4	3
3	(none)	8	8	9
4	8	25	15	25
5	25	25	25	25
6	24	25	25	25
7-18	25	25	25	25

Table 5.5: The ANF statistical test on KECCAK-f[1600] and its inverse, varying 25 bits of lane $x = y = 0$

generating round equations so that researchers can use SAGE or other computer algebra tools to attack KECCAK-f.

5.9.5 Cycle distributions

We have determined the cycle structure of KECCAK-f[25] and all its reduced-round versions. Table 5.6 lists all cycles for KECCAK-f[25] and Table 5.7 the number of cycles for all reduced-round versions. For a random permutation, the expected value of the number of cycles is $\ln(2^{25}) = 25 \ln 2 \approx 17.3$. The average of Table 5.7 is 16.3.

It can be observed that KECCAK-f[25] and all its reduced-round versions have an even number of cycles. For a permutation operating on a domain with an even number of elements, an even number of cycles implies that it is an even permutation [67], hence they are all even permutations. Actually, it is easy to demonstrate that all members of the KECCAK-f family are even permutations. We do however not think this property can be exploited in an attack or to build a usable distinguisher.

The members of the KECCAK-f family are even permutations because the composition of two even permutation is an even permutation and that all step mappings of KECCAK-f are even permutations. We cite here a number of arguments we found in [67, Lemma 2]:

- The mappings θ, π and ρ are linear. In fact all invertible linear transformations over $\mathbb{Z}_2^b$ with $b > 2$ are even permutations. This follows from the fact that each invertible binary matrix can be obtained from the identity matrix by elementary row transformations (binary addition of one row to another row), and that these elementary row transformations (considered as linear mappings) are permutations with 2^{b-1} fixed points and 2^{b-2} cycles of length 2.
- The mapping ι consists of the addition of a constant. Addition of a constant in $\mathbb{Z}_2^b$ is the identity mapping if the constant is all-zero and has 2^{b-1} cycles of length 2 if the constant is not all-zero.
- The mapping χ is an even permutation because it can be represented as a concatenation of $5w$ permutations that carry out the χ mapping for one row and leave the other $5w-1$ rows fixed. The cycle representation of each such permutation contains a number of cycles that is a multiple of 2^{25w-5} and hence even.

18447749	147821	168	12
13104259	40365	27	3
1811878	2134	14	2

Table 5.6: Cycle lengths in KECCAK-f[25]

rounds	cycles	rounds	cycles	rounds	cycles
1	14	5	18	9	14
2	12	6	20	10	20
3	16	7	18	11	18
4	16	8	18	12	12

Table 5.7: Number of cycles in reduced-round versions of KECCAK-f[25]

Chapter 6

Usage

This chapter discusses the KECCAK sponge functions from a users' point of view.

6.1 Usage scenario's for a sponge function

6.1.1 Random-oracle interface

A sponge function has the same input and output interface as a random oracle: It accepts an arbitrarily-long input message and produces an infinite output string that can be truncated at the desired length. Unlike some other constructions, a sponge function does not have a so called initial value (IV) that can be used as an additional input. Instead, any additional input, such as a key or a diversifier, can be prepended to the input message, as one would do with a random oracle. See also Section 6.3 for a related discussion.

6.1.2 Linking to the security claim

Basically, the security claim in [8, Section 3] specifies that any attacks on a member of the KECCAK family should have a complexity of order $2^{c/2}$ calls to KECCAK-f, unless easier on a random oracle.

For the first four KECCAK candidates with fixed digest length, the output length n always satisfies $n \leq c/2$. This means that using KECCAK as a hash function provides collision resistance of $2^{n/2}$, (second) preimage resistance of 2^n and resistance to length-extension. Furthermore, for any fixed subset of $m < n$ output bits, the same complexities apply with m replacing n.

For the fifth candidate KECCAK[] with its arbitrarily-long output mode, the idea is pretty much the same, except that, for any attacks that would require more than $2^{c/2} = 2^{288}$ on a random oracle, the attack may work on KECCAK[] with a complexity of $2^{c/2} = 2^{288}$.

With a random oracle, one can construct a pseudo-random function (PRF) $F_k(m)$ by prepending the message m with a key k, i.e., $F_k(m) = \mathcal{RO}(k||m)$. In such a case, the function behaves as a random function to anyone not knowing the key k but having access to the same random oracle (see also Section 3.1.5). As a consequence of the security claim, the same construction can be used with a KECCAK sponge function and the same security can be expected when the adversary does not have access to a complexity of order higher than $2^{c/2}$.

Functionality	**Expression**	**Input**	**Output**
n-bit hash function	$h = H(m)$	m	$\lfloor z \rfloor_n$
n-bit randomized hash function	$h = H_r(m)$	$r\|\|m$	$\lfloor z \rfloor_n$
n-bit hash function instance differentiation	$h = H_d(m)$	$d\|\|m$	$\lfloor z \rfloor_n$
n-bit MAC function	$t = \mathrm{MAC}(k, [\mathrm{IV},]m)$	$k\|\|\mathrm{IV}\|\|m$	$\lfloor z \rfloor_n$
Random-access stream cipher (n-bit block)	$z_i = f(k, \mathrm{IV}, i)$	$k\|\|\mathrm{IV}\|\|i$	$\lfloor z \rfloor_n$
Stream cipher	$z = f(k, \mathrm{IV})$	$k\|\|\mathrm{IV}$	as is
Mask generating and key derivation function	$\text{mask} = f(\text{seed}, l)$	seed	$\lfloor z \rfloor_l$
Deterministic random bit generator (DRBG)	$z = \mathrm{DRGB}(\text{seed})$	seed	as is
Slow n-bit one-way function	$h = H(m)$	m	$z_{N \ldots N+n-1}$
Tree and parallel hashing	see Section 6.4		

Table 6.1: Examples of usage scenario's for a sponge function

6.1.3 Examples of modes of use

In Table 6.1, we propose some possible modes of use of a sponge function.

The first five examples of Table 6.1 can be applied with any member of the KECCAK family, while the last five, as such, require the arbitrarily-long output mode of KECCAK (although less natural constructions can be found on top of the fixed digest length candidates).

An n-bit hash function can trivially be implemented using a sponge function, e.g., $H(m) = \lfloor \text{KECCAK}[](m) \rfloor_n$. If the hash function is to be used in the context of randomized hashing, a random value (i.e., the salt) can be prepended to the message, e.g., $H_r(m) = \lfloor \text{KECCAK}[](r||m) \rfloor_n$. Domain separation using the same prepending idea applies if one needs to simulate independent different hash function instances (hash function instance differentiation, see also Section 6.3) and to compute a message authentication code (MAC).

The random-access stream cipher works similarly to the SALSA20 family of stream ciphers [4]: It takes as input a key, a nonce and a block index and produces a block of key stream. It can be used with the four fixed digest length variants of KECCAK, with n the digest length. It can also be used with the arbitrarily-long output mode of KECCAK[], in which case producing blocks of $n = r$ bits of key stream is most efficient per application of KECCAK-f.

A sponge function can also be used as a stream cipher. One can input the key and some initial value and then get key stream in the squeezing phase.

A mask generating function, a key derivation function or a random bit generator can be constructed with a sponge function by absorbing the seed data and then producing the desired number of bits in the squeezing phase. Note that a deterministic random bit generator may support the function of re-seeding, i.e., injecting new seed material without throwing away its current state. This can be done by squeezing b final output bits, and then starting a new sponge feeding it with these final output bits followed by the new seed.

Finally, a slow n-bit one-way function can be built by defining as output the output bits z_N to z_{N+n-1} (and thus discarding the first N output bits) rather than its first n bits. Slow one-way functions are useful as so-called password-based key derivation functions, where the relative high computation time protects against password guessing. The function can be made arbitrarily slow by increasing N: taking $N = 10^6 r$ implies that KECCAK-f must be called a million times for a single call to the function. Note that increasing N does not result in entropy loss as KECCAK-f is a permutation.

6.2 Backward compatibility with old standards

6.2.1 Input block length and output length

Several standards that make use of a hash function assume it has an input block length and a fixed output length. A sponge function supports inputs of any length and returns an output of arbitrary length. When a sponge function is used in those cases, an input block length and an output length must be chosen. We distinguish two cases.

- For the four SHA-3 candidates where the digest length is fixed, the input block length is assumed to be the bitrate r and the output length is the digest length of the candidate $n \in \{224, 256, 384, 512\}$.
- For the fifth SHA-3 candidate KECCAK[], the output length n must be explicitly chosen to fit a particular standard. Since the input block length is usually assumed to be greater than or equal to the output length, the input block length can be taken as an integer multiple of the bitrate, mr, to satisfy this constraint.

6.2.2 Initial value

Some constructions that make use of hash functions assume the existence of a so-called initial value (IV) and use this as additional input. In the sponge construction the root state could be considered as such an IV. However, for the security of the sponge construction it is crucial that the root state is fixed and cannot be manipulated by the adversary. If KECCAK sponge functions are used in constructions that require it to have an initial value as supplementary input, e.g., as in NMAC [3], this initial value shall just be pre-pended to the regular input.

6.2.3 HMAC

HMAC [3, 56] is fully specified in terms of a hash function, so it can be applied as such using one of the KECCAK candidates. It is parameterized by an input block length and an output length, which we propose to choose as in Section 6.2.1 above.

Apart from length extension attacks, the security of HMAC comes essentially from the security of its inner hash. The inner hash is obtained by prepending the message with the key, which gives a secure MAC. The outer hash prepends the inner MAC with the key (but padded differently), so again giving a secure MAC. (Of course, it is also possible to use the generic MAC construction given in Section 6.1, which requires only one application of the sponge function.)

From the security claim in [8, Section 3], a PRF constructed using HMAC shall resist a distinguishing attack that requires much fewer than $2^{c/2}$ queries and significantly less computation than a preimage attack.

6.2.4 NIST and other relevant standards

The following standards are based either generically on a hash function or on HMAC. In all cases, at least one of the KECCAK candidates can readily be used as the required hash function or via HMAC.

- IEEE P1393 [36] requires a hash function for a key derivation function (X9.42) and a mask generating function (MGF-hash). (Note that the MGF-hash construction could be advantageously replaced by the arbitrarily-long output mode of Keccak[].)
- PKCS #1 [46] also requires a hash function for a mask generating function (MGF1).
- The key derivation functions in NIST SP 800-108 [57] rely on HMAC.
- The key derivation functions in NIST SP 800-56a [53] are generically based on a hash function.
- The digital signature standard (DSS) [50] makes use of a hash function with output size of 160, 224 or 256 bits. Output truncation is permitted so any of the five Keccak candidates can be chosen to produce the 160 bits of output.
- In the randomized hashing digital signatures of NIST SP 800-106 [52], the message is randomized prior to hashing, so this is independent of the hash function used. (With a sponge function, this can also be done by prepending the random value to the message.)
- The deterministic random bit generation (DRBG) in NIST SP 800-90 [54] is based on either a hash function or on HMAC. (Maybe the arbitrarily-long output mode of Keccak[] could be used for such an application.)

6.3 Input formatting and diversification

In a sponge function, the input is like a *white page*: It does not impose any specific structure to it. Some SHA-3 submissions (e.g., Skein [29]) propose a structured way to add optional inputs (e.g., key, nonce, personalization data) in addition to the main input message.

For Keccak, we do not wish to impose a way data would be structured. We prefer to keep the input as a white page and let anyone build upon it. Instead, we propose in this section a simple convention that allows anyone to impose his/her own format. Note that this convention could be used with any other hash function.

In the Keccak specifications [8], only three values have been assigned to the diversifier d (out of 256) when $r = 1024$ and two values when $r = 512$. There are thus many available values of d to create diversified sponge functions, but clearly not enough for anyone to choose his/her own.

The idea is to prefix the input with a namespace name. The owner of the namespace can then define the format of any input data, appended to the namespace name. To make this construction distinct from the five candidates defined in [8], we propose to assign $d = 1$ in this case, for any valid r. More specifically, we propose the namespace name to be a uniform resource identifier (URI) [32], similarly to what is done for XML [65]. The namespace name is encoded in UTF-8 [31] as a sequence of bytes, followed by the byte 0^8:

$$\text{KeccakNS}[r, c, \text{ns}](\text{data}) \triangleq \text{Keccak}[r, c, d = 1](\text{UTF8}(\text{ns})||0^8||\,\text{encode}_{\text{ns}}(\text{data})),$$

where $\text{encode}_{\text{ns}}$ is a function defined by the owner of the namespace ns. This allows domain separation: Two inputs, formatted using different namespaced conventions, will thus always be different.

For efficiency reasons, the namespace owner may design $\text{encode}_{\text{ns}}$ to put fixed bytes after the encoded namespace name and before the variable data, so as to create a constant prefix of r bits (or a multiple of r bits). This way, the state obtained after absorbing the constant prefix can be precomputed once for all.

Using a specific namespace also implies how the output of the sponge function is used. In the Keccak specifications [8], the four fixed output length candidates are diversified using d. Here we propose an additional possibility, where the namespace owner can decide what is the output length, if not arbitrarily long, or in which way the desired output length is encoded.

6.4 Parallel and tree hashing

Tree hashing (see, e.g., [49, 59]) can be used to speed up the computation of a hash function by taking advantage of parallelism in modern architectures. It can be performed on top of many hash function constructions, including sponge functions. In this section, we propose a tree hashing scheme, called KeccakTree, which explicitly uses Keccak but which could also be implemented on top of another hash function. In addition, this scheme can be seen as an example of an application of sponge functions. It does not exclude variants or other applications: By basing KeccakTree on KeccakNS, other tree hashing schemes can be defined using different namespaces.

In a nutshell, the construction works as follows. Consider a rooted tree, with internal nodes and leaves. The input message is cut into blocks, which are spread onto the leaves. Each leaf is then hashed, producing I bits of output, for a given value I. An internal node gathers the output values of its (ordered) sons, concatenates them and hashes them. This process is repeated recursively until the root node is reached. The output of the root node, also called *final node*, can be arbitrarily long.

Since the input message is arbitrarily long and a priori unknown, we have to define how the tree can grow or how a finite tree can accept a growing number of input blocks. In fact, we propose two options.

- The first option is *final node growing* (FNG). The degree of the final node grows as a function of the input message length, and the number of leaves then increases proportionally.
- The second option is *leaf interleaving* (LI), where the tree size and the number of leaves are fixed, but the message input blocks are interleaved onto the leaves.

For randomized and keyed hashing, it should be possible to prefix all the node inputs with the salt or key. To this purpose, the construction accepts such a prefix.

6.4.1 Definitions

The input of the scheme are two binary strings: the prefix (key or salt) P (from 0 to 2040 bits) and the input message M. Its tree parameters, collectively denoted A, are the following:

- the tree growing mode $G \in \{\text{LI}, \text{FNG}\}$;
- the height H of the tree;
- the degree D of the nodes;

- the leaf block size B.

When $G = \text{LI}$, the tree is a balanced rooted tree of height H. All the internal nodes have degree D. When $G = \text{FNG}$, the final node has variable degree (as a function of the input message length) and all other internal nodes have degree D.

For all nodes, the scheme uses the sponge function defined by

$$\text{K}[r, c] \triangleq \text{KECCAKNS}[r, c, \text{ns} = \text{“}\texttt{http://keccak.noekeon.org/tree/}\text{”}].$$

Its input is composed of the prefix P, a partial input message $m \in \{0,1\}^*$, a flag $v \in \{\text{final}, \text{nonfinal}\}$ and the scheme parameters A when $v = \text{final}$:

$$\text{K}[r, c](P, m, \text{nonfinal}) \quad \text{or} \quad \text{K}[r, c](P, m, \text{final}, A).$$

For simplicity, we omit the fixed parameters r and c in the sequel. The input arguments are encoded into a binary string according to Algorithm 6. The prefix length $|P|$ must be an integral number of bytes and such that $|P|/8 \in [0 \dots 255]$. The possible values of A are constrained by $H, D \in [0 \dots 255]$, B is a multiple of 64 and $B/64 \in [1 \dots 2^{16} - 1]$. In addition, $H \geq 1$ when $G = \text{FNG}$.

Algorithm 6 encode$_{\texttt{http://keccak.noekeon.org/tree/}}(P, m, v, A)$

```
Input $P, m, v, A$
Let $M = \text{enc}(|P|/8, 8)||P$
Let $l = |\text{UTF8}(\texttt{http://keccak.noekeon.org/tree/})| + 8 + |M| = 264 + |P|$
Align $M$ to a block boundary: $M = M||0^{(-l) \bmod r}$
$M = M||\text{pad}(m, 8)$
if $v = \text{final}$ then
    $M = M||(\text{enc}(0, 8)$ if $G = \text{LI}$, or $\text{enc}(1, 8)$ if $G = \text{FNG})$
    $M = M||\text{enc}(H, 8)||\text{enc}(D, 8)||\text{enc}(B/64, 16)$
    $M = M||\text{enc}(1, 8)$
else
    $M = M||\text{enc}(0, 8)$
end if
return $M$
```

We then define how the data are divided into leaves. The number L of leaves depends on the tree growing mode G. If $G = \text{LI}$, $L = D^H$. If $G = \text{FNG}$, $L = RD^{H-1}$ with $R = \left\lceil \frac{|M|}{BD^{H-1}} \right\rceil$. Input message blocks are assigned to the leaves according to Algorithm 7.

Algorithm 7 Construction of the leaves

```
For each leaf $L_j$, $0 \leq j \leq L - 1$, set $L_j$ to the empty string
for $i = 0$ to $|M| - 1$ do
    $j = \lfloor \frac{i}{B} \rfloor \bmod L$
    Append bit $i$ of $M$ to $L_j$
end for
```

The processing at each node is defined in Algorithm 8. The output of the KECCAKTREE scheme is the output of the final node obtained by calling $\text{Node}(0, 0)$.

Algorithm 8 Node(h, j)

```
if $h = H \neq 0$ (processing a leaf) then
  return $\lfloor \mathrm{K}(P, L_j, \text{nonfinal}) \rfloor_c$
else if $0 < h < H$ (processing an internal node except the final node) then
  Set $Z$ to the empty string
  for $i = 0$ to $D - 1$ do
    $Z = Z || \mathrm{Node}(h + 1, j + iD^{H-h-1})$
  end for
  return $\lfloor \mathrm{K}(P, Z, \text{nonfinal}) \rfloor_c$
else if $h = 0$ and $H > 0$ (processing the final node of a non-trivial tree) then
  Set $Z$ to the empty string
  for $i = 0$ to $R - 1$ (taking $R = D$ when $G = \mathrm{LI}$) do
    $Z = Z || \mathrm{Node}(1, iD^{H-1})$
  end for
  return $\mathrm{K}(P, Z, \text{final}, A)$
else if $h = H = 0$ (processing a trivial tree containing only a final node) then
  return $\mathrm{K}(P, M, \text{final}, A)$
end if
```

6.4.2 Discussion

The calls to Node(h, j), for equal h but different j, process independent data and so can be parallelized. Furthermore, the prefix P is always absorbed in K, both for leaves and internal nodes. The state after absorbing P can therefore be computed once for all.

If the optimal number of independent processes is known, one can simply use the LI mode ($G = \mathrm{LI}$) with $H = 1$ and D equal to this number of independent processes. Tree hashing in this case comes down to a simple parallel hashing, where the B-bit blocks of the input message are equally spread onto D different sponge functions. The D results are then combined at the final node to make the final output string.

In addition to the LI and FNG growing modes, one can make the tree grow by increasing its height H until the number of leaves L is large enough for $|M|$. Setting $G = \mathrm{LI}$ in this case does not really interleave the input blocks, but fixes the tree. Knowing whether a node is going to be the final node (if H is large enough) or not becomes significant only at the end of the absorbing phase of K. Once H is large enough, the implementation can then fix it and mark the candidate final node as final.

According to our preliminary analysis, the expected workload for differentiating this scheme from a random oracle is of the order $2^{n/2}$ with n the output size of the called compression function (i.e., in this case K with $n = c$). For this reason, this scheme would be sub-optimal if used with one of the four fixed-output length candidates of [8, Section 4] as they all have an output size n that satisfies $n \leq c/2$. When using as underlying compression function a function that is claimed to be indifferentiable with a capacity c, the optimum output size to use is also c, resulting in the absence of generic attacks with expected workload of order below $2^{c/2}$.

Chapter 7

Implementation

In this chapter, we discuss the implementation of Keccak in software and in hardware, together with its estimated performances.

7.1 Bit and byte numbering conventions

As it impacts the reference implementation, the bit and byte numbering conventions are defined in [8, Section 5.1]. In this section, we wish to detail our choices concerning the mapping between the bits of the Keccak-$f[b]$ permutation and their representation in terms of w-bit CPU words and in the SHA-3 API defined by NIST [55].

As explained in [8, Section 1], the bits in the state are numbered from 0 to $b-1$, and bit $i = z + w(5y + x)$ corresponds to the coordinates (x, y, z). From the software implementation point of view, we expect the bits in a lane (i.e., with the same coordinates (x, y)) to be packed together in a w-bit CPU word, so that, if the processor allows it, the operation ρ becomes a set of CPU word rotations.

For the ρ operation to be translated into rotation instructions in the processor, the numbering z must be either an increasing or a decreasing function of the bit numbering in the processor's conventions. So, up to a constant offset, either $z = 0$ is the most significant bit (MSB) and $z = w - 1$ is the least significant bit (LSB), or vice-versa.

The input bits of the hash function come through the `Update` function of the API, organized as a sequence of bytes. Within each block, the message bit $i = i_{\text{bit}} + 8i_{\text{byte}}$ is going to be XORed with the state bit i. To avoid re-ordering bits or bytes and to allow a word-wise XORing, the message bit numbering should follow the same convention as the state bit numbering. In particular, if $z = 0$ indicates the MSB (resp. LSB), $i_{\text{byte}} = 0$ should indicate the most (resp. least) significant byte within a word.

Since the reference platform proposed by NIST follows the little-endian (LE) convention, we adopt the following numbering of bits and bytes: When mapping a lane to a CPU word, the bit $z = 0$ is the LSB. Within a CPU word, the byte $i_{\text{byte}} = 0$ is the least significant byte. Within a byte, $i_{\text{bit}} = 0$ is the LSB. This way, the message bits can be organized as a sequence of words (except for the last bits), which can be XORed directly to the lanes of the state on a LE processor.

The convention in the `Update` function is different, and this is the reason for applying the formal bit reordering of [8, Section 5.1]. It formalizes the chosen translation between the two conventions, while having an extremely limited impact on the implementation. In practice, only the bits of the last byte (when incomplete) of the input message need to be shifted.

7.2 General aspects

For KECCAK, the bulk of the processing is done by the KECCAK-f permutation and by XORing the message blocks into the state. For an input message of l bits, the number of blocks to process, or in other words, the number of calls to KECCAK-f, is given by:

$$\left\lceil \frac{8\lfloor \frac{l}{8} \rfloor + 32}{r} \right\rceil .$$

For an output length n smaller than or equal to the bitrate, the squeezing phase does not imply any additional processing. However, in the arbitrarily-long output mode, the additional number of calls to KECCAK-f for an n-bit output is $\lceil \frac{n}{r} \rceil - 1$.

In terms of memory usage, KECCAK has no feedforward loop and the message block can be directly XORed into the state. This limits the amount of working memory to the state, the round number and some extra working memory for θ and χ. Five w-bit words of extra working memory allow the implementation of θ to compute the XOR of the sheets, while they can hold the five lanes of a plane when χ is computed.

In terms of lane operations, the evaluation of KECCAK-f[1600] uses

- 1368 XORs,
- 450 ANDs,
- 450 NOTs, and
- 522 64-bit rotations.

Almost 80% of the NOT operations can be removed by applying a *lane complementing transform* as explained in Section 7.2.1, turning a subset of the AND operations into OR operations.

On a 64-bit platform, each lane can be mapped to a CPU word. On a b-bit platform, each lane can be mapped to $64/b$ CPU words. There are different such mappings. As long as the same mapping is applied to all lanes, each bitwise Boolean operation on a lane is translated as $64/b$ instructions on CPU words. The most straightforward mapping is to take as CPU word i the lane bits with $z = bi \dots b(i+1) - 1$. In that case, the 64-bit rotations need to be implemented using a number of shifts and bitwise Boolean instructions. Another possible mapping, that translates the 64-bit rotations into a series of b-bit CPU word rotation instructions, is introduced in Section 7.2.2.

7.2.1 The lane complementing transform

The mapping χ applied to the 5 lanes in a plane requires 5 XORs, 5 AND and 5 NOT operations. The number of NOT operations can be reduced to 1 by representing certain lanes by their complement. In this section we explain how this can be done.

For the sake of clarity we denote the XOR operation by $\oplus$, the AND operation by $\wedge$, the OR operation by $\vee$ and the NOT operation by $\oplus 1$. Assume that the lane with $x = 2$ is represented its bitwise complement $\overline{a[2]}$. The equation for the bits of $A[0]$ can be transformed using the law of De Morgan ($\overline{a} \wedge \overline{b} = \overline{a \vee b}$):

$$A[0] = a[0] \oplus (a[1] \oplus 1) \wedge (\overline{a[2]} \oplus 1) = \overline{a[0]} \oplus (a[1] \vee \overline{a[2]}) \ .$$

The equation for the bits of $A[1]$ now becomes $A[1] = a[1] \oplus (\overline{a[2]} \wedge a[3])$. This results in the cancellation of two NOT operations and $A[0]$ being represented by its complement. Similarly, representing $a[4]$ by its complement cancels two more NOT operations. We have

$$\begin{aligned}
\overline{A[0]} = &\ \ a[0] \oplus (a[1] \vee \overline{a[2]}),\\
A[1] = &\ \ a[1] \oplus (\overline{a[2]} \wedge a[3]),\\
\overline{A[2]} = &\ \ a[2] \oplus (a[3] \vee \overline{a[4]}),\\
A[3] = &\ \ a[3] \oplus (\overline{a[4]} \wedge a[0]).
\end{aligned}$$

In the computation of the bits of $A[4]$ the NOT operation cannot be avoided without introducing NOT operations in other places. We do however have two options:

$$\begin{aligned}
\overline{A[4]} = &\ \ \overline{a[4]} \oplus ((a[0] \oplus 1) \wedge a[1]), \text{ or}\\
A[4] = &\ \ \overline{a[4]} \oplus (a[0] \vee (a[1] \oplus 1)).
\end{aligned}$$

Hence one can choose between computing $\overline{A[4]}$ and $A[4]$. In each of the two cases a NOT operation must be performed on either $a[0]$ or on $a[1]$. These can be used to compute $A[0]$ rather than $\overline{A[0]}$ or $\overline{A[1]}$ rather than $A[1]$, respectively, adding another degree of freedom for the implementer. In the output some lanes are represented by their complement and the implementer can choose from 4 output patterns. In short, representing lanes $a[2]$ and $a[4]$ by their complement reduces the number of NOT operations from 5 to 1 and replaces 2 or 3 AND operations by OR operations. It is easy to see that complementing any pair of lanes $a[i]$ and $a[(i+2) \bmod 5]$ will result in the same reduction. Moreover, this is also the case when complementing all lanes except $a[i]$ and $a[(i+2) \bmod 5]$. This results in 10 possible input patterns in total.

Clearly, this can be applied to all 5 planes of the state, each with its own input and output patterns. We apply a complementing pattern (or *mask*) p at the input of χ and choose for each plane an output pattern resulting in P. This output mask P propagates through the linear steps θ, ρ, π (and ι) as a symmetric difference pattern, to result in yet another (symmetric) mask $p' = \pi(\rho(\theta(P)))$ at the input of χ of the next round. We have looked for couples of masks (p, P) that are *round-invariant*, i.e., with $p = \pi(\rho(\theta(P)))$, and found one that complements the lanes in the following 6 (x, y) positions at the output of χ or input of θ:

$$P : \{(1,0), (2,0), (3,1), (2,2), (2,3), (0,4)\}.$$

A round-invariant mask P can be applied at the input of KECCAK-f. The benefit is that in all rounds 80% of the NOT operations are cancelled. The output of KECCAK-f can be corrected by applying the same mask P. The overhead of this method comes from applying the masks at the input and output of KECCAK-f. This overhead can be reduced by redefining KECCAK-f as operating on the masked state. In that case, P must be applied to the root state 0^b and during the squeezing phase some lanes (e.g., 4 when $r = 16w$) must be complemented prior to being presented at the output.

7.2.2 Bit interleaving

The technique of bit interleaving consists in coding an w-bit lane as an array of $s = w/b$ CPU words of b bits each, with word i containing the lane bits with $z \equiv i \pmod{s}$. This can be applied to any version of Keccak-f to any CPU with word length b that divides its lane length w.

For readability, we now treat the concrete case of 64-bit lanes and 32-bit CPU words. This can be easily generalized. A 64-bit lane is coded as two 32-bit words, one containing the lane bits with even z-coordinate and the other those with odd z-coordinates. More exactly, a lane $L[z] = a[x][y][z]$ is mapped onto words U and V with $U[i] = L[2i]$ and $V[i] = L[2i+1]$. If all the lanes of the state are coded this way, the bitwise Boolean operations can be simply implemented with bitwise Boolean instructions operating on the words. The main benefit is that the lane translations in ρ and θ can now be implemented with 32-bit word rotations. A translation of L with an even offset 2τ corresponds to the translation of the two corresponding words with offset τ. A translation of L with an odd offset $2\tau + 1$ corresponds to $U \leftarrow \mathrm{ROT}_{32}(V, \tau+1)$ and $V \leftarrow \mathrm{ROT}_{32}(U, \tau)$. On a 32-bit processor with efficient rotation instructions, this may give an important advantage compared to the straightforward mapping of lanes to CPU words. Additionally, a translation with an offset equal to 1 or -1 results in only a single CPU word rotation. This is the case for 6 out of the 29 lane translations in each round (5 in θ and 1 in ρ).

The bit-interleaving representation can be used in all of Keccak-f where the input and output of Keccak-f assume this representation. This implies that during the absorbing the input blocks must be presented in bit-interleaving representation and during the squeezing the output blocks are made available in bit-interleaving representation. When implementing Keccak in strict compliance to the specifications [8], the input blocks must be transformed to the bit-interleaving representation and the output blocks must be transformed back to the standard representation. However, one can imagine applications that require a secure hash function but no interoperability with respect to the exact coding. In that case one may present the input in interleaved form and use the output in this form too. The resulting function will only differ from Keccak by the order of bits in its input and output.

In hashing applications, the output is usually kept short and some overhead is caused by applying the bit-interleaving transform to the input. Such a transform can be implemented using shifts and bitwise operations. For implementing Keccak-f[1600] on a 32-bit CPU, we must distribute the 64 bits of a lane to two 32-bit words. The cost of this transform is about 2 cycles/byte on the reference platform (see Table 7.2). On some platforms, look-up tables can speed up this process, although the gain is not always significant.

7.3 Software implementation

We provide a reference and two optimized implementations of the Keccak candidates in ANSI C. The file `KeccakSponge.c` is common to the three flavors and implements the NIST API, including the functions `Init`, `Update`, `Final` and `Hash`, plus the additional function `Squeeze` (see [8, Section 5.2]).

The reference implementation calls the Keccak-f[1600] permutation written in `Keccak-PermutationReference.c`, while the optimized versions use `KeccakPermutationOptimized-32.c` and `KeccakPermutationOptimized64.c` for 32-bit and 64-bit platforms respectively.

For best performance, we allow the state to be stored in an opaque way during the hash computation. In particular, the state can have some of its lanes complemented (as in 7.2.1), the even and odd bits separated (as in 7.2.2) or the bytes in a word reordered according to the endianness of the machine. The methods that apply on the state are:

- `KeccakInitializeState()` to set the state to zero;
- `KeccakPermutation()` to apply the KECCAK-f[1600] permutation on it;
- `KeccakAbsorb1024bits()` and `KeccakAbsorb512bits()` to XOR the input, given as bytes, to the state and then apply the permutation;
- `KeccakExtract1024bits()` and `KeccakExtract512bits()` to extract output bits from the state in the squeezing phase.

In the case of the reference implementation, separate functions are provided for the θ, ρ, π, χ and ι operations for readability purposes.

Additional files are provided, i.e., to produce known answer test results, to display intermediate values in the case of the reference implementation and to measure timings in the optimized one.

7.3.1 Optimized for speed

An optimized version of KECCAK has been written in ANSI C and tested under the following platforms.

- Platform A:
  - PC running Linux openSUSE 11.0 x86_64;
  - CPU: Intel Xeon 5150 (CPU ID: 06F6) at 2.66GHz, dual core, with a bus at 1333MHz and 4Mb of level-2 cache;
  - Compiler: GCC 4.3.2 using `gcc -O3 -g0 -march=nocona` for 64-bit code or `gcc -O3 -g0 -m32` for 32-bit code.
- Platform B (reference platform proposed by NIST):
  - PC running Vista Ultimate x86 or x64, version 6.0.6001, SP1 build 6001;
  - CPU: Intel Core2 Duo E6600 at 2.4GHz;
  - For x86:
    * Microsoft Visual Studio 2008 Version 9.0.21022.8 RTM,
    * 32-bit C/C++ Optimizing Compiler Version 15.00.21022.08 for 80x86;
  - For x64:
    * Microsoft Visual Studio 2008 version 9.0.30428.1 SP1Beta1,
    * Microsoft Windows SDK 6.1 Targeting Windows Server 2008 x64,
    * C/C++ Optimizing Compiler Version 15.00.21022.08 for x64, using `cl /O2 /Ot /favor:INTEL64`.

Operation	Platform A	Platform B
KECCAK-f[1600] only	1264 c	1368 c
Squeezing with $r = 1024$	9.9 c/b	10.7 c/b
Squeezing with $r = 512$	19.8 c/b	21.4 c/b
KECCAK-f and XORing 1024 bits	1296 c	1377 c
Absorbing with $r = 1024$	10.1 c/b	10.8 c/b
KECCAK-f and XORing 512 bits	1296 c	1377 c
Absorbing with $r = 512$	20.3 c/b	21.5 c/b

Table 7.1: Performance of the KECCAK-f[1600] permutation and the XOR of the input block, compiled using 64-bit instructions ("c" is for cycles and "c/b" for cycles per byte)

Operation	Platform A	Platform B
KECCAK-f[1600] only	3528 c	4644 c
Squeezing with $r = 1024$	27.6 c/b	36.3 c/b
Squeezing with $r = 512$	55.1 c/b	72.6 c/b
KECCAK-f and XORing 1024 bits	3728 c	4905 c
Absorbing with $r = 1024$	29.1 c/b	38.3 c/b
KECCAK-f and XORing 512 bits	3664 c	4788 c
Absorbing with $r = 512$	57.3 c/b	74.8 c/b

Table 7.2: Performance of the KECCAK-f[1600] permutation and the XOR of the input block, compiled using only 32-bit instructions ("c" is for cycles and "c/b" for cycles per byte)

The code uses only plain C instructions, without assembly nor SIMD instructions. We have applied lane complementing to reduce the number of NOTs. The operations in the round function have been expanded in macros to allow some reordering of the instructions. We have tried to interleave lines that apply on different variables to enable pipelining, while grouping sets of lines that use a common precomputed value to avoid reloading the registers too often. The order of operations is centered on the evaluation of χ on each plane, preceded by the appropriate XORs for θ and rotations for ρ, and accumulating the parity of sheets for θ in the next round.

The macros are generated by KECCAKTOOLS [10], which can be parameterized to different lane sizes and CPU word sizes, possibly with lane complementing and/or bit interleaving.

For the 64-bit optimized code, unrolling 18 rounds was found often to give the best results, although one can unroll any number of rounds that divides 18. On the platforms defined above, we have obtained the figures in Table 7.1.

For the 32-bit optimized code, we have used interleaving so as to use 32-bit rotations. Unrolling 2 or 6 rounds usually gives the best results. We have then performed the same measurement, instructing the compiler to use only x86 32-bit instructions; the figures are in Table 7.2.

Please refer to eBASH [5] and our website for up-to-date figures on a wider range of platforms.

The performance on the 32-bit reference platform can be improved by using SIMD instructions, as detailed in Section 7.3.2. Also, it may be possible to find a better order of the

C operations using profiling tools or write the code directly in assembler.

While Platform B is the reference platform given by NIST, we think that the figures for Platform A are also relevant to the reference platform for the following reasons. First, the Intel Xeon 5150 [19] is identical in terms of CPU ID, number of cores, core stepping and level-2 cache size as the Intel Core 2 Duo Processor E6600 (2.4GHz) [17]. The clock speed differs, but the figures are expressed in clock cycles. Second, the operating system should not have an impact on the performance of KECCAK as the algorithm does not use any OS services. Finally, the compiler should not have an impact on the intrinsic speed of KECCAK on a given CPU, as a developer can always take the assembly code produced by one compiler and use it as in-line assembly on the other (although we did not do this exercise explicitly).

There is no precomputation in the `Init` function (when the round constants and ρ offsets are integrated into the code). The only constant overhead is taken by clearing the initial state to zero in `Init` and padding the message in `Final`. We have measured the overhead by timing the whole process from `Init` to `Final` with 1, 2, 3 and 10 blocks of data. The number of cycles of the constant overhead, in the current implementation, is of the order of 300 cycles using 64-bit code and of the order 600 cycles using 32-bit code (for the whole message).

7.3.2 Using SIMD instructions

The reference platform CPU, as well as other members of the family, supports single instruction multiple data (SIMD) instruction sets known as MMX, SSE and their successors. These include bitwise operations on 64-bit and 128-bit registers. Thanks to the symmetry of the operations in KECCAK-f, the performance of KECCAK can benefit from these instruction sets. Due to the size of the registers, the relative benefit is higher for 32-bit code than for 64-bit code.

For instance, the `pandn` instruction performs the AND NOT operation bitwise on 128-bit registers (or one register and one memory location), which can be used to implement χ. Such an instruction replaces four 64-bit instructions or eight 32-bit instructions. Similarly, the `pxor` instruction computes the XOR of two 128-bit registers (or one register and one memory location), replacing two 64-bit XORs or four 32-bit XORs.

While instructions on 128-bit registers work well for θ, π, χ and ι, the lanes are rotated by different amounts in ρ. Consequently, the rotations in ρ cannot fully benefit from the 128-bit registers. However, the rotations in θ are all of the same amount and can be combined efficiently.

Overall, the results are encouraging, as shown in Tables 7.3 and 7.4. The use of SIMD instructions on the 32-bit reference platform significantly improves the performance.

7.3.3 SIMD instructions and KECCAKTREE

Parallel evaluations of two instances of KECCAK can also benefit from SIMD instructions, for example in the context of KECCAKTREE. In particular, KECCAKTREE$[G = \text{LI}, H = 1, D = 2, B = r]$ can directly take advantage of two independent sponge functions running in parallel, each taking r bits of input alternatively. The final node then combines their outputs.

We have thus made the following exercise: we have implemented the KECCAK-f permutation with SSE2 instructions using only 64 bits of the 128-bit registers. By definition of these instructions, the same operations are applied to the other 64 bits of the same registers. It is

Operation	**Platform A**	**Platform B**
Keccak-f[1600] only	2040 c	1782 c
Squeezing with $r = 1024$	15.9 c/b	13.9 c/b
Squeezing with $r = 512$	31.9 c/b	27.8 c/b
Keccak-f and XORing 1024 bits	2064 c	1800 c
Absorbing with $r = 1024$	16.1 c/b	14.1 c/b
Keccak-f and XORing 512 bits	2064 c	1791 c
Absorbing with $r = 512$	32.3 c/b	28.0 c/b

Table 7.3: Performance of the Keccak-f[1600] permutation and the XOR of the input block, using SSE2 instructions ("c" is for cycles and "c/b" for cycles per byte)

Operation	**Platform A**	**Platform B**
Keccak-f[1600] only	2256 c	1845 c
Squeezing with $r = 1024$	17.6 c/b	14.4 c/b
Squeezing with $r = 512$	35.3 c/b	28.8 c/b
Keccak-f and XORing 1024 bits	2248 c	1872 c
Absorbing with $r = 1024$	17.6 c/b	14.6 c/b
Keccak-f and XORing 512 bits	2208 c	1872 c
Absorbing with $r = 512$	34.5 c/b	29.3 c/b

Table 7.4: Performance of the Keccak-f[1600] permutation and the XOR of the input block, using SSE2 instructions and restricted to the 32-bit mode ("c" is for cycles and "c/b" for cycles per byte)

thus possible to evaluate two independent instances of Keccak-f in parallel on a single core of the CPU.

Although performance numbers should be evaluated beyond this simple exercise, this way of working allows speeds significantly below 10 cycles/byte/core on the reference platform. Furthermore, one may consider the case $r = 1088$ and $c = 512$, for which the claimed security level is 2^{256}. While losing the power of two for the rate, the final node needs to absorb only one block ($2c < r$) and the overhead remains reasonable: one extra evaluation of Keccak-f per message. This benefits also to long messages, for which the number of cycles per byte is further reduced by 6%.

7.3.4 Protection against side channel attacks

If the input to Keccak includes secret data or keys, side channel attacks may pose a threat. One can protect against timing attacks and simple power analysis by coding the Keccak-f permutation as a fixed sequence of instructions in a straightforward way. Moreover, Keccak-f does not make use of large lookup tables so that cache attacks pose no problem. (The interleaving method of Section 7.2.2 can be implemented with table lookups, which depend on the input message only. Of course, it can also be implemented without any table at all.)

Protection against differential power analysis (DPA) can be obtained by applying several mechanisms, preferably at the same time. One of the mechanisms is called state splitting. This method was proposed in [34] and [14] and consists in splitting the state in two parts—one of which is supposed to be random—that give XORed back the actual state value. If properly implemented, this method eliminates any correlation between values inside the machine and any intermediate computation result, thereby removing the possibility for first-order DPA. We have shown how this can be applied to BaseKing in [25]. It turns out that for the linear steps the computations can be done independently on the two split parts of the state. An additional overhead is in the non-linear steps, in which operations must be performed using parts of both split states. The mechanisms described in [25] apply equally well to χ, the only non-linear part of Keccak-f.

7.3.5 Estimation on 8-bit processors

We have estimated the performance of Keccak on the Intel 8051 microprocessor. The 8051 is an 8-bit processor equipped with an 8-bit data bus and a 16-bit address bus. Initially the 8051 could only address 128 bytes of internal RAM memory, but later versions were released to allow accessing more internal RAM using a technique called Expanded RAM [18]. Nowadays many manufacturers propose variants of the original 8051 with various levels of improvements, including even more powerful addressing modes, security features and shorter instruction cycles. The variant we have selected for our estimation is the 80C51RA/RB/RC microcontroller from Intel [16], with 512 bytes of on-chip RAM, split in 3 segments: 128 bytes of low internal RAM (direct and indirect access), 128 bytes of high internal RAM (indirect access only), and finally 256 bytes of external RAM (indirect access only).

The first problem to solve when implementing Keccak on such a constrained platform like the 80C51RA/RB/RC is the memory mapping. The performance of memory accesses depends on which memory segment is addressed, and so a careful mapping must be done to ensure that most operations are done in the low internal segment. This is particularly

Step	**Performance**
θ	25920 cycles
ρ	15606 cycles
π	0 cycles
χ	22536 cycles
ι	288 cycles
Words re-ordering	11250 cycles
Total	75600 cycles

Table 7.5: Performance estimates of the Keccak-f[1600] permutation on the 8XC51RA/RB/RC

difficult for θ for which in-place evaluation requires additional temporary storage. However by following a tight schedule of operations, it is possible to maintain the complete state of Keccak in internal RAM, hence maximizing the performance.

Another problem is the implementation of 64-bit rotations in ρ using 8-bit rotate operations. At first the 8051 does not offer specific instructions to optimize this step, and so rotations are merely done by iterating several times rotate-through-carry instructions (with the exception of 4-bits rotation which can be done by swapping the byte digits). However using efficient memory transfer instructions like `XCH` that exchanges the accumulator with a byte in memory, it is actually possible to reduce the average number of cycles for rotation to only 4.3 cycles/byte.

The performance estimates for Keccak including the details for each step are given in Table 7.5, for 18 rounds. One cycle refers to the number of controller clock oscillation periods, which is 12 in the case of our selected variant. It must be noted that the figures are the result of a best-effort paper exercise. Figures for an actual implementation might vary, in particular if it uses specific manufacturer improvements available on the platform.

7.4 Hardware implementation

In this section we highlight the characteristics of Keccak when implemented in hardware. It is possible to design different architectures of Keccak. We will start describing the high-speed core design depicted in Figure 7.1. This is based on the plain instantiation of the combinational logic for computing one round, and use it iteratively. A second approach is a low area coprocessor described in section 7.4.3. One of the benefits of the proposed algorithm is that, unless intentionally forced, a general architecture can easily support the four digest lengths, and contrary to the HMAC case, no additionally circuitry is needed for supporting MAC or other functionality as KDF, MGF and similar. Our first VHDL implementations have been tested on different FPGAs by J. Strömbergson [63], highlighting some possible improvements and problems with the tools available from FPGA vendors. We have improved the VHDL code for solving the problems, and this has resulted in better results in ASIC as well.

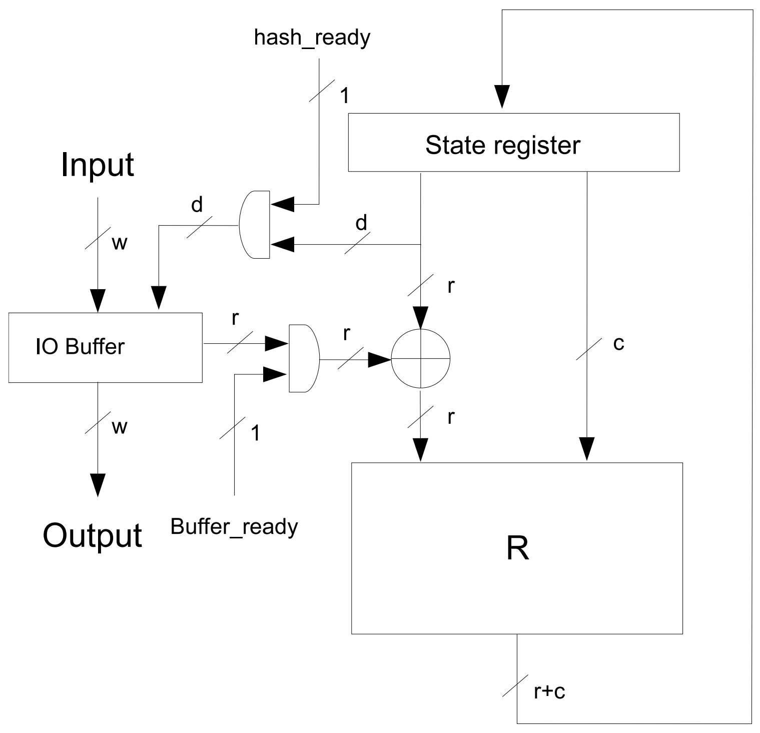

Figure 7.1: The high-speed core

7.4.1 High-speed core

The core is composed of three main components: the round function, the state register and the input/output buffer. The use of the input/output buffer allows decoupling the core from a typical bus used in a system-on-chip (SoC). These buses typically come in widths of 8, 16, 32, 64 or 128 bits. We have decided to fix its width to 64 bits. When the bitrate is equal to 1024 bits the throughput of the bus is almost saturated when processing long messages. Thus in this configuration the core will be capable of processing 128 bytes in 18 clock cycles. If the bitrate is reduced to 512 bits, the elaboration time remains the same but the amount of input is reduced by half.

The I/O buffer allows the core to compute the absorbing phase while the words of the next block are transferred through the bus. With bitrate 1024 and a bus width smaller than 64, e.g., the common value 32, the throughput of the hash engine would be bounded by the throughput of the bus.

The high-speed core has been coded in VHDL, test benches for the permutation and the hash function are provided together with C code allowing the generation of test vectors for the test benches. The core has been tested using ModelSim tools. In order to evaluate the silicon area and the clock frequency, the core has been synthesized using Synopsys Design Compiler and a 130 nm general purpose ST technology library, worst case 105°C. The core can reach a clock frequency of 526MHz, implying a throughput of 29.9 Gbit/s for bitrate 1024. Note that the 130 nanometer technology is relatively old; the use of newer technology, such as 65 nm, will very likely allow the core to reach a clock frequency of 1GHz or above.

Device	**Logic**	**Registers**	**Max Freq.**	**Throughput**
Altera StratixIII EP3SE50F484C2	4713 (38000) ALUTs	2642 (38000)	218 MHz	12.4 Gbit/s
Altera Cyclone III EP3C10F256C6	5776 (10320) LEs	2641 (10320)	133 MHz	7.5 Gbit/s
Virtex 5 XC5VLX50FF324-3	1412 (7200) slices	2640 (28800)	122 MHz	6.9 Gbit/s

Table 7.6: Performance estimation of the high speed core on different FPGAs, and in brackets the resources available in the different cases.

The area needed for having the core running at this frequency is 48 kgate, composed of 19 kgate for the round function, 9 kgate for the I/O buffer and 21 kgate for the state register and control logic. The critical path of the combinatorial logic of the round function is only 1.1 ns.

We have used Altera Quartus II Web Edition version 8.1 [15] and Xilinx ISE WebPACK version 10.1 [20] to evaluate VHDL with the tools for FPGA. These tools provide estimations of the amount of resources needed and the maximum clock frequency reached. We report in Table 7.6 the estimation of the completed place and route.

For more details on the VHDL, refer to the `readme.txt` file in the `VHDL` directory.

7.4.2 Variants of the high-speed core

The high-speed core can be modified to optimize for different aspects.

In many systems the clock frequency is fixed for the entire chip. So even if the hash core can reach a high frequency it has to be clocked at a lower frequency. In such a scenario Keccak allows to instantiate two, three or even six rounds in combinatorial logic and compute them in one clock cycle.

In the high-speed core we have decided to instantiate a separate buffer for the input/output functions. This allows to run the absorbing phase while the bus is transferring the next block to be processed. An alternative for saving area is to execute the storing of the words composing the block directly in the state register. This alternative allows to save about 8 kgate and decreases the throughput to 15 Gbit/s at 500MHz, which is still a considerably high value.

For a further reduction of the silicon area it is possible to compute one round in more than one clock cycle. Thanks to the bit slice approach a minimalist architecture could be made composed by the state register, with hardwired the π and ρ transformation, and the logic and registers for computing first θ at bit level and after π and ρ, χ at bit level.

7.4.3 Low-area coprocessor

The core presented in Section 7.4.1 operates in a stand-alone fashion. The input block is transferred to the core, and the core does not use other resources of the system for performing the computation. The CPU can program a *direct memory access* (DMA) for transferring chunks of the message to be hashed, and the CPU can be assigned to a different task while the core is computing the hash. A different approach can be taken in the design of the

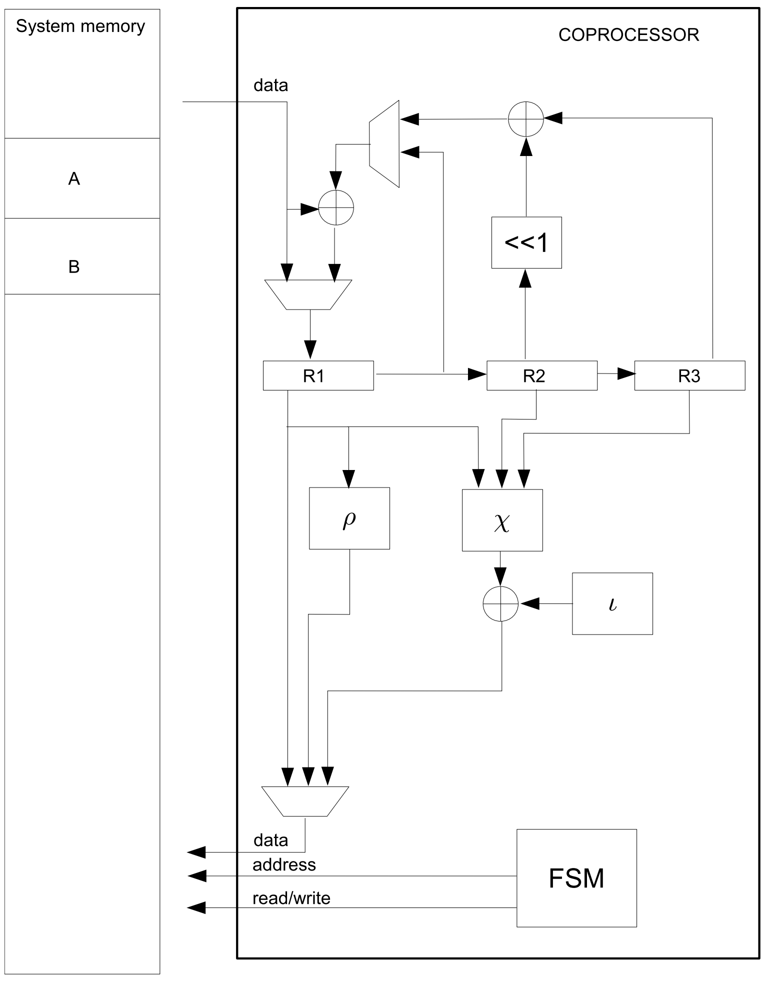

Figure 7.2: The low area coprocessor

hardware accelerator: the core can use the system memory instead of having all the storage capabilities internally. The state of KECCAK will be stored in memory and the coprocessor is equipped with registers for storing only temporary variables.

This is depicted in figure 7.2 where memory buffer labeled with A is reserved for the state, and B is reserved for temporary values. The width of the data bus for performing memory access could be of different width, we consider it of 64 bits as a first assumption, and later considerations are reported if the width is smaller.

Internally the coprocessor is divided in two parts, a finite state machine (FSM) and a data path. The data path is equipped with 3 registers for storing temporary values. The FSM computes the address to be read and set the control signals of the data path. The round is computed in different phases. First the sheet parities are computed, and the 5 lanes are stored in a dedicated area of the memory. The second phase consists in computing the θ transformation, reading all the lanes of the state, and computing the XOR with the corresponding sheet parities. After computing a lane in this way, it is rotated according to ρ and written to the position defined by π. Now the intermediate state is completely

Device	**Logic**	**Registers**	**Max Freq.**	**Throughput**
Altera StratixIII EP3SE50F484C2	855 (38000) ALUTs	242 (38000)	366 MHz	96.8 Mbit/s
Altera Cyclone III EP3C5F256C6	1559 (5136) LEs	242 (5136)	181 MHz	47.8 Mbit/s
Virtex 5 XC5VLX50FF324-3	444 (7200) slices	227 (28800)	265 MHz	70.1 Mbit/s

Table 7.7: Performance estimation of the low area coprocessor on different FPGAs, and in brackets the resources available in the different cases.

stored in the buffer B. The last step is to compute χ and add the round constant, ι, to the lane in position $(0, 0)$. For doing this the coprocessor reads 3 lanes of a plane from the intermediate state, computes χ and writes the result to the buffer A, reads another element of the intermediate value and writes the new χ, and so on for the 5 elements of the plane.

The computation of the KECCAK-f permutation takes 3870 clock cycles, every round requires 215 clock cycles, and 55 out of these are *bubbles* where the core is computing internally and not transferring data to or from the memory.

In a variant with 32-bit memory words rather than 64-bit words, the number of clock cycles doubles but only for the part relative to read and write, not for the bubbles. In such an implementation the KECCAK-f permutation will require 6750 clock cycles.

The buffer A, where the input of the permutation is written and where the output of the permutation is written and the end of the computation, is composed of 200 bytes, while the memory space for storing temporary values is composed of 240 bytes.

The low-area coprocessor has been coded in VHDL and simulated using Modelsim. As the core depicted in Section 7.4.1, the coprocessor has been synthesized using ST technology at 130 nm. The coprocessor can run up to 666.7 MHz requiring 6.5 kgate. If the core is synthesized for a clock frequency limited to 200MHz, the area requirement is reduced to 5 kgate. In both cases the amount of area needed for the registers is about 1 kgate. In the case of FPGA, the estimations are reported in table 7.7.

It is interesting to note that the low area coprocessor is capable of reaching higher frequencies compared to the high speed core.

This kind of coprocessor is suitable for smart cards or wireless sensor network where area is particularly important since it determines the cost of the device and there is not a rich operating system allowing to run different processes in parallel.

7.4.4 Protection against side channel attacks

Due to the simplicity of the round logic, and the use of simple 2-input gates, it is possible to use logic gates resistant to power analysis, like WDDL [64] or SecLib [35]. These types of logic are evolutions of the dual rail logic, where a bit is coded using two lines in such a way that all the logic gates consume the same amount of energy independently of the values. Additionally, the fact that the non-linear component of KECCAK-f is limited to binary AND gates, it lends itself very well for the very powerful protection techniques based on secret sharing proposed in [60] that can offer effective protection against glitches.

Bibliography

[1] *SAGE mathematical software*, http://www.sagemath.org/.

[2] T. Baignères, J. Stern, and S. Vaudenay, *Linear cryptanalysis of non binary ciphers*, Selected Areas in Cryptography (C. M. Adams, A. Miri, and M. J. Wiener, eds.), Lecture Notes in Computer Science, vol. 4876, Springer, 2007, pp. 184–211.

[3] M. Bellare, R. Canetti, and H. Krawczyk, *Keying hash functions for message authentication*, Advances in Cryptology – Crypto '96 (N. Koblitz, ed.), LNCS, no. 1109, Springer-Verlag, 1996, pp. 1–15.

[4] D. J. Bernstein, *The Salsa20 family of stream ciphers*, 2007, Document ID: 31364286077dcdff8e4509f9ff3139ad, http://cr.yp.to/papers.html#salsafamily.

[5] D. J. Bernstein and T. Lange (editors), *eBACS: ECRYPT Benchmarking of cryptographic systems*, http://bench.cr.yp.to, accessed 21 December 2008.

[6] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, RadioGatún*, a belt-and-mill hash function*, Second Cryptographic Hash Workshop, Santa Barbara, August 2006, http://radiogatun.noekeon.org/.

[7] ———, *Sponge functions*, Ecrypt Hash Workshop 2007, May 2007, also available as public comment to NIST from http://www.csrc.nist.gov/pki/HashWorkshop/Public_Comments/2007_May.html.

[8] ———, Keccak *specifications*, NIST SHA-3 Submission, October 2008, http://keccak.noekeon.org/.

[9] ———, *On the indifferentiability of the sponge construction*, Advances in Cryptology – Eurocrypt 2008 (N. P. Smart, ed.), Lecture Notes in Computer Science, vol. 4965, Springer, 2008, http://sponge.noekeon.org/, pp. 181–197.

[10] ———, KeccakTools *software*, January 2009, http://keccak.noekeon.org/.

[11] E. Biham, O. Dunkelman, and N. Keller, *The rectangle attack - rectangling the serpent*, Advances in Cryptology – Eurocrypt 2001 (B. Pfitzmann, ed.), Lecture Notes in Computer Science, vol. 2045, Springer, 2001, pp. 340–357.

[12] C. Bouillaguet and P.-A. Fouque, *Analysis of the collision resistance of RadioGatún using algebraic techniques*, Selected Areas in Cryptography, Lecture Notes in Computer Science, vol. 4876, Springer, 2008.

[13] R. Canetti, O. Goldreich, and S. Halevi, *The random oracle methodology, revisited*, Proceedings of the 30th Annual ACM Symposium on the Theory of Computing, ACM Press, 1998, pp. 209–218.

[14] S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi, *Towards sound approaches to counteract power-analysis attacks*, Advances in Cryptology – Crypto '99 (M. J. Wiener, ed.), Lecture Notes in Computer Science, vol. 1666, Springer, 1999, pp. 398–412.

[15] Altera corporation, *Quartus II web edition software*, `http://www.altera.com`.

[16] Intel Corporation, *Intel 8XC51RA/RB/RC hardware description*, `http://www.intel.com/design/mcs51/manuals/272668.htm`.

[17] ______, *Intel® Core^{TM}2 Duo Desktop Processor E6600*, `http://processorfinder.intel.com/details.aspx?sSpec=SL9S8`.

[18] ______, *Intel MCS 51/251 microcontrollers - expanded RAM*, `http://www.intel.com/design/mcs51/er_51.htm`.

[19] ______, *Intel® Xeon® Processor 5150*, `http://processorfinder.intel.com/details.aspx?sSpec=SL9RU`.

[20] Xilinx corporation, *ISE WebPACK software*, `http://www.xilinx.com`.

[21] D. A. Cox, J. B. Little, and D. O'Shea, *Ideals, varieties, and algorithms*, third ed., Springer, 2007.

[22] J. Daemen, *Cipher and hash function design strategies based on linear and differential cryptanalysis, PhD thesis*, K.U.Leuven, 1995.

[23] J. Daemen and C. S. K. Clapp, *Fast hashing and stream encryption with PANAMA*, Fast Software Encryption 1998 (S. Vaudenay, ed.), LNCS, no. 1372, Springer-Verlag, 1998, pp. 60–74.

[24] J. Daemen, L. R. Knudsen, and V. Rijmen, *The block cipher Square*, Fast Software Encryption 1997 (E. Biham, ed.), Lecture Notes in Computer Science, vol. 1267, Springer, 1997, pp. 149–165.

[25] J. Daemen, M. Peeters, and G. Van Assche, *Bitslice ciphers and power analysis attacks*, in Schneier [62], pp. 134–149.

[26] J. Daemen, M. Peeters, G. Van Assche, and V. Rijmen, *Nessie proposal: the block cipher* NOEKEON, Nessie submission, 2000, `http://gro.noekeon.org/`.

[27] J. Daemen and V. Rijmen, *The design of Rijndael — AES, the advanced encryption standard*, Springer-Verlag, 2002.

[28] ______, *Probability distributions of correlation and differentials in block ciphers*, Journal of Mathematical Cryptology **1** (2007), no. 3, 221–242.

[29] N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker, *The Skein hash function family*, Submission to NIST, 2008, `http://skein-hash.info/`.

[30] E. Filiol, *A new statistical testing for symmetric ciphers and hash functions*, Proc. Information and Communications Security 2002, volume 2513 of LNCS, Springer, 2002, pp. 342–353.

[31] IETF (Internet Engineering Task Force), *RFC 3629: UTF-8, a transformation format of ISO 10646*, 2003, `http://www.ietf.org/rfc/rfc3629.txt`.

[32] ———, *RFC 3986: Uniform resource identifier (URI): Generic syntax*, 2005, `http://www.ietf.org/rfc/rfc3986.txt`.

[33] G. Gielen and J. Figueras (eds.), *2004 design, automation and test in Europe conference and exposition (DATE 2004), 16-20 February 2004, Paris, France*, IEEE Computer Society, 2004.

[34] L. Goubin and J. Patarin, *DES and differential power analysis (the duplication method)*, CHES (Ç. K. Koç and C. Paar, eds.), Lecture Notes in Computer Science, vol. 1717, Springer, 1999, pp. 158–172.

[35] S. Guilley, P. Hoogvorst, Y. Mathieu, R. Pacalet, and J. Provost, *CMOS structures suitable for secured hardware*, in Gielen and Figueras [33], pp. 1414–1415.

[36] IEEE, *P1363-2000, standard specifications for public key cryptography*, 2000.

[37] A. Joux, *Multicollisions in iterated hash functions. Application to cascaded constructions*, Advances in Cryptology – Crypto 2004 (M. Franklin, ed.), LNCS, no. 3152, Springer-Verlag, 2004, pp. 306–316.

[38] J. Kelsey, T. Kohno, and B. Schneier, *Amplified boomerang attacks against reduced-round mars and serpent*, in Schneier [62], pp. 75–93.

[39] J. Kelsey and B. Schneier, *Second preimages on n-bit hash functions for much less than 2^n work*, Advances in Cryptology – Eurocrypt 2005 (R. Cramer, ed.), LNCS, no. 3494, Springer-Verlag, 2005, pp. 474–490.

[40] D. Khovratovich, *Two attacks on RadioGatún*, 9th International Conference on Cryptology in India, 2008.

[41] L. Knudsen, C. Rechberger, and S. Thomsen, *Grindahl - a family of hash functions*, Fast Software Encryption 2007 (A. Biryukov, ed.), LNCS, Springer-Verlag, 2007, pp. 39–47.

[42] L. R. Knudsen, *Truncated and higher order differentials*, Fast Software Encryption 1994 (B. Preneel, ed.), Lecture Notes in Computer Science, vol. 1008, Springer, 1994, pp. 196–211.

[43] L. R. Knudsen and V. Rijmen, *Known-key distinguishers for some block ciphers*, Advances in Cryptology – Asiacrypt 2007, 2007, pp. 315–324.

[44] D. E. Knuth, *The art of computer programming, vol. 2, third edition*, Addison-Wesley Publishing Company, 1998.

[45] T. Kohno and J. Kelsey, *Herding hash functions and the Nostradamus attack*, Advances in Cryptology – Eurocrypt 2006 (S. Vaudenay, ed.), LNCS, no. 4004, Springer-Verlag, 2006, pp. 222–232.

[46] RSA Laboratories, *PKCS # 1 v2.1 RSA Cryptography Standard*, 2002.

[47] S. K. Langford and M. E. Hellman, *Differential-linear cryptanalysis*, Advances in Cryptology – Crypto '94 (Y. Desmedt, ed.), Lecture Notes in Computer Science, vol. 839, Springer, 1994, pp. 17–25.

[48] U. Maurer, R. Renner, and C. Holenstein, *Inidifferentiability, impossibility results on reductions, and applications to the random oracle methodology*, Theory of Cryptography - TCC 2004 (M. Naor, ed.), Lecture Notes in Computer Science, no. 2951, Springer-Verlag, 2004, pp. 21–39.

[49] R. C. Merkle, *Secrecy, authentication, and public key systems, PhD thesis*, UMI Research Press, 1982.

[50] NIST, *Federal information processing standard 186-3, digital signature standard (DSS)*, March 2006.

[51] ———, *Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family*, Federal Register Notices **72** (2007), no. 212, 62212–62220, `http://csrc.nist.gov/groups/ST/hash/index.html`.

[52] ———, *NIST special publication 800-106 draft, randomized hashing digital signatures*, July 2007.

[53] ———, *NIST special publication 800-56a, recommendation for pair-wise key establishment schemes using discrete logarithm cryptography (revised)*, March 2007.

[54] ———, *NIST special publication 800-90, recommendation for random number generation using deterministic random bit generators (revised)*, March 2007.

[55] ———, *ANSI C cryptographic API profile for SHA-3 candidate algorithm submissions, revision 5*, February 2008, available from `http://csrc.nist.gov/groups/ST/hash/sha-3/Submission_Reqs/crypto_API.html`.

[56] ———, *Federal information processing standard 198, the keyed-hash message authentication code (HMAC)*, July 2008.

[57] ———, *NIST special publication 800-108, recommendation for key derivation using pseudorandom functions*, April 2008.

[58] R. L. Rivest, M. J. B. Robshaw, R. Sidney, and Y. L. Yin, *The RC6 block cipher*, AES proposal, August 1998.

[59] P. Sarkar and P. J. Schellenberg, *A parallelizable design principle for cryptographic hash functions*, Cryptology ePrint Archive, Report 2002/031, 2002, `http://eprint.iacr.org/`.

[60] M. Schlaeffer, *Masking non-linear functions based on secret sharing*, Echternach Symmetric Cryptography seminar, 2008, `http://wiki.uni.lu/esc/`.

[61] B. Schneier, *Applied cryptography*, second ed., John Wiley & Sons, 1996.

[62] B. Schneier (ed.), *Fast software encryption, 7th international workshop, fse 2000, new york, ny, usa, april 10-12, 2000, proceedings*, Lecture Notes in Computer Science, vol. 1978, Springer, 2001.

[63] J. Strömbergson, *Implementation of the Keccak hash function in FPGA devices*, `http://www.strombergson.com/files/Keccak_in_FPGAs.pdf`.

[64] K. Tiri and I. Verbauwhede, *A logic level design methodology for a secure DPA resistant ASIC or FPGA implementation*, in Gielen and Figueras [33], pp. 246–251.

[65] W3C, *Namespaces in XML 1.0 (second edition)*, 2006, `http://www.w3.org/TR/2006/REC-xml-names-20060816`.

[66] D. Wagner, *The boomerang attack*, Fast Software Encryption 1999 (L. R. Knudsen, ed.), Lecture Notes in Computer Science, vol. 1636, Springer, 1999, pp. 156–170.

[67] R. Wernsdorf, *The round functions of Rijndael generate the alternating group*, Fast Software Encryption 2002 (J. Daemen and V. Rijmen, eds.), Lecture Notes in Computer Science, vol. 2365, Springer, 2002, pp. 143–148.

[68] Wikipedia, *Cryptographic hash function*, 2008, `http://en.wikipedia.org/wiki/Cryptographic_hash_function`.

[69] ———, *Impossible differential cryptanalysis*, 2008, `http://en.wikipedia.org/wiki/Miss_in_the_middle_attack`.

[70] ———, *Random permutation statistics*, 2008, `http://en.wikipedia.org/wiki/Random_permutation_statistics`.

[71] M. R. Z'aba, H. Raddum, M. Henricksen, and E. Dawson, *Bit-pattern based integral attack*, Fast Software Encryption 2008 (K. Nyberg, ed.), Lecture Notes in Computer Science, vol. 5086, Springer, 2008, pp. 363–381.

Appendix A

Change log

A.1 From 1.1 to 1.2

- In Chapter 1, we added Section 1.1 with a 2-page specifications summary of KECCAK.
- In Section 5.3.1, we provide more explanations on difference propagation and correlation properties of χ.
- In Section 5.9.3, we present ANF tests also on the inverse of KECCAK-f.
- In Section 7.3, updates have been made to the software performance in general and regarding SIMD instructions in particular, see Sections 7.3.2 and 7.3.3.

A.2 From 1.0 to 1.1

- Sections 2.2 and 4.1 now explicitly mention the *hermetic sponge strategy*.
- Chapter 6 proposes additional usage ideas for KECCAK, including input formatting and diversification (see Section 6.3), and parallel and tree hashing (see Section 6.4). Section 6.1 now also mentions a slow one-way function.
- New techniques and updated implementation figures are added to Chapter 7.
  - Section 7.2.1 presents the lane complementing transform.
  - Section 7.2.2 shows how to use the bit interleaving technique.
  - Section 7.3 has been reorganized to show the results on the two platforms side by side.
  - Section 7.3.1 display updated performance figures.
  - Sections 7.4.1 and 7.4.3 report updated performance figures on ASIC and new figures on FPGA.