Keccak sponge function family
main document

Guido Bertoni[1]
Joan Daemen[1]
Michaël Peeters[2]
Gilles Van Assche[1]

http://keccak.noekeon.org/

Version **2.1**
June 19, 2010

[1]STMicroelectronics
[2]NXP Semiconductors

Contents

Chapter 1

Introduction

KECCAK [11] is a family of cryptographic hash functions [86] or, more accurately, sponge functions [9]. 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 [68].

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 [11], to which we assume the reader has access. While the KECCAK definition is updated once (for the SHA-3 2nd round), this present document is regularly updated, so we suggest the reader to obtain the latest version from our website `http://keccak.noekeon.org/`. Note that this document comes with a set of files containing results of tests and experiments. Moreover, also available from `http://keccak.noekeon.org/` is KECCAKTOOLS [14], a public software for aimed at helping analyze KECCAK.

The document is organized as follows. The design choices behind the KECCAK sponge functions are summarized in Chapter 2. Chapter 3 looks at the use of the sponge construction in our submission. Chapter 4 gives an overview of the different modes of use of sponge functions that go beyond plain hashing and discusses backwards compatibility with old standards. Chapter 5 gives more insight on the use of an iterated permutation in the sponge construction and introduces our *hermetic sponge* design strategy. The subsequent three chapters are dedicated to the permutations underlying KECCAK: KECCAK-f.

- Chapter 6 explains the properties of the building blocks of KECCAK-f and motivates the choices made in the design of KECCAK-f.
- Chapter 7 is dedicated to trail propagation in KECCAK-f.
- Chapter 8 covers all other analysis of KECCAK-f respectively.

Finally, Chapter 9 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 $n_\mathrm{r} = 12 + 2\ell$, where $2^\ell = b/25$. This gives 24 rounds for KECCAK-$f[1600]$.

KECCAK-$f[b](A)$
  for i in $0 \dots n_\mathrm{r} - 1$
    $A = \mathrm{Round}[b](A, \mathrm{RC}[i])$
  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.

$\mathrm{Round}[b](A, \mathrm{RC})$
  θ 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 $0 \dots 4$
    $D[x] = C[x-1] \oplus \mathrm{ROT}(C[x+1], 1)$, $\forall x$ in $0 \dots 4$
    $A[x,y] = A[x,y] \oplus D[x]$, $\forall (x,y)$ in $(0 \dots 4, 0 \dots 4)$

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

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

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

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||$0x01$||\text{byte}(d)||\text{byte}(r/8)||$0x01$||$0x00$||\dots||$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 [68, Section 2.B.1]
  - The complete specifications can be found in [11].
  - 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 Chapters 3, 5, 6, 7 and 8.
  - Tunable parameters: a summary is provided in Section 2.4, with pointers to sections with more details.
  - Recommended value for each digest size: see [11, Section 1] for the number of rounds and [11, 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 6.4.
- Requirements in [68, Section 2.B.2]
  - The estimated computational efficiency can be found in Chapter 9.
  - A description of the platforms used to generate the estimates can be found in Section 9.3.1.
  - The speed estimate on the reference platform can also be found in Section 9.3.1.
- Requirements in [68, Section 2.B.3]
  - The known answer and Monte Carlo results can be found on the optical media.
- Requirements in [68, Section 2.B.4]
  - The expected strength of KECCAK is stated in [11, Section 3].
  - The link between the security claim and the expected strength criteria listed in [68, Section 4.A] can be found in Section 4.1.2. More details can be found in Sections 3.1.3, 3.1.4, 3.1.5 and 5.7.
  - For HMAC specifically, see also Section 4.2.3.
  - Other pseudo random functions (PRF) constructions: some modes of use are proposed in Section 4.1.
- Requirements in [68, 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 [68, 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 [20] with us, Dmitry Khovratovich for discussing with us the results published in [54] and for his analysis in [2], Jean-Philippe Aumasson for his analysis in [2] and [3], Joel Lathrop for his analysis in [62] and Willi Meier for his analysis in [3], Anne Canteaut and Christina Boura for their analysis in [22, 21], Paweł Morawiecki and Marian Srebrny for their analysis in [66], Joachim Strömbergson for useful comments on the FPGA implementation, Joppe Bos for reporting a bug in the optimized implementation, all people who contributed to implementations or benchmarks of KECCAK in hardware or software, Virgile Landry Nguegnia Wandji for his work on DPA-resistant KECCAK implementations, 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. Finally we would like to thank *agentschap voor Innovatie door Wetenschap en Technologie* (IWT) for funding two of the authors (Joan Daemen and Gilles Van Assche).

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 [11] 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 [9, 10][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 5.1).

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 [8] and Grindahl [63] 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, a reseedable pseudorandom bit generator and a mask generating function (see Section 4.1).

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 6).

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 [38] and RIJNDAEL [39], 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 9.3) and a very compact co-processor circuit (see Section 9.4.3) 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 9.3.4) and hardware (see Section 9.4.5) that are not suited for most other nonlinear functions [12].

Matryoshka structure The analysis of small versions is relevant for larger versions (see Section 6.2).

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.

- $c = 2n$: for the fixed-output-length candidates, we chose a capacity equal to twice the output length n. This is the smallest capacity value such that there are no generic attacks with expected complexity below 2^n. See Section 3.3.1.
- $b = 1600$: The width of the KECCAK-f permutation is chosen to favor 64-bit architectures while supporting all required capacity values using the same permutation. See Section 3.3.2.
- Parameters for KECCAK[]: for the variable-output-length candidate KECCAK[], we chose a rate value that is a power of two and a capacity not smaller than 512 bits and such that their sum equals 1600. This results in $r = 1024$ and $c = 576$. This capacity value precludes generic attacks with expected complexity below 2^{288}. A rate value that is a power of two 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.
- $n_r = 24$: The value of n_r has been chosen to have a good safety margin with respect to even the weakest structural distinguishers and still have good performance. See Section 6.4.

2.5 The difference between version 1 and version 2 of KECCAK

For the 2nd round of the SHA-3 competition, we decided to modify KECCAK. There are basically two modifications: the increase of the number of rounds in KECCAK-f and the modification of the rate and capacity values in the four fixed-output-length candidates for SHA-3:

- Increasing the number of rounds of KECCAK-f from $12 + \ell$ to $12 + 2\ell$ (from 18 to 24 rounds for KECCAK-f[1600]): this modification is due to the distinguishers described in [3] that work on reduced-round variants of KECCAK-f[1600] up to 16 rounds. In the logic of the hermetic sponge strategy (see Section 5.1.1), we want the underlying permutation to have no structural distinguishers. Sticking to 18 rounds would not contradict this strategy but would leave a security margin of only 2 rounds against a distinguisher of KECCAK-f. In addition, we do think that this increase in the number of rounds increases the security margin with respect to distinguishers of the resulting sponge functions and attacks against those sponge functions.
- For applications where the bitrate does not need to be a power of 2, the new parameters of the fixed-output-length candidates take better advantage of the performance-security trade-offs that the KECCAK sponge function allows.

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 [9] and proven that it is indifferentiable from a random oracle in [10].

3.1.1 Indifferentiability from a random oracle

In [10] 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.

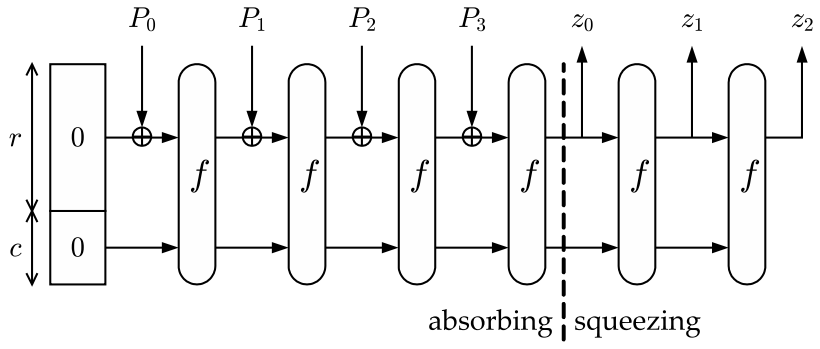

Figure 3.1: The sponge construction

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 [10] 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 [11, 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 [10] 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 [11]. This padding satisfies the conditions imposed by the indifferentiability proof in [10]. 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 [51],
- second preimages on n-bit hash functions for much less than 2^n work [53],
- herding hash functions and the Nostradamus attack [59].

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 5.3.

3.1.4 Randomized hashing

Interesting in this context is the application of randomized hashing [68]. 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 [9, 10]: 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 [9, 10], 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 $c = 2n$ for SHA3-n makes the sponge construction as strong as a random oracle with respect to the security requirements as specified in [68]. In particular, $c = 2n$ is needed by the requirement that (second) preimage resistance should be at least 2^n.

For our candidate with default parameters KECCAK[] we have chosen a capacity of 576 bits. For this choice, the expected workload of an attack should be 2^{288} calls to the underlying permutation. Note that requiring a resistance of 2^{288} is quite strong, as according to laws

of thermodynamics the energy needed by an irreversible computer to perform that many operations is unreachable [78, 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: $f_i(M) =$ KECCAK$[](M||C_i)$ or $f_i(M) =$ KECCAK$[](C_i||M)$. Note that for an appropriate set of constants C_i, Theorem 1 can be extended to this mode of use.

For a more detailed discussion of these aspects we refer to our Note on KECCAK parameters and usage [15].

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 [9], the classical attacks can be executed as a combination of these operations. In [9] 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 [9] 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 [9]. In [17] we have proven upper bounds for the success probability of recovering the state from an output sequence for the case that there is a single solution. We refer to [17] for the detailed statements. In short, the success probability of a passive attack for typical values of the bitrate upper bounded by $N2^{-c}$ with N the number of queries to f. The success probability of an active attack is upper bounded by $N\ell 2^{-c}$ with ℓ the number of calls to f in the sponge instance under attack. Both success probabilities are far below the success probability of differentiating the sponge construction from a random oracle and allow to adopt smaller capacity values in keyed modes of use. Moreover, as recovery of a pre-image implies state recovery, in applications where one-wayness is the sole requirement, here also a smaller capacity can be adopted.

Chapter 4

Usage

This chapter discusses the KECCAK sponge functions from a users' point of view. Note that the explanations are given for KECCAK, but most of the material treated in this chapter applies to any sponge function.

4.1 Usage scenario's for a sponge function

4.1.1 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 4.3 for a related discussion.

4.1.2 Linking to the security claim

Basically, the security claim in [11, 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 satisfies $n = c/2$. 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	$t = \text{MAC}(k, [\text{IV},]m)$	$k\|\|\text{IV}\|\|m$	$\lfloor z \rfloor_n$
Random-access stream cipher (n-bit block)	$z_i = f(k, \text{IV}, i)$	$k\|\|\text{IV}\|\|i$	$\lfloor z \rfloor_n$
Stream cipher	$z = f(k, \text{IV})$	$k\|\|\text{IV}$	as is
Mask generating and key derivation function	$\text{mask} = f(\text{seed}, l)$	seed	$\lfloor z \rfloor_l$
Deterministic random bit generator (DRBG)	$z = \text{DRBG}(\text{seed})$	seed	as is
Reseedable pseudo-random bit generator	see [17]		
Slow n-bit one-way function	$h = H(m)$	m	$z_{N \ldots N+n-1}$
Tree and parallel hashing	see Section 4.4		

Table 4.1: Examples of usage scenario's for a sponge function

4.1.3 Examples of modes of use

In Table 4.1, we propose some possible modes of use of a sponge function.

The first five examples of Table 4.1 can be applied with any member of the KECCAK family, while the last ones, 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 hash function instances (hash function instance differentiation, see also Section 4.3) and to compute a message authentication code (MAC).

The random-access stream cipher works similarly to the SALSA20 family of stream ciphers [6]: 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 pseudo-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. Often pseudo-random bit generator should support re-seeding, i.e., injecting new seed material without throwing away its current state. This can be implemented quite efficiently with the sponge construction and is the subject of our paper [17].

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.

4.2 Backward compatibility with old standards

4.2.1 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.

4.2.2 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 [5], this initial value shall just be pre-pended to the regular input.

4.2.3 HMAC

HMAC [5, 73] 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 4.2.1 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 4.1, which requires only one application of the sponge function.)

From the security claim in [11, 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.

4.2.4 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 [50] 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 [60] also requires a hash function for a mask generating function (MGF1).
- The key derivation functions in NIST SP 800-108 [74] rely on HMAC.
- The key derivation functions in NIST SP 800-56a [70] are generically based on a hash function.
- The digital signature standard (DSS) [67] 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 [69], 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 [71] is based on either a hash function or on HMAC.

4.3 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 [42]) 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 [11], out of the 256 possible values for the diversifier d only 5 have been assigned. 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 [11], 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) [45], similarly to what is done for XML [83]. The namespace name is encoded in UTF-8 [44] 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 [11], 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.

4.4 Parallel and tree hashing

Tree hashing (see, e.g., [65, 76]) 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 c bits of output. 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.

4.4.1 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 1. 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 8 bits and $B/8 \in [1 \dots 2^{16} - 1]$. In addition, $H \geq 1$ when $G = \text{FNG}$.

Algorithm 1 $\text{encode}_{\texttt{http://keccak.noekeon.org/tree/}}(P, m, v, A)$

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 $G = \text{FNG})$
  $M = M||\text{enc}(H, 8)||\text{enc}(D, 8)||\text{enc}(B/8, 16)$
  $M = M||\text{enc}(1, 8)$
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 2.

The processing at each node is defined in Algorithm 3. The output of the KeccakTree scheme is the output of the final node obtained by calling $\text{Node}(0, 0)$.

Algorithm 2 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**
  $j = \left\lfloor \frac{i}{B} \right\rfloor \bmod L$
  Append bit i of M to L_j
end for

Algorithm 3 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**
    $Z = Z || \text{Node}(h + 1, j + iD^{H-h-1})$
  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 $G = \text{LI}$) **do**
    $Z = Z || \text{Node}(1, iD^{H-1})$
  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

4.4.2 Soundness

In [13], we define a set of four conditions for a tree hashing mode to be *sound*. Here soundness is defined in the scope of the indifferentiability framework [64]. The advantage in differentiating a sound tree hashing mode from an ideal monolithic hash function is upper bounded by $q^2/2^{n+1}$ with q the number of queries to the underlying hash function and n the length of the chaining values.

The mode used by KECCAKTREE satisfies the four following conditions, hence is sound. For the terminology, please refer to [13].

- The mode is tree-decodable. The structure of the tree is entirely determined by the parameters encoded in the final node, except for the degree of the final node when $G = \text{FNG}$. The degree of the final node can be determined from the length of its input.
- The mode is message-complete, as Algorithm 2 assigns each input message bit to a leaf. The length of the message can be determined from the length of the leaf nodes.
- The mode is parameter-complete, as Algorithm 1 encodes all tree parameters in the final node.
- The mode enforces domain separation between final and inner nodes. Algorithm 1 encodes whether the node is final or not in the last byte.

4.4.3 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 ($G = \text{LI}$) with $H = 1$ and D equal to or greater than 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. (The performance of such a configuration is discussed in Section 9.3.3.)

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 $G = \text{LI}$ 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.

From the soundness of the construction, 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 [11, Section 4] as they all have an output size n that satisfies $n = c/2$. 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 5

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.

5.1 The philosophy

5.1.1 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.

5.1.2 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 [24] and later an alternative proof was given in [64]. 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 [24] and [64], 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.

5.1.3 The choice between a permutation and a transformation

As can be read in [9], 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 $2^{c/2}$.

5.1.4 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 [9] 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 [57]. 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.

5.2 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.

5.2.1 Differential cryptanalysis

A (XOR) *differential* over a function α consists of an input difference a' and an output difference b' and is denoted by a couple (a', b'). A pair *in* 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 pair in a differential 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 pairs it contains and its differential probability (DP) is the cardinality divided by the total number of pairs with given input difference. We define the (restriction) weight of a differential $\mathrm{w_r}(a', b')$ as minus the binary logarithm of its DP, hence we have $\mathrm{DP}(a', b') = 2^{-\mathrm{w_r}(a', b')}$. The set of values a with a a member of a pair in 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 $\mathrm{w_r}(a', b')$ independent binary equations.

It is well known (see, e.g., [40]) 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 $\lambda = 1/2$ [40]. 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 $n_\mathrm{r} + 1$ differences $(q_0, q_1, \ldots, q_{n_\mathrm{r}})$. 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 pair in 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 pairs it contains and its DP is the cardinality divided by the total number of pairs with given input difference. We define the (restriction) weight of a differential trail $\mathrm{w_r}(Q)$ 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 $\mathrm{DP}(Q) = 2^{-\mathrm{w_r}(Q)}$ is not necessarily true, although in many cases it may be a good approximation, e.g., when $\mathrm{w_r}(Q) < b - 4$. The cardinality of the trail is then given by $2^{b-1} \times \mathrm{DP}(Q)$. Now, when $\mathrm{w_r}(Q) > b - 1$, we cannot have $\mathrm{DP}(Q) = 2^{-\mathrm{w_r}(Q)}$ as the number of pairs is an integer. Typically, a trail with $\mathrm{w_r}(Q) > b - 1$ 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 trails with non-zero cardinality will only have a single pair. In other words, trails containing more than a single pair will be rare. In those circumstances, finding a trail with non-zero cardinality is practically equivalent to finding a pair in 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 differences 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 differences 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 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} [40]. 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 differences 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.

5.2.2 Linear cryptanalysis

A (XOR) *correlation* over a function α, defined by a linear mask v at the input and a linear mask u at the output 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 $v^{\mathrm{T}}a = \sum v_i a_i$ and $u^{\mathrm{T}}b = \sum u_i b_i$ with $b = \alpha(a)$ and the summations taken over GF(2). This correlation is a real number in the interval $[-1, 1]$. We define the (correlation) weight of a correlation by:

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

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 [4]. 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 [34].

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} [40]. 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 $n_\mathrm{r}+1$ masks $(q_0, q_1, \ldots, q_{n_\mathrm{r}})$. 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 correlation weight of a linear trail by

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

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.

5.2.3 Algebraic expressions

In this secton we discuss distinguishers exploiting particular properties of algebraic expressions of iterated mappings, more particular that of 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 $\mathrm{GF}(2^d)$. The derivations are very similar, the only difference is that the coefficients are in $\mathrm{GF}(2^d)$ 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 4.

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 $\{0, 1\}$ [43]. 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 $\mathrm{GF}(2)^b$.

Algorithm 4 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

If b is small, the ANF of the permutation f can be computed explicitly by varying the b bits of input and applying Algorithm 4. A statistical test on the ANF of the output bit 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 [43].

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:

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

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 $i \geq n \Rightarrow e_i = 0$.

A powerful type of attack that exploits algebraic expressions with a low degree are *cube attacks*, recently introduced in [41]. Cube attacks recover secret bits from polynomials that take as input both secret and tweakable public variables. Later *cube testers* were introduced in [1], that detect nonrandom behaviour rather than perform key extraction and can attack cryptographic schemes described by polynomials of relatively high degree. Cube testers are very well suited for building structural distinguishers.

5.2.4 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 [33].

5.2.5 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.

5.2.6 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 [88] 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 ($G \approx 0.624$).

5.3 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 pair $\{a, a^*\}$ with $\widehat{a} = \widehat{a^*} = 0^c$ in 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 $0^c = \widehat{S_f[\text{empty string}]}$.

5.3.1 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 pair (a, a^*) in this trail, the intermediate values of a satisfy $\mathrm{w_r}(Q)$ conditions. If $\mathrm{w_r}(Q)$ 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 pairs $\{a, a \oplus a'\}$ in it with given $\widehat{a}$. We consider two cases:

- $\mathrm{w_r}(Q) < r$: it is likely that the trail contains pairs 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, $a \neq a^*$.
- $\mathrm{w_r}(Q) \geq r$: the probability that the trail contains a pair with $\widehat{a} = 0^c$ is $2^{r-\mathrm{w_r}(Q)}$.

If several trails are available, one can extend this attack by trying it for different trails until a pair in one of them is found with $\widehat{a} = 0^c$. 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 the trail in the second block contains a 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 $\mathrm{w}(\mathcal{Y}) \approx \mathrm{w_r}(Q) - r$: 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 pair can be found that is in the trail, a CICO problem must be solved with $\mathrm{w}(\mathcal{X}) = c$ and $\mathrm{w}(\mathcal{Y}) \approx \mathrm{w_r}(Q) - r$ for determining the first block of the inputs.

Note that if there are no trails with weight below b, the expected number of pairs per trail is smaller than 1 and trails containing more than a single pair will be rare. In this case, even if a trail with non-zero cardinality 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 $\lceil c/r \rceil$.

5.3.2 Exploiting a differential

In the search for inner collisions, all pairs $(a, a \oplus a')$ with $\widehat{a} = 0^c$ in 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 it contains, 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 pairs in 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 pairs in a differential is expected to be a set that has no simple algebraic characterization and we expect the most efficient way to determine pairs in a differential is to try different outputs of f with the required difference and computing the corresponding inputs.

5.3.3 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

5.4 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 $e = \lceil r/c \rceil$.

5.5 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.

5.6 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 $\lceil b/r \rceil$.

5.7 Classical hash function criteria

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

5.7.1 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 5.3 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.

5.7.2 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 5.4 and 5.6, 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 $\mathrm{w}(\mathcal{Y}) = n$.

5.7.3 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 [9], which is discussed in Section 5.4. 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 5.7.2.

5.7.4 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 5.6. 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.

5.7.5 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: $\text{PRF}[k](M) = \text{sponge}(k||M)$. As explained in Section 4.1, 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}.

5.7.6 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 6

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 to motivate certain design choices. Subsequently, we briefly discuss the applicability of a number of cryptanalytic techniques to KECCAK-f.

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

6.1 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:*

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

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 $2^{2^d 25} - 2^{2^{d-1} 25}$.

6.2 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 6.5.2 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.

6.3 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 z 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

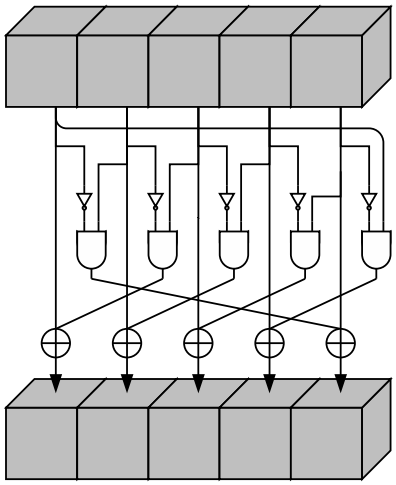

Figure 6.1: χ applied to a single row

denote by 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 9.2.2).

6.3.1 Properties of χ

Figure 6.1 contains a schematic representation of χ and Algorithm 5 its pseudocode.

Algorithm 5 χ

```
for y = 0 to 4 do
  for x = 0 to 4 do
    A[x, y] = a[x, y] ⊕ ((NOT a[x + 1, y]) 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 6.3.1.1 and Section 6.3.1.2 and refer to [34, 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 [34, 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 [8], PANAMA [35] and several other ciphers [34]. 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.

6.3.1.1 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 [33] with $2^{\mathrm{w_r}(a',b')}$ elements. Moreover, the cardinality of a differential (a', b') over χ is either zero or a power of two. The corresponding (restriction) weight $\mathrm{w_r}(a', b') = \mathrm{w_r}(a')$ is an integer that only depends on the input difference a'. A possible differential imposes $\mathrm{w_r}(a')$ linear conditions on the bits of input a.

We now provide a recipe for constructing the affine variety of output differences corresponding to an input difference, 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 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 $\delta(i, i+1)$.

This algorithm is implemented in KECCAKTOOLS [14]. The (restriction) weight of a difference is equal to its Hamming weight plus the number of patterns 001. The all-1 input difference 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 differences, 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 $a_{i+1} = B_i$.

The generation of these conditions given a differential trail is implemented in KECCAKTOOLS [14].

6.3.1.2 Correlation properties

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

We now provide a recipe for constructing the affine variety of input masks corresponding to an output mask, 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 mask. First initialize the offset to 0 and the basis to the empty set. Then for each of the 1-runs $a_s a_{s+1} \dots a_{s+\ell-1}$ 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.
- If $a_i a_{i+1} = 10$ extend the basis with $\delta(i+1)$ and $\delta(i+2)$.

This algorithm is implemented in KECCAKTOOLS [14]. The (correlation) weight of a mask is equal to its Hamming weight plus the number of 1-runs of odd length. The all-1 output mask 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 mask, 10 have weight 2 and 21 have weight 4.

6.3.2 Properties of θ

Figure 6.2 contains a schematic representation of θ and Algorithm 6 its pseudocode.

Algorithm 6 θ

```
for x = 0 to 4 do
    C[x] = a[x, 0]
    for y = 1 to 4 do
        C[x] = C[x] ⊕ a[x, y]
    end for
end for
for x = 0 to 4 do
    D[x] = C[x − 1] ⊕ ROT(C[x + 1], 1)
    for y = 0 to 4 do
        A[x, y] = a[x, y] ⊕ 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 6.5.3 for a more detailed treatment of this.

In fact, 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.

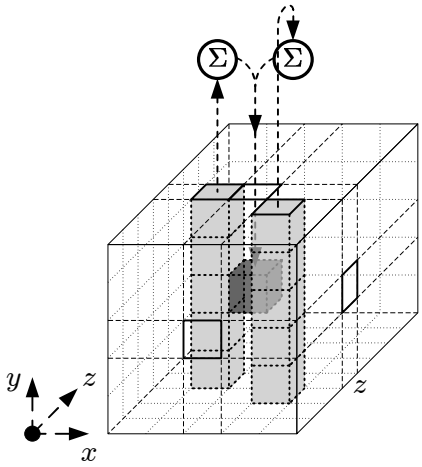

Figure 6.2: θ applied to a single bit

6.3.2.1 The inverse mapping

Computing the inverse of θ can be done by adopting a polynomial notation. 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 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 :

$$1 + \bar{y}\left(x + x^4 z\right) \text{ with } \bar{y} = \sum_{i=0}^{4} y^i = \frac{1+y^5}{1+y} \ . \tag{6.1}$$

The inverse of θ corresponds with the multiplication by the polynomial that is the inverse of polynomial (6.1). For $w = 64$, we have computed this with the open source mathematics software SAGE [80] 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:

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

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

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

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 $x^4z = t^{193}$, 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 with a single active bit results in a difference with about half of the bits active. Similarly, a mask at the output of θ^{-1} determines a mask at its input with about half of the bits active.

6.3.2.2 Propagation of linear masks

A linear Boolean function defined by a linear mask u at the output of a linear function has non-zero correlation to a single linear Boolean function at its input. Given the matrix representation of the linear function, it is easy to express the relation between the input and output mask. Given $b = Ma$, we have:

$$u^{\mathrm{T}}b = u^{\mathrm{T}}Ma = (M^{\mathrm{T}}u)^{\mathrm{T}}a \ .$$

It follow that $u^{\mathrm{T}}b$ is correlated to $v^{\mathrm{T}}a$ with $v = M^{\mathrm{T}}u$ with correlation 1. We say that a linear mask u at the output of a linear mapping M *propagates to* $v = M^{\mathrm{T}}u$ at its input. We denote the mapping defined by M^{T} the *transpose* of M.

As θ is linear, we have $v = \theta^{\mathrm{T}}(u)$, with u a linear mask at the output of θ, v a linear mask at its input and where θ^{T} the transpose of θ. We now determine the expression for the transpose of θ in the formalism of [11]. Let $b = \theta(a)$ and

$$\sum_{x,y,z} u[x][y][z]b[x][y][z] = \sum_{x,y,z} v[x][y][z]a[x][y][z] \ .$$

Filling in the value of $b[x][y][z]$ from the specification of θ in [11] and working this out yields:

$$\sum_{x,y,z} u[x][y][z]b[x][y][z] =$$

$$\sum_{x,y,z} \left(u[x][y][z] + \sum_{y'} u[x+1][y'][z] + \sum_{y'} u[x-1][y'][z+1] \right) a[x][y][z]$$

It follows that:

$$v = \theta^{\mathrm{T}}(u) \Leftrightarrow v[x][y][z] = u[x][y][z] + \sum_{y'} u[x+1][y'][z] + \sum_{y'} u[x-1][y'][z+1] \qquad (6.2)$$

In polynomial notation the application of θ^{T} is a multiplication by

$$1 + \bar{y}\left(x^4 + xz^4\right) \ .$$

Algorithm 7 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

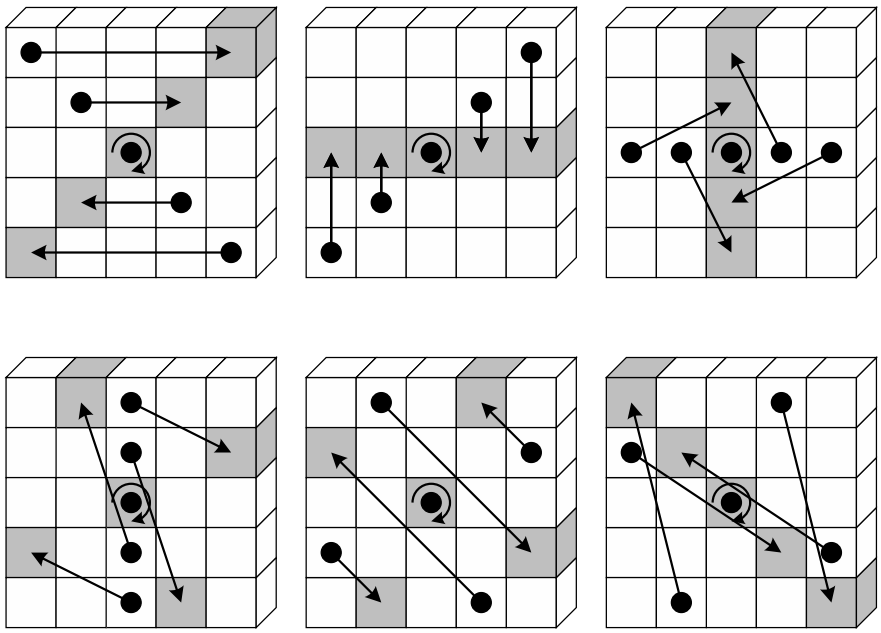

Figure 6.3: π applied to a slice. Note that $x = y = 0$ is depicted at the center of the slice.

6.3.3 Properties of π

Figure 6.3 contains a schematic representation of π and Algorithm 7 its pseudocode.
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;
- $y = x$ axis: rising 1-slope;

	$x=3$	$x=4$	$x=0$	$x=1$	$x=2$
$y=2$	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 6.1: The offsets of ρ

- $y=-x$ 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 6.5.6. Implementing π in hardware requires no gates but results in wiring.

As π is a linear function, a linear mask u at the output propagates to the linear mask v at the input with $v=\pi^{\mathrm{T}}(u)$ (see Section 6.3.2.2). Moreover, we have $\pi^{\mathrm{T}}=\pi^{-1}$, yielding $u=\pi(v)$. This follows directly from the fact that π is a bit transposition and that subsequently its matrix is orthogonal: $M^{\mathrm{T}}M=I$.

6.3.4 Properties of ρ

Figure 6.4 contains a schematic representation of ρ, while Table 6.1 lists its translation offsets. Algorithm 8 gives pseudocode for ρ.

Algorithm 8 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

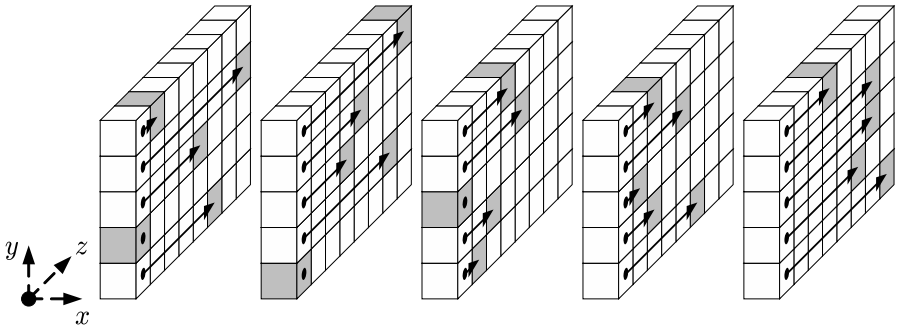

Figure 6.4: ρ applied to the lanes. Note that $x = y = 0$ is depicted at the center of the slices.

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[8], PANAMA [35] and STEPRIGHTUP [34]. In hardware its computational cost corresponds to wiring.

As ρ is a linear function, a linear mask u at the output propagates to the linear mask v at the input with $v = \rho^{\mathrm{T}}(u)$ (see Section 6.3.2.2). Moreover, we have $\rho^{\mathrm{T}} = \rho^{-1}$, yielding $u = \rho(v)$. This follows directly from the fact that ρ is a bit transposition and that subsequently its matrix is orthogonal: $M^{\mathrm{T}}M = I$.

6.3.5 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.

6.3.6 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.

6.4 Choice of parameters: the number of rounds

We here provide our estimate for our SHA-3 candidates specified in [11, Section 4] of how many rounds in KECCAK-f[1600] are sufficient to provide resistance against four types of distinguishers or attacks:

- Construction of structural distinguisher for KECCAK-f[1600]: 21 rounds.
- Construction of structural distinguisher for any of the five candidates: 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 and the following two chapters and the third-party analysis in [2, 62, 3, 22, 21, 66]. By having 24 rounds in KECCAK-f[1600], we take a security margin even with respect to the weakest known structural distinguishers of KECCAK-f as described in [3, 22, 21]. In addition, we take a high security margin with respect to structural distinguishers of the KECCAK sponge function and the two types of attack against the five candidates.

6.5 Differential and linear cryptanalysis

6.5.1 A formalism for describing trails adapted to KECCAK-f

The propagation of differential and linear trails in KECCAK-f is very similar. Therefore we introduce a formalism for the description of trails that is to a large extent common for both types of trails. Differential trails describe the propagation of differences through the rounds of KECCAK-f and linear trails the propagation of masks. We will address both with the term *patterns*.

As explained in Section 6.3.1, for a given difference a at the input of χ, the set of possible output differences is a linear affine variety. For a given mask a at the *output* of χ, the set of input masks with non-zero correlation to the given output mask is also a linear affine variety. Hence, to make the pattern propagation similar, for differential trails we consider the propagation from input to output and for linear trails we consider the propagation from output to input.

A difference at the input of χ is denoted by a_i and we call it a pattern *before* χ (in round i). A difference at the output of χ is denoted by b_i and we call it the pattern *after* χ. Similarly, a mask at the output of χ is denoted by a_i and we call it a pattern *before* χ. A mask at the input of χ is denoted by b_i and we call it the pattern *after* χ. In both cases we denote the linear affine variety of possible patterns after χ compatible with a_i by $\mathcal{B}(a_i)$.

Thanks to the fact that χ is the only nonlinear step in the round, a difference b_i after χ fully determines the difference a_{i+1} before χ of the following round: we have $a_i = \pi(\rho(\theta(b_i)))$. We denote the linear part of the round by λ, so:

$$\lambda = \pi \circ \rho \circ \theta \ .$$

Similarly, a mask b_i after χ fully determines the mask a_{i+1} before the χ of the following round. Now we have $a_i = \theta^{\mathrm{T}}(\rho^{\mathrm{T}}(\pi^{\mathrm{T}}(b_i))) = \theta^{\mathrm{T}}(\rho^{-1}(\pi^{-1}(b_i)))$. Here again, we denote this linear transformation by λ, so in this case we have:

$$\lambda = \theta^{\mathrm{T}} \circ \rho^{-1} \circ \pi^{-1} \ .$$

Note that the way $\mathcal{B}(a_i)$ is formed depends on whether we consider differential or linear trails. Moreover, the meaning of λ depends on whether we consider differential or linear trails.

Consider now the set obtained by applying λ to all elements of $\mathcal{B}(a_i)$. Thanks to the linearity of λ this is again a linear affine variety and we denote it by $\mathcal{A}(a_i)$.

We now define a ℓ-round *trail* Q by a sequence of state patterns a_i with $0 \leq i \leq \ell$. Every a_i denotes a state pattern before χ and a_i must be *compatible with* a_{i-1}, i.e., $a_i \in \mathcal{A}(a_{i-1})$. We use b_i to denote the patterns after χ, i.e., $a_{i+1} = \lambda(b_i)$. So we have:

$$a_0 \xrightarrow{\chi} b_0 \xrightarrow{\lambda} a_1 \xrightarrow{\chi} b_1 \xrightarrow{\lambda} a_2 \xrightarrow{\chi} b_2 \xrightarrow{\lambda} \dots a_\ell. \tag{6.3}$$

The restriction weight of a differential trail Q is the number of conditions it imposes on the absolute values on the members of a right pair. It is given by

$$\mathrm{w_r}(Q) = \sum_{0 \leq i < \ell} \mathrm{w_r}(a_i) \ .$$

Note that the restriction weight of the last difference a_ℓ does not contribute to that of the trail. Hence the weight of any ℓ-round trail is fully determined by its ℓ first differences. For weight values well below the width of the permutation, a good approximation for the DP of a trail is given by $\mathrm{DP}(Q) \approx 2^{-\mathrm{w_r}(Q)}$. If $\mathrm{w_r}(Q)$ is near the width b, this approximation is no longer valid due to the fact that the cardinality of a trail is an integer. While the mapping ι has no role in the existence of differential trails, it does in general impact their DP. For trails with weight above the width, it can make the difference between having cardinality zero or non-zero.

The correlation weight of a linear trail over an iterative mapping determines its contribution to a correlation between output and input defined by the masks a_0 and a_ℓ. The correlation weight of a trail is given by

$$\mathrm{w_c}(Q) = \sum_{0 \leq i < \ell} \mathrm{w_c}(a_i) \ .$$

Here also the correlation weight of a_ℓ does not contribute and hence the weight of any ℓ-round trail is fully determined by its ℓ first masks. The magnitude of the correlation contribution

of a trail is given by $2^{-\mathrm{w_c}(Q)}$. 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.

In our analysis we focus on the weights of trails. As the weight of a ℓ-round trail is determined by its first ℓ patterns, in the following we will ignore the last pattern and describe ℓ-round trail with only ℓ patterns a_i, namely a_0 to $a_{\ell-1}$.

6.5.2 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.

6.5.3 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 $10w$.

6.5.4 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 (a_0) is obtained by taking a difference a_0

with a single active bit and has weight 2. For a linear trail this is obtained by a mask a_0 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 before χ restricted to a row has a single active bit, the same difference is a possible difference after χ. Hence for difference with zero or one active bits per row, χ can behave as the identity. Similarly, for masks 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 a_0.

For all widths, the two-round trails with minimum weight are χ-zero trails. For a differential trail, we choose for a_0 a difference with two active bits that are in the same column. After χ the difference 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 a_1 with two active bits in different rows. As the weight of both a_0 and a_1 is 4, the resulting trail has weight 8. For linear trails, the two active bits in a_0 must be chosen such that after ρ and π they are in the same column. with a similar reasoning it follows that the minimum trail weight is also 8. Note that the low weight of these trails is due to the fact that the difference at the input of θ in round 0 is in the CP-kernel.

6.5.5 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 differences a_0 and a_1 are in the CP-kernel. As in a χ-zero trail χ behaves as the identity and a_0 is in the CP-kernel, we have $a_1 = \pi(\rho(a_0))$. Hence, we can transfer the conditions that a_0 is in the kernel to conditions on a_1, or vice versa.

We will now look for patterns a_0 where both a_0 and $\pi(\rho(a_0))$ are in the CP-kernel. a_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 a_1.

The minimum number of active bits in a_0 is four, where both a_0 and a_1 have two active columns with two active bits each. 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 a_0: $\{0, 1\}$ in one column and $\{2, 3\}$ in another one. In a_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 a_1 and the points $\{1, 2\}$ and $\{3, 0\}$ are in planes in a_0. This implies that projected on the (x, y) plane, the four points of a_0 form a rectangle with horizontal and vertical sides. Similarly, in a_1 they form a parallelogram with vertical sides and sides that are falling 2-slopes.

The (x, y) coordinates of the four points in a_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:

$$\begin{array}{rcl} z_0 + r[x_0][y_2] & = & z_2 + r[x_2][y_2] \mod w \ , \\ z_2 + r[x_2][y_0] & = & z_0 + r[x_0][y_0] \mod w \ , \end{array} \tag{6.4}$$

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

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

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:

$$r[x_0][y_0] - r[x_0][y_2] + r[x_2][y_2] - r[x_2][y_0] = 0 \bmod w \ . \tag{6.5}$$

If this equation is not satisfied, the equations (6.4) have no solution.

Consider now $w = 1$. In that case Equation (6.5) is always satisfied. However, in order to be χ-zero, the points must be in different rows, and hence in different planes, both in a_0 and a_1, and this is not possible for a rectangle.

If $\ell \geq 1$, Equation (6.5) has a priori a probability of $2^{-\ell}$ of being satisfied. Hence, we can expect about $2^{-\ell}100$ rectangles to define a state a_0 with both a_0 and $\pi(\rho(a_0))$ 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 (6.5), there are 2^ℓ patterns a_0, one for each choice of z_0. Second, if Equation (6.5) is not satisfied for ℓ but it is for some $\ell' < \ell$, it implies a pattern a_0 with $2^{\ell-\ell'}4$ points rather than 4 for which both a_0 and $\pi(\rho(a_0))$ are in the kernel.

These patterns can be generalized by extending the number of active bits: a pattern a_0 with both a_0 and $\pi(\rho(a_0))$ 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 a_0 and the points $\{2i+1, 2i+2\}$ are in the same column in a_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:

$$d(V) = \sum_{i=0}^{2e-1} (-1)^i r[\text{point } i]. \tag{6.6}$$

Now, the vortex results in a valid pattern a_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 a_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 a_0 with $2e$ active bits, even for lane length 64.

Moreover, one can construct patterns a_0 containing two or more vortices, provided that they do not result in a row with two active bits in either a_0 or a_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.

6.5.6 Beyond three-round trails: choice of π

We will now try to extend this to four-round trails: we try to find patterns a_0 such that a_0, a_1 and a_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 a_0 with sheets and planes as sides results in a parallelogram in a_1 with falling 2-slopes and columns as sides and in a parallelogram in a_2 with rising 2-slopes and falling 2-slopes as sides. Hence the four points in a_2 cannot group 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 a_0, it is $\{1,2\},\{3,4\},\{5,0\}$ in a_1 and $\{1,4\},\{2,5\},\{3,0\}$ in a_2. The grouping in a_1 simply implies that $\{1,2\},\{3,4\},\{5,0\}$ are grouped in planes in a_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 a_2 on the (x,y) coordinates. Grouping in columns (sheets) in a_2 implies grouping in planes in a_1 and subsequently grouping in rising 1-slopes in a_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 a_2. Similar to Equation (6.4) 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 a_0, a_1 and a_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 a_0 and a_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 a_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 a_1 to be in the kernel imposes that the number of points in each plane of a_0 must be even. This adds 5 parity conditions. However, one is redundant with the ones due to a_0 as the total parity of the activity pattern over the state is even. Hence there are $2^{16} - 1$ such patterns. Additionally requiring a_2 to be in the kernel imposes that the number of points in each rising 1-slope of a_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 a_0, a_1 and a_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 a_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 a_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].

6.5.7 Truncated trails and differentials

Truncated trails deal with the propagation of activity patterns rather than differences [55]. 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 [54] truncated cryptanalysis was applied to RADIOGATÚN [8], 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.

6.5.8 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.

6.5.9 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 [55]: 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 [87]: we expect the KECCAK-f permutations to behave as random permutations. If so, the cardinality of differentials has a Poisson distribution with $\lambda = 1/2$ [40] 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 [61]: 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 $\mathrm{DP} \approx 2^{-\mathrm{w_r}(Q_d)}$. Additionally, if we assume the correlation is dominated by a single low-weight linear trail, we have $\mathrm{DP} \approx 2^{-\mathrm{w_c}(Q_l)}$. This gives for the number of required pairs: $2^{2(\mathrm{w_r}(Q_d)+\mathrm{w_c}(Q_l))}$. The number of required pairs to exploit a trail in a simple differential or linear attack is of the order $2^{\mathrm{w_c}(Q)}$. Hence, over a number of rounds, the differential-linear distinguisher is more powerful than a simple differential or linear distinguisher if $\mathrm{w_r}(Q_d)+\mathrm{w_c}(Q_l) < \mathrm{w_c}(Q)/2$. 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 [84, 52] and rectangle attacks [18]: These attacks chain (sets of) differentials over a small number of rounds to construct distinguishers over a larger number of rounds. These are also likely candidates for good structural distinguishers, for the same reason as differential-linear ones.

- Integral cryptanalysis (Square attacks) [36]: this type of cryptanalysis lends itself very well to ciphers that treat the state in blocks. It was applied to bit-oriented ciphers in [89]. 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.

6.6 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 $w = 1$. We refer to Section 8.2 for the results of some experiments for solving CICO-solving algebraically.

6.7 Strength in keyed mode

In keyed modes we must consider attack scenario's such as explained in Section 5.7.5. 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 6.6. 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.

6.8 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. We refer to Section 8.1.2 for some experimentally obtained evidence of this.

Chapter 7

Trail propagation in KECCAK-f

As explained in Section 5.2, the existence of differential or linear trails in KECCAK-f with a weight below the width of KECCAK-f may result in a structural distinguisher of KECCAK-f. In this chapter we report on our investigations related to lower bounds for weights of such trails.

In Section 7.1 we define different types of weight and in Section 7.2 we discuss the relevant properties of θ. In Section 7.3 we discuss the techniques used to come up with the lower bounds on trail weights and report on the results obtained. Finally, in Section 7.4 we report on experiments that allows us to get an idea what to expect for higher width values.

7.1 Relations between different kinds of weight

In this section, we recall or define the various kinds of weight we use in the sequel. We speak of patterns that may have the shape of a state, slice, plane, sheet, row, column, lane or even a single bit. Some types of weight are only defined for certain shapes and we will indicate if it is the case.

We call a bit equal to 1 in a pattern an *active* bit and a bit equal to zero a *passive* bit.

Definition 6. *The* Hamming weight *of a pattern a is the number of active bits in the pattern and is denoted by $||a||$.*

Definition 7. *The (propagation)* weight *of a pattern a, denoted by $\mathrm{w}(a)$, is a generic term for either the restriction weight or the correlation weight of a pattern before χ. Since χ operates on rows, the pattern a must consist of full rows implying that the weight is only defined for states, slices, planes and rows.*

Note that the size of the linear affine varieties $\mathcal{B}(a_i)$ and $\mathcal{A}(a_i)$ is determined by the propagation weight of a_i:

$$|\mathcal{B}(a_i)| = |\mathcal{A}(a_i)| = 2^{\mathrm{w}(a_i)} \ .$$

These two weights relate to the properties of χ, which are detailed in Section 6.3.1. Since χ operates on each row independently, the restriction and correlation weights of a pattern can be computed row per row and the results are summed. The weights for all row patterns are listed explicitly in Tables 7.1 and 7.2.

Difference a	$\mathrm{w_r}(a)$	$\mathrm{w_r^{rev}}(a)$	$\|\|a\|\|$	$\|a\|_{\mathrm{row}}$
00000	0	0	0	0
10000	2	2	1	1
11000	3	2	2	1
10100	3	2	2	1
11100	4	2	3	1
11010	3	3	3	1
11110	4	3	4	1
11111	4	3	5	1

Table 7.1: Weights of all row differences (up to cyclic shifts)

Mask a	$\mathrm{w_c}(a)$	$\mathrm{w_c^{rev}}(a)$	$\|\|a\|\|$	$\|a\|_{\mathrm{row}}$
00000	0	0	0	0
10000	2	2	1	1
11000	2	2	2	1
10100	4	2	2	1
11100	4	2	3	1
11010	4	2	3	1
11110	4	2	4	1
11111	4	4	5	1

Table 7.2: Weights of all row masks (up to cyclic shifts)

Definition 8. *For a pattern b after χ, we define the* minimum reverse weight $\mathrm{w^{rev}}(b)$ *as the minimum weight over all compatible a. Namely,*

$$\mathrm{w^{rev}}(b) = \min_{a \,:\, b \in \mathcal{B}(a)} \mathrm{w}(a).$$

This weight applies to states, slices, planes and rows.

Given an ℓ-round trail $Q = (a_1 \dots a_\ell)$, it is easy to find the $\ell + 1$-round trail $Q' = (a'_0, a'_1 \dots a'_\ell)$ with minimum weight such that $a_i = a'_i$ for $1 \le i \le \ell$. In this case, $\mathrm{w}(Q') = \mathrm{w}(Q) + \mathrm{w^{rev}}(\lambda^{-1}(a_1))$.

It may also be useful to express the *number of active rows* in a given pattern.

Definition 9. *For a pattern a before (or after) χ, the number of active rows, denoted by $\|a\|_{\mathrm{row}}$ is simply the number of rows whose value is non-zero. This weight applies to states, slices, planes and rows.*

The different kinds of weight for all row patterns are given in Tables 7.1 and 7.2. We now give some relations between the various kinds of weights. The following bounds relate the Hamming weight to the weight:

$$\hat{\mathrm{w}}(||a||) = ||a|| - \left\lfloor \frac{||a||}{5} \right\rfloor + [1 \text{ if } ||a|| = 1 \pmod 5] \le \mathrm{w}(a) \le 2||a||,$$

$$\left\lceil \frac{\mathrm{w}(a)}{2} \right\rceil \le ||a|| \le \left\lfloor \frac{5\mathrm{w}(a)}{4} \right\rfloor.$$

The following bounds relate the number of active rows to the weight:

$$2\|a\|_{\text{row}} \leq \text{w}(a) \leq 4\|a\|_{\text{row}},$$
$$\left\lceil \frac{\text{w}(a)}{4} \right\rceil \leq \|a\|_{\text{row}} \leq \left\lfloor \frac{\text{w}(a)}{2} \right\rfloor.$$

Given the Hamming weight, the minimum reverse restriction weight can be lower bounded as follows:

$$\text{w}_\text{r}^{\text{rev}}(a) \geq \hat{\text{w}}_\text{r}^{\text{rev}}(||a||) = 3\left\lfloor \frac{||a||}{5} \right\rfloor + \begin{cases} 0 & \text{if } ||a|| = 0 \pmod 5, \\ 1 & \text{if } ||a|| = 1 \pmod 5, \\ 2 & \text{if } ||a|| = 2 \pmod 5, \\ 2 & \text{if } ||a|| = 3 \pmod 5, \\ 3 & \text{if } ||a|| = 4 \pmod 5. \end{cases}$$

Given the Hamming weight, the minimum reverse correlation weight can be lower bounded as follows:

$$\text{w}_\text{c}^{\text{rev}}(a) \geq \hat{\text{w}}_\text{c}^{\text{rev}}(||a||) = 2\left\lceil \frac{||a||}{4} \right\rceil.$$

Other relations on the minimum reverse weight follow:

$$\text{w}^{\text{rev}}(b) \leq 2||b||,$$
$$\left\lceil \frac{\text{w}^{\text{rev}}(b)}{2} \right\rceil \leq ||b|| \leq \left\lfloor \frac{5\text{w}^{\text{rev}}(b)}{4} \right\rfloor,$$
$$2\|b\|_{\text{row}} \leq \text{w}^{\text{rev}}(b) \leq 4\|(\|_{\text{row}}b),$$
$$\left\lceil \frac{\text{w}^{\text{rev}}(b)}{4} \right\rceil \leq \|b\|_{\text{row}} \leq \left\lfloor \frac{\text{w}^{\text{rev}}(b)}{2} \right\rfloor.$$

And finally,

$$\text{w}^{\text{rev}}(b) \leq \text{w}(b).$$

Definition 10. *A weight function $f(a)$ is said to be* monotonous *if setting a passive bit of a pattern a to active does not decrease $f(a)$. More formally, let the partial ordering $a \leq a'$ be defined as $a[x][y][z] = 1 \Rightarrow a'[x][y][z] = 1$. Then, f is a monotonous weight if $\forall a, a' : a \leq a'$, we have $f(a) \leq f(a')$.*

All weights defined in this section are monotonous. This follows directly from Tables 7.1 and 7.2.

7.2 Propagation properties related to the linear step θ

Definition 11. *The* column parity *(or* parity *for short) $P(a)$ of a pattern a is a pattern $P(a) = p[x][z]$ defined as the parity of the columns of a, namely $p[x][z] = \sum_y a[x][y][z]$. The parity is defined for states, slices, sheets and columns. The parity of a state has the shape of a plane, the parity of a slice has the shape of a row, the parity of a sheet has the shape of a lane and the parity of a column is a bit.*

Equivalently, the parity is the result of applying the operator $\bar{y}$ (see Section 6.3.2), and the value C in Algorithm 6 is the parity of a state. A column is *even* (resp. *odd*) if its parity is 0 (resp. 1). When the parity of a pattern is zero (i.e., all its columns are even), we say it is in the *CP-kernel*, as in Section 6.5.3. Note that the column parity defines a partition on the set of possible states.

Definition 12. *The* θ-effect *of a state* a *before* θ *is a pattern* $E(a)[x][z]$ *defined as the result of applying the operator* $\bar{y}(x+x^4z)$ *to the state, or equivalently by applying the operator* $(x+x^4z)$ *to its parity, i.e.,* $E(a)[x][z] = P(a)[x-1][z] + P(a)[x+1][z-1]$.

In difference propagation, the θ-effect is also the value of D in Algorithm 6. In mask propagation, the θ-effect is defined by θ^{T} rather than θ itself and the expression becomes $E(a)[x][z] = P(a)[x+1][z] + P(a)[x-1][z+1]$. (see Section 6.3.2.2). Note that the θ-effect always has an even Hamming weight. A column of coordinates (x, z) is *affected* iff $E(a)[x][z] = 1$; otherwise, it is *unaffected*.

The θ-effect entirely determines the effect of applying θ to a state a. For a fixed θ-effect $e[x][z]$, θ just adds a constant pattern $e[x][y][z]$ of bits to the state, constant in each column, namely $e[x][y][z] = e[x][z]$ for all y. Since the θ-effect has even Hamming weight, it means that the number of affected columns is even.

Definition 13. *The* θ-gap *is defined as the Hamming weight of the* θ*-effect divided by two.*

Hence, if the θ-gap of a state at the input of θ is g, the number of affected columns is $2g$ and applying θ to it results in $10g$ bits being flipped.

When a state is in the CP-kernel, the θ-gap is zero. However, the θ-gap is also zero when the parity is all-one, i.e., when all columns have odd parity before θ.

We have defined the θ-gap using the θ-effect, but it can also be defined using the parity itself. For this, we need to represent the parity $p[x][z]$ differently. We do the coordinate transformation mapping the (x, z) coordinates to a single coordinate t as specified in Section 6.3.2.1 (i.e., t goes to $(x, z) = (-2t, t)$) and denote the result by $p[t]$.

In this representation, a *run* is defined as a sequence R of consecutive t-coordinates, $R = \{s, s+1, \ldots, s+n-1\}$, such that $p[s-1] = 0$, $p[t] = 1 \forall t \in R$ and $p[s+n] = 0$. The following lemma links the number of runs to the θ-gap.

Lemma 1. *The parity* p *has* θ*-gap* g *iff* $p[t]$ *has* g *distinct runs.*

7.3 Exhaustive trail search

In Section 7.3.2 we will show an efficient method for generating all two-round trails up to a given propagation weight T_2. We will then show in section Section 7.3.3 how these trails can be extended to more rounds. In general, we want to generate all ℓ-round trails up to some propagation weight T_ℓ. We start with Section 7.3.1 deriving the minimum value of T_2 given ℓ and T_ℓ. Finally, we report on our bounds obtained in Section 7.3.4

7.3.1 Upper bound for the weight of two-round trails to scan

The idea is have bounds on ℓ-round trails by starting from all two-round trails of weight up to T_2 and extending them both forwards and backwards. More precisely, each two-round trail is extended n rounds backwards and $\ell-2-n$ rounds forwards, for each value of $n \in \{0, \ldots, \ell-2\}$. This way, we cover all trails that have a two-round subtrail with weight up to T_2.

Lemma 2. *To list all ℓ-round trails of weight not higher than T_ℓ exhaustively, it is necessary to start from all 2-round trails with weight up to (and including) T_2, with $T_2 = \left\lfloor \frac{2T_\ell}{\ell} \right\rfloor$ if ℓ is even, or $T_2 = \left\lfloor \frac{2(T_\ell - 2)}{\ell - 1} \right\rfloor$ if ℓ is odd.*

Proof. For a sequence of weights $W = w_1, w_2, \ldots, w_\ell$, let $\delta(W) = \min_{i=1\ldots\ell-1}(w_i + w_{i+1})$. We want to make sure that $\delta(W) \leq T_2$ for all W such that $\sum_i w_i \leq T_\ell$.

If ℓ is even, $T_\ell \geq \sum_{i=1}^{\ell} w_i = \sum_{j=1}^{\ell/2}(w_{2j-1} + w_{2j}) \geq \frac{\ell}{2}\delta(W)$ and thus $\delta(W) \leq \frac{2T_\ell}{\ell}$. Setting $T_2 = \left\lfloor \frac{2T_\ell}{\ell} \right\rfloor$ satisfies the condition.

If ℓ is odd, we can always assume that $w_\ell \geq 2$ (as in any non-trivial trail) and thus we can consider the same problem with sequences of $\ell - 1$ weights and $T_{\ell-1} = T_\ell - 2$. □

So we have all the necessary ingredients to make exhaustive search of trails up to a given weight, within the limits of a reasonable computation time, and we can use that to find trails with minimum weight.

7.3.2 Constructing two-round trails

In this section we describe an efficient method for constructing all two-round trails (a_0, a_1) with $\mathrm{w}(a_0) + \mathrm{w}(a_1) \leq T_2$ for a given value T_2. For any such trail, we know that there exists a trail (a_0', a_1) with $\mathrm{w}(a_0') = \mathrm{w}^{\mathrm{rev}}(\lambda^{-1}(a_1)) \leq \mathrm{w}(a_0)$. The quantity $\mathrm{w}^{\mathrm{rev}}(\lambda^{-1}(a)) + \mathrm{w}(a) \leq T_2$ imposes a lower bound on the weight of a 2-round trail trail that has a as its state before χ of the second round and we give it the following name.

Definition 14. *The* propagation branch number *of a state a before χ is denoted by $B_{\mathrm{p}}(a)$ and given by:*

$$B_{\mathrm{p}}(a) = \mathrm{w}^{\mathrm{rev}}(\lambda^{-1}(a)) + \mathrm{w}(a) \ .$$

Hence, rather than explicitly constructing all two-round trails, we can generate the set of states a such that $B_{\mathrm{p}}(a) \leq T_2$. We call this set $\alpha(T_2)$. It contains the second members a_1 of all two-round trails Q with $\mathrm{w}(Q) \leq T_2$. Each state in this set $\alpha(T_2)$ then serves as a starting point for trail extension. For the resulting trails, we know that the subtrail (a_{i-1}, a_i) with $a_i = a$ has a weight of at least $B_{\mathrm{p}}(a)$.

In generating $\alpha(T_2)$ we use a strategy that exploits the monotonicity of the propagation weight and the properties of θ in terms of the Hamming weight of its input/output. In Section 7.1 we listed equations providing lower bounds on the weight and reverse minimum weight of a state as a function of its Hamming weight. Hence, $||a||$ and $||\lambda^{-1}(a)||$ gives a lower bound on $B_{\mathrm{p}}(a)$.

Definition 15. *The* Hamming branch number *of a state a before χ is denoted by $B_{\mathrm{h}}(a)$ and given by:*

$$B_{\mathrm{h}}(a) = ||\lambda^{-1}(a)|| + ||a|| \ .$$

We can now just generate all states up to some given propagation branch number by generating all states up to a sufficiently high Hamming branch number value.

Now let a' be the state before θ corresponding with a. In difference propagation, we have $a' = \lambda^{-1}(a)$ and in mask propagation this is $a' = \theta^{\mathrm{T}^{-1}}(a)$. It follows that in difference propagation, we have $||a'|| = ||\lambda^{-1}(a)||$ and $||\theta(a')|| = ||a||$ and hence $B_{\mathrm{h}}(a) = ||a'|| + ||\theta(a')||$. In mask propagation we can similarly derive $B_{\mathrm{h}}(a) = ||a'|| + ||\theta^{\mathrm{T}}(a')||$. Hence we can instead

move our reference from a to a' and just compute the value of a from a'. With a slight abuse of notation we will write $B_{\mathrm{h}}(a')$ and $B_{\mathrm{p}}(a')$ to denote $B_{\mathrm{h}}(a)$ and $B_{\mathrm{p}}(a)$.

Consider now the set of states a' with a given parity p, i.e., $P(a') = p$. As the θ-effect e is fully determined by the parity, all these states have the same parity effect. It follows that θ is reduced to the addition of a constant value, facilitating the computation of $B_{\mathrm{h}}(a')$ and the deduction of lower bounds on $B_{\mathrm{p}}(a')$.

Similar to the Hamming branch number of a state, we can define the Hamming branch number of a parity.

Definition 16. *The Hamming branch number of a parity p before θ is defined as the minimum Hamming branch number over all states with the given parity:*

$$B_{\mathrm{h}}(p) = \min_{a' \ :P(a')=p} B_{\mathrm{h}}(a') \ .$$

In a state a' we can use its parity p to partition its columns $a'[x][z]$ in four kinds: odd ($P(a')[x][z] = 1$) and even ($P(a')[x][z] = 0$), combined with affected ($E(a')[x][z] = 1$) and unaffected ($E(a')[x][z] = 0$). We can use this to easily compute $B_{\mathrm{h}}(p)$ for any parity p.

- An unaffected odd column has at least one active bit before θ and is preserved after it. Hence, it contributes at least 2 to $B_{\mathrm{h}}(a')$. As this minimal case can be constructed, the contribution to $B_{\mathrm{h}}(p)$ is equal to 2.
- An affected (odd or even) column having n active bits before θ has $5-n$ bits afterwards, hence contributes exactly 5 to $B_{\mathrm{h}}(a')$ and to $B_{\mathrm{h}}(p)$.
- An unaffected even column can have zero active bits, hence does not contribute to $B_{\mathrm{h}}(p)$.

Hence, it turns out that

$$B_{\mathrm{h}}(p) = 5||E(p)|| + 2||p \cdot (\bar{0} + E(p))|| = 10g + 2u_{\mathrm{o}},$$

with g the θ-gap of p, $\cdot$ the componentwise product, $\bar{0}$ the all-1 state and u_{o} the number of unaffected odd columns.

We can now generate all states a' up to some given propagation branch number and with given parity p in two phases.

- In a first phase we generate all states a' with $B_{\mathrm{h}}(a') = B_{\mathrm{h}}(p)$. We call those states *branch-parity-minimal*.
- In a second phase, we can generate states a' that are not branch-parity-minimal by taking branch-parity-minimal states and adding pairs of active bits in columns such that the parity is unchanged.

The generation of branch-parity-minimal states is done as follows:

- For each unaffected odd column, put a single active bit. There are 5 possibilities: one for each positions y.
- For each affected even column, put an even number of active bits. There are 2^4 possibilities.
- For each affected odd column, put an odd number of active bits. There are in 2^4 possibilities.

The number of branch-parity-minimal states a' with given parity p is thus $2^{8g}5^{|p(1+e)|}$.

From the monotonicity of the weights it follows that in the set of states a' with given parity p, the subset of branch-parity-minimal states contain the states that minimize $B_{\mathrm{p}}(a')$. Adding a pair of active bits in a single column of a' leaves its parity intact and thanks to the monotonicity cannot decrease $B_{\mathrm{p}}(a')$. From this, we devise the following strategy to generate all states a' with given parity p with $B_{\mathrm{p}}(a') \leq T_2$.

For each branch-parity-minimal state a' with $B_{\mathrm{p}}(a') \leq T_2$ do the following:

- Output a'.
- Iteratively construct states a' by adding pairs of active bits in each column, as long as $B_{\mathrm{p}}(a') \leq T_2$. To avoid duplicates the active bits shall have y coordinates with larger values than any active bits in the columns.

To generate all states a' for which $B_{\mathrm{p}}(a') \leq T_2$ we must do this for all parities with a small enough θ-gap. Actually, we can compute a lower bound on $B_{\mathrm{p}}(a')$ given only $P(a')$. We then have to consider only those parities for which this bound is lower than or equal to T_2. We compute it in the following way. First, we consider the Hamming branch number $B_{\mathrm{h}}(p)$ and assume that $|\lambda^{-1}(a)| = B_{\mathrm{h}}(p) - n$ and $|a| = n$ for some value n. Then, we use the bounds found in Section 7.1 and minimize over n. Note that we have checked that the minimum is always at $n = 1$, hence:

$$\begin{aligned} B_{\mathrm{p}}(a') &\geq \min_{n \in \{1 \ldots B_{\mathrm{h}}(p)-1\}} \hat{\mathrm{w}}^{\mathrm{rev}}(B_{\mathrm{h}}(p) - n) + \hat{\mathrm{w}}(n) \\ &= \hat{\mathrm{w}}^{\mathrm{rev}}(B_{\mathrm{h}}(p) - 1) + 2 \\ &= \hat{\mathrm{w}}^{\mathrm{rev}}(10g + 2u_{\mathrm{o}} - 1) + 2. \end{aligned}$$

Hence the θ-gap of the parity p imposes a lower bound to the propagation branch number of a state.

Then, it is possible to determine the maximum θ-gap $g_{\max}$ above which the lower bound is above T_2. If we further relate $g_{\max}$ to the number of runs in the parity, as in Lemma 1, we can generate all possible parities we need to consider by generating those with up to $g_{\max}$ runs.

Notice that both χ and λ are invariant by translation along z. It is thus necessary to keep only a single member of the states (or parities) in $\alpha(T_2)$ that are equal modulo the translations along z.

7.3.3 Extending trails

We can now use the elements in $\alpha(T_2)$ to recursively generate longer and longer trails up to some given length and some given weight. We can extend the trails in two directions: forward and backward. Given a trail Q, extending the trail forward (resp. backward) means constructing trails Q' of which Q is a prefix (resp. suffix). It can also be in both directions, i.e., adding a number of steps as a prefix and another number of steps as a suffix of Q.

In the forward direction, the general idea is the following. Given the last state $a_{\ell-1}$ of the trail Q, we characterize the affine space $\mathcal{A}(a_{\ell-1})$ as an offset and a basis, i.e.,

$$\mathcal{A}(a_\ell) = s + \langle t_1, t_2, \ldots, t_{\mathrm{w}(a_\ell)} \rangle \ .$$

We can then loop through the affine space, produce the state values a_ℓ and check the weight $\mathrm{w}(a_\ell)$. If the weight is low enough for the extended trail to be interesting in the search, we can append a_ℓ to Q and recursively continue the search from there if necessary.

In the backward direction, we cannot use an affine space representation of a_{-1} as a function of a_0. As the weight is determined by the to-be-found state a_{-1}, we can list, for each active row of b_0, the possible input rows and their corresponding weight in increasing order. The weight of a_{-1} is the sum of the weights of each individual row, and we can take advantage of this to choose input rows such that the weight stays below or equal to a given threshold. We set each active row to the input row with lowest weight, the total weight $\mathrm{w}(a_{-1})$ being equal to $w^{\mathrm{rev}}(b_0)$ for the output state b_0. Then, we can generate all other input states a_{-1} by looping through the input rows, locally knowing up to which weight we can go.

7.3.4 Linear and differential trail bounds for $w \leq 8$

We have investigated the different instances of KECCAK-$f[b]$ starting from the smallest widths $b = 25w$, resulting in the lower bounds for trail weights listed in Table 7.3. Thanks to the Matryoshka structure, a lower bound on trails for KECCAK-$f[b]$ implies a lower bound on symmetric trails for all larger widths. More specifically, a differential or linear trail for KECCAK-$f[25w]$ with weight W corresponds with a w-symmetric trail for KECCAK-$f[25w']$ with weight $W' = W\frac{w'}{w}$. For instance, in Table 7.3 the column DC, $w = 8$ expresses a lower bound of 46 on the weight of 4-round trails in KECCAK-$f[200]$. This also expresses a lower bound for 4-round symmetric trails in KECCAK-f versions with larger width: 92 for 2-symmetric trails in KECCAK-$f[400]$, 184 for 4-symmetric trails in KECCAK-$f[800]$ and 268 for 8-symmetric trails in KECCAK-$f[1600]$.

For $w = 1$, five rounds are sufficient to have no trails with weight below 25, the width of the permutation. For $w = 2$, 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 the bounds for width 25 and those for width 50 grows. We expect this effect to be similar for larger widths.

For $w = 4$, the search was complete up to weight 36 and 38 for 4 rounds, for differential and linear trails respectively:

- For 4 rounds, the differential trail with minimum weight has weight 30. For the small number of trails found up to weight 36, we checked that these trails cannot be chained together. Hence, this guarantees that a 8-round differential trail has at least weight $36+37 = 73$. For 5 and 6 rounds, the best trails we have found so far have weight 54 and

Number of rounds	DC				LC			
	$w=1$	$w=2$	$w=4$	$w=8$	$w=1$	$w=2$	$w=4$	$w=8$
2	8	8	8	8	8	8	8	8
3	16	18	19	20	16	16	20	20
4	23	29	30	46	24	30	38	46
5	30	42	≤ 54		30	40	≤ 66	
6	37	54	≤ 85		38	52	≤ 94	

Table 7.3: Minimum weight of w-symmetric trails

85, respectively (but these do not provide bounds). For the 16 rounds of KECCAK-f[100], we can guarantee that there are no differential trails of weight below $2 \times 73 = 146$.

- For 4 rounds, the linear trail with minimum weight has weight 38. For 5 and 6 rounds, the best trails we have found so far have weight 66 and 94, respectively (but these do not provide bounds). For the 16 rounds of KECCAK-f[100], we can guarantee that there are no linear trails of weight below $4 \times 38 = 152$.

For $w = 8$, the search was complete up to weight 49 and 48 for 4 rounds, for differential and linear trails respectively:

- For 4 rounds, the differential trail with minimum weight has weight 46. For the small number of trails found up to weight 49, we checked that these trails cannot be chained together. Hence, this guarantees that a 8-round differential trail has at least weight $49 + 50 = 99$. For the 18 rounds of KECCAK-f[200], we can guarantee that there are no differential trails of weight below $2 \times 99 + 8 = 206$.
- For 4 rounds, the linear trail with minimum weight has weight 46. For the small number of trails found up to weight 48, we checked that these trails cannot be chained together. Hence, this guarantees that a 8-round differential trail has at least weight $48 + 50 = 98$. For the 18 rounds of KECCAK-f[200], we can guarantee that there are no linear trails of weight below $8 \times 98 + 8 = 204$.

7.4 Tame trails

In this section we report on our investigations related to the search for 3-round and 4-round differential trails with low weight and for high width. In this context, we consider differential trails for which the intermediate states b_i are in the CP-kernel and call such trails *tame*. Linear trails are expected to behave in qualitatively the same way.

This is a generalization of the kernel vortices introduced in Section 6.5.5 where only patterns are considered for which both χ and θ behave as the identity. In kernel vortices only intermediate patterns at the input of χ where considered with a single active bit per row. In the trails considered in this section, this restriction is no longer present.

7.4.1 Construction of tame trails

Let us start with 3-round differential trails. The weight of such a trail is defined by three states: (a_0, a_1, a_2). This trail is tame if $b_0 = \lambda^{-1}(a_1)$ and $b_1 = \lambda^{-1}(a_2)$ are both in the

CP-kernel. We have written a program that generates all values of a_1 of a given Hamming weight, such that b_0 is in the CP-kernel and there is at least one three-round differential trail (a_0, a_1, a_2) that is tame, i.e., with b_1 in the CP-kernel. The latter condition imposes that a_1 must be such that the intersection of $\mathcal{A}(a_1)$ and the CP-kernel is not empty. As χ operates on rows and the CP-kernel is determined by individual columns, this can be treated slice by slice. In other words, every slice of a_1 must be such that its linear affine variety of possible output patterns contains patterns in the CP-kernel.

We call such a slice *tame* and a state with only tame slices also tame. For slice patterns with few active bits it can be easily verified whether it is tame. A slice with no active bits is tame, a slice with a single active bit can never be tame and a slice with two active bits is tame iff the active bits are in the same column. As the number of active bits grows, the proportion of slice patterns that are not tame decreases exponentially. As there are only 2^{25} different slice patterns, the tameness check can be precomputed and implemented by a simple table-lookup. Generating all states a_1 of a given (small) Hamming weight, that are tame and for which b_0 is in the CP-kernel can be done efficiently.

We can now construct all valid states a_1 as the combination of a number of *kernel chains*. A kernel chain is set of active bits determined by a sequence of bit positions c_i in a_1. A kernel chain forms a set of active bits with the following properties:

- When moved to position b_0 the kernel chain it is in the CP-kernel.
- In position a_1 every slice contains exactly two bits of the kernel chain and is tame, except the slice containing the initial bit c_0 and the slice containing the final bit c_{2n+1}, that each just contain a single slice and are not tame.

Actually, a kernel chain is a generalization of a kernel vortex. It follows that in b_0, the bits c_{2i} and c_{2i+1} are in the same column and in a_1, the bits c_{2i+1} and c_{2i+2} are in the same column. This implies that the total number of kernel chains of a given length $2n$ starting from a given position is only 4^{2n-1}. Clearly, any combination of kernel chains is in the CP-kernel in b_0. Likewise, all slices in a_1 that contain only two kernel chain bits (excluding the initial and final bits) are tame. Now we must arrange the initial and final kernel chain bits such that the slides in a_1 that contain them are tame. The first possibility is that the bits c_0 and c_{2n-1} are in the same column in a_1: this kernel chain forms a kernel vortex. The second possibility is to group the initial and final bits of kernel chains in *tame knots*. We call a slice in a_1 with more than 2 active bits a *knot*. We construct states a_1 by combining kernel chains such that their initial and final bits are grouped in a set of knots. If all knots are tame, the state is tame. For a given Hamming weight x, valid states may exist with 0 up to $\lfloor x/3 \rfloor$ knots.

In our program we first fix the number of knots and their slice positions and then efficiently search for all valid states. Table 7.4 lists the number of valid states a_1 (modulo translation over the z-axis) for all KECCAK-f widths and up to a Hamming weight of 14. The question marks mark the limitations of our search algorithm: we have not yet been able to compute those values due to time constraints. It can be seen that for a given Hamming weight, overall the number of valid states decreases with increasing width. For a given width, the number of valid states increases with increasing Hamming weight.

7.4.2 Bounds for three-round tame trails

Starting from the valid patterns a_1 we have for each one searched for the 3-round tame trail with the smallest (restriction) weight. This was done in the following way. Given a_1 we

Width	Hamming weight					
	4	6	8	10	12	14
25	825	12100	95600	465690	1456725	?
50	150	13835	905135	22392676	?	?
100	48	2712	137078	6953033	?	?
200	10	481	24037	1143550	56824109	?
400	4	83	4006	164806	7290847	?
800	0	28	918	30771	1154855	44788752
1600	0	10	304	8231	259567	8399589

Table 7.4: The number of valid difference patterns a_1 per KECCAK-f width and Hamming weight

Width	Hamming weight					
	4	6	8	10	12	14
25	18	20	22	≤ 22	?	?
50	18	22	25	28	?	?
100	19	24	29	≤ 36	?	?
200	20	29	33	38	?	?
400	24	30	35	40	47	?
800	-	35	41	47	53	58
1600	-	35	41	48	56	62

Table 7.5: Minimum differential trail weight values of tame 3-round trails per KECCAK-f width and Hamming weight of a_1.

compute the minimum weight of a_0 by taking $\mathrm{w}^{\mathrm{rev}}(\lambda^{-1}(a_1))$. In the forward direction, we iterate over all states a_2 for which b_1 is in the CP-kernel. The results are given in Table 7.5. Cases containing a dash indicate that there are no tame 3-round trails for the given width and Hamming weight. Entries of the form "$\leq n$" indicate that not all the patterns were investigated. It can be seen that increasing the width results only in a limited growth of the (restriction) weight. Clearly, the limited diffusion due to the existence of the CP-kernel results in low-weight trails.

7.4.3 Bounds for four-round tame trails

We have conducted the a similar search for the 4-round tame trails with the smallest (restriction) weight. This was done in the same way as for 3-round trails with this difference: rather than stopping at a_2, we iterate over all states $a_3 \in \mathcal{A}(a_2)$ in the CP-kernel and compute its restriction weight. To speed up the search we apply pruning if the restriction weight of (a_0, a_1, a_2) is such that a_3 cannot result in a trail with a lower restriction weight than one found for the given category. The results are given in Table 7.6. While for small widths the increase in width does not substantially increase the restriction weights, for higher widths it can be observed that the minimum Hamming weight of a_1 for which there exist tame 4-round differential trails increases dramatically. For width 400 we have to go up to a Hamming weight of 12 to find a tame 4-round trail and for 800 and 1600 no tame 4-round trails exist

Width	**Hamming weight**					
	4	6	8	10	12	14
25	25	28	29	≤ 31	?	?
50	30	31	34	38	?	?
100	30	36	41	≤ 48	?	?
200	-	56	56	61	?	?
400	-	-	-	-	90	?
800	-	-	-	-	-	-
1600	-	-	-	-	-	-

Table 7.6: Minimum differential trail weight values of tame 4-round trails per KECCAK-f width and Hamming weight of pattern after first χ

for a_1 patterns with Hamming weight below 16. As for the minimum restriction weights of the 4-round tame trails found, the increase in 30 to 56 from width 100 to 200 and to 90 for width 400 suggests that the CP-kernel plays a much smaller role than for 3-round trails.

Chapter 8

Analysis of KECCAK-f

In this chapter we report on analysis and experiments performed on reduced-round versions of KECCAK-f, either by the designers or third parties. In Section 8.1 we describe our experiments based on the algebraic normal form representations. In Section 8.2 we report on the outcome of attempts to solve CICO problem instances. In Section 8.3 we describe statistical properties of reduced-round versions of KECCAK-f[25]. Finally, in Section 8.4 we discuss distinguishers that exploit the low algebraic degree of the round function and its inverse.

8.1 Algebraic normal form

In this section, explain how the algebraic normal form can be used to evaluate the pseudo-randomness of KECCAK-f in different aspects.

8.1.1 Statistical tests

There are several ways to describe KECCAK-f algebraically. One could compute the algebraic normal form (ANF, see Section 5.2.3) with elements in GF(2), GF(2^5), GF(2^{25}) or GF(2^w), 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.

We performed several statistical tests based on the ANF of KECCAK-$f[b, n_\mathrm{r} = n]$, from $b = 25$ to $b = 1600$ and their inverses in GF(2). The number of rounds n is also varied from 1 to its nominal value, although in practice we can limit ourselves to a reasonable number of rounds after which no significant statistical deviation can be found.

In general, the test consists in varying 25 bits of input and counting the number of monomials of degree d of all b 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.

The different tests determine which input bits are varied and/or which variant of the KECCAK-f permutation is used.

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
6-24	25	25	25	25

Table 8.1: The BIS ANF statistical test on KECCAK-f[1600] and its inverse

- **Bits in slice (BIS)** In this test, the 25 bits of the slice $z = 0$ are varied. For $b > 25$, the remaining $b - 25$ bits are set to zero.

- **Bits in lane (BIL)** In this test, the first 25 bits of lane $x = y = 0$ are varied. This test applies only to $b \geq 800$. The remaining $b - 25$ bits are set to zero.

- **Bits in two lanes, kernel** In this test, the first 25 bits of lane $x = y = 0$ and of lane $(x, y) = (0, 1)$ are varied simultaneously. The remaining $b - 50$ bits are set to zero. The idea behind this test is that θ will behave like the identity in the first round since the input state is always in the column-parity kernel (see 6.5.3).

- **BIS, symmetry in lanes** In this test, the 25 bits of the slice $z = 0$ are varied. The $b-25$ other bits are set as $a[x][y][z] = a[x][y][0]$ so that all lanes contain either all zeroes or all ones. The output bits $a[x][y][0]$ are xored into $a[x][y][z]$ for all $z > 0$. Only the $b - 25$ output bits with $z > 0$ are considered for the test—this test applies only when $b > 25$. The purpose of this test is discussed in Section 8.1.2.

- **BIS, slide** This test is like the first BIS test, except that the function tested is different. Instead of testing it against KECCAK-$f[b, n_\mathrm{r} = n]$ itself, we test it against the function slide$[b, n]$ defined in Equation (8.1). The purpose of this test is discussed in Section 8.1.3.

The results for KECCAK-f[1600] are summarized in Tables 8.1 and 8.2. 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.

All results of the tests can be found in the file `ANF-Keccak-f.ods`. It is interesting to observe the fast increase of degree of the monomials that are densely present in the algebraic description of the KECCAK-f permutations. Note that the round function has only degree two and thus no monomial of degree higher than 2^i can appear after after i rounds. The degree of the inverse of the round function is three and thus no monomial of degree higher than 3^i can appear after after i inverse rounds.

Taking the worst case among all these tests, the KECCAK-f permutations and their inverses pass the test for the maximum degree tested here (i.e., $\max(25, b - 1)$) after 6 to 8 rounds, depending on the width.

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
7-24	25	25	25	25

Table 8.2: The BIL ANF statistical test on KECCAK-f[1600] and its inverse

8.1.2 Symmetric trails

The design of KECCAK-f has a high level of symmetry. Due to this, the weight of symmetric trails may no longer be relevant for the security. (See Section 6.5.2 for more details.) We investigate how an attacker may be able to exploit the symmetry in his advantage.

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.

To find a pair in a differential trail of weight W requires the attacker to fulfill W conditions on the absolute values when following that trail. In the case of a b'-symmetric case, the conditions are repeated b'/b times on translated sets of bits. The question is to determine whether the symmetry induced by this duplication can be exploited by the attacker, even with the asymmetry introduced by ι.

In the absence of ι, KECCAK-$f[b]$ presented with a symmetric input behaves as b parallel identical instances of KECCAK-f[25]. In such a modified permutation, a symmetric trail with weight eb would only impose e conditions (on the symmetric absolute values) rather than eb.

To determine how much asymmetry ι introduces on the absolute values, we express the KECCAK-$f[b]$ permutation on a different set of variables and compute the ANF on it. The change of variables is defined as:

$$a'[x][y][0] = a[x][y][0],$$
$$a'[x][y][z] = a[x][y][z] \oplus a[x][y][0], z > 0.$$

In the absence of asymmetry (i.e., without ι), $a'[x][y][z]$, $z > 0$ remains zero if the input of the permutation is 1-symmetric, i.e., if $a[x][y][z]$ depends only on x and y.

The attacker can try to introduce a symmetric difference and to keep it symmetric through the rounds. In these new variables, it is equivalent to keeping $a'[x][y][z] = 0$ for $z > 0$. By

analyzing the ANF in these new variables, it gives the degree of the equations to solve to keep the symmetry in the state at a given round by adjusting the input.

The results of these tests can be found in the file `ANF-Keccak-f.ods`. The maximum degree to pass test is indicated in the *BIS, sym lane* columns. To impose dense non-linear equations to the attacker, KECCAK-$f[50]$ to KECCAK-$f[400]$ need at least 3 rounds, while KECCAK-$f[800]$ and KECCAK-$f[1600]$ need at least 4 rounds. A dense number of monomials with maximum degree tested here (i.e., $\max(25, b-1)$) is reached after 6 rounds for all widths. Similar conclusions apply to the inverse of KECCAK-f: The maximum degree tested is reached after 6 rounds for the inverse of KECCAK-$f[50]$, 5 rounds for the inverse of KECCAK-$f[100]$ and of KECCAK-$f[200]$ and 4 rounds for the other inverses. According to this test, the attacker should have a very difficult time to keep the differences symmetric after such a number of rounds.

8.1.3 Slide attacks

Slide attacks [19, 47] are attacks that exploit symmetry in a primitive that consists of the iteration of a number of identical rounds. We investigate how much asymmetry must be applied for these attacks to be non-effective.

In the absence of ι, all rounds are identical. In such a case, this allows for distinguishing properties of KECCAK-f: the distribution of the cycle lengths will be significantly different from that of a typical randomly-chosen permutation.

To evaluate the amount of inter-round asymmetry brought by ι, we performed a specific test based on ANF. We compute the ANF of the function slide$[b, n]$, which is obtained by XORing together the output of the first n rounds of KECCAK-$f[b]$ and the output of n rounds starting from the second one. Alternatively, it is defined as

$$\text{slide}[b, n] = (\text{round}_{n-1} \circ \cdots \circ \text{round}_0) \oplus (\text{round}_n \circ \cdots \circ \text{round}_1), \tag{8.1}$$

where round_i is the round permutation number i. In the absence of ι, the slide$[b, n]$ function would constantly return zeroes. With ι, it returns the difference between two sets of n rounds slid by one round.

The results of these tests can be found in the file `ANF-Keccak-f.ods`. The maximum degree to pass test is indicated in the *BIS, slide* columns. The degree increases more slowly than for other tests such as *bits in slice*. However, the maximum degree tested here (i.e., $\max(25, b-1)$) is reached after 8 rounds for KECCAK-$f[25]$ and after 6 rounds for all other widths. For the inverse of KECCAK-$f[25]$, the maximum degree tested is reached after 7 rounds and, for the other inverses, after 4 rounds.

8.2 Solving CICO problems algebraically

8.2.1 The goal

As explained in Sections 5.3 to 5.7 the security of a sponge function relies critically on the infeasibility of solving non-trivial CICO problems for its underlying permutation. The resistance of KECCAK-f against solving non-trivial CICO problems is hence critical for the security of KECCAK. We have developed software to generate the sets of equations in a form that they can be fed to mathematics software such as MAGMA or SAGE, and subsequently using this software to solve instances of the CICO problem for different sets of parameters.

8.2.2 The supporting software

KECCAKTOOLS [14] supports the generation of round equations of all supported width values, compatible with SAGE or other computer algebra tools. It can generate the equations for all operations within a round, θ, ρ, π, χ and ι, or a sequence of them. The functions that generate the equations are based on the same code that actually evaluates the KECCAK-f permutations. Using C++ template classes, the evaluation of the operations is made symbolically, hence producing equations. This way of working reduces the chances of making mistakes, as it avoids to duplicate the description of KECCAK-f in C++.

The format of the generated equations has been chosen to match the syntax of SAGE [80]. Also, the bits inside the state are named in such a way that their alphabetical order matches the bit numbering defined at the level of the sponge construction. This feature comes in handy when defining a concrete problem to solve, e.g., finding a pre-image or a collision. One needs to separate bits that are known and fixed from those that are the unknowns of the problem instance. This separation is defined at the sponge level and can be done alphabetically on the bit names.

We have installed a SAGE server version 1.4 [80] and automated the tests using Python scripts [26] interpreted by the SAGE server.

8.2.3 The experiments

In our experiments we have used SAGE to solve a wide range of CICO problems applied to the KECCAK-f permutations. In all problems a subset of the input bits and output bits are fixed to randomly generated values. More particularly, the range of CICO problems we investigated can be characterized with the following parameters:

- Number of rounds n_r: the number of rounds of KECCAK-f
- Width b: the width of KECCAK-f
- Rate r: the number of unknown input bits
- Output length n: the number of known output bits

There is a variable for every bit at the input of each round and for the output bits, totalling to $b(n_\mathrm{r}+1)$ variables. There is a round equation for each output bit of every round, expressing the output bit as a Boolean expression of the input bits. Additionally there is an equation for each input or output bit that is fixed, simply expressing the equality of the corresponding variable with a binary constant. Hence in total the number of equations is $bn_\mathrm{r}+(b-r)+n = b(n_\mathrm{r}+1)+n-r$.

We use SAGE to solve the CICO problems using Ideals and Gröbner bases [33]. We provide a short intuitive explanation here and refer to [33] for thorough treatment of Ideals and Gröbner bases. To solve a CICO problem we do the following:

- Define a ring R of Boolean polynomials, providing the $b(n_\mathrm{r}+1)$ variable names and specifying the so-called term ordering [33].
- Define an ideal over the ring, providing the $b(n_\mathrm{r}+1)+n-r$ equations as generator polynomials.
- Compute a Gröbner basis for this ideal.

The generated Gröbner basis consists of a sequence of polynomials (equivalent to equations) that allows to easily generate all solutions of the set of equations generating the ideal. If there is no solution, it simply consists of the polynomial 1 (implying the equation $1 = 0$ that has no solution). If there is one solution, it simply consists of a sequence of equations of type $x + 1$ (implying $x = 1$) or x (implying $x = 0$). The order of the variable names and term ordering provided in the definition of the ring R have an impact on the Gröbner basis. The sequence of polynomials in the Gröbner basis satisfy an ordering that is determined by the order of variable names and the term ordering. This is not just the way the basis is presented but has an impact on the way it is generated and hence may impact its computational and memory complexity.

For executing these functions, SAGE calls an underlying library dedicated to Boolean polynomials and monomials. This library is called PolyBoRi [23] and is part of the SAGE distribution. As opposed to the generic polynomial ideal oriented functions in SAGE, it heavily exploits the fact that Boolean polynomials can be modelled in a very simple way, with both coefficients and degree per variable in $\{0, 1\}$. The ring of Boolean polynomials is not a polynomial ring, but rather the quotient ring of the polynomial ring over the field with two elements modulo the field equations $x^2 = x$ for each variable x. Therefore, the usual polynomial data structures seem not to be appropriate for fast Gröbner basis computations. The PolyBoRi authors state that they introduces a specialised data structure for Boolean polynomials, capable of handling these polynomials more efficiently with respect to memory consumption and also computational speed.

If the number of equation is smaller than the number of variables, we expect there to be multiple solutions. If the number of equations is larger than the number of variables, chances are that there is no solution at all. A priori, the expected number of solutions of a CICO problem is 2^{r-n}. In our experiments we have focused on the case $r = n$ where we expect to have one solution on the average and across experiments we expect the number of solutions to have a Poisson distribution with $\lambda = 1$. This is an interesting case as it corresponds with the CICO problems encountered when searching a (second) pre-image.

We investigated the case $n = r$ for values of r up to 12, values of n_r up to 8 and widths from 25 to 400. We used lexicographical order with the output variables declared first and the input variables declared last, as from preliminary experiments this turned out to be the most efficient choice. For each set of parameters, a number of CICO problems was solved with at least 5 problems resulting in a solution and 5 problems resulting in the absence of a solution. We arranged the output of SAGE for analysis in the file `Keccak-CICO-results.ods`.

Analysing this data, we can make the following observations:

- For small values of r PolyBoRi is efficient in computing the Gröbner basis, and this for all values of n_r and width b.

- For certain parameter choices, solving CICO problems that have no solution takes significantly less time than CICO problems that have a solution. The difference is especially large for large widths and small rate values. When the rate increases, the computation times for the two cases (solution and no solution) converge.

- Doubling the width, keeping all other parameters constant also roughly doubles the computation time. Hence the computation time appears to grows roughly linearly in the width.

- Increasing the number of rounds from n_r to n_r+1, keeping all other parameters constant, results in an increase roughly independent from the value of n_r. Hence the computation time appears to grow linearly in the number of rounds.
- The effect of increasing bitrate by 1 from r to $r+1$, keeping all other parameters constant, appears to increase the computation time by a factor that weakly increases with r and the number of rounds. At $r = 12$ its value is around 3. If we may extrapolate this behaviour, solving a CICO problem quickly becomes infeasible with this method as r grows.

Hence for this class of CICO problems the case of a small rate and small output length can be relatively easily solved. Although surprising at first sight, this poses no threat to the security of KECCAK-f as such a CICO problem can be efficiently solved by exhaustive search for any permutation. It suffices to try all 2^r possible values of the r unknown input bits, apply the permutation and verify whether the generated output has the correct value in the known bit positions.

8.2.4 Third-party analysis

[2] reports on attempts of solving instances of the CICO problem for different widths of KECCAK-f using a so-called triangulation tool. Solutions were found for KECCAK-f[1600] reduced to three rounds. For more rounds the fast backwards diffusion in θ apparently prevented solving the CICO problems.

[66] reports on attempts of solving instances of the CICO problem for different widths of KECCAK-f by first expressing the round equations in conjunctive normal form (CNF) and then run SAT solvers on the resulting set of equations. The SAT solver performed better than exhaustive search for some CICO problem instances of KECCAK-f[1600], KECCAK-f[200] and KECCAK-f[50] reduced to three rounds. For more rounds the SAT solvers were less efficient than exhaustive search.

8.3 Properties of KECCAK-f[25]

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 verify certain properties.

First, we report on the algebraic normal form investigations, applied to KECCAK-f[25]. Second, we have reconstructed significant parts of the distribution of differential probabilities and input-output correlation of KECCAK-f[25] and its reduced-round versions. Third, we have determined the cycle structure of KECCAK-f[25] and all its reduced-round versions.

As a reference for the distributions, we have generated a pseudorandom permutation operating on 25 bits using a simple algorithm from [58] taking input from a pseudorandom bit generator based on a cipher that is remote from KECCAK and its inventors: RC6 [75]. We denote this permutation by the term Perm-R.

8.3.1 Algebraic normal statistics

The results of the ANF analysis of KECCAK-f[25] are displayed in Table 8.3. Starting from 7 rounds, all monomials up to order 24 exist and appear with a fraction close to one half. Since

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

KECCAK-f[25] is a permutation, the monomial of order 25 does not appear.

8.3.2 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 [40]. Moreover, [40] also determines the distribution of the maximum cardinality of a large number of differentials over a random permutation. According to [40, 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 8.1 shows that Perm-R exhibits a distribution that follows quite closely the theoretically predicted one. The maximum observed cardinality is 11.

Figure 8.2 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 7.3) but apparently no such trail was encountered in our sampling.

Figure 8.3 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

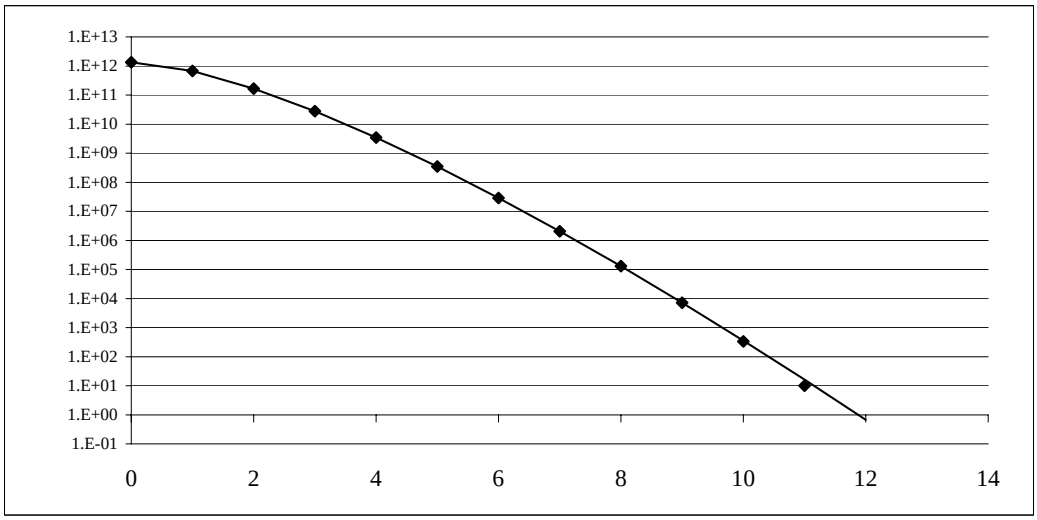

Figure 8.1: Cardinality histogram of sampling of Perm-R

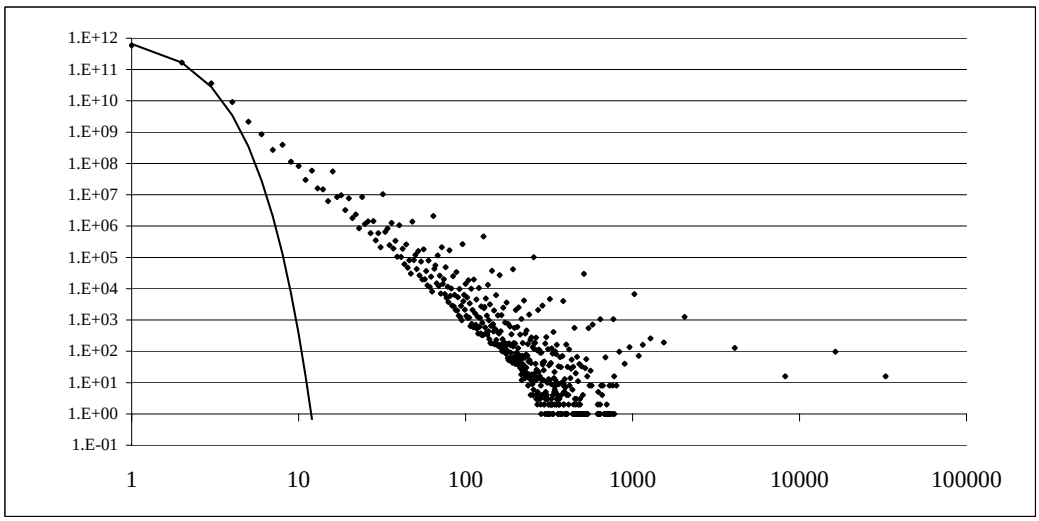

Figure 8.2: Cardinality histogram of sampling of 2-round version of KECCAK-f[25]

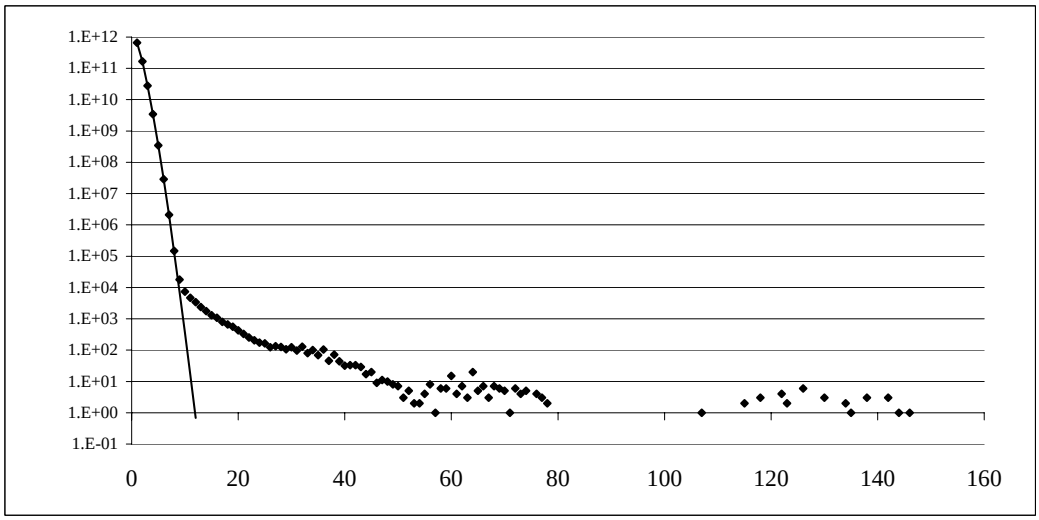

Figure 8.3: Cardinality histogram of sampling of 3-round version of KECCAK-f[25]

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 7.3) were not encountered in our sampling.

Figure 8.4 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 7.3) 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 8.5 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.

8.3.3 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 correlations of 2^{39} couples (v, u) where u ranges over 2^{14} different non-zero output masks 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 masks. For the KECCAK-f[25] variants we tested as output masks 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 [40]. Moreover, [40] also determines the distribution of the maximum correlation magnitude of a large number of couples (v, u) over a random permutation. According to [40, 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,

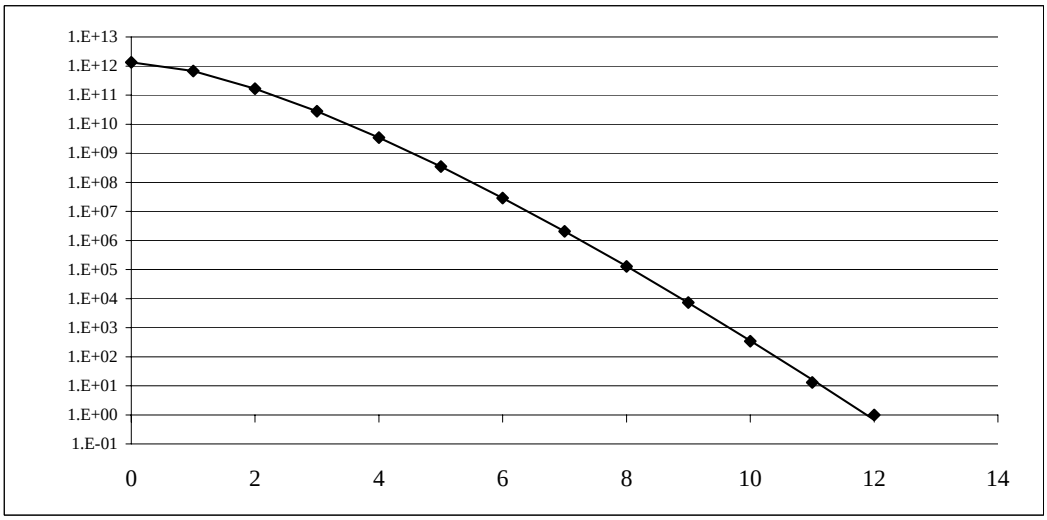

Figure 8.4: Cardinality histogram of sampling of 4-round version of KECCAK-f[25]

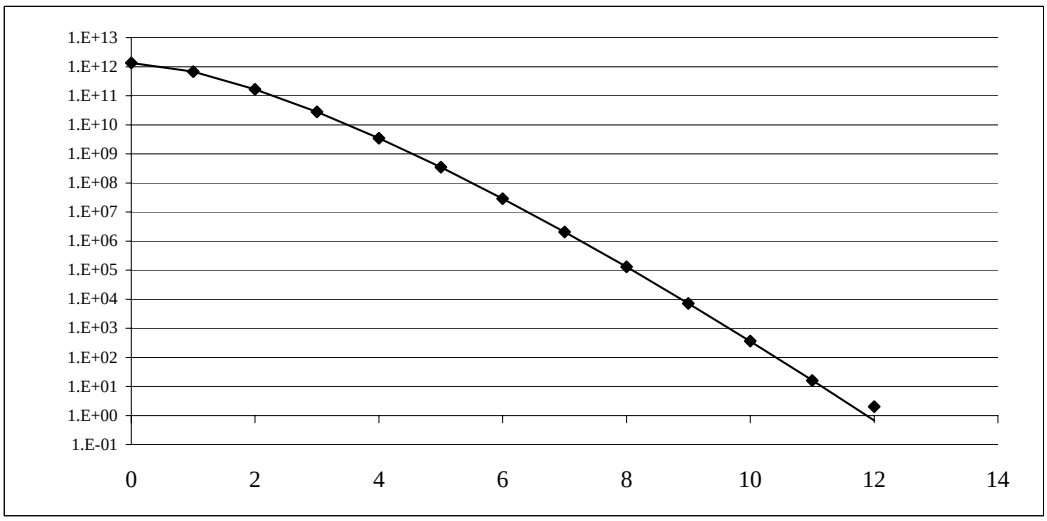

Figure 8.5: Cardinality histogram of sampling of KECCAK-f[25]

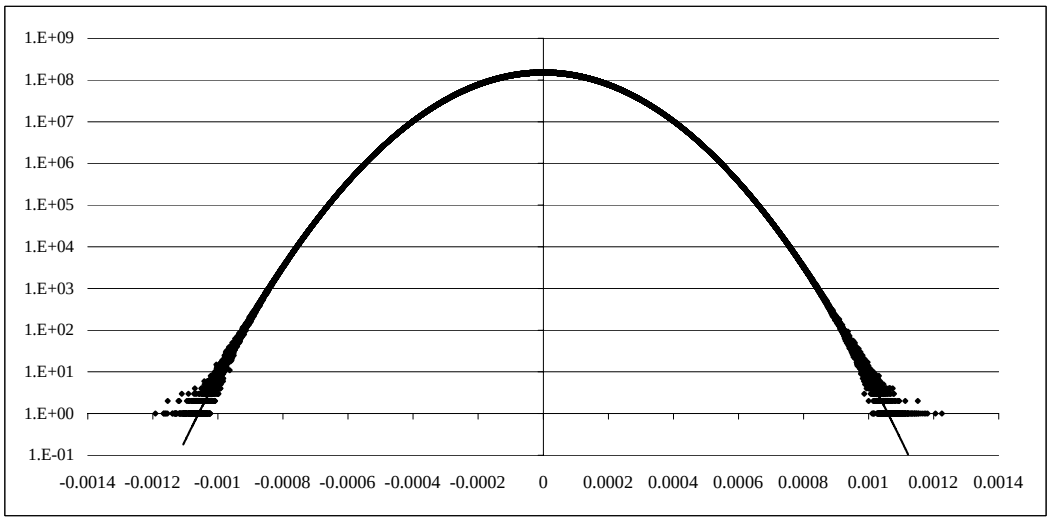

Figure 8.6: Correlation histogram of sampling of Perm-R

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 8.6 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 histogram 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 8.7 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, rather than full degree 24. The largest correlation magnitude encountered 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. By measuring the correlation of the same pair of masks for variants of the two-round version of KECCAK-f[25] where different constant vectors are XORed in between the two rounds, it turns out that the correlation value is either 2^5 or -2^{-5}. This implies that the correlation is the result of a single trail. The 2-round linear trails with weight 8 (see Table 7.3) were apparently not encountered in our sampling.

Figure 8.8 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. The largest correlation magnitude encountered is 0.003479. This is a correlation magnitude that cannot be obtained by a single linear trail. 3-round linear trails with weight 16 would give correlation magnitude

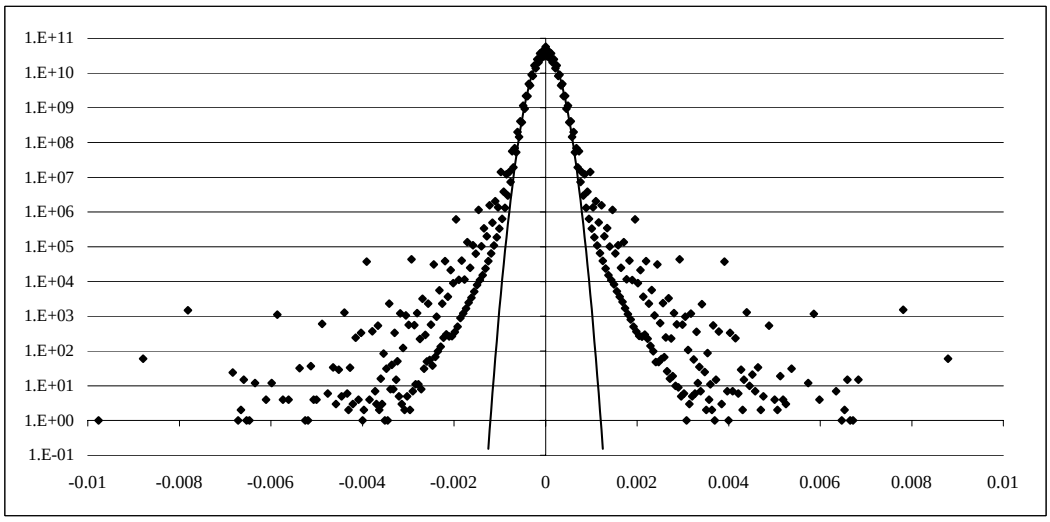

Figure 8.7: Correlation histogram of sampling of 2-round version of KECCAK-f[25]

$2^{-8} \approx 0.0039$. 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 higher weight. By measuring the correlation of this pair of masks in variants of the three-round version of KECCAK-f[25] where different constant vectors are XORed in between the rounds, we obtain 491 different values. This implies that this correlation has contributions from at least 9 trails.

Figure 8.9 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. By measuring the correlation of this pair of masks in variants of the four-round version of KECCAK-f[25] where different constant vectors are XORed in between the rounds, we obtain many different values implying that this correlation is the result of a large amount of trails. Moreover, the value of the correlation exhibits a normal distribution.

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}.

Finally, Figure 8.10 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.

8.3.4 Cycle distributions

We have determined the cycle structure of KECCAK-f[25] and all its reduced-round versions. Table 8.4 lists all cycles for KECCAK-f[25] and Table 8.5 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 8.5 is 16.3.

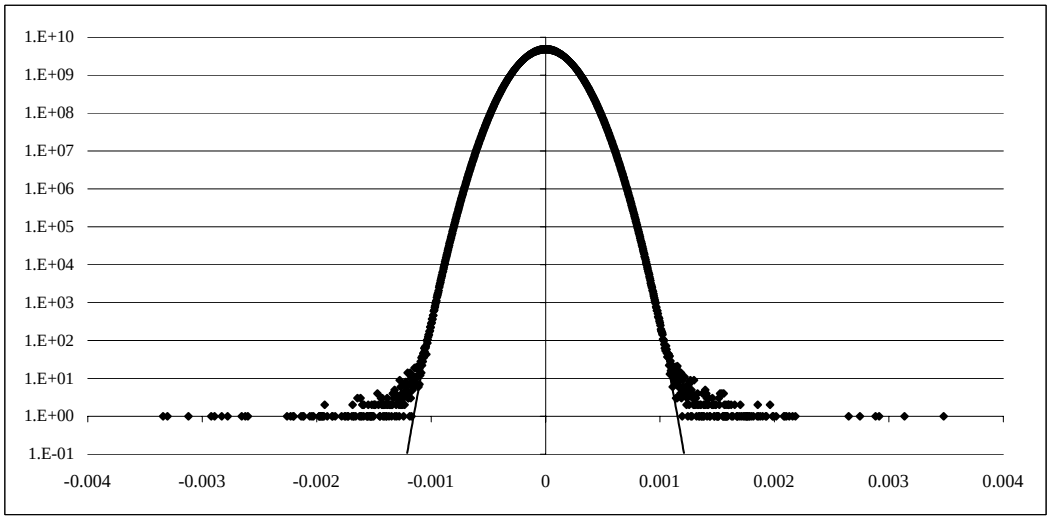

Figure 8.8: Correlation histogram of sampling of 3-round version of KECCAK-f[25]

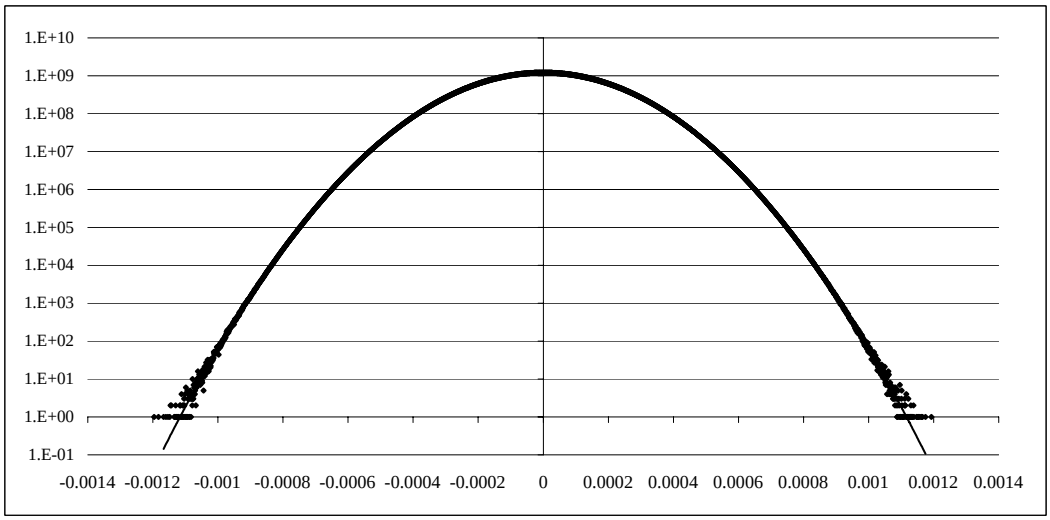

Figure 8.9: Correlation histogram of sampling of 4-round version of KECCAK-f[25]

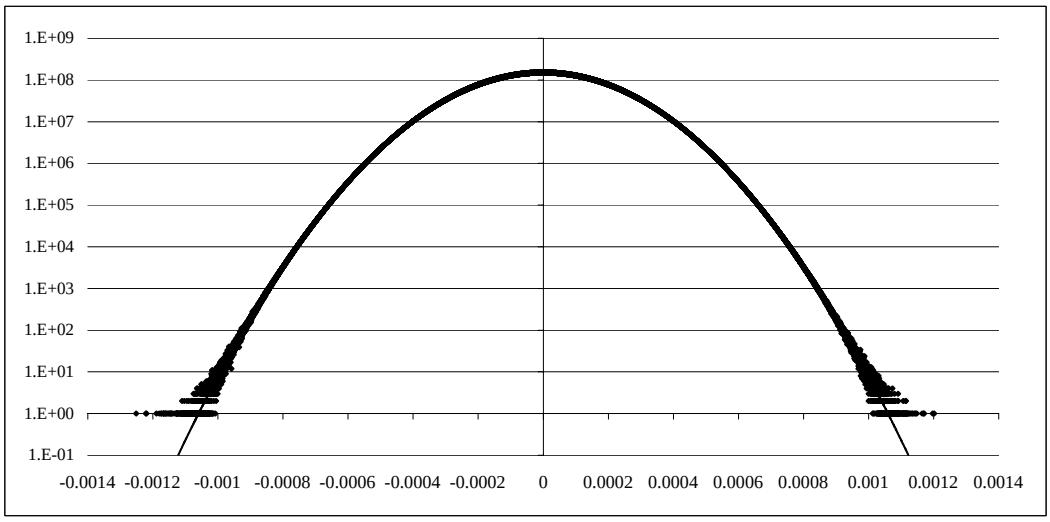

Figure 8.10: Correlation histogram of sampling of KECCAK-f[25]

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 [85], 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 [85, 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 composition 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 8.4: 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 8.5: Number of cycles in reduced-round versions of KECCAK-f[25]

8.4 Distinguishers exploiting low algebraic degree

The KECCAK-f round function and it inverse have low algebraic degrees, 2 and 3 respectively. This has been exploited in third-party cryptanalysis to construct a number of distinguishers.

[2] reports on observations showing that KECCAK-f[1600] reduced to 3 and 4 rounds does not have an ideal algebraic structure and conjectures that this can be observed up to reduced-round versions of ten rounds, for which the algebraic degree is at most 1024 and hence not maximal.

[62] tested the resistance of KECCAK against cube attacks by executing efficient cube attack against the 224-bit output-length version of KECCAK configured as an HMAC and calling a version of KECCAK-f[1600] reduced to 4 rounds or less. Based on their analysis, the author suggests that a cube attack against such a configuration would only be practical against reduced-round versions up to 7 rounds.

The papers [3, 22] describe very simple and elegant distinguishers against reduced-round versions of KECCAK-f[1600]. They are based on the simple observation that the degree of n KECCAK-f rounds is at most 2^n and that the degree of n inverse rounds is at most 3^n. We cite [3]:

Suppose one fixes $1600 - 513 = 1087$ *bits of the initial state to some arbitrary value, and consider the* 2^{513} *states obtained by varying the 513 bits left. Our main observation is that applying the 9-round* KECCAK-f *to each of those states and xoring the* 2^{513} *1600-bit final states obtained yields the zero state. This is because, for each of the 1600 Boolean components, the value obtained is the order-513 derivative of a degree-512 polynomial, which by definition is null.*

If these states are chosen in some intermediate round and one computes the corresponding inputs and outputs, this is also the case for those inputs as long as the polynomials expressing the inputs in terms of the intermediate state bits have small enough degree. Hence one has a systematic way to construct a set of inputs that XOR to zero and for which the corresponding outputs XOR to zero, which is qualitatively different from the generic method [16]. Additionally, if the positions of the free bits in the intermediate round are chosen carefully, one may even reduce the degree of the forward and/or backward polynomials. Using these simple elements, [3] constructs distinguishers for up to 16 rounds of KECCAK-f[1600].

This last one makes use of 6 backward rounds and 10 forward rounds.

Given an upper bound of N free bits one can construct a zero-sum distinguisher with $\log_2 N$ forward rounds and $\log_3 N$ backward rounds. While in a distinguisher using only the forward direction maximum degree is reached after 11 rounds, the start-in-the-middle technique allows going up to 16 rounds.

This was further refined in [22] where the authors show a distinguisher for KECCAK-f[1600] up to 18 rounds and later in [21], where they extended it to 20 rounds. The new ideas are the following. First, the Walsh spectrum of χ allows one to bound the degree of the inverse rounds more tightly. In particular, while 7 inverse rounds are expected to be of maximal degree (as $3^7 > 1599$), they show that it cannot be higher than 1369. Second, by aligning the bits that are varied and those that are fixed on row boundaries, χ works independently and bijectively row per row. This can be generalized to cover several rounds.

In general, for all widths, the number of rounds is now expressed as $12 + 2\ell$ instead of $12 + \ell$. The motivation behind this is that the applicability of the zero-sum distinguishers is limited by the maximum number of free bits N, namely the width. Doubling the width allows to add a forward round in the distinguisher and possibly a backward round. So when doubling the width, roughly two additional rounds are required to provide resistance against zero-zum distinguishers.

Further discussions about the applicability of the zero-sum distinguishers can be found in [16].

Chapter 9

Implementation

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

9.1 Bit and byte numbering conventions

As it impacts the reference implementation, the bit and byte numbering conventions are defined in [11, 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 [72].

As explained in [11, 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 NIST reference platform follows the little-endian (LE) convention, we adopt the following numbering of bits and bytes: when mapping a lane to a CPU word, 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), to 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 [11, 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.

r	c	Relative performance
576	1024	$\div 1.778$
832	768	$\div 1.231$
1024	576	1
1088	512	$\times 1.063$
1152	448	$\times 1.125$
1216	384	$\times 1.188$
1280	320	$\times 1.250$
1344	256	$\times 1.312$
1408	192	$\times 1.375$

Table 9.1: Relative performance of KECCAK$[r, c]$ with respect to KECCAK[].

9.2 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 $\lceil \frac{n}{r} \rceil - 1$.

When evaluating KECCAK, the processing time is dominantly spent in the evaluation of KECCAK-f. In good approximation, the throughput of KECCAK for long messages is therefore proportional to r. In the sequel, we will often write performance figures for the default bitrate $r = 1024$. To estimate the performance for another bitrate, Table 9.1 provides the performance relative to the default bitrate. This is valid for long messages; for short messages, the processing time is determined by the required number of calls to KECCAK-f (e.g., one when $l \leq r - 25$).

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

- 1824 XORs,
- 600 ANDs,
- 600 NOTs, and
- 696 64-bit rotations.

Almost 80% of the NOT operations can be removed by applying a *lane complementing transform* as explained in Section 9.2.1, 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 9.2.2.

9.2.1 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.

9.2.2 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 [11], 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 9.3). On some platforms, look-up tables can speed up this process, although the gain is not always significant.

9.3 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 [11, Section 5.2]).

The reference implementation calls the KECCAK-f[1600] permutation written in `KeccakPermutationReference.c`, while the optimized versions use `KeccakPermutationOptimized32.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 9.2.1), the even and odd bits separated (as in 9.2.2) 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 `KeccakAbsorb()` to XOR the input, given as bytes, to the state and then apply the permutation;
- `KeccakExtract1024bits()` and `KeccakExtract()` 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.

9.3.1 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 4.4.1 using `gcc -O3 -g0 -march=barcelona` for 64-bit code and adding `-m32` to produce only 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`.

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 [14], 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 24 rounds was found often to give the best results, although one can unroll any number of rounds that divides 24. On the platforms defined above, we have obtained the figures in Table 9.2.

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 9.3.

Please refer to eBASH [7] and our website for up-to-date figures on a wider range of platforms.

Operation	Platform A	Platform B
KECCAK-f[1600] only	1648 c	1845 c
Squeezing with $r = 1024$	12.9 c/b	14.4 c/b
KECCAK-f and XORing 1024 bits	1680 c	1854 c
Absorbing with $r = 1024$	13.1 c/b	14.5 c/b

Table 9.2: 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	4408 c	5904 c
Squeezing with $r = 1024$	34.4 c/b	46.1 c/b
KECCAK-f and XORing 1024 bits	4600 c	6165 c
Absorbing with $r = 1024$	35.9 c/b	48.2 c/b

Table 9.3: 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 performance on the 32-bit reference platform can be improved by using SIMD instructions, as detailed in Section 9.3.2. 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 [31] 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) [30]. 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).

9.3.2 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.

Operation	Platform A	Platform B
KECCAK-f[1600] only	2520 c	2394 c
Squeezing with $r = 1024$	19.7 c/b	18.7 c/b
KECCAK-f and XORing 1024 bits	2528 c	2412 c
Absorbing with $r = 1024$	19.8 c/b	18.8 c/b

Table 9.4: 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	2816 c	2475 c
Squeezing with $r = 1024$	22.0 c/b	19.3 c/b
KECCAK-f and XORing 1024 bits	2832 c	2475 c
Absorbing with $r = 1024$	22.1 c/b	19.3 c/b

Table 9.5: 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)

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 9.4 and 9.5. The use of SIMD instructions on the 32-bit reference platform significantly improves the performance.

9.3.3 SIMD instructions and KECCAKTREE

Parallel evaluations of two instances of KECCAK can also benefit from SIMD instructions, for example in the context of KECCAKTREE (see Section 4.4). In particular, KECCAKTREE[$r = 1024, c = 576, G = \text{LI}, H = 1, D = 2, B = 64$] can directly take advantage of two independent sponge functions running in parallel, each taking 64 bits of input alternatively. The final node then combines their outputs.

We have implemented the KECCAK-f[1600] 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 thus possible to evaluate two independent instances of KECCAK-f[1600] in parallel on a single core of the CPU.

In this instance of KECCAKTREE, the message bits can be directly input into the 128-bit SSE2 registers without any data shuffling. Using leaf interleaving and a block size B of 64 bits, 64 bits of message are used alternatively by the first sponge then by the second sponge. This

Operation	Platform A	Platform B
2 × (KECCAK-f and XORing 1024 bits)	2520 c	2511 c
Absorbing with $r = 1024$	9.8 c/b	9.8 c/b
2 × (KECCAK-f and XORing 1088 bits)	2528 c	2520 c
Absorbing with $r = 1088$	9.3 c/b	9.3 c/b

Table 9.6: Performance of KECCAKTREE$[r + c = 1600, G = \text{LI}, H = 1, D = 2, B = 64]$ on a single core using SSE2 instructions to compute two KECCAK-$f[1600]$ permutations in parallel ("c" is for cycles and "c/b" for cycles per byte)

matches how the data are organized in the SSE2 registers, where 64 bits are used to compute one instance of KECCAK-$f[1600]$ and the other 64 bits to compute the second instance.

This way of working allows speeds 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 ($Dc < r$) 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%. The performance figures for $r = 1024$ and $r = 1088$ are given in Table 9.6.

9.3.4 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 9.2.2 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 [48] and [25] 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 [37]. 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. As discussed in [12], the mechanisms described in [37] apply equally well to χ, the only non-linear part of KECCAK-f.

9.3.5 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

Step	Performance
θ	34560 cycles
ρ	20808 cycles
π	0 cycles
χ	30048 cycles
ι	384 cycles
Words re-ordering	15000 cycles
Total	100800 cycles

Table 9.7: Performance estimates of the KECCAK-f[1600] permutation on the 8XC51RA/RB/RC

released to allow accessing more internal RAM using a technique called Expanded RAM [29]. 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 [28], 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 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 9.7, for 24 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.

9.4 Hardware Implementations

In this section we report on our hardware implementations of KECCAK. For an overview and links to third-party hardware implementations of KECCAK we refer to `http://keccak.noekeon.org/third_party.html`.

KECCAK allows to trade off area for speed and vice versa. Different architectures reflect

different trade-offs. The two architectures we have investigated and implemented reflect the two ends of the spectrum: a high-speed core and a low-area coprocessor. Thanks to the symmetry and simplicity of its round function, KECCAK allows to trade off area for speed and vice versa. Different architectures reflect different trade-offs. We have concentrated on two architectures at the two ends of the spectrum: a high-speed core and a low-area coprocessor.

We have coded our architectures in VHDL for implementation in ASIC and FPGA. For more details on the VHDL code, refer to the `readme.txt` file in the `VHDL` directory of the SHA-3 submission package.

In these efforts we have concentrated on two instances of KECCAK:

KECCAK$[r = 1024, c = 576]$: the instance of KECCAK with default parameter values. It is built on top of KECCAK-$f[1600]$, the largest instance of the KECCAK-f family.

KECCAK$[r = 40, c = 160]$: the instance of KECCAK with the smallest instance of the KECCAK-f family such that the capacity still provides a security level that is sufficient for many applications. It makes use of KECCAK-$f[200]$.

It should be noted that during the design of KECCAK particular effort has been put to facilitate the hardware implementation. The round function is based only on simple Boolean expressions and there is no need for adders or S-boxes with complex logic (typically used in many cryptographic primitives). Avoiding these complex sub-blocks allow having a very short critical path for reaching very high frequencies. Another beneficial aspect of KECCAK is that, unless intentionally forced, a general architecture implementing KECCAK-f and the sponge construction can easily support all variants (rates, capacities) and use cases (MAC, MGF, KDF, PRNG) for a given lane size.

9.4.1 High-speed core

The architecture of the high-speed core design is depicted in Figure 9.1. It is based on the plain instantiation of the combinational logic for computing one KECCAK-f round, and use it iteratively.

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).

In the absorbing phase, the I/O buffer allows the simultaneous transfer of the input through the bus and the computation of KECCAK-f for the previous input block. Similarly, in the squeezing phase it allows the simultaneous transfer of the output through the bus and the computation of KECCAK-f for the next output block.

These buses typically come in widths of 8, 16, 32, 64 or 128 bits. We have decided to fix its width to the lane size w of the underlying KECCAK-f permutation. This limits the throughput of the sponge engine to w per cycle. This imposes only a restriction if r/w (i.e., the rate expressed in number of lanes) is larger than the number of rounds of the underlying KECCAK-f.

In a first phase 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. We were able to introduce the lane size as a parameter, allowing us to generate VHDL for all the lane sizes supported by KECCAK.

These first VHDL implementations have been tested on different FPGAs by J. Strömbergson [81], highlighting some possible improvements and problems with the tools available from FPGA

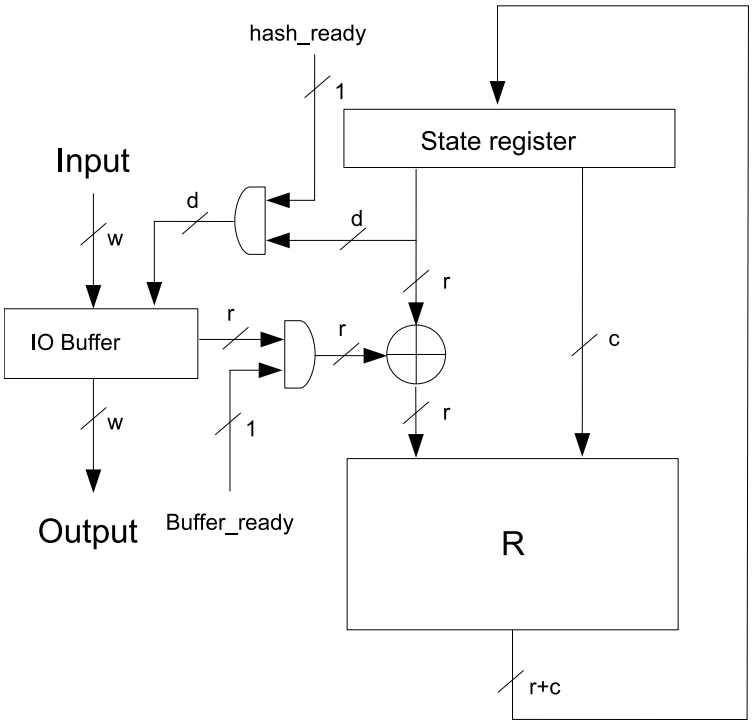

Figure 9.1: The high-speed core

vendors. We have improved the VHDL code for solving the problems, and this has resulted in better results in ASIC as well.

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.

9.4.2 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 instantiating two, three, four 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 performing the KECCAK-f rounds simultaneous with the input during the absorbing phase or output during the squeezing phase. An alternative for saving area is to execute the storing of the lanes composing the input blocks and extracting the lanes composing the output blocks directly in the state register.

9.4.2.1 KECCAK$[r = 1024, c = 576]$

In this instantiation the width of the bus is 64 bits. The bitrate of 1024 bits and the number of rounds of KECCAK-$f[1600]$ being 24 implies a maximum rate of 43 bits per cycle.

Number of round instances	Size	Critical Path	Frequency	Throughput
$n = 1$	48 kgates	1.9 ns	526 MHz	22.44 Gbit/s
$n = 2$	67 kgates	3.0 ns	333 MHz	28.44 Gbit/s
$n = 3$	86 kgates	4.1 ns	244 MHZ	31.22 Gbit/s
$n = 4$	105 kgates	5.2 ns	192 MHz	32.82 Gbit/s
$n = 6$	143 kgates	6.3 ns	135 MHZ	34.59 Gbit/s

Table 9.8: Performance estimation of variants of the high speed core of KECCAK$[r = 1024, c = 576]$.

The critical path of the core is 1.9 nanoseconds, of which 1.1 nanoseconds in the combinatorial logic of the round function. This results in a maximum clock frequency of 526MHz and throughput of 22.4 Gbit/s. 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.

An alternative without separate I/O buffer allows saving about 8 kgate and decreases the throughput to 12.8 Gbit/s at 500MHz.

Thanks to the low critical path in the combinational logic, it is possible to instantiate two or more rounds per clock cycle. For instance, implementing two rounds gives a critical path of 3 nanoseconds, allowing to run the core at 333MHz reaching a throughput of 28Gbit/s. Such a core will consume 1024 bits every 12 clock cycle, thus the bus width must grow too to keep up with the throughput per cycle. Note that contrary to many cryptographic algorithm, in KECCAK the processing does not impose the bottleneck in term of hardware implementation. Table 9.8 summarizes the throughputs for different variants.

9.4.2.2 KECCAK$[r = 40, c = 160]$

In this instantiation the width of the bus is 8 bits. The bitrate of 40 bits and the number of rounds of KECCAK-f[200] being 18 implies a maximum rate of 2.2 bits per cycle.

The critical path of the core is 1.8 nanoseconds, of which 1.1 nanoseconds in the combinatorial logic of the round function. This results in a maximum clock frequency of 555MHz and throughput of 1.23 Gbit/s. The area needed for having the core running at this frequency is 6.5 kgate, composed of 3 kgate for the round function, 3.1 kgate for the state register and control logic and less than 400 gates for the I/O buffer.

An alternative without separate I/O buffer allows saving about 400 gate and decreases the throughput to 0.96 Gbit/s at 555MHz.

9.4.3 Low-area coprocessor

The core presented in Section 9.4.1 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 hardware accelerator: the core can use the system memory instead of having all the storage

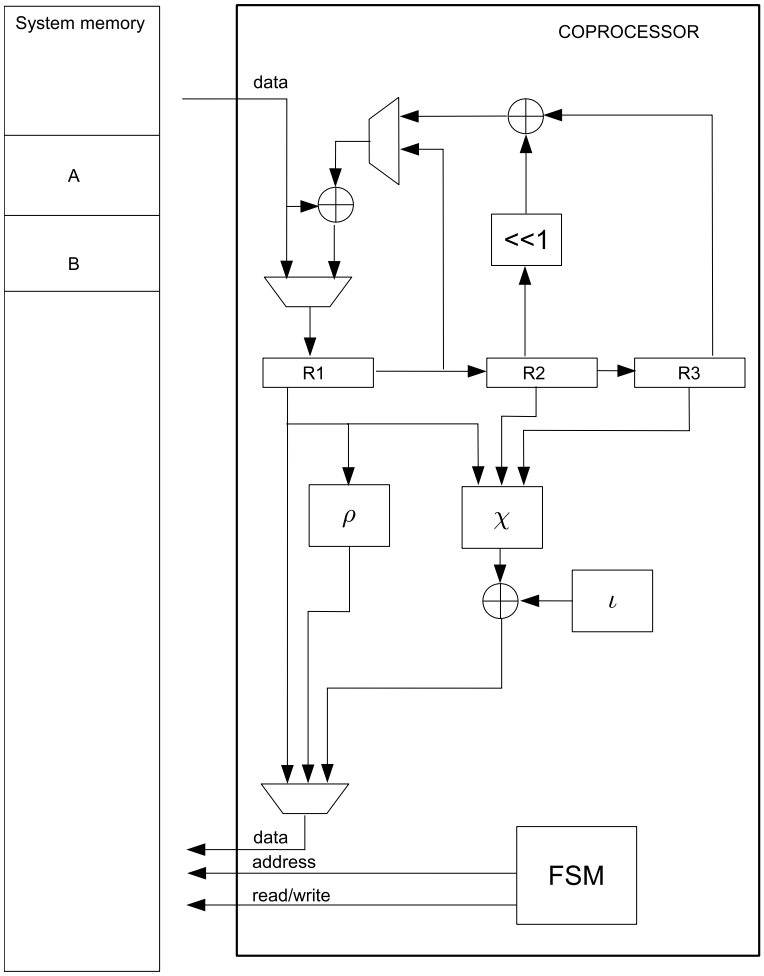

Figure 9.2: The low area coprocessor

capabilities internally. The state of KECCAK will be stored in memory and the coprocessor is equipped with registers for storing only temporary variables.

This kind of coprocessor is suitable for smart cards or wireless sensor networks where area is particularly important since it determines the cost of the device and there is no rich operating system allowing to run different processes in parallel.

The architecture is depicted in figure 9.2 where memory buffer labeled with A is reserved for the state, and B is reserved for temporary values. For the width of the data bus for performing memory access different values can be taken. We consider it equal to the lane size as a first assumption, and discuss later the implications if a smaller width is taken.

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 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 one round of the KECCAK-f permutation takes 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 memory words half the lane size, the number of clock cycles doubles but only for the part relative to read and write, not for the bubbles. In such an implementation one round of KECCAK-f requires 375 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 has the size of the state (25 times the lane size), while the memory space for storing temporary values has the size of the state times 1.2.

The low-area coprocessor has been coded in VHDL and simulated using Modelsim. As the core depicted in Section 9.4.1, the coprocessor has been synthesized using ST technology at 130 nm.

9.4.3.1 KECCAK$[r = 1024, c = 576]$

In this instantiation the computation of the KECCAK-f permutation takes 5160 clock cycles. The coprocessor has a critical path of 1.5 nanoseconds and can run up to 666.7 MHz resulting in a throughput of 132 Mbit/s. The area needed for attaining this clock frequency is 6.5 kgate. If the core is synthesized for a clock frequency limited to 200MHz, the area requirement is reduced to 5 kgate and the corresponding throughput is 39 Mbit/s. In both cases the amount of area needed for the registers is about 1 kgate.

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

9.4.3.2 KECCAK$[r = 40, c = 160]$

In this instantiation the computation of the KECCAK-f permutation takes 3870 clock cycles. The coprocessor has a critical path of 1.4 nanoseconds and can run up to 714 MHz resulting in a throughput of of 6.87 Mbit/s. The area for attaining this clock frequency is 1.6 kgate, If the core is synthesized for a clock frequency limited to 200MHz (500MHz), the area requirement is reduced to 1.3 (1.4) kgate and the corresponding throughput is 1.9 (4.8) Mbit/s. In both cases the amount of area needed for the registers is in the order of 100 gates.

9.4.4 FPGA implementations

We have used Altera Quartus II Web Edition version 9 [27] and Xilinx ISE WebPACK version 11.1 [32] to evaluate VHDL with the tools for FPGA. These tools provide estimations of the amount of resources needed and the maximum clock frequency reached.

Device	Logic	Registers	Max Freq.	Throughput
Altera StratixIII EP3SE50F484C2	4684 (38000) ALUTs	2641 (38000)	206 MHz	8.7 Gbit/s
Altera Cyclone III EP3C10F256C6	5770 (10320) LEs	2641 (10320)	145 MHz	6.1 Gbit/s
Virtex 5 XC5VLX50FF324-3	1330 (7200) slices	2640 (28800)	122 MHz	5.2 Gbit/s

Table 9.9: Performance estimation of the high speed core of KECCAK$[r = 1024, c = 576]$ on different FPGAs, and in brackets the resources available in the different cases.

Device	Logic	Registers	Max Freq.	Throughput
Altera StratixIII EP3SE50F484C2	594 (38000) ALUTs	333 (38000)	206 MHz	412 Mbit/s
Altera Cyclone III EP3C10F256C6	732 (10320) LEs	333 (10320)	145 MHz	290 Mbit/s
Virtex 5 XC5VLX50FF324-3	190 (7200) slices	340 (28800)	122 MHz	244 Mbit/s

Table 9.10: Performance estimation of the high speed core of KECCAK$[r = 40, c = 160]$ on different FPGAs, and in brackets the resources available in the different cases.

9.4.4.1 High-speed core

We report in Tables 9.9 and 9.10 the estimation of the completed place and route for the high-speed core for the large and small versions respectively.

9.4.4.2 Low-area coprocessor

In the case of FPGA, the estimations are reported in tables 9.11 and 9.12.

Device	Logic	Registers	Max Freq.	Throughput
Altera StratixIII EP3SE50F484C2	855 (38000) ALUTs	242 (38000)	359 MHz	71.2 Mbit/s
Altera Cyclone III EP3C5F256C6	1570 (5136) LEs	242 (5136)	183 MHz	36.3 Mbit/s
Virtex 5 XC5VLX50FF324-3	448 (7200) slices	244 (28800)	265 MHz	52.5 Mbit/s

Table 9.11: Performance estimation of the low area coprocessor of KECCAK$[r = 1024, c = 576]$ on different FPGAs, and in brackets the resources available in the different cases.

Device	Logic	Registers	Max Freq.	Throughput
Altera StratixIII EP3SE50F484C2	131 (38000) ALUTs	32 (38000)	359 MHz	3.7 Mbit/s
Altera Cyclone III EP3C5F256C6	205 (5136) LEs	30 (5136)	183 MHz	1.9 Mbit/s
Virtex 5 XC5VLX50FF324-3	62 (7200) slices	30 (28800)	265 MHz	2.7 Mbit/s

Table 9.12: Performance estimation of the low area coprocessor of KECCAK$[r = 40, c = 160]$ on different FPGAs, and in brackets the resources available in the different cases.

9.4.5 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 [82] or SecLib [49]. 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 [77] that can offer effective protection against glitches.

Bibliography

[1] J.-P. Aumasson, I. Dinur, W. Meier, and A. Shamir, *Cube testers and key recovery attacks on reduced-round MD6 and Trivium*, Fast Software Encryption (O. Dunkelman, ed.), Lecture Notes in Computer Science, vol. 5665, Springer, 2009, pp. 1–22.

[2] J.-P. Aumasson and D. Khovratovich, *First analysis of Keccak*, Available online, 2009, `http://131002.net/data/papers/AK09.pdf`.

[3] J.-P. Aumasson and W. Meier, *Zero-sum distinguishers for reduced Keccak-f and for the core functions of Luffa and Hamsi*, Available online, 2009, `http://131002.net/data/papers/AM09.pdf`.

[4] 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.

[5] 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.

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

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

[8] 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/`.

[9] ———, *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`.

[10] ———, *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.

[11] ———, Keccak *specifications, version 2*, NIST SHA-3 Submission, September 2009, `http://keccak.noekeon.org/`.

[12] ———, *Note on side-channel attacks and their countermeasures*, Comment on the NIST Hash Competition, May 2009, `http://keccak.noekeon.org/NoteSideChannelAttacks.pdf`.

[13] ———, *Sufficient conditions for sound tree and sequential hashing modes*, Cryptology ePrint Archive, Report 2009/210, 2009, `http://eprint.iacr.org/`.

[14] ———, KECCAKTOOLS *software*, June 2010, `http://keccak.noekeon.org/`.

[15] ———, *Note on* KECCAK *parameters and usage*, Comment on the NIST Hash Competition, February 2010, `http://keccak.noekeon.org/NoteOnKeccakParametersAndUsage.pdf`.

[16] ———, *Note on zero-sum distinguishers of* KECCAK-f, Comment on the NIST Hash Competition, January 2010, `http://keccak.noekeon.org/NoteZeroSum.pdf`.

[17] ———, *Sponge-based pseudo-random number generators*, CHES (S. Mangard and F.-X. Standaert, eds.), Lecture Notes in Computer Science, vol. 6225, Springer, August 2010, pp. 33–47.

[18] 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.

[19] A. Biryukov and D. Wagner, *Slide attacks*, in Knudsen [56], pp. 245–259.

[20] 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.

[21] C. Boura and A. Canteaut, *Zero-sum distinguishers on the Keccak-f permutation with 20 rounds (working draft)*, private communication, 2010.

[22] ———, *A zero-sum property for the Keccak-f permutation with 18 rounds*, Available online, 2010, `http://www-roc.inria.fr/secret/Anne.Canteaut/Publications/zero_sum.pdf`.

[23] M. Brickenstein and A. Dreyer, *PolyBoRi: A framework for Gröbner-basis computations with Boolean polynomials*, Journal of Symbolic Computation **44** (2009), no. 9, 1326–1345, Effective Methods in Algebraic Geometry.

[24] 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.

[25] 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.

[26] The Python community, *Python Programming Language*, Python Software Foundation, 2009, `http://www.python.org/`.

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

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

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

[30] ———, *Intel® Core™2 Duo Desktop Processor E6600*, `http://processorfinder.intel.com/details.aspx?sSpec=SL9S8`.

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

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

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

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

[35] 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.

[36] 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.

[37] J. Daemen, M. Peeters, and G. Van Assche, *Bitslice ciphers and power analysis attacks*, in Schneier [79], pp. 134–149.

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

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

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

[41] I. Dinur and A. Shamir, *Cube attacks on tweakable black box polynomials*, Cryptology ePrint Archive, Report 2008/385, 2008, `http://eprint.iacr.org/`.

[42] 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/`.

[43] 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.

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

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

[46] 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.

[47] M. Gorski, S. Lucks, and T. Peyrin, *Slide attacks on a class of hash functions*, Asiacrypt (J. Pieprzyk, ed.), Lecture Notes in Computer Science, vol. 5350, Springer, 2008, pp. 143–160.

[48] 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.

[49] S. Guilley, P. Hoogvorst, Y. Mathieu, R. Pacalet, and J. Provost, *CMOS structures suitable for secured hardware*, in Gielen and Figueras [46], pp. 1414–1415.

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

[51] 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.

[52] J. Kelsey, T. Kohno, and B. Schneier, *Amplified boomerang attacks against reduced-round mars and serpent*, in Schneier [79], pp. 75–93.

[53] 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.

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

[55] 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.

[56] L. R. Knudsen (ed.), *Fast software encryption, 6th international workshop, fse '99, rome, italy, march 24-26, 1999, proceedings*, Lecture Notes in Computer Science, vol. 1636, Springer, 1999.

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

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

[59] 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.

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

[61] 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.

[62] J. Lathrop, *Cube attacks on cryptographic hash functions, Master's thesis*, Available online, 2009, `http://www.cs.rit.edu/~jal6806/thesis/`.

[63] L.Knudsen, C. Rechberger, and S. Thomsen, *The Grindahl hash functions*, FSE (A. Biryukov, ed.), Lecture Notes in Computer Science, vol. 4593, Springer, 2007, pp. 39–57.

[64] U. Maurer, R. Renner, and C. Holenstein, *Indifferentiability, 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.

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

[66] Paweł Morawiecki and Marian Srebrny, *A sat-based preimage analysis of reduced KECCAK hash functions*, Cryptology ePrint Archive, Report 2010/285, 2010, `http://eprint.iacr.org/`.

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

[68] ———, *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`.

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

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

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

[72] ———, *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`.

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

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

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

[76] 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/`.

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

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

[79] 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.

[80] W. A. Stein et al., *Sage Mathematics Software*, The Sage Development Team, 2009, `http://www.sagemath.org/`.

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

[82] K. Tiri and I. Verbauwhede, *A logic level design methodology for a secure DPA resistant ASIC or FPGA implementation*, in Gielen and Figueras [46], pp. 246–251.

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

[84] D. Wagner, *The boomerang attack*, in Knudsen [56], pp. 156–170.

[85] 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.

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

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

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

[89] 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 2.0 to 2.1

A.1.1 Restructuring of document

- The chapter on usage has now become Chapter 4 and comes right after Chapter 3 on the sponge construction, as its content is not specific for KECCAK but rather generic for sponge functions.
- The chapter on the KECCAK-f permutations has been split in three:
  - Chapter 6: explaining the properties of the building blocks of KECCAK-f and motivating the choices made in the design.
  - Chapter 7: dedicated to propagation of differential and linear trails in KECCAK-f.
  - Chapter 8: covering all other analysis and investigations of KECCAK-f.

A.1.2 Addition of contents

- Added a reference to [15] in Section 3.3.3.
- Added explanation on our new upper bounds for the success probability of state recovery [17] in Section 3.4.2.
- Added the reseedable pseudo-random bit generator mode of use in Table 4.1.
- Modified statement on safety margin of KECCAK-f with respect to structural distinguishers reflecting recent zero-sum distinguishers, in Section 6.4.
- New techniques for determining lower bounds on the weight of differential and linear trails in Chapter 7.
- New bounds on differential and linear trails in Section 7.3.4.
- Results of new tests on the algebraic normal form in Section 8.1, amongst others focusing on symmetry properties in Sections 8.1.2 and 8.1.3.
- Results of new experiments of algebraically solving CICO problems in Section 8.2.

- New hardware implementation results for a version of KECCAK with width 200 in Section 9.4.
- Added references to new third-party cryptanalysis results where applicable.

A.1.3 Corrections and editorial changes

- Abandoned *right pairs* terminology in favor of more natural terminology *pairs in a differential or trail* starting from Section 5.2.1.
- Abandoned *selection vector* terminology in favor of more generally used terminology *mask* starting from Section 5.2.2.
- Corrected expressions for correlation weight in Section 5.2.2 by removing the division by two.
- Introduced a consistent KECCAK-oriented terminology for trail propagation in Section 6.5.1.

A.2 From 1.2 to 2.0

- Adaptation of the specifications summary to the modified number of rounds of KECCAK-f in Section 1.1.
- Update of the acknowledgements in Section 1.3.
- Update of Section 2.4 to reflect the modified parameters.
- Addition of Section 2.5 motivating the parameter change between version 1 and version 2 of KECCAK.
- Update of Section 3.3 to reflect the new choice for capacity and bitrate for the fixed-output-length SHA-3 candidates
- Update of Section 5.2.3 to include an introduction of cube attacks and testers.
- Update of our estimates for the required number of rounds in KECCAK-f for providing resistance against different types of distinguishers and attacks in Section 6.4.
- Addition of two entries in Table 7.3 in Section 7.3.4.
- Mentioning of the CICO-solving results of [2] in Section 6.6.
- Addition of Section 8.4 discussing third-party cryptanalysis based on low-degree polynomials.
- Addition of Section 7.4 reporting on our results of particular trails called tame trails.
- Encoding of B in bytes (instead of multiples of 64 bits) in Section 4.4.
- Discussion on soundness of the tree hashing mode in Section 4.4.2.
- Update of performance figures in Chapter 9.

A.3 From 1.1 to 1.2

- In Chapter 1, we added Section 1.1 with a 2-page specifications summary of KECCAK.
- In Section 6.3.1, we provide more explanations on difference propagation and correlation properties of χ.
- In Section 8.1, we present ANF tests also on the inverse of KECCAK-f.
- In Section 9.3, updates have been made to the software performance in general and regarding SIMD instructions in particular, see Sections 9.3.2 and 9.3.3.

A.4 From 1.0 to 1.1

- Sections 2.2 and 5.1 now explicitly mention the *hermetic sponge strategy*.
- Chapter 4 proposes additional usage ideas for KECCAK, including input formatting and diversification (see Section 4.3), and parallel and tree hashing (see Section 4.4). Section 4.1 now also mentions a slow one-way function.
- New techniques and updated implementation figures are added to Chapter 9.
  - Section 9.2.1 presents the lane complementing transform.
  - Section 9.2.2 shows how to use the bit interleaving technique.
  - Section 9.3 has been reorganized to show the results on the two platforms side by side.
  - Section 9.3.1 display updated performance figures.
  - Sections 9.4.1 and 9.4.3 report updated performance figures on ASIC and new figures on FPGA.