KECCAK implementation overview

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

http://keccak.noekeon.org/

Version **3.2**
May 29, 2012

[1]STMicroelectronics
[2]NXP Semiconductors

Contents

1 General aspects **7**
1.1 Specifications summary . . . 7
1.2 Bit and byte numbering conventions . . . 9
1.2.1 Some justification for our choice . . . 10
1.3 Operation count . . . 10
1.4 Memory . . . 12

2 Implementation techniques **13**
2.1 Bit interleaving . . . 13
2.2 The lane complementing transform . . . 14
2.3 Extending the state for smoother scheduling . . . 15
2.4 Plane-per-plane processing . . . 16
2.4.1 Early parity . . . 17
2.4.2 Combining with bit interleaving . . . 18
2.5 Efficient in-place implementations . . . 18
2.5.1 Combining with bit interleaving . . . 19
2.6 Processing slices . . . 22
2.6.1 Processing consecutive slices . . . 22
2.6.2 Processing interleaved slices . . . 23

3 Software **25**
3.1 PC and high-end platforms . . . 25
3.1.1 Using 64-bit instructions . . . 25
3.1.2 Using SIMD instructions . . . 26
3.1.3 SIMD instructions and tree hashing . . . 26
3.1.4 Batch or tree hashing on a graphics processing unit . . . 27
3.2 Small 32-bit platforms . . . 27
3.2.1 Implementation on a ARM Cortex-M0 and -M3 . . . 28
3.3 Small 8-bit platforms . . . 28
3.3.1 Implementation on a Atmel AVR processor . . . 29

4 Hardware **31**
4.1 Introduction . . . 31
4.2 High-speed core . . . 31
4.3 Variants of the high-speed core . . . 33
4.3.1 KECCAK$[r = 1024, c = 576]$ . . . 33
4.3.2 KECCAK$[r = 40, c = 160]$ . . . 33
4.4 Mid-range core . . . 34
4.4.1 Description . . . 34

4.4.2 Results for Keccak[$r = 1024, c = 576$] . . . 35
4.5 Low-area coprocessor . . . 38
4.5.1 Keccak[$r = 1024, c = 576$] . . . 40
4.5.2 Keccak[$r = 40, c = 160$] . . . 40
4.6 FPGA implementations . . . 40

5 Protection against side-channel attacks 43
5.1 Introduction . . . 43
5.2 Power analysis . . . 44
5.2.1 Different types of countermeasures . . . 45
5.2.2 Secret sharing . . . 46
5.3 Software implementation using two-share masking . . . 46
5.3.1 Simplifying the software implementation . . . 47
5.4 Hardware using three-share masking . . . 48
5.4.1 One-cycle round architecture . . . 48
5.4.2 Three-cycle round architecture . . . 50
5.4.3 Synthesis results . . . 50
5.5 Computing in parallel or sequentially? . . . 52

A Change log 59
A.1 From 3.1 to 3.2 . . . 59

Introduction

This document gives an overview of the implementation aspects of Keccak, in software and hardware, with or without protection against side-channel attacks.

Acknowledgments

We wish to thank (in no particular order) Joachim Strömbergson for useful comments on our 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, Joris Delclef, Jean-Louis Modave, Yves Moulart and Armand Linkens for giving us access to fast hardware, Daniel Otte and Christian Wenzel-Benner for their support on embedded platforms, Renaud Bauvin for his implementation in Python. Francesco Regazzoni and Bernhard Jungk for fruitful discussions on the hardware implementation.

Chapter 1

General aspects

In this chapter, we first briefly review the KECCAK specifications formally defined in [10]. We then detail the bit and byte numbering conventions, followed by general statements about the operation count and memory usage of KECCAK.

1.1 Specifications summary

This section offers 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 KECCAK, we refer to [10].

Any instance of the KECCAK sponge function family makes use of one of the seven KECCAK-f permutations, denoted KECCAK-$f[b]$, where $b \in \{25, 50, 100, 200, 400, 800, 1600\}$ is the width of the permutation. These KECCAK-f permutations are iterated constructions consisting of a sequence of almost identical rounds. The number of rounds n_r depends on the permutation width, and is given by $n_r = 12 + 2\ell$, where $2^\ell = b/25$. This gives 24 rounds for KECCAK-$f[1600]$.

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

Round$[b](A, \mathrm{RC})$

θ step

$C[x] = A[x,0] \oplus A[x,1] \oplus A[x,2] \oplus A[x,3] \oplus A[x,4],\quad \forall x \text{ in } 0\ldots4$

$D[x] = C[x-1] \oplus \mathrm{ROT}(C[x+1],1),\quad \forall x \text{ in } 0\ldots4$

$A[x,y] = A[x,y] \oplus D[x],\quad \forall (x,y) \text{ in } (0\ldots4, 0\ldots4)$

ρ and π steps

$B[y, 2x+3y] = \mathrm{ROT}(A[x,y], r[x,y]),\quad \forall (x,y) \text{ in } (0\ldots4, 0\ldots4)$

χ step

$A[x,y] = B[x,y] \oplus ((\mathrm{NOT}\, B[x+1,y])\ \mathrm{AND}\ B[x+2,y]),\quad \forall (x,y) \text{ in } (0\ldots4, 0\ldots4)$

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$[r, c]$ sponge function, with parameters capacity c and bitrate r, 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](M)
  PADDING
  P = M||0x01||0x00||...||0x00
  P = P ⊕ 0x00||...||0x00||0x80

  INITIALIZATION
  S[x,y] = 0,                                   ∀(x,y) in (0...4,0...4)

  ABSORBING PHASE
  for every block P_i in P
    S[x,y] = S[x,y] ⊕ P_i[x+5y],                ∀(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],                              ∀(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 Bit and byte numbering conventions

Internally, the state of KECCAK-$f[b]$ is organized in three dimensions and its bits are identified with coordinates $x, y \in \mathbb{Z}_5$ and $z \in \mathbb{Z}_w$, for $b = 25w$ and $w \in \{1, 2, 4, \ldots, 64\}$. Externally, the sponge and duplex constructions require to have a linear numbering of the bits from 0 to $b-1$. Here, the bit index $i = z + w(5y + x)$ externally corresponds to the coordinates (x, y, z) internally.

A lane (i.e., bits with the same coordinates (x, y)) contains bits with w consecutive indices. If an input or output block is organized as lanes, the outer part of the state spans $\lceil \frac{r}{w} \rceil$ lanes. In a typical software implementation, the bits in a lane are packed together in a w-bit CPU word. Thus, this allows to organize the input and output in terms of CPU words. In addition, one can implement the operation ρ as a set of CPU word rotations (if the processor supports it).

When it comes to expressing the bit positions within a CPU word or a byte, or the byte positions within a CPU word, we take the following conventions.

1. The bits within a byte or within a CPU word are numbered from zero onwards, with the bit number i being the coefficient of 2^i when a byte or a word needs to be represented as an integer.

2. The bytes within a CPU word are numbered from zero onwards, with the byte at offset i being the coefficient of 256^i when a words needs to be represented as an integer ("little-endian" convention).

The consequence of the first convention is that, as in ρ, the operation consisting of moving bits of coordinates (x, y, z) to $(x, y, z + \delta \mod w)$ becomes a rotation "to the left" by δ positions if the lane is in a CPU word.

The consequence of the two conventions jointly is that the bytes consisting an input block do not have to be shuffled before being XORed into the state when the state is represented as an array of lanes on a little-endian processor. Similarly, the bytes consisting an output block can be taken as is from the state in the same representation.

Note that the SHA-3 API defined by NIST [27] follows the opposite convention for bit numbering and this is the reason behind the formal bit reordering described in [11].

An example of the formal bit reordering can be found in the files `KeccakSpongeIntermediateValues_*.txt` in [8].

1.2.1 Some justification for our choice

In this subsection, we detail our choice of conventions 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 [27]. (This part can be safely skipped for readers not interested.)

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 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 reordering 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.

Hence, the choice is bound to either follow the little endian or the big endian convention. We found numbering the bits (or bytes) with increasing powers of 2 (or 256) a bit easier to express than to follow a decreasing rule. Furthermore, in its call for SHA-3, NIST defined a reference platform that happens to be little endian [26]. So we decided to follow the conventions detailed above.

The convention in the `Update` function of NIST's API is different, and this is the reason for applying the formal bit reordering described in [11]. It formalizes the chosen translation between the two conventions, while having a tiny impact on the implementation. In practice, only the bits of the last byte (when incomplete) of the input message need to be shifted.

1.3 Operation count

For KECCAK, the bulk of the processing goes into the KECCAK-f permutation and the XOR of 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:

$$\left\lceil \frac{l+2}{r} \right\rceil .$$

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 1.1: Relative performance of Keccak-$f[r + c = 1600]$ with respect to Keccak[].

For an output length n smaller than or equal to the bitrate, the squeezing phase does not imply any additional processing. However, if more output bits are needed, 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 for a given permutation width b. For instances with $r + c = 1600$, we will often write performance figures for the default bitrate $r = 1024$. To estimate the performance for another bitrate, Table 1.1 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 $l \leq r - 2$).

On a platform supporting operations on w-bit words, each lane can be mapped to such a word. If the platform supports only smaller, m-bit words, each lane has to be mapped to b/m such words. There are different kinds of mappings. As long as the same mapping is applied to all lanes, each bitwise Boolean operation on a lane is translated as b/m instructions on CPU words. The most straightforward mapping is to take as CPU word i the lane bits with $z = mi \ldots m(i + 1) - 1$. In that case, the b-bit rotations need to be implemented using a number of shifts and bitwise Boolean instructions. Another possible mapping, that translates the b-bit rotations into a series of m-bit CPU word rotation instructions, is introduced in Section 2.1.

If we use a b-bit platform, the evaluation of Keccak-$f[b]$ uses in terms of lane operations

- $76n_\text{r}$ XORs,
- $25n_\text{r}$ ANDs and $25n_\text{r}$ NOTs, and
- $29n_\text{r}$ b-bit rotations.

For Keccak-$f[1600]$ specifically, this becomes

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

Some instruction sets propose an instruction that combines the AND and NOT operations in one operation as in χ, i.e., $c \leftarrow (\text{NOT } a) \text{ AND } b$. If such an instruction is not available on

a CPU, almost 80% of the NOT operations can be removed by applying a *lane complementing transform* as explained in Section 2.2, turning a subset of the AND operations into OR operations.

In [9], we provide a simple implementation (called `Simple`) that maps a lane to a CPU word of 8, 16, 32 or 64 bits, and hence is suitable for KECCAK-f[200], KECCAK-f[400], KECCAK-f[800] and KECCAK-f[1600].

1.4 Memory

In terms of memory usage, KECCAK has no feedforward loop, as opposed to many other constructions, and the message block can be directly XORed into the state. This can benefit applications for which the message is formatted on the fly or does not need to be kept after being hashed. This also applies where the hashing API has to implement a message queue. In general a message queue must be allocated, which can be avoided for sponge functions or similar.

The amount of working memory is limited 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 being computed.

An efficient memory-constrained implementation technique is presented in Section 2.5.

Examples of implementations that use a limited amount of memory can be found in the `Compact`, `Compact8`, `Inplace` and `Inplace32BI` implementations in [9]. Some of these implementations also provide an API with a message queue such as `Init`, `Update` and `Final`, for which no extra memory needs to be allocated.

Chapter 2

Implementation techniques

In this chapter, we introduce techniques that can be used to optimize software and hardware implementations, namely bit interleaving, lane complementing, plane-per-plane processing, efficient in-place evaluation and consecutive or interleaved slices processing.

2.1 Bit interleaving

The technique of bit interleaving consists in coding an w-bit lane as an array of $s = w/m$ CPU words of m bits each, with word ζ containing the lane bits with $z \equiv \zeta \pmod{s}$. Here, s is called the *interleaving factor*. This technique can be applied to any version of KECCAK-f to any CPU with word length m that divides its lane length w.

We first treat the concrete case of 64-bit lanes and 32-bit CPU words, so a factor-2 interleaving. 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_0 and U_1 with $U_0[j] = L[2j]$ and $U_1[j] = L[2j+1]$. 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 $U_0 \leftarrow \mathrm{ROT}_{32}(U_1, \tau + 1)$ and $U_1 \leftarrow \mathrm{ROT}_{32}(U_0, \tau)$. 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. For KECCAK-$f[1600]$ specifically, this is the case for 6 out of the 29 lane translations in each round (5 in θ and 1 in ρ).

More generally, let us consider an interleaving factor s. The words U_ζ, with $\zeta \in \mathbb{Z}_s$, represent the lane L. All the operations on the ζ coordinate are considered modulo s. A translation of L by τ positions causes each word U_ζ to be moved to $U_{\zeta'}$ with $\zeta' = \zeta + \tau$, and to be rotated by $\lfloor \frac{\tau}{s} \rfloor + 1$ bits if $\zeta' < \tau \bmod s$ or by $\lfloor \frac{\tau}{s} \rfloor$ bits if $\zeta' \geq \tau \bmod s$.

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 [10], 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 sponge 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. On some platforms, look-up tables can speed up this process, although the gain is not always significant.

> Several examples of implementations making use of bit interleaving can be found in [9], notably the `Reference32BI` and `Simple32BI` implementations. Also, the `Optimized32` implementation can be set to use look-up tables by defining the `UseInterleaveTables` symbol in `KeccakF-1600-opt32-settings.h`. Intermediate values to help debug a bit-interleaved implementation can be found in the file `KeccakPermutationIntermediateValues32BI.txt` in [8]. Finally, KECCAKTOOLS can generate code that make use of bit interleaving for any lane size and any (smaller) CPU word size [12].

2.2 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 ($\bar{a} \wedge \bar{b} = \overline{a \vee b}$):

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

So we have

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

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.

The `Optimized32` and `Optimized64` implementations can be set to use lane complementing by defining the `UseBebigokimisa` symbol in `KeccakF-1600-opt*-settings.h` [9]. KECCAKTOOLS can generate code that make use of lane complementing for any lane size [12].

2.3 Extending the state for smoother scheduling

A naive serialized implementation of the KECCAK-f round function would perform the different steps sequentially: one phase per step. In such a scheduling the full state is processed four times per round, typically resulting in high load-and-store overhead. One may reduce this number by combining multiple steps in a single phase. For example, the two bit transpositions ρ and π can be computed in a single phase. In this section we describe a technique to reduce the number of phases to two.

The main idea is to adopt a *redundant state representation that includes the column parities*. The main phase updates the state bits and can be serialized down to the level of rows. The other phase updates the column parity bits and can be serialized down to the level of bits. At the expense of some additional storage overhead, the two phases can be combined into one. This allows efficient and compact serialized hardware architectures and software scheduling.

Let us consider the computation of a row of the round output given the round input in extended representation. The last step is ι that simply consists of possibly flipping a bit of the row. Before that, there is χ that can be applied to the row. This is preceded by the bit transpositions π and ρ that can be implemented by taking the bits from the appropriate positions of the state. The step θ requires adding two parity bits to each of the five bits of the row. If one extends the state with the θ-effect rather than the column parity, the step θ requires the addition of a single bit of the θ-effect to each bit of the row. It follows that the computation of a single row only requires as input 5 bits of the state and 5 bits of the θ-effect. In hardware, in principle one can compute rows serially by a small circuit that implements χ restricted to a row and the addition of 5 θ-effect bits. ρ and π are materialized

by RAM addressing. In software there is no point in computing single rows. However, one may compute sets of rows in parallel by making use of the bitwise Boolean instructions and cyclic shifts. The simplest such grouping is by taking all the rows in a plane. Here ρ consists of five cyclic shifts and π requires addressing. One can also apply bit-interleaving and group all rows in a plane with even or odd z coordinates.

The computation of the θ-effect from a state can done bit-by-bit. For a given column, one can initialize its column parity to zero and accumulate the bits of a column to it one by one. This can again be grouped: lane-by-lane or using bit-interleaving. This computation can however also be integrated in a single phase together with the computation of the round output, as will be explained in Section 2.4.1. After χ has been applied to a row, one accumulates each of the bits in the corresponding column parity. After all rows of a slice have been computed, the corresponding column parities will have their correct value. Now these column parities must simply be combined into the θ-effect. Note that this scheduling implies the storage of the θ-effect of the round input and storage for the θ-effect of the round output that is being computed.

The five bits of the round input used in the computation of a row of the round output are not used in the computation of any other row. So the registers containing these round input bits may be overwritten by the computed round output bits. This allows reducing working memory for storing intermediate computation results to the working state, the θ-effect of the round input and possible the column parity effect being computed. This is interesting in software implementations on platforms with restricted RAM. It goes at the expense of more complex addressing as the position of statebits depends on the round number.

2.4 Plane-per-plane processing

This section describes a general technique to schedule the order of operations for high efficiency, both in software and serialized hardware architectures. The main idea is to process one plane at a time centered around χ, i.e., $y = 0, 1, \ldots, 4$ from the point of view of the coordinates at the output of the round.

Let us take the point of view of the evaluation of χ applied to the five lanes of a plane y, denoted $B[\cdot, y]$. We can track each lane $B[x, y]$ at the input of χ to a lane at the beginning of the round $A[x', y']$, with $(x', y')^{\mathrm{T}} = M^{-1}(x, y)^{\mathrm{T}}$ and M^{-1} the inverse of the matrix used in π. Between $A[x', y']$ and $B[x, y]$ are the application of θ and a cyclic shift ρ.

Assuming the parity effect of θ is already computed, as the values $D[\cdot]$ in [10, Algorithm 3], we have:

$$B[x, y] = \mathrm{ROT}((A[x', y'] \oplus D[x']), r[x', y']), \text{ with } \begin{pmatrix} x' \\ y' \end{pmatrix} = M^{-1} \begin{pmatrix} x \\ y \end{pmatrix},$$

for a fixed y. Then, the plane y at the output of the round is obtained as

$$E[x, y] = B[x, y] \oplus ((\mathrm{NOT}\, B[x+1, y])\, \mathrm{AND}\, B[x+2, y]),$$

without forgetting to XOR $E[0, 0]$ with the round constant to implement ι. Note that, since y is fixed, the variable B is a plane and can be indexed only by its x coordinate. To get the parities of the columns before θ, denoted $C[\cdot]$, the simplest solution is to compute them as a first phase in the round. All the steps are summarized in Algorithm 1. Note that all lane addresses can be fixed by loop unrolling. The output variables $E[\cdot, \cdot]$ can be used as input for the next round.

Most software implementations available actually use the plane-per-plane processing [9]. The code generated with KeccakTools's `KeccakFCodeGen::genCodePlanePerPlane()` follows this idea too [12]. The code generation can be combined with lane complementing if desired.

Algorithm 1 Plane-per-plane processing

```
for x = 0 to 4 do
  C[x] = A[x,0] ⊕ A[x,1] ⊕ A[x,2] ⊕ A[x,3] ⊕ A[x,4]
end for
for x = 0 to 4 do
  D[x] = C[x−1] ⊕ ROT(C[x+1],1)
end for
for y = 0 to 4 do
  for x = 0 to 4 do
    B[x] = ROT((A[x',y'] ⊕ D[x']), r[x',y']), with (x', y')^T = M^{-1} (x, y)^T
  end for
  for x = 0 to 4 do
    E[x,y] = B[x] ⊕ ((NOT B[x+1]) AND B[x+2])
  end for
end for
E[0,0] = E[0,0] ⊕ RC[i]
```

2.4.1 Early parity

To compute the parities of the columns, an alternate option is to compute $C[\cdot]$ ahead during a round for the next one: $C[x]$ accumulates the values $E[x,y]$ on the fly as they are computed. We call this *early parity* and it is described in Algorithm 2. Of course, when using the early parity, the parity C must still be computed before the first round and does not need to be computed during the last round.

Algorithm 2 Plane-per-plane processing with early parity (ι not shown)

```
Assuming C[·] already computed
for x = 0 to 4 do
  D[x] = C[x−1] ⊕ ROT(C[x+1],1)
end for
C = (0,0,0,0,0)
for y = 0 to 4 do
  for x = 0 to 4 do
    B[x] = ROT((A[x',y'] ⊕ D[x']), r[x',y']), with (x', y')^T = M^{-1} (x, y)^T
  end for
  for x = 0 to 4 do
    E[x,y] = B[x] ⊕ ((NOT B[x+1]) AND B[x+2])
    C[x] = C[x] ⊕ E[x,y]
  end for
end for
```

KECCAKTOOLS's function `KeccakFCodeGen::genCodePlanePerPlane()` has a parameter `prepareTheta` that tells whether to include the computation of $C[\cdot]$ ahead for early parity.

2.4.2 Combining with bit interleaving

Plane-per-plane processing can be combined with bit interleaving. We adapt Algorithm 1 by writing it in terms of operations on m-bit words. All the variables are now m-bit words, instead of w-bit lanes, and we add an extra coordinate ζ when necessary, with $\zeta \in \mathbb{Z}_s$ and $s = w/m$ the interleaving factor. All the operations on the x, y (resp. ζ) coordinates are considered modulo 5 (resp. modulo s).

The resulting algorithm is displayed in Algorithm 3. In the main loop (starting from the 10th line), the coordinates x, y and ζ are from the point of view of the output of the round. The coordinates x', y' and ζ' express which input word has to be fetched.

Algorithm 3 Plane-per-plane processing with bit interleaving of factor s

for $x = 0$ to 4 **and** $\zeta = 0$ to $s-1$ **do**
  $C[x,\zeta] = A[x,0,\zeta] \oplus A[x,1,\zeta] \oplus A[x,2,\zeta] \oplus A[x,3,\zeta] \oplus A[x,4,\zeta]$
end for
for $x = 0$ to 4 **do**
  $D[x,0] = C[x-1,0] \oplus \mathrm{ROT}(C[x+1,s-1],1)$
  for $\zeta = 1$ to $s-1$ **do**
    $D[x,\zeta] = C[x-1,\zeta] \oplus C[x+1,\zeta-1]$
  end for
end for
for $y = 0$ to 4 **and** $\zeta = 0$ to $s-1$ **do**
  for $x = 0$ to 4 **do**
    Let $(x',y')^{\mathrm{T}} = M^{-1}(x,y)^{\mathrm{T}}$ and $\zeta' = \zeta - r[x',y'] \mod s$
    Let $r = \lfloor \frac{r[x',y']}{s} \rfloor + 1$ if $\zeta < r[x',y'] \mod s$, or $r = \lfloor \frac{r[x',y']}{s} \rfloor$ otherwise
    $B[x] = \mathrm{ROT}((A[x',y',\zeta'] \oplus D[x',\zeta']), r)$
  end for
  for $x = 0$ to 4 **do**
    $E[x,y,\zeta] = B[x] \oplus ((\mathrm{NOT}\, B[x+1])\ \mathrm{AND}\, B[x+2])$
  end for
end for
$E[0,0,\zeta] = E[0,0,\zeta] \oplus \mathrm{RC}_s[i,\zeta]$ **for** $\zeta = 0$ to $s-1$

> The code generated with KECCAKTOOLS's function `KeccakFCodeGen::genCodePlanePerPlane()` can be combined with bit interleaving [12].

2.5 Efficient in-place implementations

In this section, we adapt the addressing in the plane-per-plane processing to minimize the memory needed to compute KECCAK-f.

In the previous section, the computation of the round function extracted its input from variables denoted A and stored the output into variables E. These two sets of variables, each of the size of b bits, must be distinct to avoid overwriting variables required for subsequent computations; hence, $2b$ bits are needed to store both A and E. In this section, we propose a way to do plane-per-plane processing using only the A set of variables, hereby reducing the memory consumption by b bits. To do this, we allow $A[x,y]$ to store lanes of coordinates not necessarily $(x,y)^{\mathrm{T}}$.

When processing a plane y as in Algorithm 1, χ needs five lanes coming from the coordi-

nates $M^{-1}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x+3y \\ x \end{pmatrix}$ for all $x \in \mathbb{Z}_5$ at the input of the round. This determines a line with slope 1 going through $(0, 2y)$ (input variety). The five lanes at the output of χ are per construction on a line with slope 0 going through $(0, y)$ (output variety). To avoid overwriting other variables, we must store the five lanes of the output variety in the memory location that held the input variety. We choose that the relationship between the lane coordinates and the memory location is represented by a linear matrix $N \in \mathbb{Z}_5 \times \mathbb{Z}_5$. At the end of the first round, the lane at $(x, y)^{\mathrm{T}}$ is stored at location $N(x, y)^{\mathrm{T}}$. To satisfy the mapping of the output variety onto the input variety, N must be of the form $\begin{pmatrix} 1 & a \\ 1 & b \end{pmatrix}$ with $a + b = 2$. We choose to set $a = 0$ so as to leave the x coordinate unchanged during the mapping, hence

$$N = \begin{pmatrix} 1 & 0 \\ 1 & 2 \end{pmatrix}.$$

At the beginning of the second round, the input variety appears rotated by N in memory so the lane at $(x, y)^{\mathrm{T}}$ is stored at location $N^2(x, y)^{\mathrm{T}}$ at the end of the second round. In general at round i, the lanes with coordinates $(x, y)^{\mathrm{T}}$ are stored at location $N^i(x, y)^{\mathrm{T}}$.

The input of χ with coordinates $(x, y)^{\mathrm{T}}$ requires the lane with coordinates $M^{-1}(x, y)^{\mathrm{T}}$ before π, which is located at coordinates $N^i M^{-1}(x, y)^{\mathrm{T}}$ in memory. Notice that $N^i M^{-1}(x, y)^{\mathrm{T}} = (x + 3y, \dots)$, so there is a shift in the x coordinate inherent to the lane-per-lane processing. This can be explicitly compensated for by multiplying all coordinates by $MN = \begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix}$ and integrating this shift before or after χ, as χ is translation-invariant. This leads to a simpler expression: The input of χ with coordinates $(x + 2y, y)^{\mathrm{T}}$ requires the lane with coordinates $N(x, y)^{\mathrm{T}}$ before π, which is located at coordinates $N^{i+1}(x, y)^{\mathrm{T}}$ in memory. The resulting algorithm can be found in Algorithm 4.

Note the following properties of N:

- The matrix N has order 4, so after every 4 rounds the variable A will contain the state without transposition. Unrolling 4 rounds implies that all the coordinates can be constants, and the memory or register indexing can be hardcoded.
- Expressions like $N^j(x, y)^{\mathrm{T}}$ are of the form $(x, f_i(x, y))^{\mathrm{T}}$. So in an implementation that does not unroll the rounds, only the y coordinate changes compared to Algorithms 1 and 2.

Finally, note that Algorithm 4 can also be combined with early parity, lane complementing and bit interleaving techniques, the latter being detailed below.

> The `Inplace` and `Inplace32BI` implementations explicitly use the techinque presented in this section on KECCAK-$f[1600]$, one using 64-bit rotations and the other one using bit interleaving for 32-bit rotations [9]. KECCAKTOOLS's `KeccakFCodeGen::genCodeInPlace()` can generate code for 4 unrolled rounds using this in-place technique, with factor-2 interleaving or without it [12].

2.5.1 Combining with bit interleaving

Similarly to Section 2.4.2, we adapt Algorithm 4 by writing it in terms of operations on m-bit words. All the variables are now m-bit words, instead of w-bit lanes, and we add an extra coordinate ζ when necessary, with $\zeta \in \mathbb{Z}_s$ and $s = w/m$ the interleaving factor. All the operations on the x, y (resp. ζ) coordinates are considered modulo 5 (resp. modulo s).

Algorithm 4 In-place processing of round i

> **for** $x = 0$ to 4 **do**
> $C[x] = A[N^i(x,0)^{\mathrm{T}}] \oplus A[N^i(x,1)^{\mathrm{T}}] \oplus A[N^i(x,2)^{\mathrm{T}}] \oplus A[N^i(x,3)^{\mathrm{T}}] \oplus A[N^i(x,4)^{\mathrm{T}}]$
> **end for**
> **for** $x = 0$ to 4 **do**
> $D[x] = C[x-1] \oplus \mathrm{ROT}(C[x+1], 1)$
> **end for**
> **for** $y = 0$ to 4 **do**
> **for** $x = 0$ to 4 **do**
> $B[x+2y] = \mathrm{ROT}((A[N^{i+1}(x,y)^{\mathrm{T}}] \oplus D[x]), r[N(x,y)^{\mathrm{T}}])$
> **end for**
> **for** $x = 0$ to 4 **do**
> $A[N^{i+1}(x,y)^{\mathrm{T}}] = B[x] \oplus ((\mathrm{NOT}\, B[x+1])\, \mathrm{AND}\, B[x+2])$
> **end for**
> **end for**
> $A[0,0] = A[0,0] \oplus \mathrm{RC}[i]$

While the memory location of a given lane vary from round to round according to the $N^i(x,y)^{\mathrm{T}}$ rule, the ζ coordinate of words varies as well. This is because, five words are fetched and processed from a given set of five memory locations, and the resulting five words must be stored into the same five locations, for the implementation to be in-place. Let us first illustrate this with an example using interleaving with factor $s = 4$ (e.g., to compute KECCAK-f[1600] with 16-bit operations) and using the notations of Algorithm 3. At the beginning of the first round $i = 0$, words are stored in memory locations with the identical coordinates. Assume that we want to process the quarter-plane $y = 2$ and $\zeta = 0$. At some point, we need to compute the word $B[0]$, i.e., word $(x, y, \zeta) = (0, 2, 0)$ at the input of χ. To do so, we need the word $A[1, 0, 3]$ since $(x', y')^{\mathrm{T}} = (1, 0)^{\mathrm{T}} = M^{-1}(x, y)^{\mathrm{T}}$ and $r[0, 0] = 1$ so $\zeta' = \zeta - 1 = 3 \pmod s$. At the beginning of the round, word $(1, 0, 3)$ is stored in memory at $(1, 0, 3)$. We will need to reuse the memory locations that we just processed. So, to respect the $N^i(x,y)^{\mathrm{T}}$ rule, all words belonging to lane $(1, y) = (1, 2)$ have to be stored at memory locations $(1, 0, \cdot)$ since $N(1, 2)^{\mathrm{T}} = (1, 0)^{\mathrm{T}}$. In particular, word $(1, y, \zeta) = (1, 2, 0)$ needs to be stored at memory location $(1, 0, 3)$ since it is the only memory location free at this point within $(1, 0, \cdot)$. So the third coordinate of the word (0 in this example) can become different from the third coordinate of the memory location to store it (3 in this example). For the other words of the lane, there will be a constant shift between the third coordinates.

In general, the relationship between the third coordinate of words and that of the memory location depends on the rotation constants. We model this relationship as follows. At the beginning of round i, word ζ of lane $(x, y)^{\mathrm{T}}$ is stored at memory location $(x', y', \zeta + O(i, x', y'))$ with $(x', y')^{\mathrm{T}} = N^i(x, y)^{\mathrm{T}}$. Here $O(i, x, y)$ is viewed as a property of the memory cells $(x, y, \cdot)$: it tells by how many positions the words are shifted.

Using the notations of Algorithm 3, word (x', y', ζ') is read in order to evaluate output word (x, y, ζ). Word (x', y', ζ') is located in memory at $(x'', y'', \zeta' + O(i, x'', y''))$, with $(x'', y'')^{\mathrm{T}} = N^i(x', y')^{\mathrm{T}}$, which can now be reused. Hence some output word within the part of the plane being processed $(\cdot, y, \zeta)$ is stored at the same memory position $(x'', y'', \zeta' + O(i, x'', y''))$ at the end of round i. Looking only at the third coordinate, we use memory location $\zeta' + O(i, x'', y'') = \zeta - r[x', y'] + O(i, x'', y'') \pmod s$ to store word ζ at the beginning of round $i + 1$. Since $r[x', y'] = r[N^{-i}(x'', y'')^{\mathrm{T}}]$, we identify $O(i + 1, x'', y'') =$

$O(i, x'', y'') - r[N^{-i}(x'', y'')^{\mathrm{T}}]$ and we obtain:

$$O(i, x, y) = -\sum_{j=0}^{i-1} r[N^{-j}(x, y)^{\mathrm{T}}] \pmod{s}.$$

The resulting algorithm is displayed in Algorithm 5.Note that $O(i, 0, 0) = 0$ for all rounds i, so the last line (evaluation of ι) can be simplified.

Algorithm 5 In-place processing of round i with bit interleaving of factor s

for $x = 0$ to 4 **and** $\zeta = 0$ to $s - 1$ **do**
  Let $\zeta_y = \zeta + O(i, N^i(x, y)^{\mathrm{T}})$
  $C[x, \zeta] = A[N^i(x, 0)^{\mathrm{T}}, \zeta_0] \oplus A[N^i(x, 1)^{\mathrm{T}}, \zeta_1] \oplus A[N^i(x, 2)^{\mathrm{T}}, \zeta_2] \oplus A[N^i(x, 3)^{\mathrm{T}}, \zeta_3] \oplus A[N^i(x, 4)^{\mathrm{T}}, \zeta_4]$
end for
for $x = 0$ to 4 **do**
  $D[x, 0] = C[x - 1, 0] \oplus \mathrm{ROT}(C[x + 1, s - 1], 1)$
  for $\zeta = 1$ to $s - 1$ **do**
    $D[x, \zeta] = C[x - 1, \zeta] \oplus C[x + 1, \zeta - 1]$
  end for
end for
for $y = 0$ to 4 **and** $\zeta = 0$ to $s - 1$ **do**
  for $x = 0$ to 4 **do**
    Let $(x'', y'')^{\mathrm{T}} = N^{i+1}(x, y)^{\mathrm{T}}$ and $\zeta' = \zeta - r[N(x, y)^{\mathrm{T}}] \mod s$
    Let $r = \lfloor \frac{r[N(x,y)^{\mathrm{T}}]}{s} \rfloor + 1$ if $\zeta < r[N(x, y)^{\mathrm{T}}] \mod s$, or $r = \lfloor \frac{r[N(x,y)^{\mathrm{T}}]}{s} \rfloor$ otherwise
    $B[x + 2y] = \mathrm{ROT}((A[x'', y'', \zeta' + O(i, x'', y'')] \oplus D[x, \zeta']), r)$
  end for
  for $x = 0$ to 4 **do**
    Let $(x'', y'')^{\mathrm{T}} = N^{i+1}(x, y)^{\mathrm{T}}$
    $A[x'', y'', \zeta + O(i + 1, x'', y'')] = B[x] \oplus ((\mathrm{NOT}\, B[x + 1])\ \mathrm{AND}\ B[x + 2])$
  end for
end for
$A[0, 0, \zeta + O(i + 1, 0, 0)] = A[0, 0, \zeta + O(i + 1, 0, 0)] \oplus \mathrm{RC}_s[i, \zeta]$ **for** $\zeta = 0$ to $s - 1$

2.5.1.1 Special case: interleaving factor 2

We now look at the specific case of an in-place implementation using an interleaving of factor $s = 2$, e.g., to implement KECCAK-$f[1600]$ using 32-bit operations or to implement KECCAK-$f[400]$ using 8-bit operations. After 4 rounds, one would expect all the lanes to come back to their initial positions but not necessarily the word offsets within the lanes, requiring another 4 rounds for $O(8, x, y)$ to be zero (modulo 2). However, the nice thing about this special case is that $O(4, x, y) = 0 \pmod{2}$ for all coordinates x, y. This implies that after 4 rounds, all the words within the lanes are also back to their initial positions.

To verify that $O(4, x, y) = -\sum_{j=0}^{3} r[N^{-j}(x, y)^{\mathrm{T}}] = 0 \pmod{2}$ for all x, y coordinates, we start with two properties of N:

- The matrix N and its powers leave the set $x + y = 0$ unchanged, i.e., the set $x + y = 0$ is an eigenspace with eigenvalue 1.

- Except for the points in the set $x + y = 0$, the points $N^i(x, y)^T$ stay in column x, take all the positions such that $y \neq -x$ and come back to their initial position after a cycle of length 4.

For coordinates satisfying $x + y = 0$, $r[N^{-j}(x, y)^T]$ is a constant so clearly 2 divides $O(4, x, y)$. For the other coordinates, it suffices to check that the sum of the round constants over the 4 coordinates $y \neq -x$ in each column is zero modulo 2.

> The `Inplace32BI` implementation exploits this by unrolling 4 rounds [9].

2.5.1.2 Special case: interleaving factor 4

We now look at the specific case of an in-place implementation using an interleaving of factor $s = 4$, e.g., to implement KECCAK-f[1600] using 16-bit operations or to implement KECCAK-f[800] using 8-bit operations. Extending the same reasoning from the previous case $s = 2$, we see that $O(4, x, y) \in \{0, 2\} \pmod 4$ for all coordinates x, y. This implies that after 8 rounds, we have $O(8, x, y) = 2O(4, x, y) = 0 \pmod 4$ and all the words within the lanes are back to their initial positions.

If unrolling 8 rounds is a problem, we can unroll 4 rounds and compute $O(4, x, y)$ to see how words are stored. Fortunately, $O(4, x, y) = 0 \pmod 4$ for all x, y coordinates except $(0, 1)$, $(0, 2)$, $(0, 3)$, $(0, 4)$, for which $O(4, x, y) = 2 \pmod 4$. So after 4 rounds it suffices to interchange $A[x, y, \zeta]$ with $A[x, y, \zeta + 2]$ for the x, y coordinates in the set above.

2.6 Processing slices

In this section, we discuss implementation techniques that are essentially hardware-oriented. They consist in processing together a slice or a group of slices, either with consecutive z coordinates or with constant coordinate z modulo some number.

2.6.1 Processing consecutive slices

The idea of processing consecutive slices in a hardware circuit comes from Jungk and Apfelbeck [20]. While the state of KECCAK-f[$25w$] can be seen as an array of 25 lanes of w bits, the transposed view is to see it as an array of w slices of 25 bits each. The function of ρ is to disperse bits across different slices, but all the other operations work in a slice-oriented way. More precisely, π and χ work in each slice independently, and for θ the output slice z depends on the input slices $z - 1$ and z. Hence, a hardware implementation can process groups of n slices with consecutive z coordinates, where n divides w. The number n of consecutive slices can serve as a parameter for speed-area trade-offs.

The round function needs to be rescheduled to perform the operations as $\pi \circ \rho \circ \theta \circ \iota \circ \chi$. Note that ρ and π can be interchanged, so that π can be grouped with the other slice-oriented operations. To start the permutation, one has to compute $\pi \circ \rho \circ \theta$ and to end it $\iota \circ \chi$.

> Jungk and Apfelbeck implemented KECCAK-f[1600] using $n = 8$ consecutive slices, with extra registers to manage the fact that θ makes the last slice of a group interact with the first slice of the next group. Inter-slice dispersion ρ is implemented in part by an appropriate addressing of RAM and in part by extra registers. This resulted in a compact implementation [20], see also Table 4.5.

2.6.2 Processing interleaved slices

An alternative to the idea of consecutive slices is to group slices in an interleaved way. For a given interleaving factor s, the slices are grouped with constant coordinate z modulo s.

This technique has similar properties as for processing consecutive slices. For instance, we expect it allows similar speed-area trade-offs in hardware implementations, as the number of slices to be grouped can be chosen. In contrast, however, ρ can be partly done within a group of slices. This removes the need for the extra registers dedicated to ρ, but some extra memory could be needed to store the parity (as in Section 2.3) so as to implement θ without interdependencies between groups of slices.

At this time, we are not aware of an existing implementation making use of this technique.

Chapter 3

Software

In this chapter, we give an overview of software implementations. Please note that we do not provide extensive benchmark results of KECCAK on various platforms. Instead, we refer to eBASH and XBX for detailed and up-to-date benchmarking data on a wide range of platforms [2, 33]. In addition, a summary of the data coming from eBASH and XBX can be found on our web page [7].

3.1 PC and high-end platforms

3.1.1 Using 64-bit instructions

The platforms supporting 64-bit instructions are in general well-suited for KECCAK-f[1600], as a lane can be mapped onto a CPU word. The x86-64 instruction set present in the CPU of most recent PCs is a widely-used example. This instruction set does not have a combined AND+NOT instruction, so lane complementing can be used to reduce the number of NOT instructions. At the time of writing, the permutation KECCAK-f[1600] plus XORing 1024 bits takes slightly more than 1600 cycles on a typical x86-64-based machine, hence enabling to absorb long messages with KECCAK[] at about 12.6 cycles/byte. KECCAK[] performs very well on IA64-based machines too, at around 6-7 cycles/byte [7, 2].

In [9], we provide an implementation suitable for 64-bit CPUs called `Optimized64`. The code uses only plain C instructions, without assembly nor SIMD instructions. If needed, we have applied lane complementing to reduce the number of NOTs. The operations in the round function have been expanded in macros generated by KECCAKTOOLS [12]. 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.
In `KeccakF-1600-opt64-settings.h`, the number of rounds unrolled can be set to any number dividing 24, and the use of lane complementing can be turned on or off.

- For x86-64, the fastest option is to unroll 24 rounds (`#define Unrolling 24`) and to use lane complementing (`#define UseBebigokimisa`).
- For IA64, unrolling 6 rounds and *not* using lane complementing give the fastest results.

3.1.2 Using SIMD instructions

Some platforms do not have plain 64-bit instructions but have single-instruction multiple-data (SIMD) units capable of processing 64-bit or 128-bit data units.

In the family of Intel, AMD and compatible CPUs, a majority supports SIMD instruction sets known as MMX, SSE, AVX, XOP 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. 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 implementation of the cyclic shifts in ρ depends significantly on the instructions available. In SSE, no rotation instructions are present but only (left and right) shifts by the same amount. Consequently, the rotations in ρ cannot fully benefit from the 128-bit registers, whereas the rotations in θ are all of the same amount and can be combined efficiently. In the XOP instruction set, however, the instruction `vprotq` allows one to do cyclic shifts of two 64-bit words in parallel. Furthermore, the rotations can be of different amounts for each of the two words. Using XOP instead of plain x86-64 instructions on some recent AMD processors increases the throughput by a factor 1.75.

If a modern 64-bit CPU such as an Intel Core 2 Duo or an AMD Athlon 64 is restricted to 32-bit instructions (e.g., if the installed operating system uses only a legacy 32-bit mode), using the 128-bit registers of SSE yields faster throughput than using plain 32-bit instructions on such platforms. In such a case, the evaluation of KECCAK-f[1600] and XORing 1024 bits takes about 2500 cycles, hence enabling to absorb long messages with KECCAK[] at about 20 cycles/byte.

Some older PCs have 32-bit CPUs and a MMX unit with 64-bit SIMD registers. In such a case, it is also interesting to use SIMD instructions. E.g., on an Intel Pentium 3, the evaluation of KECCAK-f[1600] and XORing 1024 bits takes about 5200 cycles, hence enabling to absorb long messages with KECCAK[] at about 41 cycles/byte. This represents a 40% speedup compared to an implementation using only 32-bit instructions.

Finally, some ARM processors propose the NEON instruction set, which provides 64-bit registers and 64-bit operations. Furthermore, some of these operations can be combined so as to execute two of them in one clock cycle in a SIMD fashion. With our current implementations, using NEON gives a 60% speedup compared to an implementation using only 32-bit instructions.

> In [9], we provide implementations for 64-bit MMX and 128-bit SSE instructions. This can be set in `KeccakF-1600-opt64-settings.h`, by defining either `UseMMX` or `UseSSE`. Files `Keccak*-crypto_hash-inplace-armgcc-ARMv7A-NEON.s` contain assembly implementations using NEON.

3.1.3 SIMD instructions and tree hashing

Parallel evaluations of two instances of KECCAK can also benefit from SIMD instructions, for example in the context of tree hashing. In particular, a tree hashing mode as defined in [4] using KECCAK[] and parameters $(G = \text{LI}, H = 1, D = 2, B = 64, C = c = 576)$ 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 or XOP 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 the tree hashing mode, the message bits can be directly input into the 128-bit 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 function then by the second sponge function. This matches how the data are organized in the SIMD registers, where 64 bits are used to compute one instance of Keccak-f[1600] and the other 64 bits to compute the second instance.

On a PC with an Intel Sandy Bridge architecture, we have measured the speed of the double evaluation of Keccak-f[1600] and the XOR of two blocks of 1024 bits. This takes about 1800 cycles, hence enabling to absorb long messages at about 7 cycles/byte. Furthermore, one may consider the case $r = 1088$ and $C = c = 512$, 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 to about 6.7 cycles/byte. On recent AMD processors with the XOP instruction set, the throughput is even higher thanks to the SIMD rotation instruction (less than 6 cycles/byte with $r = 1088$).

3.1.4 Batch or tree hashing on a graphics processing unit

Using a graphics processing unit (GPU) allows for a high number of instances of Keccak to be computed in parallel. This can be useful in the case of the batch evaluation of several hash functions or of several key streams, or in the case of tree hashing. We are aware of at least two implementations of Keccak on a graphics processing unit [19, 30].

3.2 Small 32-bit platforms

In this section, we consider the implementation of Keccak on a 32-bit platform without a SIMD unit.

A first question to ask ourselves—at least if this choice is possible due to interoperability constraints—is to consider using either Keccak-f[800] or Keccak-f[1600]. On the one hand, for a given security level—hence a given capacity c—a sponge function using Keccak-f[1600] allows processing 800 more input bits per call to the permutation than one using Keccak-f[800]. On the other hand, the memory footprint of Keccak-f[800] is smaller as its the state uses only 25 words of 32 bits, while this number raises to 50 for Keccak-f[1600]. A lower number of words usually means a lower number of load/store operations from/to memory.

If one goes for Keccak-f[1600], the best is to make use of the bit interleaving technique (see Section 2.1). This way, all the operations are done on 32-bit words, including the rotations. If one goes for Keccak-f[800], each lane is simply mapped to a CPU word.

For 32-bit platforms in general, good starting points are the following.

- The target `Simple32BI` contains a simple yet optimized implementation of Keccak-f[1600] in C using bit interleaving (see file `Keccak-simple32BI.c`).
- Alternatively, the target `Inplace32BI` is almost as simple as `Simple32BI`, with the added benefit of using less RAM.
- The target `Simple` can be used to instantiate a simple yet optimized implementation of Keccak-f[800] in C, mapping a lane to a CPU word (see file `Keccak-simple.c` and set cKeccakB to 800 in `Keccak-simple-settings.h`).

3.2.1 Implementation on a ARM Cortex-M0 and -M3

The ARM Cortex-M3 is a 32-bit RISC microcontroller core with 16 registers. Its instruction set contains a combined AND+NOT instruction called `bic`, so that lane complementing does not give a speed advantage. The 32-bit rotations are natively supported and such rotations can be combined with other operations thanks to its barrel shifter. The ARM Cortex-M0 is similar, with a smaller footprint and a restricted instruction set.

With our best implementations, Keccak[] takes about 144 cycles/byte (for -M0) or 95 cycles/byte (for -M3) for long messages and 272 bytes of RAM on the stack, according to our measurements.

In [9], we provide implementations in assembly of Keccak-f[1600] for ARM Cortex-M0 and -M3; see the files `Keccak-inplace32BI-armgcc-ARMv6M.s` and `-v7M.s`. It uses bit interleaving and takes advantage of the barrel shifter to combine some XORs and rotations in one instruction.

3.3 Small 8-bit platforms

In this section, we consider the implementation of Keccak on a small 8-bit platform. As for small 32-bit platforms, the first question to ask ourselves is which instance of Keccak-f to implement. Generally speaking, larger instances allow having larger rates and processing more input bits for a given capacity c. However, the required memory also increases.

The bit interleaving technique, described in Section 2.1, could be used to represent the 64-bit, 32-bit and 16-bit lanes of Keccak-f[1600], Keccak-f[800] and Keccak-f[400], respectively, with bytes. Byte-level rotations, however, are not often present on 8-bit platforms. Rotations that propagate a bit through the carry are usually natively supported and can be used to perform rotations over lanes of any size, by chaining such rotations through all the bytes of a lane. So, in practice, we did not find bit interleaving to be advantageous in this case, although it may become so on some specific platforms.

If a small capacity of $c = 160$ bits or so is enough for a given application, one can use Keccak-f[200], which uses only 8-bit operations and can be made very compact, as the state only uses 25 bytes.

For small 8-bit platforms in general, the target `Compact8` in [9] is a good starting point to implement Keccak-f[1600] on an 8-bit processor. Also, the target `Simple` can be used to instantiate a simple yet optimized implementation of Keccak-f[200], mapping a lane to a byte (see file `Keccak-simple.c` and set cKeccakB to 200 in `Keccak-simple-settings.h`), although this implementation was not optimized for size.

3.3.1 Implementation on a Atmel AVR processor

The AVR platform uses an 8-bit RISC processor with 32 single-byte registers. With our best implementation on this processor, KECCAK[] takes about 1110 cycles/byte for long messages and 281 bytes of RAM on the stack, according to our measurements.

> In [9], we provide an implementation in assembly of KECCAK-f[1600] for Atmel AVR; see the files `Keccak-avr8*`.

Chapter 4

Hardware

In this chapter we report on our hardware implementations of Keccak without protection against side-channel attacks. For an overview and links to third-party hardware implementations of Keccak we refer to [6].

4.1 Introduction

Thanks to the symmetry and simplicity of its round function, Keccak allows trading off area for speed and vice versa. Different architectures reflect different trade-offs. We have investigated and implemented three architectures to reflect the two ends of the spectrum (a high-speed core and a low-area coprocessor) plus a mid-range core.

> We have coded our architectures in VHDL for implementation in ASIC and FPGA [5]. For more details on the VHDL code, refer to the `readme.txt` file in the VHDL directory.

In these efforts we focused 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 smallest instance of Keccak with a capacity providing a security level 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, PRG) for a given lane size.

4.2 High-speed core

The core presented in this section 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.

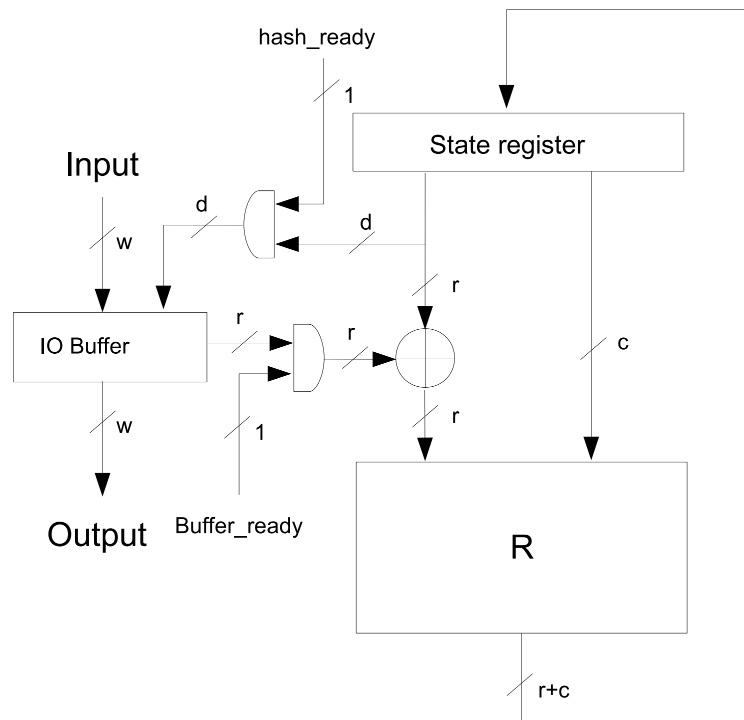

Figure 4.1: The high-speed core

The architecture of the high-speed core design is depicted in Figure 4.1. 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 KECCAK-f 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 [31], highlighting some possible improvements and problems with the tools available from FPGA vendors. We have improved the VHDL code for solving the problems, and this has given 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

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 4.1: Performance estimation of variants of the high speed core of Keccak$[r = 1024, c = 576]$.

130 nm general purpose ST technology library, worst case 105°C.

4.3 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.

An alternative for saving area is to XOR the lanes composing the input blocks directly into the state register and to extract the lanes composing the output blocks directly from it.

4.3.1 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.

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 4.1 summarizes the throughputs for different variants.

4.3.2 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.

4.4 Mid-range core

Our mid-range core takes inspiration from the work of Jungk and Apfelbeck in [20]. It cuts KECCAK's state in typically 2 or 4 pieces, so naturally fitting between the fast core (1 piece) and Jungk and Apfelbeck's compact implementation (8 pieces). As a result, we get a circuit not as fast as the fast core but more compact.

4.4.1 Description

One of the known techniques for trading silicon area at the cost of performances is the so called folding. At first sight it might be not trivial to find a way for folding the KECCAK round.

An interesting solution has been proposed by Jungk and Apfelbeck in [20]. They reorganize the round in such a way that the two computational parts, χ and θ are close one to the other, and not separated by registers or memory. Since these two transformations are slice oriented, it is possible to build a computational unit that compute the sequence of χ, ι and θ (called a *central round*) on a subset of the slices that compose the state. As the order of the transformations within the round is changed, it is necessary to have an *initial round* and a *final round*. The initial round is composed only by θ while the final round is like a central round but missing θ. The amount of slices to be processed in one shot can be decided at design time, it is convenient that the amount of slices is a divisor of the lane width, thus a power of 2 for KECCAK.

Jungk and Apfelbeck in [20] adopted a RAM for storing the state and the computational part processes 8 slices per iteration. The choice of processing 8 slices has been made for facilitating the routing of the data in the FPGA. In contrast, our mid-range core is ASIC oriented, and we wanted to investigate how the performance scales as a function of the number of slices and of the clock frequency.

The state is divided in N_b blocks, each consisting of a set of adjacent slices. The implementation is parametrized by N_b, and hence determines the amount of folding. The different parts of the core are depicted in Figures 4.2, 4.3, 4.4 and 4.5. To make the description of the core easier to follow, we take the practical example of $N_b = 4$, so with the state divided in four blocks of 16 consecutive slices (400 bits) each in the case of KECCAK-$f[1600]$.

The core is logically divided in two parts: the computational parts and the register block. The latter is not a plain 1600 bit register, as there is some logic for routing the data in the different phases of the computation. When the absorption of the input is executed, and supposing an input interface of 64 bits, the 64 bit word is accumulated in the lane that spans the four blocks. Based on the desired rate, the absorption will take as many clock cycles as the number of words composing the rate.

Once the absorption is done, the state is ready for being processed by the rounds:

- The initial round is composed only by the θ transformation. For computing the θ transformation on one block it is necessary to have access also to the slice of the adjacent block. Thus while computing θ on the block containing the slices form z to $z + 15$, it is necessary to have the parity of slice $z - 1$. In the case of the first block it is necessary to have access to the 25 bits composing the slice $z - 1$ and use them for computing the

	N_b				
MHz	2	4	8	16	32
333	25.5	21.1	19	17.8	17.3
500	28.3	22.3	19.6	18.5	17.6
625	35.1	26.5	22	19.8	18.2

Table 4.2: Gate count (KGE) for KECCAK$[r = 1024, c = 576]$ as a function of the clock frequency and the number of blocks N_b

θ-effect. For the other blocks a 5 bit register (in the θ parity block) stores the parity of the slice, and use it for the computation of θ. In the central round the computation of θ is done after χ and ι, while in the initial round it is necessary to bypass χ and ι (see multiplexer MUX1 in Figure 4.2).

- In the central rounds, the first step is the computation of ρ π, which is done in one clock cycle and is just a pure wiring. All the 1600 bits of the state are read and written in the correct position. Now block by block the sequence of χ ι θ is processed. For the computation of θ the same care taken in the initial round is applied, on the first block an additional slice is read from the register block for computing the θ parity, while for the other blocks the needed θ parity is stored in a special register since computed in the previous block.
- For the final round it is necessary to skip θ (see multiplexer MUX2 in Figure 4.2).

The block register is organized in a kind of shift register fashion, as depicted in Figure 4.3. We can see four different phases in the block register behavior. Two of them are related to the computation of the rounds, while the other two are for the absorption and squeezing phases of the sponge construction. The block register outputs the content of a block and the slice $z - 1$ when needed to compute the θ parity. In this phase the register block receives back a block of slices, and performs a shifting of the blocks to process the next block and store the previous one.

A different routing is put in place when ρ and π are computed, it is necessary to read all the state and write back the entire state, as depicted in Figure 4.4.

4.4.2 Results for KECCAK$[r = 1024, c = 576]$

Assuming that N_b divides the number of slices w, the number of cycles needed to compute the rounds is N_b for the initial round and $N_b + 1$ for the central and final rounds. The total latency of the permutation is $n_r + (n_r + 1)N_b$, so $24 + 25N_b$ for KECCAK-$f[1600]$.

We have implemented in VHDL the mid range core, with the parameter N_b ranging from 2 to 32. The cores have been synthesized using Synopsys Design Compiler and the technology library from STMicroelectornics at 130 nm, as for the other cores. We report in Table 4.2 the silicon area required expressed in gate count as a function of the clock frequency and the parameter N_b. With the technology library used, there is a negligible area saving with clock frequency lower than 333MHz, thus we do not report the data below this frequency, while the maximum frequency that the cores can reach is 625 MHz.

It is also possible to report the throughput of the mid range cores as a function of the clock frequency and the parameter N_b, as represented in Table 4.3. The data refers to a rate of $r = 1024$ bits, thus requiring additional 16 clock cycles for absorbing a block of the message. Throughput is in the so-called long message scenario, where the squeezing of the output is

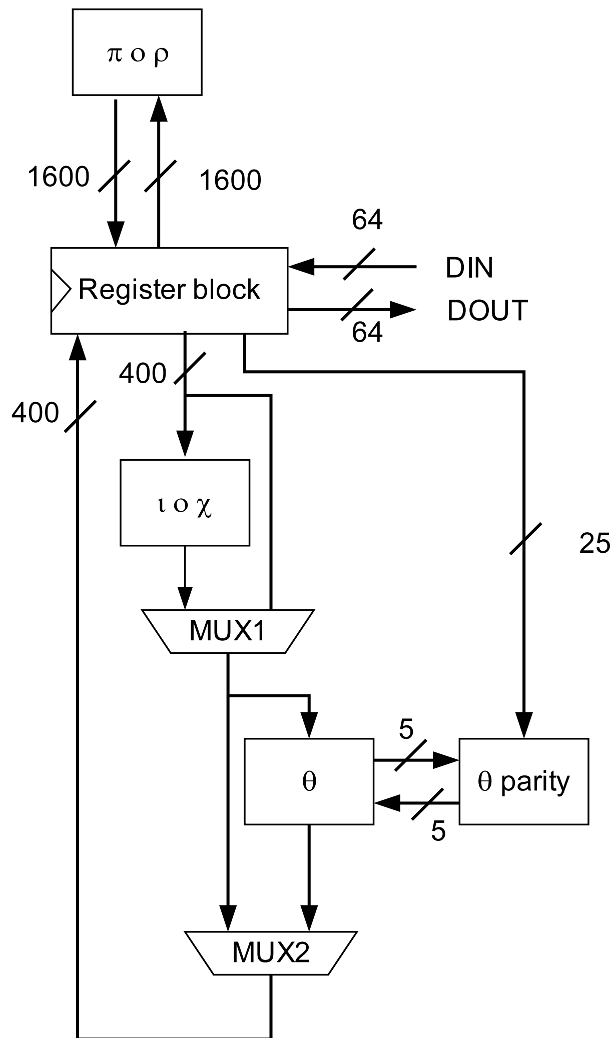

Figure 4.2: The mid-range core architecture, assuming KECCAK-$f[1600]$ and $N_b = 4$ as example.

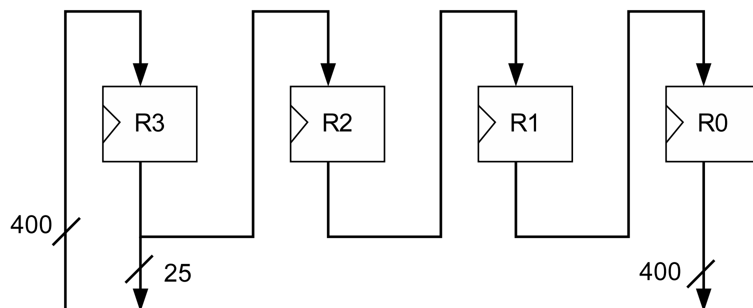

Figure 4.3: The mid-range core register block, when applying θ, assuming KECCAK-$f[1600]$ and $N_b = 4$ as example.

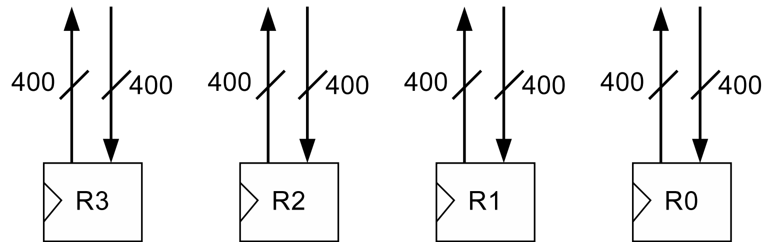

Figure 4.4: The mid-range core register block, when applying the ρ an π transformations, assuming KECCAK-f[1600] and $N_b = 4$ as example.

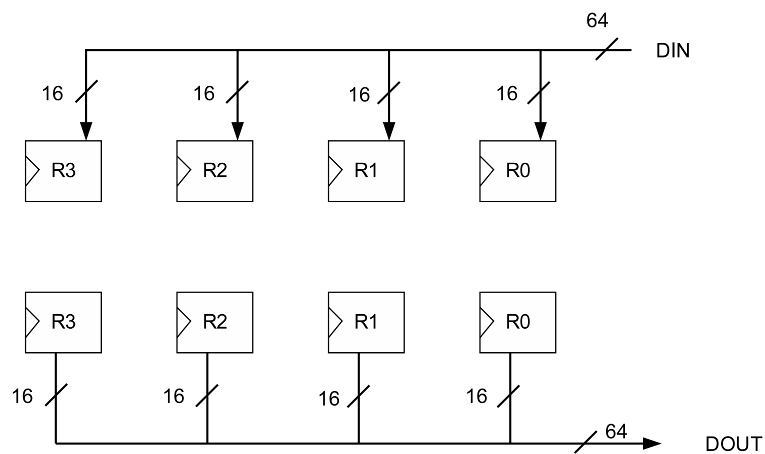

Figure 4.5: The mid-range core register block, when doing absorption (top) and squeezing (bottom), assuming KECCAK-f[1600] and $N_b = 4$ as example.

MHz	N_b 2	4	8	16	32
333	3.79	2.43	1.42	0.77	0.40
500	5.68	3.65	1.16	1.16	0.60
625	7.11	4.57	2.66	1.45	0.76

Table 4.3: Throughput (Gbit/s) of KECCAK$[r = 1024, c = 576]$ as a function of the clock frequency and the number of blocks N_b

Architecture	**Size**	**Throughput**
	(at 500MHz)	
High-speed core	48.0 KGE	21.3 Gbit/s
Mid-range core with $N_b = 2$	28.3 KGE	5.7 Gbit/s
Mid-range core with $N_b = 4$	22.3 KGE	3.6 Gbit/s

Table 4.4: Comparison of the mid-range core and of the high-speed core for KECCAK$[r = 1024, c = 576]$ at 500MHz.

negligible. As expected, the increase of the parameter N_b allows to reduce the silicon area, but also increase the latency for computing the permutation. Overall seems like the most interesting values of N_b are 2 and 4 when looking at the figure of merit throughput per area (see also Figure 4.4 for a comparison with the high-speed core). In all the configurations, there is a fixed cost in term of area for the storing of the state and the logic associated with the register block. The scaling of the area affecs only the area of the computational part for χ and θ.

4.5 Low-area coprocessor

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 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 4.6 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.

- In a first phase the sheet parities are computed, and the 5 lanes of the parity are stored in a dedicated area of the memory.
- The second phase computes θ, ρ and π. One-by-one the lanes of the state are read

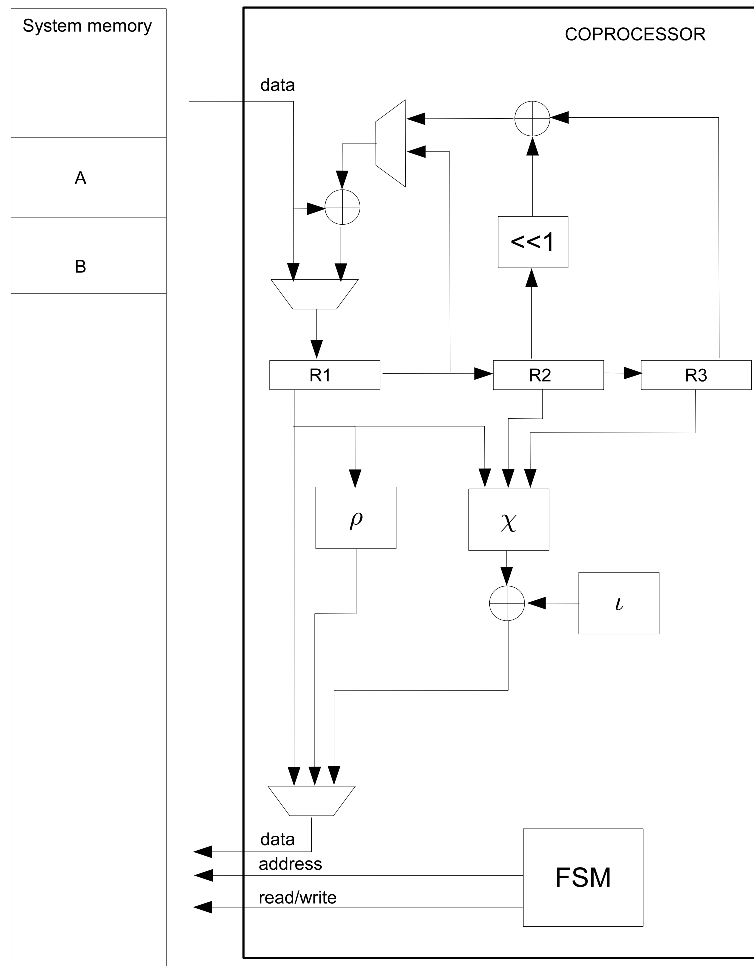

Figure 4.6: The low area coprocessor

and subject to θ by adding the corresponding sheet parities, to ρ by translation by the appropriate offset and to π by writing to a memory cell with the appropriate address. Now the intermediate state is completely stored in the buffer B.

- The last phase is to compute χ and add the ι round constant 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. Out of these, 55 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 4.2, the coprocessor has been synthesized using ST technology at 130 nm.

4.5.1 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.

4.5.2 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.

4.6 FPGA implementations

In Table 4.5, we report some figures we obtained on the coprocessor, as well as results from third parties, making use of a lane-oriented, slice-oriented or high-speed architecture.

Architecture	**Thr.** (Mbit/s)	**Freq.** (MHz)	**Slices** (+RAM)	**Latency** (clock cycles)	**Efficiency** (Mbit/s/slice)
Lane-wise	52	265	448	5160	0.12
Lane-wise [29]	501	520	151 (+3)	1062	3.32
Slice-wise [20]	813	159	372	200	2.18
High-speed [15]	12789	305	1384	24	9.24

Table 4.5: Performance of KECCAK$[r = 1024, c = 576]$ on Virtex 5 (scaled to $r = 1024$ when necessary). "(+3)" means that 3 RAM blocks are used in addition to the slices.

Chapter 5

Protection against side-channel attacks

If the input to Keccak includes secret data or keys, side channel attacks may pose a threat. In this chapter, we report on Keccak implementations that offer a high level of resistance against power analysis by using the technique of masking (secret sharing).

5.1 Introduction

Sponge functions, among which Keccak, can be used in a wide range of modes covering the full range of symmetric cryptography functions. We refer to [10, 4] for examples. This includes functions that take as argument a secret key. Some other functions do not take a secret key but take as input data that should remain secret such as pseudorandom sequence generators or commit-challenge-response protocols. If such functionality is desired on devices to which an adversary has some kind of physical or logical access, protection against side channel and fault attacks is appropriate [3].

Side channel and fault attacks are attacks that do not exploit an inherent weakness of an algorithm, but rather a characteristics of the implementation. For their security cryptographic primitives inevitably rely on the fact that an adversary does not have access to intermediate computation results. As a consequence, even partial knowledge of intermediate computation results can give a complete breakdown of security, e.g., by allowing computation of the key. However, actual implementations may leak information on these results via characteristics such as computation time, power consumption or electromagnetic radiation. Another condition for the security of cryptographic primitives is that they are executed without faults. An implementation that makes faults, or can be manipulated to make faults, may be completely insecure.

We here concentrate on countermeasures against power analysis and electromagnetic radiation that make use of the algebraic properties of the step functions of Keccak. In the remainder of this chapter we will speak only about power analysis implying also electromagnetic analysis. The main difference between power analysis and electromagnetic analysis is that in the latter the adversary can make more sophisticated measurements and that the physical and electronic countermeasures are different. The countermeasures at the algorithmic level are however the same for both.

As far as timing attacks are concerned, it is straightforward to implement Keccak in such a way that its execution time is independent of the input it processes, both in software as in hardware. Moreover, Keccak-f does not make use of large lookup tables so that cache timing attacks pose no problem. The interleaving method of Section 2.1 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 fault attacks can be achieved by countermeasures that are independent of the cryptographic primitive being protected: fault detection at software level, at hardware level and by performing computations multiple times and verifying that the results are equal. Particularly good sources of information on side channel attacks and countermeasures are the proceedings of the yearly Cryptographic Hardware and Embedded Systems (CHES) conferences (`http://www.chesworkshop.org/`) and the text book [28]. A good source of information on fault attacks and countermeasures are the proceedings of the yearly Fault Diagnosis and Tolerance in Cryptography (FDTC) workshops (`http://conferenze.dei.polimi.it/FDTC10/index.html`).

5.2 Power analysis

The general set-up of power (and electromagnetic) analysis is that the attacker gets one or more traces of the measured power consumption. If only a single trace suffices to mount an attack, one speaks about simple power analysis (SPA). However, the dependence of the signal in the variables being computed is typically small and obscured by noise. This can be compensated for by taking many traces, each one representing an execution of the cryptographic primitive with different input values. These many traces are then subject to statistical methods to retrieve the key information. These attacks are called differential power analysis (DPA) [21]. An important aspect in these attacks is that the traces must be *aligned*: they must be combined in the time-domain such that corresponding computation steps coincide between the different traces.

In DPA one distinguishes between *first order* DPA and *higher order* DPA. In first-order, the attacker is limited to considering single time offsets of the traces. In m-th order the attacker may incorporate up to m time offsets in the analysis. Higher-order attacks are in principle more powerful but also much harder to implement [28].

In correlation power analysis (CPA) [13] one exploits the fact that the power consumption may be correlated to the value of bits (or bitwise differences of bits) being processed at any given moment. In short, one exploits this by taking many traces and partitioning them in two subsets: in one set, a particular bit, the *target bit*, is systematically equal to 0 and in the other it is equal to 1. Then one adds the traces in each of the two sets and subtracts the results giving the *compound trace*. If now at any given time the power consumption is correlated to the target bit, one sees a high value in the compound trace. One can use this to retrieve key information by taking a target bit that depends on part of the key and trying different partitions based on the hypothesis for that part of the key. If wrong key guesses result in a partition where the bits of intermediate results are more or less balanced, the compound trace of the correct key guess will stand out. Note that if the power consumption is correlated to bit values or differences, it is also correlated to the Hamming values of words or Hamming distances between words.

Later, more advanced ways to measure the distance between distributions were introduced. In particular, mutual information analysis (MIA) [17] is a generalization of CPA in the sense that instead of just exploiting the fact that different bit values may result in different expected power consumption values, it is able to exploit the difference between the distributions of the power consumption for a bit being 0 or 1 respectively. So in short, when the power consumption distributions of a bit equal to 0 or 1 have equal mean values but different shapes, CPA will not work while MIA may still be able to distinguish the two distributions.

5.2.1 Different types of countermeasures

In the light of power analysis attacks, one must attempt implementing the cryptographic primitives such that the effort (or cost) of the adversary for retrieving the key is too high for her to be interesting. An important countermeasure is implementing the cryptographic primitives such that the power consumption and electromagnetic radiation leak as little as possible on the secret keys or data. Countermeasures can be implemented at several levels:

Transistor level Logical gates and circuits are built in such a way that the information leakage is reduced;

Platform level The platform supports features such as irregular clocking (clock jitter), random insertion of dummy cycles and addition of noise to power consumption;

Program level The order of operations can be randomized or dummy instructions can be inserted randomly to make the alignment of traces more difficult;

Algorithmic level The operations of the cryptographic algorithm are computed in such a way that the information leakage is reduced;

Protocol level The protocol is designed such that it limits the number of computations an attacker can conduct with a given key.

As opposed to protection against cryptographic attacks, protection against side channel attacks is never expected to be absolute: a determined attacker with a massive amount of resources will sooner or later be able to break an implementation. The engineering challenge is to put in enough countermeasures such that the attack becomes too expensive to be interesting. Products that offer a high level of security typically implement countermeasures on multiple levels.

The countermeasures at transistor level are independent of the algorithm to be implemented. Examples are wave dynamic differential logic (WDDL) [32] or SecLib [18]. 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. They imply dedicated hardware for cryptography which takes more area, requires dedicated industrialization processes and is in general more expensive. Moreover, while these countermeasures may significantly reduce the information leakage, there always remains some leakage.

The countermeasures at platform level are also independent of the algorithm to be implemented. By randomizing the execution timing, alignment of power traces is made more difficult. The addition of noise to the power consumption increases the required number of traces. The timing randomization is particularly efficient against higher-order DPA as there the signals must be aligned in multiple places and any misalignment severely limits the effectiveness of attack. The addition of noise is also very efficient against attacks that look for dependencies in higher moments of the distributions, such as MIA (see Section 5.5).

The countermeasures at program level are partially dependent on the algorithm to be implemented. Insertion of dummy instructions is possible in any algorithm while changing the order of operations may be easier for some algorithms than for others.

The countermeasures at protocol level are also independent of the algorithm. However, what can be done at this level depends on the requirements of the application and in many cases the possibilities are limited.

Finally, the countermeasures at algorithmic level depend on the basic operations used in the algorithm. This is the type of countermeasures where the choice of operations in the

cryptographic primitive is relevant. One of the countermeasures of this type is that of secret sharing (or masking) and KECCAK is particularly well suited for it.

5.2.2 Secret sharing

Masking is a countermeasure that offers protection against DPA at the algorithmic level. It consists of representing variables processed by a cryptographic primitive by two or more shares (as in secret sharing) where the (usually bitwise) sum of the shares is equal to the *native* variable. Subsequently the program or circuit computes the cryptographic primitive using the shares in such a way that the processed variables are independent from the native variables. Whether this is possible depends on the details of the cryptographic primitive and the type of masking. In any case, to achieve independence for a native variable, all but one of its shares must be generated (pseudo-)randomly for each execution of the cryptographic primitive. Clearly, the generation of the shares, the masking operation of the input words and unmasking operation of output, usually considered out of scope of DPA attacks, must also be carefully implemented to limit information leakage. For masking to be effective, the adversary shall have as little information as possible on the value of the shares.

Taking two shares offers protection against first-order DPA under the condition that all processed bits and their joint behavior at any time are independent from native variables. Providing protection against m-th order DPA requires at least $m + 1$ shares.

Computing a linear function λ on the shares of a variable is straightforward. If we represent a native variable x by its shares x_i with $x = \sum_i x_i$ we can compute the shares y_i of $y = \lambda(x)$ by simply applying λ on the individual shares: $y_i = \lambda(x_i)$. A function that consists of the addition of a constant can be performed by adding it to a single share. As all separate operations are performed on shares that are independent of native variables, this provides protection against first-order DPA.

Computing a nonlinear function on the shares assuring all variables processed are independent of native variables depends on the nonlinear function at hand. This is in general a non-trivial problem. We refer to [3] for some examples. In this chapter we limit ourselves to the nonlinear step mapping χ in the round function of KECCAK-f:

$$x_i \leftarrow x_i + (x_{i+1} + 1)x_{i+2} \,. \tag{5.1}$$

5.3 Software implementation using two-share masking

For software implementations, we have studied the application of masking with two shares, denoted by a and b. The step mapping χ is very similar to nonlinear step mapping γ in BASEKING, the cipher that is the subject of [14] and hence the techniques shown there can be readily applied. We must now compute the shares of x at the lefthand side of Equation (5.1) in such a way that all intermediate variables are independent of native variables. This can be realized by implementing following equations:

$$\begin{aligned} a_i &\leftarrow a_i + (a_{i+1} + 1)a_{i+2} + a_{i+1}b_{i+2} \\ b_i &\leftarrow b_i + (b_{i+1} + 1)b_{i+2} + b_{i+1}a_{i+2} \,. \end{aligned} \tag{5.2}$$

To achieve independence from native variables, the order in which the operations are executed is important. If the expressions are evaluated left to right, it can be shown that all intermediate variables are independent from native variables. The computations of all terms except the rightmost one involve only variables of a single share, hence here independence from x is automatic. For the addition of the mixed term to the intermediate result of the

computation, the presence of $a[i]$ (or $b[i]$) as a linear term in the intermediate variable results in independence.

We realize that the transformation within $\mathrm{GF}(2)^{10}$ depicted by Equation (5.2) is not a permutation. The uniformity of the shares is therefore not preserved. A variant of this method that provides a permutation is given in Section 5.3.1 below.

Although it means that some entropy is lost, this seems unfeasible to exploit in practice according to our experiments.

At the algorithm level, the effect of the introduction of two shares in the computation of the KECCAK-f round function is rather simple. The linear part λ can be executed on the two shares separately, roughly doubling the workload. In the nonlinear step mapping χ the computation of a state word according to Equation (5.1), taking a XOR, AND and NOT instruction, is replaced by the computation of the shares according to Equations (5.2), taking in total 4 XOR, 4 AND and 2 NOT instructions. The addition of round constants ι and addition of input blocks can be performed on one share only.

As the order of execution is important, it is not sufficient to write a program in C or some other high-level language and compile it. The compiler may optimize away the desired order. An option is to compile and inspect the machine code afterwards, but the method that provides the highest level of assurance is to program in assembly language. It is however not sufficient to check only the sequencing of instructions. In general, the operations on two shares of the same variable are preferably executed in registers *physically isolated* from each other. More particularly, the program shall be such that at any given moment there is no register (or bus) content or transition that is correlated to a native variable. For example, if the number of shares is two and a register containing x_0 is loaded with x_1, the power consumption can depend on the number of bits switched (e.g., on the Hamming weight of $x_0 \oplus x_1$) and it is likely to leak information on x. This can be solved by setting the register to zero in between. Another example is the simultaneous processing of two shares of a variable in different parts of the CPU. As the power consumption at any given moment depends on all processing going on, it depends on both shares and and hence there may be a dependence on the native variable. Clearly, care must be taken when attempting to build side-channel resistant implementations. So if sufficient care is taken, this provides provable resistance against first-order DPA. Higher-order DPA is in principle still possible but as explained in [14, Appendix A] very sensitive to noise and clock jitter. On smart cards, the addition of noise, dummy cycles and clock jitter are typically supported by the platform.

5.3.1 Simplifying the software implementation

The protection using two-share masking can be simplified by randomly choosing one of the shares only once per evaluation of the permutation. Say that b is fixed during the rounds. Then, Equation (5.2) can be changed into

$$\begin{aligned} a_i &\leftarrow a_i + (a_{i+1}+1)a_{i+2} + a_{i+1}b_{i+2} + (b_{i+1}+1)b_{i+2} + b_{i+1}a_{i+2} \\ b_i &\leftarrow b_i \,. \end{aligned} \tag{5.3}$$

Here again, the order of execution is important. Evaluating the XORs from left to right, each intermediate value is independent of the native values.

To compute the entire round, one has to apply the linear part λ onto both a and b. Alternatively, one can precompute $y = b \oplus \lambda(b) \oplus x$, with x is a random value chosen once per evaluation of the permutation, and then update a as follows:

$$\begin{aligned} a &\leftarrow \lambda(a) \oplus y \oplus x \\ b &\leftarrow b \,. \end{aligned} \tag{5.4}$$

To simplify further, the mask b can be chosen such that b and $\lambda(b)$ differ only over a restricted support (e.g., a few lanes). Hence, the operation in Equation (5.4) require less XORs and the value x can be restricted to the support of $b \oplus \lambda(b)$.

5.4 Hardware using three-share masking

At first sight, masking with two shares may offer protection against first order CPA. However, in dedicated hardware the occurrence of *glitches* [23] may result in correlation between the power consumption and native variables. In a combinatorial circuit computation is typically not monotonous but intermediate signals may switch several times per clock cycle resulting in the (possible transient) appearance of native variables. Hence, due to glitches, two-share masking cannot provide provable protection against first order CPA.

A solution to this problem was proposed in [24]. We refer to [24, 25] for an in-depth treatment and limit ourselves here to the explanation of the basic concept. The simple but powerful idea is to take as many shares as needed such that in any computation at least one of the shares is not taken as input. In this way, all intermediate variables are guaranteed to be independent from native variables for the same reason that the one-time pad offers perfect secrecy. For linear functions two shares are sufficient to realize this. For nonlinear functions, the required number of shares depends on the particular function.

As can be read in [24, 25], for most cryptographic algorithms, the mere number of shares required makes the application of this technique very expensive. Fortunately, χ with its simple structure is well suited for applying this technique and only three shares are required. This suitability is quite unique among cryptographic primitives. If we denote the shares by a, b and c, a possible computation of χ applied to three shares is given by:

$$\begin{aligned}
a_i &\leftarrow b_i + (b_{i+1}+1)b_{i+2} + b_{i+1}c_{i+2} + c_{i+1}b_{i+2} \\
b_i &\leftarrow c_i + (c_{i+1}+1)c_{i+2} + c_{i+1}a_{i+2} + a_{i+1}c_{i+2} \\
c_i &\leftarrow a_i + (a_{i+1}+1)a_{i+2} + a_{i+1}b_{i+2} + b_{i+1}a_{i+2}\,.
\end{aligned} \tag{5.5}$$

Clearly, the computation of each share takes as input only components of the other two shares and it provides provable security against first order CPA, even in the presence of glitches.

For pipe-lining of non-linear layers, a uniformity condition is required, implying that the transformation operating on the shares is a permutation [24, 25]. We realize that the transformation within $\mathrm{GF}(2)^{15}$ depicted by Equation (5.5) does not satisfy this condition [1]. In Keccak-f, the quadratic step χ does not require pipe-lining to get correctness and incompleteness with three shares. In principle, the potential problem is a gradual loss of entropy as no new randomness is added to the shares. According to our experiments, this property seems however unfeasible to exploit in practice. Furthermore, variants processing several rows jointly exist, which reduce the loss.

The three lines of Equation (5.5) are equivalent and only differ in the input shares and output share. We denote the computation of the nonlinear step χ' resulting in a share given the two other shares by, e.g., $a \leftarrow \chi'(b, c)$.

In the following subsections we present two architectures that make use of three shares. Other architectures can be derived based on different partitioning or sequences of the computational steps composing the round. For instance a low area coprocessor, as those presented in Section 4.5, can be protected using the secret sharing techniques.

5.4.1 One-cycle round architecture

A first architecture computes one round in one clock cycle and is depicted in Figure 5.1.

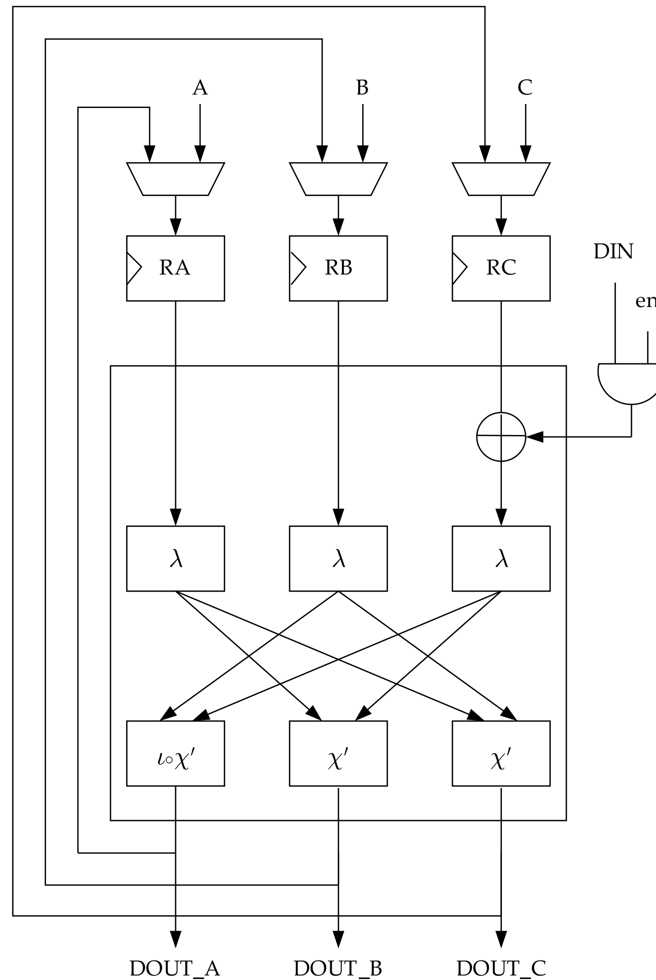

Figure 5.1: Protected architecture computing one round in one clock cycle.

Before processing, the three shares a, b and c are generated from a random source. As the initial state of KECCAK should be set equal to zero implying $a + b + c = 0$, the shares a and b can be generated randomly and c computed as $c = a + b$. The hardware for generating the three shares is out of the scope of our study, we just consider them as input to the core.

The combinatorial logic implements the round function and input data block absorbing. It has a layered structure. In a first (linear) layer, the absorbing of the input data block, D_{in}, is implemented by adding it to one of the shares and then λ is applied to the three shares by three separate combinatorial blocks. In a second layer, the nonlinear step mapping χ is computed on the output of the first layer according to Equations (5.5) by three separate combinatorial blocks implementing χ'. Each block takes as input two shares and generates one share. The blocks only differ in the presence of ι in the leftmost block as ι only needs to be added to a single share.

We can estimate the cost of the secret sharing technique in terms of silicon area by comparing this architecture with our unprotected one-cycle round architecture in Section 4.2, based on Figure 5.1 and Equations (5.5). The number of registers required for storing the state is three times larger than the unprotected version. The cost of the linear part is three times larger as well. Regarding the nonlinear part, we have three blocks χ' instead of one χ and the cost of every χ' is also larger than that of χ. While χ requires basically an AND gate,

a NOT and a XOR for each bit of the state, χ' requires three AND gates, one NOT and three XOR for a single share. So roughly, the protected nonlinear part is expected to be nine times larger than the unprotected χ.

5.4.2 Three-cycle round architecture

A second architecture reduces the amount of silicon at the cost of performance and is depicted in Figure 5.2. Since the round logic is composed of three equal blocks for λ and three equal blocks for χ', we instantiate only one block of each and try to use them as much as possible.

Instead of three registers this architecture requires four registers, some multiplexing and a careful schedule. The schedule has some similarity to pipelining techniques.

For explaining the schedule we refer to Table 5.1. We use $\lambda(R_0)$ to denote the application of λ to register R_0, and $\chi'(R_1, R_2)$ to denote the application of χ' using registers R_1 and R_2 as inputs. The three initial values of the shares are indicated as A, B and C. The values after the first round by A', B' and C', after the second round by A'', etc. At clock cycle zero the registers do not yet contain any relevant data, as indicated with a $-$. Instead of loading the three shares in parallel, as done in the previous architecture, one share per clock cycle is loaded into register R_0 during the three initial clock cycle.

The content of R_1 is in general the result of λ applied to R_0, with the D_{in} enabled and XORed at the input of $\lambda(R_0)$ before an initial Keccak-f round.

The content of R_0 is the result of χ' applied to either the couple R_2 and R_1 or R_2 and R_3. While in the one-cycle round architecture ι was applied only to one of the three shares, here it is applied to all of the three shares. This does not change the result of the computation and simplifies the control logic.

The registers R_2 and R_3 are used for maintaining the values required for the computation of χ'. In the second clock cycle B is loaded into R_0, and R_1 receives $\lambda(A)$. In the third clock cycle C is loaded into R_0, R_1 receives $\lambda(A)$ and $\lambda(A)$ is moved from R_1 to R_2. In the fourth clock cycle no more shares need to be loaded. Instead R_0 receives χ' applied to the content of R_1 and R_2, which means $\lambda(A)$ and $\lambda(B)$. It follows that R_0 now contains the share C' after the first round. R_1 receives $\lambda(C)$, $\lambda(B)$ is moved from R_1 to R_2 and $\lambda(A)$ is moved from R_2 to R_3. In the fifth clock cycle R_0 receives χ' applied to the content of R_1 and R_2, being $\lambda(B)$ and $\lambda(C)$ and hence contains the share A after the first round. R_1 receives $\lambda(C')$, $\lambda(C)$ is moved from R_1 to R_2, while $\lambda(A)$ remains in R_3. Since shares A' and C' as input of the second round have already been computed, in the next clock cycle share B' must be computed, requiring $\lambda(A)$ and $\lambda(C)$, and they are in R_1 and R_3 respectively before clock cycle five. Thus we have described also what will be computed in the next clock cycle and this is basically the end of a round. Starting from the next clock cycle there will be a loop of three clock cycles where the alternation of data contained in the registers are those for computing a round.

Using this circuit, a Keccak computation takes 3 initial cycles plus 24 cycles per execution of Keccak-f.

5.4.3 Synthesis results

We have implemented both architectures in VHDL and synthesized it for understanding the maximum frequency and silicon area demand. We have used the same technology library adopted for the unprotected implementation reported in Section 4.2, a 130 nm general purpose library from STMicroelectronics, and the Synopsys Design Compiler.

Table 5.2 summarizes the gate count and performance numbers of the different implementations, together with the numbers for the one-cycle round unprotected architecture de-

cycle	R_0	R_1	R_2	R_3
0	-	-	-	-
1	input, A	-	-	-
2	input, B	$\lambda(A + D_{\text{in}})$	-	-
3	input, C	$\lambda(B)$	$\lambda(A + D_{\text{in}})$	-
4	$\chi'(R_2, R_1), C'$	$\lambda(C)$	$\lambda(B)$	$\lambda(A + D_{\text{in}})$
5	$\chi'(R_2, R_1), A'$	$\lambda(C')$	$\lambda(C)$	$\lambda(A + D_{\text{in}})$
6	$\chi'(R_2, R_3), B'$	$\lambda(A')$	$\lambda(C')$	$\lambda(C)$
7	$\chi'(R_2, R_1), B''$	$\lambda(B')$	$\lambda(A')$	$\lambda(C')$
8	$\chi'(R_2, R_1), C''$	$\lambda(B'')$	$\lambda(B')$	$\lambda(C')$
9	$\chi'(R_2, R_3), A''$	$\lambda(C'')$	$\lambda(B'')$	$\lambda(B')$
10	$\chi'(R_2, R_1), A'''$	$\lambda(A'')$	$\lambda(C'')$	$\lambda(B'')$
11	$\chi'(R_2, R_1), B'''$	$\lambda(A''')$	$\lambda(A'')$	$\lambda(B'')$
12	$\chi'(R_2, R_3), C'''$	$\lambda(B''')$	$\lambda(A''')$	$\lambda(A'')$

Table 5.1: The content of the registers during the computation.

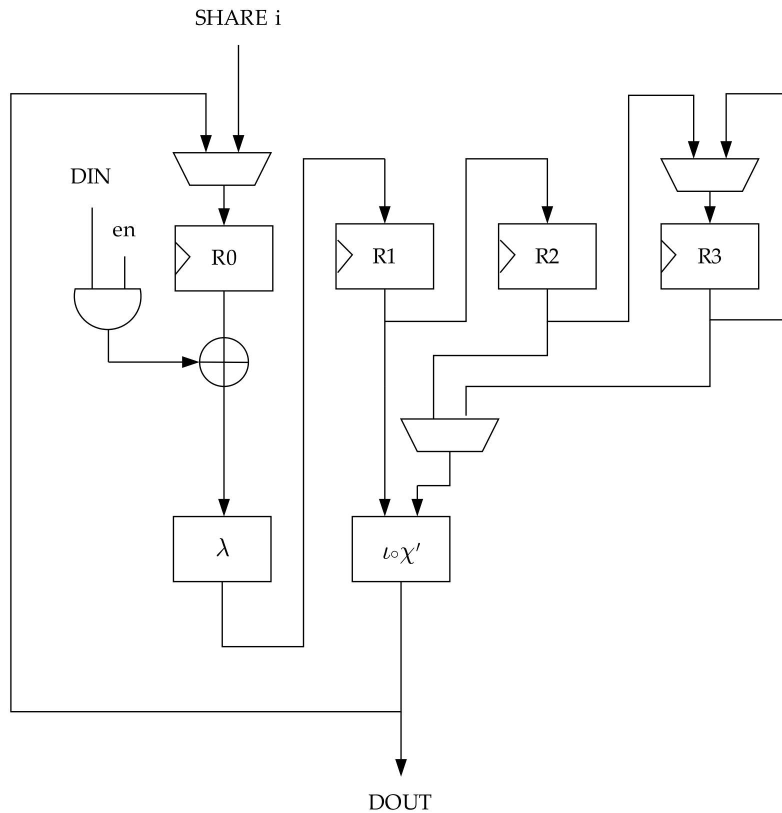

Figure 5.2: Protected architecture computing one round in three clock cycles.

Core ($r = 1024$)	Size KGE	Frequency MHz	Throughput Gbit/s.
Unprotected one-cycle	48	526	22.4
One-cycle (fast)	183	500	21.3
One-cycle (compact)	127	200	8.5
Three-cycle (fast)	115	714	10.1
Three-cycle (medium)	106	500	7.1
Three-cycle (compact)	95	200	2.8

Table 5.2: Performance and gate count of the different implementations

scribed in Section 4.2. It gives the total gate count, the maximum frequency and the throughput assuming a bitrate of 1024 bits.

The gate count figures are in line with the estimations made in Section 5.4.1. All implementations have an I/O buffer of 9 Kilo gate equivalent (KGE) for connecting the core to a system bus. This allows to load the I/O buffer, 64-bit per clock cycle, simultaneously with the application of Keccak-f on the previous input block.

In the three-cycle round architecture the computation of one round is executed in 3 clock cycles and the initialization of the state requires also 3 clock cycles. Thanks to the very short critical path, the fast variant of the three-cycle round architecture can reach 714 MHz, still resulting in a very competitive speed of 10 Gbit/s. If we compare the one-cycle round and the three-cycle round architectures both running at 500Mhz, we can see that the three-cycle requires 40% less silicon area at a cost of a throughput reduction by a factor 3.

It is interesting to note that the strategy adopted in the design of the function allows implementing this countermeasure with a cost consisting only of silicon area with almost no penalty in terms of throughput: it is reduced only by 5%, from 22.4 Gbit/s. to 21.3 Gbit/s. when using a rate of 1024 bits.

5.5 Computing in parallel or sequentially?

The use of three shares computed at different times gives a provable resistance against first-order DPA, but not against DPA of higher order. Higher-order DPA involves taking measurements corresponding to the computation of the different shares, and this task is more complex due to its higher sensitivity to noise than first-order DPA [14, Appendix A].

In the architectures presented in the previous sections, several computations take place in parallel and the power consumption at a given time depends on all these computations. For this reason, these architectures are not provably resistant against first-order DPA. For instance, in [25], the authors analyzed an implementation with three shares working in parallel. Storing the three shares in a register causes a power consumption to depend on the three shares simultaneously. In the absence of noise, the distribution of the consumption depends on the native variable being stored and hence is vulnerable to MIA. In this section, we explain that the introduction of noise has a much stronger impact on the feasibility of this attack for a two-share or three-share implementation than for an unmasked one, and that the qualitative difference is similar to the one between first-order and higher-order DPA. Furthermore, we argue that a masked implementation has per construction a higher algorithmic noise level than an unmasked one.

With three shares, a native bit equal to 0 (resp. 1) can be represented as 000, 011, 101 or 110 (resp. 001, 010, 101 or 111). If the three shares are processed simultaneously, the power

consumption can leak the Hamming weight of the shares, which means 0 or 2 for a native bit 0, or 1 or 3 for a native bit 1. Clearly, these two distributions are different and can be distinguished.

We start the discussion by constructing a simple model where the power consumption is equal to the Hamming weight plus an additive Gaussian noise of standard deviation σ, expressed in the same unit as the impact of the Hamming weight on the power consumption. In this model, the distribution of the power consumption for three shares is as in Figure 5.3(c), with $\sigma = 0.2$. Similarly, we can look at an unmasked implementation, where the power consumption is 0 or 1 plus the Gaussian noise, and at masking with two shares. In this last case, the Hamming weight is 0 or 2 for a native bit 0 (represented as 00 or 11) or 1 for a native bit 1 (represented as 01 or 10). The two cases can be found in Figures 5.3(a) and 5.3(b), respectively.

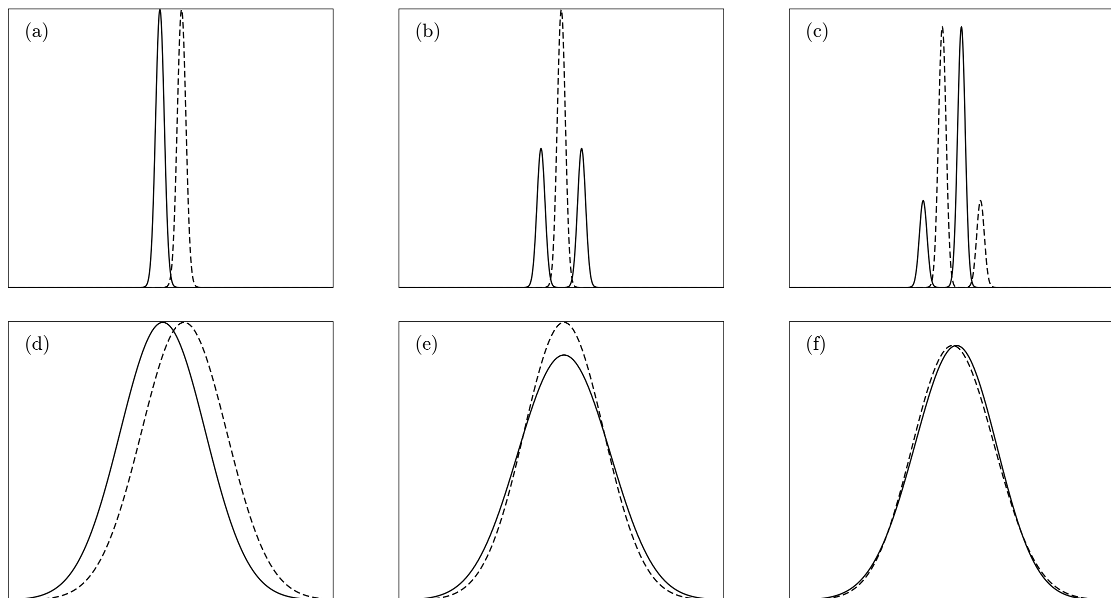

Figure 5.3: Distribution of the power consumption for a simple model. The solid line shows the distribution for a bit with native value 0 and the dashed line for a bit 1. Sub-figures (a), (b) and (c) show the case of one, two and three shares, respectively, with a noise level of $\sigma = 0.2$. Sub-figures (d), (e) and (f) follow the same model with $\sigma = 2$.

Following this model, we compute the number of samples that are needed to distinguish between the distribution for a native bit 0 and the distribution for a native bit 1. We follow the same reasoning as in [14, Appendix A]. The number z of samples needed to distinguish one distribution over the other is inversely proportional to the Kullback-Leibler divergence between the two distributions [22], $z = 1/D(f|g)$ with

$$D(f|g) = \int f(x)(\log(f(x)) - \log(g(x))dx.$$

In this model, the scaling of z as a function of σ is different for one, two or three shares. For one share, $z \sim 2\sigma^2$ samples are needed to distinguish a native bit 0 from a native bit 1 from unmasked values, whereas about z^2 samples are needed for the same noise level when two shares are used, and this number grows to about z^3 samples for three-share masking. Hence, the difference between the one-share and two-share and three-share implementation

is qualitatively the same as the one between first-order and second-order and third-order DPA.

The real-world behavior is likely to differ from this simple model. Nevertheless, we expect a significantly higher sensitivity to noise for three shares than for one. Qualitatively, the three pairs of distributions are different. For one share, the mean is different for native bits 0 and 1. For two shares, the two distributions have the same mean but a different variance. For three shares, the two distributions have the same mean and variance; they differ only starting from their third moment. Figures 5.3(d), 5.3(e) and 5.3(f) illustrate this with the simple model and a higher noise level $\sigma = 2$.

So far, we have assumed that the three levels of masking are subject to the same noise level σ. However, the masking by itself introduces noise, as $m - 1$ shares are randomly and independently chosen for every execution of the algorithm. The very dependence of the processing of a bit in the power consumption that an attacker exploits turns against her, as it becomes an additional source of noise due the randomization. For instance, in the one-cycle-one-round implementation of Keccak-f[1600] with three shares, the noise due to the Hamming weight of 3200 random bits must be compared to a small number of unmasked bit values that a differential power analysis attempts at recovering.

Bibliography

[1] E. Alemneh, *Sharing nonlinear gates in the presence of glitches*, Master's thesis, August 2010, http://essay.utwente.nl/59599/.

[2] D. J. Bernstein and T. Lange (editors), *eBACS: ECRYPT Benchmarking of cryptographic systems*, http://bench.cr.yp.to, accessed 21 December 2010.

[3] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, *Note on side-channel attacks and their countermeasures*, Comment on the NIST Hash Competition Forum, May 2009, http://keccak.noekeon.org/NoteSideChannelAttacks.pdf.

[4] ______, *Cryptographic sponge functions*, January 2011, http://sponge.noekeon.org/.

[5] ______, *Hardware implementation of* KECCAK *in VHDL*, 2011, http://keccak.noekeon.org/.

[6] ______, KECCAK *hardware performance figures page*, 2011, http://keccak.noekeon.org/hw_performance.html.

[7] ______, KECCAK *software performance figures page*, 2011, http://keccak.noekeon.org/sw_performance.html.

[8] ______, *Known-answer and Monte Carlo test results*, 2011, http://keccak.noekeon.org/.

[9] ______, *Reference and optimized implementations of* KECCAK, 2011, http://keccak.noekeon.org/.

[10] ______, *The* KECCAK *reference*, January 2011, http://keccak.noekeon.org/.

[11] ______, *The* KECCAK *SHA-3 submission*, January 2011, http://keccak.noekeon.org/.

[12] ______, KECCAKTOOLS *software*, April 2012, http://keccak.noekeon.org/.

[13] E. Brier, C. Clavier, and F. Olivier, *Correlation power analysis with a leakage model*, CHES (M. Joye and J.-J. Quisquater, eds.), Lecture Notes in Computer Science, vol. 3156, Springer, 2004, pp. 16–29.

[14] J. Daemen, M. Peeters, and G. Van Assche, *Bitslice ciphers and power analysis attacks*, Fast Software Encryption 2000 (B. Schneier, ed.), Lecture Notes in Computer Science, vol. 1978, Springer, 2000, pp. 134–149.

[15] K. Gaj, E. Homsirikamol, M. Rogawski, R. Shahid, and M. U. Sharif, *Comprehensive evaluation of high-speed and medium-speed implementations of five SHA-3 finalists using Xilinx and Altera FPGAs*, The Third SHA-3 Candidate Conference, 2012.

[16] 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.

[17] B. Gierlichs, L. Batina, P. Tuyls, and B. Preneel, *Mutual information analysis*, CHES (E. Oswald and P. Rohatgi, eds.), Lecture Notes in Computer Science, vol. 5154, Springer, 2008, pp. 426–442.

[18] S. Guilley, P. Hoogvorst, Y. Mathieu, R. Pacalet, and J. Provost, *CMOS structures suitable for secured hardware*, in Gielen and Figueras [16], pp. 1414–1415.

[19] G. Hoffmann, Keccak *implementation on GPU*, 2010, http://www.cayrel.net/spip.php?article189.

[20] B. Jungk and J. Apfelbeck, *Area-efficient FPGA implementations of the SHA-3 finalists*, International Conference on ReConFigurable Computing and FPGAs (ReConfig), 2011, to appear.

[21] P. C. Kocher, J. Jaffe, and B. Jun, *Differential power analysis*, Advances in Cryptology – Crypto '99 (M. Wiener, ed.), Lecture Notes in Computer Science, vol. 1666, Springer, 1999, pp. 388–397.

[22] S. Kullback and R. A. Leibler, *On information and sufficiency*, Ann. Math. Statist. **22** (1951), no. 1, 79–86, http://projecteuclid.org/euclid.aoms/1177729694.

[23] S. Mangard, N. Pramstaller, and E. Oswald, *Successfully attacking masked AES hardware implementations*, CHES (J.R. Rao and B. Sunar, eds.), Lecture Notes in Computer Science, vol. 3659, Springer, 2005, pp. 157–171.

[24] S. Nikova, V. Rijmen, and M. Schläffer, *Secure hardware implementation of nonlinear functions in the presence of glitches*, ICISC (P. J. Lee and J. H. Cheon, eds.), Lecture Notes in Computer Science, vol. 5461, Springer, 2008, pp. 218–234.

[25] ———, *Secure hardware implementation of nonlinear functions in the presence of glitches*, J. Cryptology **24** (2011), no. 2, 292–321.

[26] NIST, *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.

[27] ———, *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.

[28] E. Oswald S. Mangard and T. Popp, *Power analysis attacks — revealing the secrets of smartcards*, Springer-Verlag, 2007.

[29] İ. San and N. At, *Compact Keccak hardware architecture for data integrity and authentication on FPGAs*, Information Security Journal: A Global Perspective (2012), to appear.

[30] G. Sevestre, Keccak *tree hashing on GPU, using Nvidia Cuda API*, 2010, http://sites.google.com/site/keccaktreegpu/.

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

[32] K. Tiri and I. Verbauwhede, *A logic level design methodology for a secure DPA resistant ASIC or FPGA implementation*, in Gielen and Figueras [16], pp. 246–251.

[33] C. Wenzel-Benner and J. Gräf, *XBX: keeping two eyes on embedded hashing*, `http://xbx.das-labor.org/`, accessed 21 December 2010.

Appendix A

Change log

A.1 From 3.1 to 3.2

- Added a section on slice processing in Section 2.6.
- Added comments on the XOP and NEON instruction sets and on the ARM Cortex-M0 in Chapter 3.
- Removed outdated performance estimation on Intel 8051 processor from Chapter 3.
- Added the mid-range core in Section 4.4.
- Removed old FPGA results from Section 4.6.
- Added comment on non-uniformity of Equations (5.2) and (5.5).
- Added Section 5.3.1 on simplifying the software implementation.