Bachelor thesis
Computer Science

Radboud University

Side channel protected Keyak on ARM Cortex-M4

Author:
Martin Meyers
s4497899

First supervisor/assessor:
PhD, Ilya Kizhvatov
`i.kizhvatov@science.ru.nl`

Second supervisor:
Prof., Joan Daemen
`j.daemen@cs.ru.nl`

Second assessor:
Prof., Lejla Batina
`lejla@cs.ru.nl`

August 27, 2017

Abstract

Authenticated encryption provides both confidentiality and authentication where non authenticated encryption only provides confidentiality. One authenticated encryption protocol is Keyak. Keyak encrypts and decrypts messages in sessions and is one of the algorithms that is competing in the CAESAR competition.

One of the things implementations of cryptographic algorithms need to protect against are side channel attacks. If there is no protection against side channel attacks, information about the key can leak, which makes it easier for an attacker to find the key. A specific side channel attack is differential power analysis (DPA), which is a statistical analysis on the power usage of an implementation. To protect against DPA, masking can be used. This means that a secret variable is split up into two or more shares that together make up the secret variable.

We present a masked implementation of Keyak on an ARM Cortex M4 that is capable of reusing randomness. To do so, we describe Keyak, what DPA is, what masking is and how reusing randomness works in more detail. We then show what decisions we made and what features are implemented.

Contents

1 Introduction **2**

2 Keyak **3**

2.1 Motorist mode 3

2.2 Keccak-p 4

2.3 Named instances 5

3 Side channel attacks and countermeasures **7**

3.1 Side channel attacks 7

3.2 Masking 8

4 Research **10**

4.1 Two share Keyak 10

4.1.1 Code decisions 10

4.1.2 Masking 12

4.1.3 Reuse randomness 13

4.1.4 Implementation 13

5 Related Work **15**

6 Conclusion **16**

Chapter 1

Introduction

One of the things implementations of cryptographic algorithms should protect against are side channel attacks. Side channel attacks are attacks that use information like power consumption to find out details about the used key. We present an implementation of River Keyak that has protection against side channel attacks (on an ARM Cortex M4). There is no other publicly available protected implementation for Keyak on this platform yet, so Keyak could not be used on this platform if a side channel protected implementation is necessary. With this paper, we change that.

Keyak is one of the algorithms that made it into the third round of the CAESAR competition [2], which means that it could become one of the algorithms that 'wins'. CAESAR is a competition for authenticated encryption. Authenticated crypto has the advantage over non-authenticated crypto that there are no separate computations needed to provide authentication in addition to confidentiality.

The specific side channel attack that our implementation has protection against, is differential power analysis (DPA). With this technique an attacker uses multiple measurements of the device's power consumption or electromagnetic emanations to search for leaks (for more information, see section 3.1). Other techniques can abuse other leaks of information, like timing.

To protect against DPA, the implementation uses masking. That means that the state s is split up into two shares, a and b (this is explained in more detail in section 3.2). If an attacker now wants to find information about state s, he would need to gain knowledge about both share a and share b. Finding information from the two shares is harder than finding information directly from state s.

Chapter 2

Keyak

The benefit of authenticated encryption is that it provides both confidentiality and authentication where non-authenticated crypto only provides confidentiality.

Keyak is an encryption protocol that supports authenticated encryption in sessions [6]. In a session, it encrypts messages into cryptograms and decrypts cryptograms into messages. A message consists of plaintext and metadata. A cryptogram consists of the ciphertext, the metadata and a tag. The metadata in the cryptogram is not encrypted. The tag is calculated over all the messages from the start of the session and can be used to verify the authenticity of the complete message history.

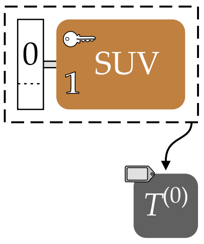
Keyak uses the motorist mode (see section 2.1) with the Keccak-p permutation (see section 2.2). The parameters that the motorist mode has can vary. However, there are five named instances where those parameters are fixed (see section 2.3).

2.1 Motorist mode

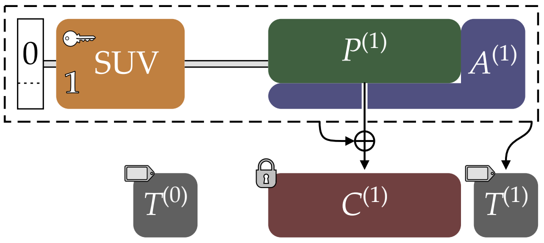
The motorist mode works by keeping a state and absorbing data into that state. The state starts as an all-zero state. To start a session, the secret key and possibly a nonce are required. If the secret key is used multiple times, a nonce must be present. The key and nonce are also called the Secret Unique Value (SUV). Starting the session is depicted in Figure 2.1a.

When encrypting, the state provides a keystream, and after the message is absorbed, it provides a tag that authenticates all absorbed data. Figure 2.1b shows a schematic of this.

When decrypting, the state provides a keystream and it absorbs the cryptogram. At the end, it also provides the tag that is used to verify the authenticity of the received data.



(a) Absorbing the Secret Unique Value (SUV) into the state



(b) Encrypting a message that contains both plaintext and metadata. Note that the metadata is not encrypted, but it is authenticated by the tag.

Figure 2.1: A motorist session[6]

The motorist mode has three different layers:

- the Piston, which keeps the state, performs the basic functions needed to set and get the state and performs the permutation
- the Engine, which controls the piston
- the Motorist, which is the user interface. It can be used to start a session, wrap a message into a cryptogram or unwrap a message from a cryptogram.

The mode also has support for parallelism. If used, the message is distributed over multiple pistons, and each piston calculates a cryptogram of its own part. To make sure the tag from the cryptograms still authenticates all messages from the start of the session, a special operation called knotting is performed. For more on how the knotting works, see [6, p. 13].

2.2 Keccak-p

Keccak-p is derived from Keccak-f, which is defined in [3]. Keccak-$p[b, n_r]$ consists of the application of the last n_r rounds of Keccak-$f[b]$. It is a sequence of n_r rounds where each round consists of five steps (from [6]):

$$Round = \iota \circ \chi \circ \pi \circ \rho \circ \theta, \text{ with}$$

$$\begin{array}{ll}
\theta: & a[x,y,z] \leftarrow a[x,y,z] + \sum_{y'=0}^{4} a[x-1,y',z] + \sum_{y'=0}^{4} a[x+1,y',z-1] \\
\rho: & a[x,y,z] \leftarrow a[x,y,z-(t+1)(t+2)/2] \\
\pi: & a[x,y] \leftarrow a[x',y'] \text{ ,where } \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x' \\ y' \end{pmatrix} \\
\chi: & a[x] \leftarrow a[x] + (a[x+1]+1)a[x+2] \\
\iota: & a \leftarrow a + \mathrm{RC}[i_r]
\end{array}$$

with $0 \leq t \leq 24$ and $\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}^t \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix}$ in $\mathrm{GF}(5)^{2\times 2}$
or $t = -1$ if $x = y = 0$.
The additions and multiplications are in GF(2). The rounds are identical with the exception of the round constants (RC). $a[x,y,z]$ denotes a bit in the state of Keccak. The state in Keccak is three dimensional, which is where the x, y and z come from. If either x, y or z is missing, then the statement is valid for all values of the missing variable. The bit $a[x,y,z]$ is bit $s[w(5y+x)+z]$ in the (one dimensional) state s. w is the maximum value for z and is defined by the width of the permutation b ($w = 2^l$ and $b = 25 \times 2^l$).

The steps in Keccak each perform a specific function. All but the χ step are linear steps.

- θ is the mixing step, it provides diffusion to the round
- ρ and π are transportation steps, providing dispersion
- χ is the non-linear step, and it is the only step that prevents the round function from being linear.

For a more detailed description of Keccak-p, see section 2.1 of the Keyak documentation [6] or the Keccak reference [3].

2.3 Named instances

There are five named instances of Keyak, listed in Table 2.1. All five instances use 12 rounds for the Keccak-p permutation. The main differences between these are the width of the permutation (b) and the amount of pistons used (Π).

For all named instances other than River Keyak, the alignment unit W is 64 bits, and for River Keyak, the alignment unit is 32 bits. This means that

Name	b	Π
River Keyak	800	1
Lake Keyak	1600	1
Sea Keyak	1600	2
Ocean Keyak	1600	4
Lunar Keyak	1600	8

Table 2.1: Named instances of Keyak

for River Keyak, the lanes are 32 bits long, while for the other instances, the lanes are 64 bits long. The capacity c of all the named instances is 256 bits.

From this, we can calculate the squeezing rate R_s, which determines how many bits of the state are never used as output. We can also calculate R_a, which determines how many bits of the state can be used for absorbing data. For River Keyak, $R_s = 68$ bytes and $R_a = 96$ bytes. For all other instances, $R_s = 168$ bytes and $R_a = 192$ bytes.

Chapter 3

Side channel attacks and countermeasures

3.1 Side channel attacks

Cryptographic protocols are made to be mathematically sound. The protocol itself is designed to be secure against attacks by adversaries with access to input and output. However, this does not mean that implementations of the protocol are safe against more powerful attackers. Even if the protocol is implemented correctly, there can still be manners in which information about a plaintext or a key can leak, like side channel attacks.

One type of side channel attack uses power consumption during the execution of the implementation to find out more about the key that is being used. One of those attacks is called differential power analysis (DPA). It is a statistical attack that exploits the correlation between the power usage and the variables that are used during calculations.

There are two forms of DPA. First order DPA and higher order DPA. Higher order DPA attacks differ from first order DPA attacks in the sense that they consider more samples from the same power consumption trace [14]. Because we only defend against first order DPA, we'll not go into detail on how higher order DPA works.

A first order DPA attack starts with collecting multiple traces of power consumption. This set of traces is then split into two subsets, based on a key hypothesis (guessed part of the key). Then the average power consumption of the two subsets is compared. If the key hypothesis is incorrect, the set of traces will be randomly divided over the subsets. Statistically, the difference in the averages of the two subsets will therefore be close to zero. If the key hypothesis is correct, the averages will be different than zero. With enough traces, it is possible to distinguish the two cases [15, 16]. Figure 3.1 shows how differences between the averages of the traces can look like.

With a properly designed cryptographic protocol that absorbs the key

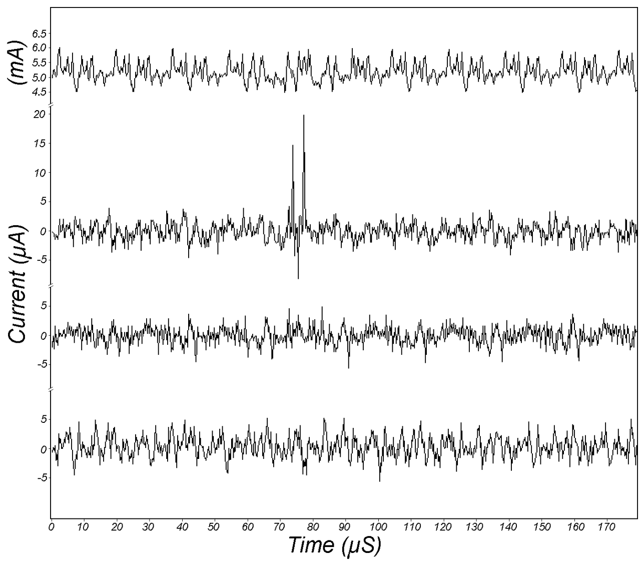


Figure 3.1: DPA traces, from top to bottom: one power reference and three traces of the difference between the averages of the two subsets, the first with a correct key hypothesis and the other two with an incorrect key hypotheses [15]

and nonce into a state, DPA will only need to be prevented when the state is permuted after key and nonce absorption. After that, the state is fully decorrelated, which makes DPA with state hypothesis impossible. If an attacker can force multiple sessions with the same nonce, the state after the key and nonce absorption will be the same in those sessions, and the attacker can attack that state with multiple plaintexts and metadata. This is why DPA prevention in Keyak is only needed for the permutation after key and nonce absorption, and the key and nonce combination must be unique.

3.2 Masking

Masking is a method to defend against DPA. The idea is that the input is split up into multiple shares. The *order* of masking is the amount of shares that are being used. If properly implemented, an n order mask defends against $(n-1)$ order DPA [5]. Because we are trying to defend against first order DPA attacks, we will only discuss second order masking.

In second order masking, the input is split up into two shares (share a and share b) that together make up the original (native) state s, in such a way that $s = a + b$ [9, p. 14, 15]. So the non-masked function would be performed as depicted in Figure 3.2, and the masked function would be performed as depicted in Figure 3.3.

$$\begin{tikzcd}
\text{State } s & I \arrow[r] & f(I)
\end{tikzcd}$$

Figure 3.2: Non-masked function f

$$\begin{tikzcd}
\text{Share } a & I \arrow[r] & {\alpha = I + R} \arrow[r] \arrow[dr] & {f'(\alpha, \beta)} \arrow[r] & f(I) \\
\text{Share } b & R \arrow[r] \arrow[ur] & {\beta = R} \arrow[r] \arrow[ur] & {f'(\beta, \alpha)} \arrow[r] \arrow[ur] & {f'(\beta, \alpha)}
\end{tikzcd}$$

Figure 3.3: Masked variant of function f

In the masked variant (Figure 3.3), share b starts off with the random value R. This value is unknown to an adversary and must be fresh. Share a starts off with the value I, which is the input for the function (and the same as in Figure 3.2). The first step is to bitwise add share b into share a. This makes sure that the shares independently don't have any information about the native state s, but by combining them, s can still be calculated.

The next step is the function step. It uses an altered function, which is called the shared function. This shared function is needed because otherwise, the shares together would not add up to the native state anymore [1]. To function correctly, some data from share b is needed in the function of share a and vice-versa.

The last step combines the two shares together by bitwise adding share b into share a again, recreating the content of the native state from the non-masked function in share a.

Because DPA prevention is only needed for the permutation after key and nonce absorption (see 3.1), masking is only needed for the first permutation of each session. This means that the randomness for the second share is also only needed at the start of each Keyak session. When the permutation continues in non shared mode, the permuted randomness will stay available in share b.

Chapter 4

Research

4.1 Two share Keyak

Because the implementation is made for the ARM Cortex M4, which is a 32 bit system, we chose to implement a masked version of River Keyak. River Keyak is the named Keyak instance that operates on 32 bit lanes, using an 800 bit permutation. Because the lanes are 32 bits and the ARM Cortex M4 is a 32 bit system, the lanes fit nicely into the 32 bit words of the system. By creating a second order masked implementation in software, we have an implementation that should be resistant to first order DPA attacks.

Because masking is only needed on the permutation after absorbing the key and the nonce (see 3.1 and 3.2), the impact of the masking on the speed is limited.

To make sure the implementation would be easy to use, we also wanted to try and keep the original interface that was already present in the code that was already available. This is explained in more detail in section 4.1.1.

The last thing we wanted to do was to reuse the randomness. This has big advantages on embedded devices (for which the implementation is made) because creating good random data on those devices can be really costly. Why and how we did this can be found at 4.1.3.

In short, the goals of this implementation are

- To create a masked implementation of Keyak,
- Keeping the original interface, and
- Reuse the randomness

4.1.1 Code decisions

We used code that was already available and modified it to make it a masked implementation. The code that we used came from the Keccak Code Package [4].

The masking needed to be implemented in assembly because there are certain operations that can leak information if not used carefully. For example, by overwriting a register that contains the first lane from share a with the first lane of share b, information about the difference between the two shares is leaked. And because the native state $s = a + b$, information about the difference between the two shares is information about the state. Therefore, a lot of control over the registers and memory is needed to make sure that there are no leaks. The control over what variable is placed in what register is only really present in assembly.

In the code that we were using, the Keccak function has its own interface. This interface is called the SnP interface. It specifies functions for:

- initializing the state, which sets the state to all-zero,
- adding to the state, which bitwise adds data into the state,
- overwriting the state, which overwrites data in the state,
- permuting the state, which performs the permutation on the state, and
- extracting data from the state, which gets bytes from the state.

The interface can be found in the Keccak Code Package [4, at SnP/SnP-documentation.h]. To be able to create and merge the two shares, we added two functions to that interface: `share` and `merge`.

To be able to keep track of the two shares and for some book keeping, we also had to change the representation of the state. Where it was just the state in the unmasked version, we needed to expand that to make room for both shares and two flags: the `shared flag` and the `random available flag`. The `shared flag` is to let later functions know that the state is currently divided into two shares and the `random available flag` is to let them know that randomness is still available and doesn't have to be recreated. Reusing randomness is explained in section 4.1.3 and more about the new state can be found in section 4.1.4.

The `share` function creates the second share. It does so by copying random data into both shares in such a way that both shares are equal. This means that the native state (consisting of two shares) is all-zero. It also sets the `shared flag` and the `random available flag`.

The `merge` function removes the `shared flag` and makes the two shares into one again. It does this by bitwise adding the first and second share together and putting the result in the first share. This means that the first share now contains the same result as it would have after an unmasked permutation. The second share is untouched, so the randomness can be reused as described in section 4.1.3.

A schematic overview of what the `share` and `merge` function do can be seen in Figure 4.1.

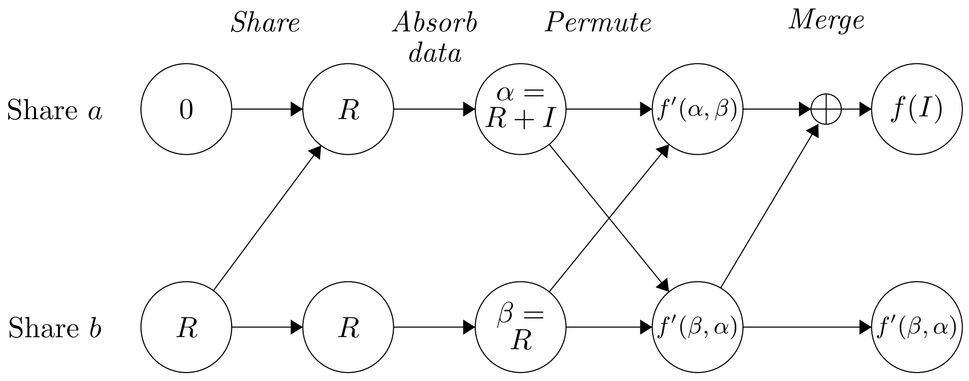


Figure 4.1: Overview of the masked operation

4.1.2 Masking

To make the masked implementation of Keyak, the biggest part that has to be changed is the code of the permutation. The other parts only have to be edited slightly for book keeping (keeping track if the state should be shared or not, to call the share and merge function and to supply random bytes).

The linear layers of Keccak (θ, ρ and π) can be done on both shares without a change. This is due to the fact that linear layers have the property that $f(a+b) = f(a) + f(b)$. In this case, a and b can be seen as the shares, and $a+b$ as the original state s. However, in the round function of Keccak, not all layers are linear. Layer χ is not linear, so performing the layer on both shares without changing it will not work. Without masking, the layer does this: (x_i is $a[x]$ and x_{i+1} is $a[x+1]$ from 2.2)

$$x_i \leftarrow x_i + (x_{i+1} + 1)x_{i+2}$$

Lemma 1. To make layer χ work with two shares, it can do the following (as described in [5]):

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

Proof. Note that $x_i = a_i + b_i$. If we apply that, we get:

$$\begin{aligned} x_i &\leftarrow a_i + (a_{i+1} + 1)a_{i+2} + a_{i+1}b_{i+2} + b_i + (b_{i+1} + 1)b_{i+2} + b_{i+1}a_{i+2} \\ &= a_i + b_i + a_{i+1}a_{i+2} + a_{i+2} + b_{i+1}b_{i+2} + b_{i+2} + a_{i+1}b_{i+2} + b_{b+1}a_{i+2} \\ &= x_i + (a_{i+1} + b_{i+1})a_{i+2} - b_{i+1}a_{i+2} + (b_{i+1} + a_{i+1})b_{i+2} - a_{i+1}b_{i+2} \\ &\quad + a_{i+2} + b_{i+2} \\ &= x_i + (x_{i+1} + 1)a_{i+2} + (x_{i+1} + 1)b_{i+2} \\ &= x_i + (x_{i+1} + 1)(a_{i+2} + b_{i+2}) \\ &= x_i + (x_{i+1} + 1)x_{i+2} \end{aligned}$$

□

The other layer that needs to be edited is layer ι (adding the round constant). If the round constant is added in both shares, it would vanish when the two shares are merged. Therefore, it should only be added into one of the two shares.

4.1.3 Reuse randomness

Computers are deterministic systems, which makes it costly to create randomness on them. This is why we chose to implement the ability to preserve randomness when merging to be able to use it again when sharing the state again. Instead of having to generate new randomness every time we start a Keyak session, it is possible to reuse the randomness from the initialization of the previous session. Because the randomness in the old instance already has been transformed by the (shared) permutation, the values that are being used are decorrelated. In the example of Figure 4.1, the $f'(\beta, \alpha)$ from share b will be used as the R in the next session.

4.1.4 Implementation

Because we wanted to keep the changes to the interface of Keccak to a minimum, we couldn't add a second parameter for the masked state to the interface. To solve this, we increased the size of the original state from 100 to 204 bytes. The first 100 are for share a, the second 100 are for share b, the 201st byte is for the masked flag and the 202nd byte is for the random available flag. The remaining two are for alignment. This makes the state 204 bytes large, which is 1632 bits. 1632 is the smallest size greater than 1616 (which is 202 bytes, the minimum size that is needed), that is divisible by 32, which makes initializing the new state a bit easier and faster. So the new state consists of:

Function	Size	Description
Share a	100 bytes	The first share when the permutation is shared (a)
		The state when the permutation is not shared (s)
Share b	100 bytes	The second share when the permutation is shared (b)
		Nothing or randomness when the permutation is not shared
Share flag	1 byte	1 if the permutation is shared, 0 otherwise
Random available flag	1 byte	1 if randomness is still available in share b, 0 otherwise
Empty bytes	2 bytes	Alignment purposes

To write and test the code we used the Keil debugger[1]. The code can be found at `https://gitlab.science.ru.nl/mmeyers/BachelorThesis/` in the code directory.

[1]`http://www.keil.com/`

Chapter 5

Related Work

Implementations of cryptographic algorithms using a mask is not new. It has been around for some time [1]. A lot of the masked implementations and research on masked implementations is being done with hardware masking ([8] and [10] for example). This is also already done for Keccak and Keyak [13, 18]. However, software implementations are cheaper to implement (because no special hardware is needed), which makes them interesting as well. Taking the benefits of both is also something that researchers have been looking into [12].

Also very interesting is the work that is being put into attacking Keccak and Keyak from the cryptanalysis point of view. At the time of writing, there are only attacks against Keccak and Keyak when less than 12 rounds are performed [11, 7, 17]. However, because the attacks work up to the 8th round, the 12 rounds have sufficient security margin.

Chapter 6

Conclusion

We have presented a second order masked implementation of Keyak for an ARM Cortex M4. To speed up the initialization, the randomness from the old instance can be reused in the new instance. The implementation isn't tested yet, so we cannot claim that it offers protection against first order DPA. This is something that will need to be done in later research.

Bibliography

[1] Mehdi-Laurent Akkar and Christophe Giraud. *An Implementation of DES and AES, Secure against Some Attacks*, pages 309–318. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[2] D. J. Bernstein. Ceasar submissions. `http://competitions.cr.yp.to/caesar-submissions.html`.

[3] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The keccak reference, 2011. `http://keccak.noekeon.org/Keccak-reference-3.0.pdf`.

[4] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Keccak code package. `https://github.com/gvanas/KeccakCodePackage`.

[5] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Keccak implementation overview. `http://keccak.noekeon.org/Keccak-implementation-3.2.pdf`.

[6] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Caesar submission: Keyak v2, 2016. `http://keyak.noekeon.org/Keyakv2-doc2.2.pdf`.

[7] Wenquan Bi, Zheng Li, Xiaoyang Dong, Lu Li, and Xiaoyun Wang. Conditional cube attack on round-reduced river keyak. *Designs, Codes and Cryptography*, Jul 2017.

[8] Begül Bilgin, Joan Daemen, Ventzislav Nikov, Svetla Nikova, Vincent Rijmen, and Gilles Van Assche. *Efficient and First-Order DPA Resistant Implementations of Keccak*, pages 187–199. Springer International Publishing, Cham, 2014.

[9] Suresh Chari, Charanjit Jutla, Josyula R. Rao, and Pankaj Rohatgi. A cautionary note regarding evaluation of aes candidates on smartcards. `http://csrc.nist.gov/archive/aes/round1/conf2/papers/chari.pdf`.

[10] Cong Chen, Mohammad Farmani, and Thomas Eisenbarth. A tale of two shares: Why two-share threshold implementation seems worthwhile - and why it is not. `https://eprint.iacr.org/2016/434.pdf`.

[11] Itai Dinur, Paweł Morawiecki, Josef Pieprzyk, Marian Srebrny, and Michał Straus. *Cube Attacks and Cube-Attack-Like Cryptanalysis on the Round-Reduced Keccak Sponge Function*, pages 733–761. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[12] Hannes Gross and Stefan Mangard. Reconciling d+1 masking in hardware and software. Cryptology ePrint Archive, Report 2017/103, 2017. `http://eprint.iacr.org/2017/103`.

[13] Hannes Gross, David Schaffenrath, and Stefan Mangard. Higher-order side-channel protected implementations of keccak. Cryptology ePrint Archive, Report 2017/395, 2017. `http://eprint.iacr.org/2017/395`.

[14] Marc Joye, Pascal Paillier, and Berry Schoenmakers. *On Second-Order Differential Power Analysis*, pages 293–308. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[15] Paul Kocher, Joshua Jaffe, and Benjamin Jun. *Differential Power Analysis*, pages 388–397. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[16] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1(1):5–27, Apr 2011.

[17] Kexin Qiao, Ling Song, Meicheng Liu, and Jian Guo. New collision attacks on round-reduced keccak. Cryptology ePrint Archive, Report 2017/128, 2017. `http://eprint.iacr.org/2017/128`.

[18] Niels Samwel and Joan Daemen. Dpa on hardware implementations of ascon and keyak. In *Proceedings of the Computing Frontiers Conference*, CF'17, pages 415–424, New York, NY, USA, 2017. ACM.