BACHELOR

The Keccak sponge function family

Szanto, Justin B.

Award date:
2018

Eindhoven University of Technology
Department of Mathematics and Computer Science

The Keccak sponge function family

August 21, 2017

Justin Szanto
j.b.szanto@student.tue.nl

Abstract

In this paper we study the Keccak sponge function family. We describe its workings in detail and discuss multiple applications. Furthermore we explore some attacks on Keccak of which we discuss the cube attack in detail. Lastly we analyse the permutation function used in Keccak and determine that it acts closely to a random permutation.

Contents

1 Introduction

In 2007 the National Institute of Standards and Technology (NIST) anounced a competition to develop a cryptographic hash function to be standardized as SHA-3, the successor of SHA-2 and SHA-1. In 2012 Keccak was selected as the winner of the competition and has been standardized as SHA-3 in 2015 [14]. In this paper we aim to give an introduction to Keccak and to perform some basic cryptanalysis of Keccak.

We first introduce the notion of a sponge function, which plays an important role in the definition of Keccak and potentially many other cryptographic functions. In the second part of this paper we explore attacks on Keccak and perform some experiments using a technique known as the cube attack [7]. Finally we explore the periodicity and differential properties of Keccak.

1.1 Sponge functions

Keccak is a family of sponge functions. We shall give a brief introduction about sponge functions here. The notion of a sponge function was first introduced by Bertoni et al. as a a reference for expressing security claims. Later it was also levereged as a design tool for cryptographic primitives and was used for the design of Keccak. For the formal definition of sponge functions we refer the reader to [3]. A sponge function is a special type of function for two reasons. Firstly there is no restriction on the input, any finite length bitstring can be given as input. Secondly the length of the output is no way determined by the input, nor is it restricted in any way. The length of the output can be any chosen number of bits. Sponge functions provide a generic method to implement various cryptographic primitives.

A sponge function is constructed using a fixed-length permutation function and a reversible padding rule [1]. A schematic overview of a sponge construction is given in figure 1. The sponge function uses a

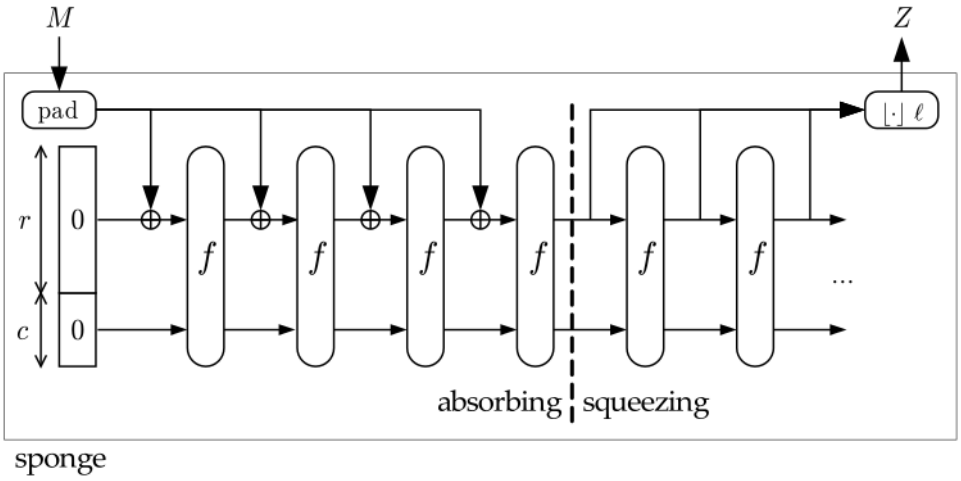

Figure 1: The sponge construction, from [3].

fixed-length permutation (or transformation) function f which operates on a fixed size state of b bits, which we refer to as the *width* of the function. The *bitrate* of the sponge function is denoted by r and the *capacity* is denoted by c. The variables r and c must be chosen such that $b = r + c$.

Sponge functions use two phases; an absorbing (input) phase and a squeezing (output) phase. In the absorbing phase a reversible padding rule is applied to the message M and it is split in to blocks of length r. First the initial state is set to a bitstring of length b consisting of only zeroes. All input blocks of length r are xor'ed in to the first r bits of the state, interleaved by applications of the function f. Once all input blocks have been absorbed, the function switches to the squeezing phase. In the squeezing phase the first r bits of the state are repeatedly output interleaved by applications of the function f, until the

[1] Padding rules are used to add bits to the message to extend the message such that it consists of a round number of blocks of length r.

desired number of output bits ℓ has been reached. The last c bits of the state are never outputted. This process is shown in figure 1.

Sponge functions have a large number of different applications within cryptography, including hashing, encryption, pseudo random number generation and MAC computation.

1.1.1 Width, bitrate and capacity

Important parameters of a sponge construction include the width $b = r + c$, the bitrate r and the capacity c. Together with the permutation (or transformation) function f these parameters define the security level and speed of the sponge function in the following ways.

- The width b defines the size of the state of the sponge function. The function f must take b bits as input and return b bits as output. A larger value of b generally implies that a single application of f will take longer.
- The c-bit section of the state is never output nor is it influenced directly by the input. Given that f may be a invertible function, an attacker can trivially retrieve the input M when given the entire state of the sponge function. Therefore a larger value of c implies a greater level of security. In section 1.1.4 we will show more details on how c influences the security level of various applications of sponge functions.
- The bitrate r defines the number of bits which can be absorbed between applications of f and the number of bits which can be outputed between applications of f. A small value for r may increase the level of security, however at the same time it will negatively impact the speed of the sponge function.

1.1.2 The duplex construction

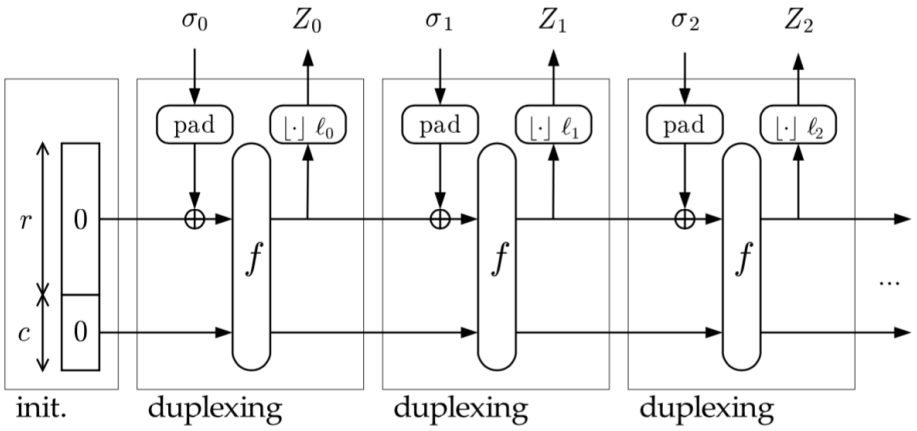

Figure 2: The sponge duplex construction, from [3].

An alternative method of utilizing sponge functions is known as the duplex mode. Rather than absorbing all input at once, the duplex mode provides one block of output after each block of input. A graphic representation of this construction is given in figure 2. In duplex mode the sponge construction can no longer accept arbitrarely sized inputs. The inputs must consist of blocks σ_i such that after applying the padding rule the block has exactly length r. Therefore the length of σ_i will be at most r. The output length ℓ must be at most r-bits. The output of a single duplexing call is the same as the output of a regular sponge function with the same parameters and therefore it offers the same level of security. A proof of this is given in [3]. This construction is useful for constructing pseudo random bit sequence generators and for authenticated encryption schemes, both will be covered in more detail in section 1.1.3.

1.1.3 Applications of sponge functions

The sponge construction can be applied to a wide range of cryptographic functions. In this section we briefly cover some examples of applications of sponge functions.

Hash function
A sponge function can be used as $n-$bit hash function by supplying the message as input to the sponge function and truncating the output to n bits. A randomized $n-$bit hash function can be constructed in a similar way, by appending a random string to the message.

Password based key derivation function (PBKDF
PBKDFs are very similar to hash functions as they aim at providing the same properties as hash functions, e.g. non-reversible, no collisions, etc. However when considering PBKDFs it is desireable to have a function which is much slower than a hash function, making it harder to attack using bruteforce methods. This can easily be achieved using a sponge function, by appending a large number of zeroes to the message. This results in more rounds of the internal permutation or transformation function and therefore slows down the sponge function.

MAC function
A MAC function takes a secret key, an initial vector (IV) and a message as input. If we concatenate these 3 inputs, feed them in to a sponge function and truncate the output to n bits, this gives us a $n-$bit MAC.

Stream cipher
If we provide an $n-$bit secret key combined with an initial vector (IV) as input to the sponge function then we can use the output as a key stream. While we may choose a large value for n, the effective strength of the stream cipher will be limited by c. The effective strength is given by $\texttt{min}(n, c)$.

Pseudo random bit generator
By providing as random seed as input to the sponge function we can consider the output to be a random stream of bits. However as with all pseudo random bit generators, the number of bits output before the output either starts repeating or becomes predictable is limited. To overcome this problem reseedable pseudo random bit generators can be used.

Reseedable pseudo random bit generator
The duplex construction as described in section 1.1.2 can be used to construct a reseedable pseudo random bit generator. This is done by maintaining a buffer with entropy. Once random output is required we feed one block of entropy in to the sponge function. If not enough entropy is available it is also possible to feed zeroes as input, which will result in more pseudo random bits based on the previous entropy.

Extendable output function (XOF)
An extendable output function (XOF) is a hash function with a variable length output. Some applications may require multiple different output lengths of a hash function. Instead of using combinations of existing hash functions to achieve this, an XOF provides an easy solution to this problem. An example application of XOFs is RSA-FDH [2]. RSA-FDH is a signature scheme using RSA encryption together with hashing. The length of the hash function output must be equal to the (variable) length of the RSA modulus. For the formal definition and motivation we refer the reader to [14].

XOFs can be implemented using sponge functions in the same way as hash functions by truncating the output to the desired length.

1.1.4 Sponge functions as a reference of security claims

When evaluating the security of cryptographic hash functions it is often assumed that these functions behave the same as a random oracle (a function which returns a uniformly random output) and therefore have the same level of security as a random oracle. These security claims are always based on the output length of the function. This model poses a problem when assesing function with a variable length output. For these functions the inner state may be shorter than the output length and therefore we can't make any security claims by comparing it to a random oracle with the same output length. To overcome these problems we can use a *random sponge* as a model for security claims. A random sponge is a sponge construction with a random permutation (or transformation) function f. When making security claims we can express the success probability of a certain attack using the sponge parameters, b, r and c. In a simpler model, refered to as the *flat sponge* we only consider the capacity c. A few examples of security claims using sponge functions are as follows.

Hash function collisions
Consider a hash function constructed using a sponge function with capacity c and output length n. If $c < n$ the optimal strategy to find a collision is to find an inner collision. Since we can control r bits of the state we only need to find a collision in the c random bits, which by the birthday attack can be done in $2^{c/2}$ operations. If $c > n$ it is more efficient to find a collision directly in the output which takes $2^{n/2}$ operations. Therefore the total number of operations required to find a collisions is on average $2^{\min(n,c)/2}$.

Stream cipher output prediction
We consider the stream cipher which we defined earlier, with a $n-$bit key. To predict the output we need to recover the state of the sponge function. If $n < c$ the optimal strategy is to find the n bits of the key in order to recover the initial state, which will take 2^n operations. If $c < n$ the optimal strategy is to recover the c bits of the state, which will take 2^c operations. So the the total number of operations required to predict the output of a stream cipher is on average $2^{\min(n,c)}$.

2 The Keccak sponge function family

We give a brief introduction to Keccak here, for the full specification please refer to [4]. Keccak makes use of the sponge construction in combination with the permutation function Keccak-$f(b)$ with $b \in \{25, 50, 100, 200, 400, 800, 1600\}$ and r and c chosen such that $b = r + c$. Keccak-$f(b)$ applies n_r-rounds of a sequence of 5 permutation steps on a b-bit state given by a three-dimensional array of size $5 \times 5 \times b/25$. The number of rounds n_r is given by $n_r = 12 + 2\log_2(b/25)$.

We shall now briefly explain each of the 5 permutation steps as described in the Keccak reference [4]. We denote the state by the three dimensional array $A[x][y][z]$, with $x, y \in \mathbb{Z}_5$ and $z \in \mathbb{Z}_w$ with $w = b/25$. All elements in A are elements of $\mathbb{Z}_2$ and all operations are performed in $\mathbb{Z}_2$. A schematic overview of the state array can be found in figure 7.

The θ step
In the θ step a single bit and two of its neighboring columns are summed acting as a diffusion step. The step is defined by

$$A[x][y][z] \leftarrow A[x][y][z] + \sum_{y'=0}^{4} A[x-1][y'][z] + \sum_{y'=0}^{4} A[x+1][y'][z-1]$$

for all x, y and z. A schematic representation of this step is given in figure 3. This step is linear and invertible.

The ρ step In the ρ step, bits are rotated within lanes, aimed at providing inter-slice diffusion. The step is defined by

$$A[x][y][z] \leftarrow A[x][y][z - (t+1)(t+2)/2]$$

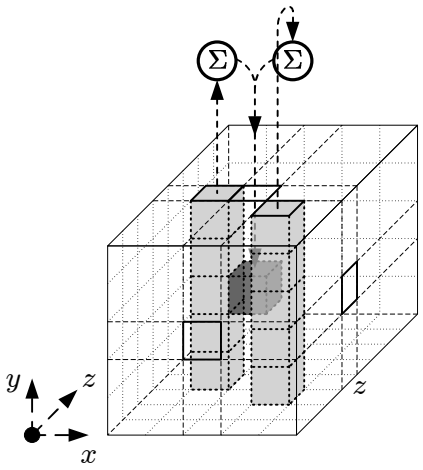

Figure 3: The θ step applied to a single bit, from [4].

with $t \in \{0, 1, 2, ..., 24\}$ and satifying

$$\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}^t \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix}$$

or $t = -1$ if $x = y = 0$ for all x, y and z. A schematic representation of this step is given in figure 4. This step is linear and invertible.

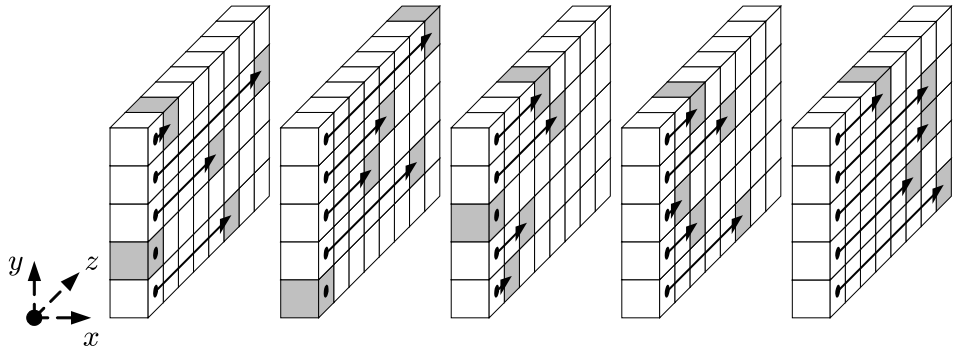

Figure 4: A schematic representation of the ρ step, from [4]

.

The π step The π step is a transposition of the lanes, providing a dispersion aimed at long term diffusion. Without it KECCAK-f would exhibit periodic trails of low weight [4]. The π step is defined by

$$A[x][y] \leftarrow A[x'][y']$$

with

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x' \\ y' \end{pmatrix}$$

for all x and y. A schematic representation of this step is given in figure 5. This step is linear and invertible.

The χ step The χ step can be seen as the parallel application of $5w$ S-boxes [2] operating on 5 bit rows. The χ step is defined by

$$A[x] \leftarrow (A[x + 1] + 1)A[x + 2]$$

[2] An S-box is a basic component of many crytographic systems which performs substitutions.

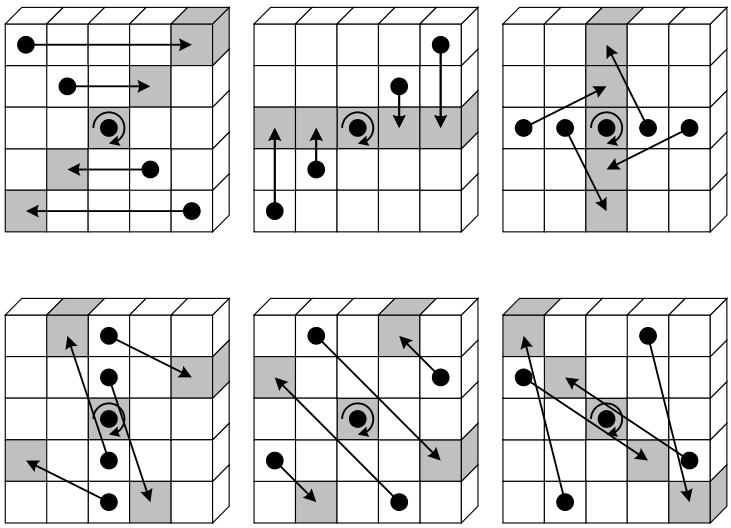

Figure 5: A schematic representation of the π step, from [4]
.

for all x. A schematic representation of this step is given in figure 6. This step is the only non-linear step, having degree 2. Without this step Keccak-f would be linear.

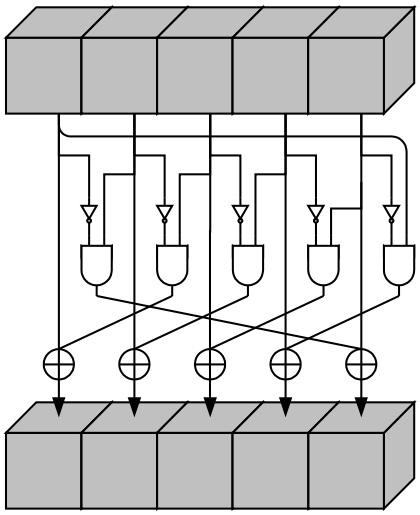

Figure 6: A schematic representation of the χ step, from [4]
.

The ι step The ι step consists of adding a round constant to the state. This ensures that every round is slightly different.

2.1 Keccak as SHA-3

Keccak can be used to compute a cryptographic hash of a message M by absorbing the entire message M in the absorbing phase. Followed by outputting the desired length l of the hash in the squeezing phase. After winning the SHA-3 contest in 2015, NIST standardized 4 hash functions based on Keccak and 2 extendable-output functions (XOFs) based on Keccak in the standard for the SHA-3 family [14]. These 6 functions consists of Keccak with various different parameters, which are given in Table 1.

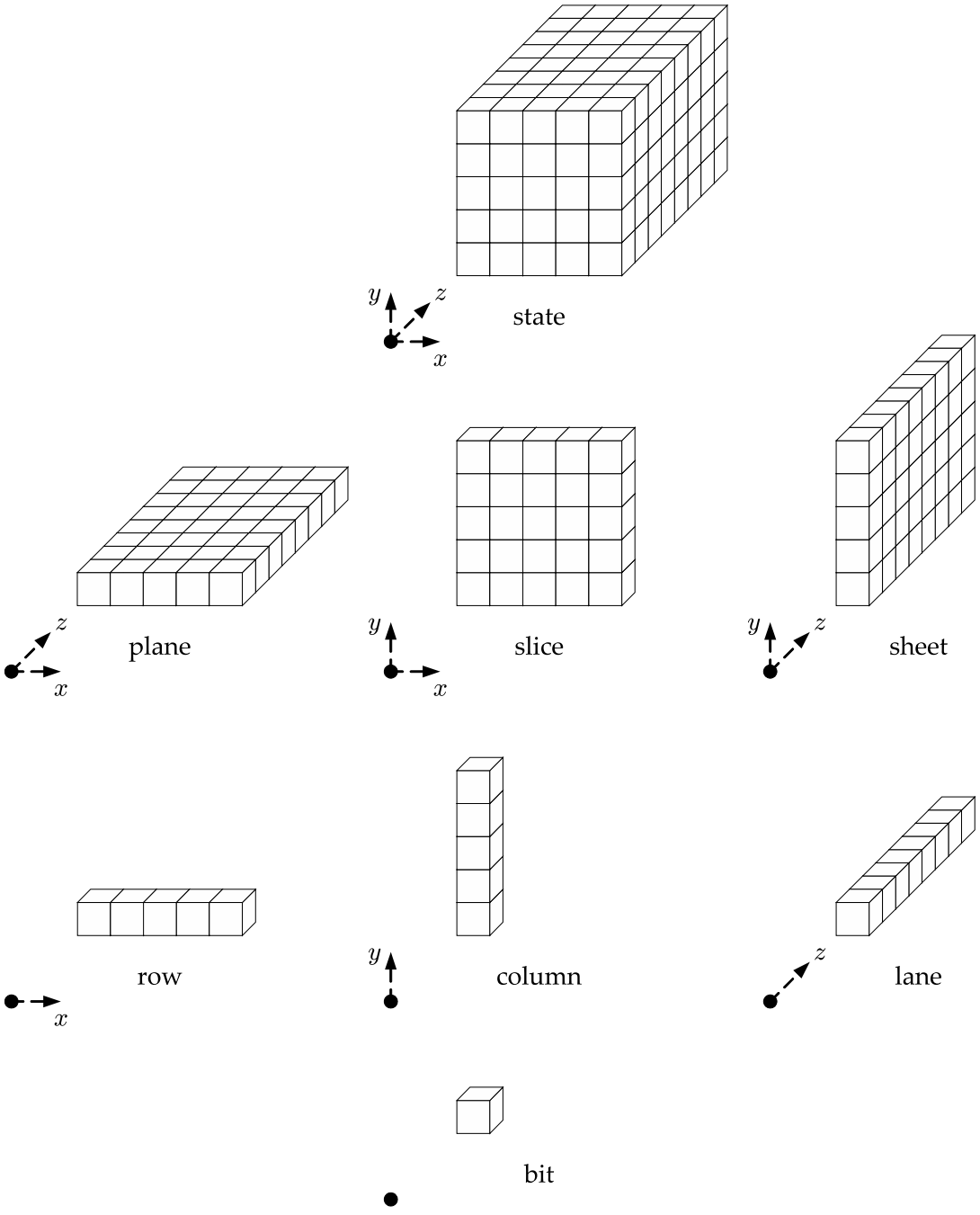

Figure 7: A schematic representation of the KECCAK-f state, from [4].

	rate (r)	Capacity (c)	Output length (l)	Security strength in bits		
				Collission	Preimage	2nd Preimage
SHAKE128	1344	256	d	$\min(d/2, 128)$	$\geq \min(d, 128)$	$\min(d, 128)$
SHAKE256	1088	512	d	$\min(d/2, 256)$	$\geq \min(d, 256)$	$\min(d, 256)$
SHA3-224	1152	448	224	112	224	224
SHA3-256	1088	512	256	128	256	256
SHA3-384	832	768	384	192	384	384
SHA3-512	576	1024	512	256	512	512

Table 1: The standardized hash functions SHA3-224/256/384/512 and XOFs SHAKE128/256.

3 Attacks on Keccak

Since its submission to the NIST contest, a lot of cryptanalysis has been performed on Keccak. So far a few practical attacks (e.g. a few days of computations) have been found for round reduced versions of Keccak. No serious flaws have been discovered yet. The cryptanalysis done on Keccak can be split into the following categories.

- **Differential cryptanalysis:** A general form of cryptanalysis in which the effect on the output of a function is studied when the attacker provides two different inputs which contain a constant difference. By studying the outputs the attacker attempts to find statistical differences between the two. Practical attacks leading to collisions in 5 round Keccak were found by Dinur et al. and are described in [6] and [5]. Similar results have been found by Qiao et al. in [15].
- **Cube attacks:** We will describe this attack in detail in section 4. Using this technique pratical attacks against 6-round Keccak have been found in [**Dinur2014**], [8] and [9].
- **Fault attacks / sidechannels:** An attack in which faulty values assigned to intermediate variables are used to recover the secret. Attacks stated in [10], [11] and [1] show that it takes a large number (> 80) of faults to recover a significant number of bits of the state of Keccak.
- **Rotational Cryptanalysis:** Many forms of symmetric cryptography make use of functions based on three operations: modular additions, rotations and XORs. In rotational cryptanalysis a system is analysed based on these 3 operations. In [13] this technique was used to find a pre-image attack on 4 round Keccak.
- **Distinguishing attacks:** In distinguishing attacks the attacker aims to distinguish the output of a cryptographic function from random output. In [16] it is shown that under certain inputs the sums of Keccak outputs after 9 rounds have symmetric properties.
- **Malicious implementations:** In [12] a malicious implementation of Keccak is given, in which only the round constants used in the ι step of the permutation differ from a correct implementation. When computing hashes with this implementation the amount of computations required to collisions and (2nd) pre-images is significantly reduced.

4 Cube attacks on Keccak

The cube attack is a chosen plaintext attack [3] for key-recovery. It was introduced by Dinur and Shamir in [7]. Suppose each output bit of a cipher can be represented by a black-box polynomial $p(x_1, x_2, \ldots, x_{n+m}) : \mathbb{Z}_2^{n+m} \to \{0, 1\}$ of degree d, in n secret variables and $m \geq d - 1$ public (tweakable) variables which can be manipulated by the attacker. Then an attacker can recover n secret bits in $2^{d-1}n + n^2$ operations on average. The attack consists of an offline preprocessing phase which has to be done once for each polynomial p and an online phase in which the secret bits are recovered. For polynomials with a small degree this attack is very fast, even for large values of n.

The main observation

The attack is based on a theorem by Dinur and Shamir in [7]. Before we present this theorem, we first

[3] An attack in which the attacker is able to obtain ciphertexts for some chosen plaintexts.

define the notion of a *cube* in this context. Let I be a subset of k tweakable variables of p with $0 < k < d$. A cube C_I contains 2^k vectors which contains all possible combinations of 0/1 in the variables in I.

Theorem 1 *(Dinur, Shamir) For all cubes C_I with I having length k, the degree of*

$$L(x) = \sum_{(x_1,x_2,\ldots,x_k)\ \in\ C_I} p(x_1, x_2, \ldots, x_k, x_{k+1}, \ldots, x_{n+m})$$

is at most $d - k$.

Proof
Write p as

$$p(x_1, x_2, \ldots, x_k, x_{k+1}, \ldots, x_{n+m}) = t_I\, L(x_1, x_2, \ldots, x_k, x_{k+1}, \ldots, x_{n+m}) + Q(x_1, x_2, \ldots, x_k, x_{k+1}, \ldots, x_{n+m})$$

in which $t_I = x_1 x_2 x_3 \ldots x_k$ and t_I does not divide Q and L contains no terms from t_I. In this way L has degree $d-k$. Since t_I is zero unless all bits in the cube are equal to one, we can write the sum as follows:

$$\sum_{(x_1,x_2,\ldots,x_k)\in C_I} t_I L(x_1, x_2, \ldots, x_k, x_{k+1}, \ldots, x_{n+m}) + Q(x_1, x_2, \ldots, x_k, x_{k+1}, \ldots, x_{n+m})$$
$$= L(1, 1, \ldots, 1, x_{k+1}, \ldots, x_{n+m}) + \sum_{(x_1,x_2,\ldots,x_k)\in C_I} Q(x_1, x_2, \ldots, x_k, x_{k+1}, \ldots, x_{n+m}).$$

Q is a sum of monomials, for each monomial we can distinguish two cases:

- The monomial contains l variables which are in I: This monomial is zero unless all l variables are equal to 1. In the cube there are 2^{k-l} vectors for which this is the case. Therefore it gets summed an even number of times and it is equal to zero.
- The monomial does not contain any variables which are in I: The monomial is summed 2^k times, which is an even number and the sum is equal to zero.

Therefore the contribution of Q to the sum is zero and the proof is complete.

If we choose $k = d - 1$ this theorem implies that we obtain a linear function. In the preprocessing phase we aim to construct multiple of these linear functions. In the online phase we obtain the values of these functions after which we can solve a system of linear equations to obtain the values of the secret bits.

Preprocessing (offline) phase
This phase begins by chosing one or more cubes C_I. We choose a set of $d-1$ variables from the set of tweakable (public) variables. For such a cube we compute a so called *superpoly* $L(x)$ belonging to a cube C_I. It is defined by the following sum.

$$L(x) = \sum_{(x_1,x_2,\ldots,x_k)\in C_I} p(x_1, x_2, \ldots, x_k, 1, 1 \ldots, 1, x_{m+1}, \ldots x_{n+m})$$

According to theorem 1 $L(x)$ has degree 1 and since all tweakable variables which are not part of the cube are set to 1 it must have the following form.

$$L(x) = a_1 x_1 + a_2 x_2 + \cdots + a_n x_n + c$$

Since all computations are in Z_2 we can easiliy compute the coefficients $a_1, a_2, ..., a_n$ by interpolation. E.g. by computing:

$$\begin{aligned} a_1 &= L(1, 0, 0, 0, \ldots, 0) \\ a_2 &= L(0, 1, 0, 0, \ldots, 0) \\ &\vdots \\ a_n &= L(0, 0, 0, 0, \ldots, 1) \\ c &= L(0, 0, 0, 0, \ldots, 0) \end{aligned}$$

As we will see in the online phase, we need to compute multiple different superpoly's such that when given the values of $L(x)$ for all chosen cubes C_I we can solve the set of equations and obtain the values of the secret bits $x_1, x_2, ..., x_j$.

Online phase

In the online phase all the secret bits are set to a fixed value and are obviously unknown to the attacker. For each cube C_I as chosen in the preprocessing phase we compute

$$b_t = \sum_{(x_1,x_2,\ldots,x_k)\in C_I} p(x_1, x_2, \ldots, x_k, x_{k+1}, \ldots, x_{n+m})$$

If we use the superpolys found in the preprocessing phase, we obtain a linear equation

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n + c = b_t$$

Doing this for all cubes C_I yields a system of equations which can be solved to obtain the values of the secret bits.

Example

The cube attack is best explained using a small example. Suppose a single bit of the output of some cryptosystem is given by the following polynomial.

$$p(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8) = x_1x_2x_5 + x_3x_8 + x_2x_5x_6 + x_1x_6 + x_6x_7x_3 + x_8x_2x_5 + x_4 + x_5$$

in which x_1, x_2, x_3 and x_4 are secret variables and x_5, x_6, x_7 and x_8 are tweakable variables which the attacker can influence. Suppose someone sets the secret bits to $(x_1, x_2, x_3, x_4) = (1, 0, 1, 0)$

We begin the offline phase by (randomly) choosing some cubes C_I in the public variables. The degree of p is 3, therefore we choose cubes for which $|I| = 2$.

$$\begin{aligned} C_{\{5,6\}} &= \{(0,0,1,1), (0,1,1,1), (1,0,1,1), (1,1,1,1)\} \\ C_{\{6,7\}} &= \{(1,0,0,1), (1,0,1,1), (1,1,0,1), (1,1,1,1)\} \end{aligned}$$

Next we compute the three superpolys corresponding to these cubes using the method described earlier.

$$\begin{aligned} L_1(x) &= x_2 \\ L_2(x) &= x_3 \end{aligned}$$

As we see here, we do not have enough superpolys to recover the value of x_1 and x_4. We possibly could try more cubes to try and obtain more superpolys to help us recover these values, however to keep this example small we will restrict ourselves to the two which we have found. Next we move on to the online phase and compute the values of b_t for our cubes. We set the values of (x_1, x_2, x_3, x_4) to $(1, 0, 1, 0)$, however we only use the outcome of p for our computations.

$$\begin{aligned} b_1 &= \sum_{(x_5,x_6,x_7,x_8)\in C_{\{5,6\}}} p(x_1, x_2, \ldots, x_8) = \sum_{(x_5,x_6,x_7,x_8)\in C_{\{5,6\}}} x_8 + x_6 + x_6x_7 + x_4 + x_5 = 0 \\ b_2 &= \sum_{(x_5,x_6,x_7,x_8)\in C_{\{6,7\}}} p(x_1, x_2, \ldots, x_8) = \sum_{(x_5,x_6,x_7,x_8)\in C_{\{6,7\}}} x_8 + x_6 + x_6x_7 + x_4 + x_5 = 1 \end{aligned}$$

This yields the following system of linear equations.

$$\begin{aligned} x_2 &= 0 \\ x_3 &= 1 \end{aligned}$$

From these equations we directly obtain the values of x_2 and x_3 which match the values which we assigned to them.

4.1 Cube attacks on round reduced keccak

When using KECCAK in stream cipher mode as described in section 1.1.3 we can apply the cube attack to KECCAK. Assuming we use it with a 128 bit secret key, a 128 bit IV and the attacker can modify the IV, we have 128 secret variables and 128 tweakable variables.

As seen in section 2 each round of KECCAK-f has degree 2, therefore the degree d of KECCAK-f is 2^{n_r}. Since the running time for this attack is exponential in d, it is not feasible to perform it the full 24 round version of KECCAK. We therefore perform an attack on a 3 round version of KECCAK.

The implementation of the attack proceeds in the following way. In the offline phase we repeatedly randomly select $d-1$ variables from the set of 128 tweakable variables and create a cube using these variables. For each cube we perform the steps needed to obtain a superpoly corresponding to this cube. We repeat this step for the first 128 bits of output. Most superpolys found are either zero or constant and therefore not of any use for us. To find a sufficient number of linearly independent superpolys we repeatedly pick random cubes until we have obtained a sufficient amount to be able to recover all 128 secret bits. Since we often find duplicate or linearly related superpolys we need to find many more than 128 polynomials before we are able to recover all 128 secret bits.

For a 3 round variant of KECCAK, finding a sufficient number of superpolynomials took aound 1 hour on a standard desktop computer, while utilizing only 1 CPU core. The code and superpolynomials used can be found in appendices A and B.

We repeated this attack for various smaller key sizes. In figure 8 the relation between the number of found superpolynomials and the number of recoverable secret bits is given. We can easily model this process using probability theory.

Since we choose random cubes for our attack and assume that KECCAK-f is a random permutation, it is reasonable to assume that the variables which we are able to recover with a given superpolynomial are random as well. If we have n secret bits and have already recoverd j secret bits, then the probability of finding a new secret bit is $\frac{n-j}{n}$. Now let X_j be the number of superpolynomials required to find a new secret bit if we already have recoverd j bits. For $j \geq 0$ we have $X_j \sim \text{Geom}(\frac{n-j}{n})$. Let Y_m be the number of superpolynomials required to recover k secret bits, then the expected value of Y_k is given by

$$\mathbb{E}[Y_k] = \sum_{j=0}^{k} \mathbb{E}[X_j] = \sum_{j=0}^{k} \frac{1 - \frac{n-j}{n}}{\frac{n-j}{n}} = \sum_{j=0}^{k} \frac{j}{n-j}.$$

As seen in figure 8 this model does not exactly match the actual outcome. This is probably due to the model not fully accounting for the method in which variables are recovered. We may find many linear equations of which the solutions depend on a single variable which is yet to be found. Once this variable is found multiple variables are recovered at once. Further research would be required to divise a better model for this behavior. ‘

5 State periodicity of Keccak-f[b]

An important factor which influences the security of sponge functions, is the number of bits which can be output by the sponge function before the output starts repeating itself. We call a fixed-length permutation (or transformation) function f and an initial state s periodic if

$$f^n(s) = s$$

for some finite value of n. In which $f^n(s)$ denotes the composition of f with itself n times. We define the *period* of f for s as the smallest n for which the equation given above holds. If after the absorbing phase we are left with a periodic state s with period n, then the output in the squeezing phase will start repeating itself after $b \cdot n$ bits.

In general every state of the sponge function must be either periodic or become periodic within a finite number of steps. This is due to the state having a fixed length of b bits, limiting the possible number of states to 2^b. This also implies that any sponge function can output at most $r \cdot 2^b$ unique bits in the squeezing phase. However in most cases this number will be much lower.

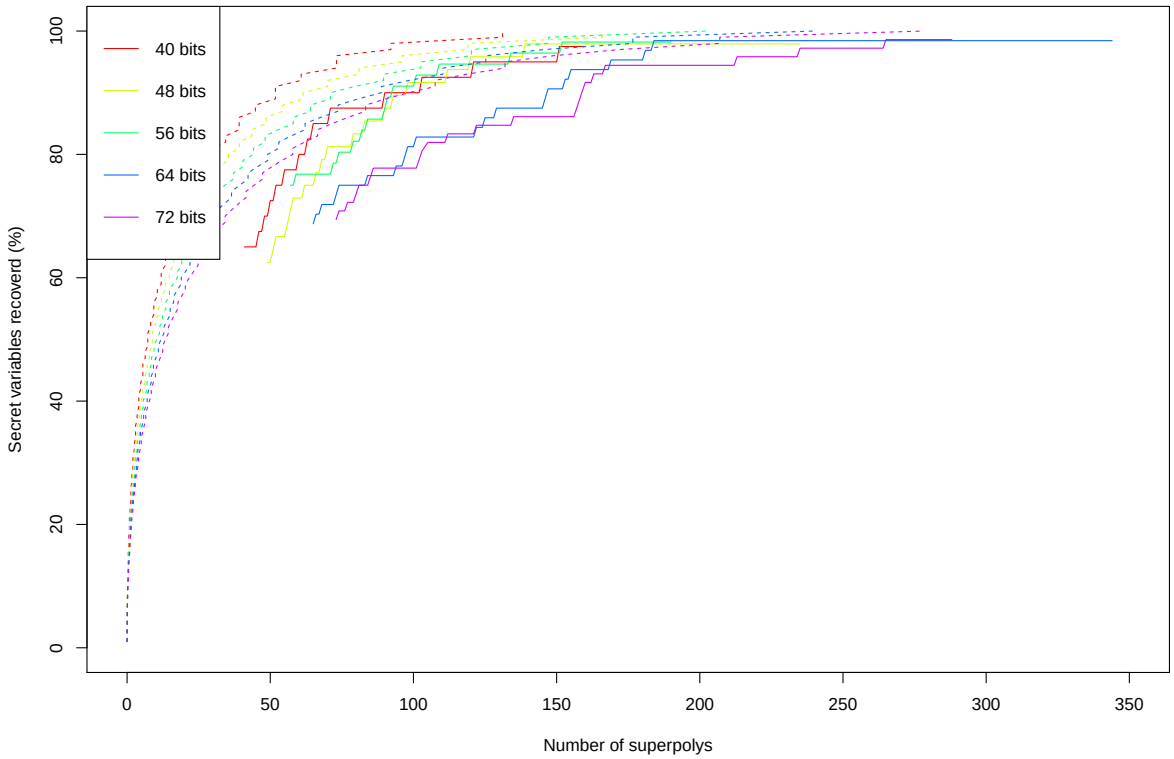

Figure 8: The number of superpolys required to recover a certain percentage of the secret given secrets of various lengths. The actual results are given by the solid lines and the expected values of Y_k are given by the dashed lines.

.

Now we consider the permutation function KECCAK-f[b] which is used in KECCAK's sponge construction. We attempt to find the periods of the different states of this function. To do this we construct a graph in which every all states of length b are represented by vertices. We draw a directed edge from a vertex u to a vertex v if $f(u) = v$. Each edge has outdegree 1 and since KECCAK-f[b] is a permutation function, each edge also has indegree 1. After constructing this graph we search for all disjunct cycles in this graph.

We performed this search for KECCAK-f[b] using $b = 25$, finding all periods took aproximately 5 minutes on a standard desktop computer. Doing this computation for larger values of b was not feasible however. Since b must be a multiple of 25, the next step is $b = 50$, which would require 2^{50} computations of KECCAK-f[b], requiring aproximately $2.8 \cdot 10^6$ CPU hours, unless we are able to make some significant performance improvements. The C code used to perform these computations is given in appendix C. The results of these computations are given in Table 2. We observe that KECCAK-f[25] has 12 cyclic groups, which, when joined together are equal to the entire state space $\mathbb{Z}_2^b$. The two largests groups together account for aproximately 94% of all possible states. Therefore with high probability the output of the sponge function will have a large period. However in some rare cases, the sponge function will have a highly periodic output.

To guarantee the security of KECCAK, its permutation function KECCAK $-f[b]$ must behave like a random permutation. Based on a few statistical properties of random permutations we can verify that this is the case.

Total number of cycles

First we consider the number of different permutations of length n in which some state s is part of a cycle of length k. We can pick $k-1$ additional states to be part of the cycle from $n-1$ different states. Leading to $\binom{n-1}{k-1}$ combinations. Assuming s is the "first" state in the cycle, we can order the remaining

Initial state (s)	Period
000000000000000000000000	13104259
000000000000000000000001	18447749
000000000000000000011110	1811878
000000000000000001011010110	147821
000000000000000001111011000	40365
000000000011000100011100	2134
000000001000010000010100	168
000000010100011101111110	27
000101110011001010100011	2
000110101101110010111011	12
001101110111110001010011	3
010011011011011110111100	14

Table 2: The periods of KECCAK-f[25] and a corresponding generator.

$k-1$ states in $(k-1)$ different ways. Finally we can order the other $n-k$ numbers in $(n-k)!$ different ways. This means that the total number of different combinations in which s is part of a cycle of length k is

$$\binom{n-1}{k-1}(k-1)!(n-k)! = (n-1)!.$$

Since there are $n!$ possible permutations, the probability of having a cycle of length k containing s is $\frac{1}{n}$. This means that the lengths of the cycles are uniformly distributed. Next the expected number of permutations of length k is given by

$$\frac{1}{k}\sum_{1}^{n}\frac{1}{n} = \frac{1}{k}.$$

Finally the total expected number of cycles for a permutation of length n is given by

$$\sum_{k=1}^{n}\frac{1}{k} = H_n \approx \ln(n). \tag{1}$$

For KECCAK-f[25] the number of expected cycles therefore is approximately 17. In our expirment we found 12 cycles. If we assume KECCAK-f[b] behaves the same for larger values of b then we can make the following prediction for KECCAK-f[1600]. We would expect to have aproximately $ln(2^{1600}) \approx 1109$ cycles.

Cycle length

Let X denote the length of a cycle containing some state s. As seen in the previous part $\mathbb{P}(X=k) = \frac{1}{n}$ for a random permutation. Using this probability we can compute the expected cycle length as follows.

$$\mathbb{E}[X] = \sum_{k=1}^{n}\frac{1}{n}\cdot k = \frac{n+1}{2}$$

For KECCAK-f[25] the expected cycle length would be aproximately $1.68 \cdot 10^7$. By summing the squares of the cycle lengths found in our experiment an dividing them by $n = 2^b$ we find the actual mean cycle length. We found it to be $1.55 \cdot 10^7$. Therefore we can conclude that the permutation KECCAK-f[25] behaves very close to a random permutation. If we once again assume KECCAK-f[b] behaves the same for larger values of b then we can make the following prediction for KECCAK-f[1600]. We would expect the mean cycle length to be $\frac{2^{1600}+1}{2} \approx 2.22 \cdot 10^{481}$.

6 Conclusions and discussion

We found that KECCAK proves to be secure when implemented correctly for one of its many applications. Multiple different types of attacks on KECCAK only are practical for a number of rounds significanly less than the full 24 rounds. When studying the periodic behavior of KECCAK-f[25] no oddities were found and the function behaves closely to a random permutation. However this does raise a big question; do the results found for KECCAK-f[25] also apply to KECCAK-f[1600]? There is currently no method to experimentally verify this, however the KECCAK-f[1600] permutation is very similar to KECCAK-f[25].

The cube attack was studied in more detail and proved to be succesful for a small number of rounds. However it does not appear to be a very practical attack agains KECCAK for the following two reasons:

- The running time is exponential in the degree of the cryptographic function being attacked. The running time of KECCAK is exponential in the number of rounds, leading to a very large overal running time.
- It requires a large amount of queries in the online phase. This large amount of queries may be feasible in some circumstances, e.g. a hardware implementation of KECCAK to which the attacker has access. However in many cases it is not feasible.

For this reason we think the cube attack will not pose any risk for KECCAK in the near future.

References

[1] Nasour Bagheri, Navid Ghaedi, and Somitra Kumar Sanadhya. "Differential fault analysis of SHA-3". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 9462. Springer, Cham, Dec. 2015, pp. 253–269.

[2] Mihir Bellare and Phillip Rogaway. *The exact security of digital signatures-How to sign with RSA and Rabin*. Vol. 1070. Springer-Verlag, 1996, pp. 399–416. arXiv: 9780201398298.

[3] Guido Bertoni et al. *Cryptographic sponge functions*. 2011. URL: http://sponge.noekeon.org/CSF-0.1.pdf (visited on 02/26/2017).

[4] Guido Bertoni et al. *The Keccak reference*. 2011. URL: http://keccak.noekeon.org/Keccak-reference-3.0.pdf.

[5] Itai Dinur, Orr Dunkelman, and Adi Shamir. "Collision Attacks on Up to 5 Rounds of SHA-3 Using Generalized Internal Differentials". In: *FSE*. 2013, pp. 219–240. URL: https://eprint.iacr.org/2012/672.pdf.

[6] Itai Dinur, Orr Dunkelman, and Adi Shamir. "New attacks on Keccak-224 and Keccak-256". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 7549 LNCS. 2012, pp. 442–461.

[7] Itai Dinur and Adi Shamir. "Cube Attacks on Tweakable Black Box Polynomials". In: *EUROCRYPT 2009*. Vol. 2. 2009, pp. 278–299.

[8] Itai Dinur et al. "Cube attacks and cube-attack-like cryptanalysis on the round-reduced Keccak sponge function". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 9056. 2015, pp. 733–761.

[9] Senyang Huang et al. "Conditional cube attack on reduced-round Keccak sponge function". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 10211 LNCS. Springer, Cham, Apr. 2017, pp. 259–288.

[10] Pei Luo et al. "Algebraic Fault Analysis of SHA-3." In: *IACR Cryptology ePrint Archive* 2017 (2017), p. 113.

[11] Pei Luo et al. "Differential Fault Analysis of SHA3-224 and SHA3-256". In: *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, Aug. 2016, pp. 4–15.

[12] Pawel Morawiecki. "Malicious Keccak." In: *IACR Cryptology ePrint Archive* 2015 (2015), p. 1085.

[13] Paweł Morawiecki, Josef Pieprzyk, and Marian Srebrny. "Rotational cryptanalysis of round-reduced Keccak". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 8424 LNCS. 2014, pp. 241–262.

[14] National Institue of Standards and Technology. *FIPS PUB 202 SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. 2015.

[15] Kexin Qiao et al. “New collision attacks on round-reduced KECCAK”. In: *EUROCRYPT 2017*. Vol. 10212 LNCS. 2017, pp. 216–243.

[16] Dhiman Saha, Sukhendu Kuila, and Dipanwita Roy Chowdhury. “SymSum: Symmetric-Sum Distinguishers Against Round Reduced SHA3.” In: *IACR Trans. Symmetric Cryptol.* 2017.1 (2017), pp. 240–258.

A Source code - Cube attack

```
#define SECRETBYTES 16
#define TWEAKBYTES 16
#define TWEAKVARS 7 // 2**rounds - 1
#define OUTPUTBYTES 16
#define EQUATIONS 1024 // Size of the equations buffer
/*
#define TESTING 0
#define VERBOSE 0*/

// http://www.mathcs.emory.edu/~cheung/Courses/255/Syllabus/1-C-intro/bit-array.html
#define SetBit(A,k)     ( A[(k/8)] |= (1 << (k%8)) )
#define ClearBit(A,k)   ( A[(k/8)] &= ~(1 << (k%8)) )
#define GetBit(A,k)     ( (A[(k/8)] & (1 << (k%8))) ? 1 : 0 )
#define SetZero(A)      ( memset(A, 0, sizeof(A)) )

void printByteArray(unsigned char A[], int len) {
printf("[%i", GetBit(A, 0));
for (int i = 1; i < len*8; i++) {
printf(", %i", GetBit(A, i));
}
printf("]\n");
}

void printArray(unsigned char C[], int len, int zeroesAsSpaces) {
printf("[ %s", C[0] ? "1" :  (zeroesAsSpaces ? " " : "0"));
for (int i = 1; i < len; i++) {
if (i % (8*SECRETBYTES) == 0)
printf("\n");
printf(", %s", C[i] ? "1" : (zeroesAsSpaces ? " " : "0"));
}
printf("]\n\n");
}

/*
Gauss elimination
*/
#define mat_elem(a, y, x) (a + ((y) * (8 * SECRETBYTES) + (x)))

void swap_row(unsigned char *a, int r1, int r2)
{
unsigned char tmp, *p1, *p2;
int i;

if (r1 == r2) return;
for (i = 0; i < 8 * SECRETBYTES; i++) {
p1 = mat_elem(a, r1, i);
p2 = mat_elem(a, r2, i);
tmp = *p1, *p1 = *p2, *p2 = tmp;
}
}

void gauss_eliminate(unsigned char *eqns, int eqnsCount)
{
#define A(y, x) (*mat_elem(eqns, y, x))
int i, j, col, row, max_row,dia;
double max, tmp;

for (dia = 0; dia < 8 * SECRETBYTES; dia++) {
max_row = dia, max = A(dia, dia);

for (row = dia + 1; row < eqnsCount; row++)
if ((tmp = (A(row, dia))) > max)
```

```
63 max_row = row, max = tmp;
64
65 swap_row(eqns, dia, max_row);
66
67 for (row = dia + 1; row < eqnsCount; row++) {
68 tmp = 1;
69 for (col = dia+1; col < 8 * SECRETBYTES; col++)
70 {   A(row, col) -= tmp * A(dia, col);
71 A(row, col) %= 2;
72 }
73 A(row, dia) = 0;
74 }
75 }
76 #undef A
77
78 }
79 /**
80 Returns the number of variables for which we can find a solution
81 */
82 int hasUniqueSolution(unsigned char *eqns, int eqnsCount) {
83 #define A(y, x) (*mat_elem(eqns, y, x))
84 if (eqnsCount < 8 * SECRETBYTES) {
85 return 0;
86 }
87 int row, col, varCount;
88 varCount = 0;
89 for (row = 0; row < 8 * SECRETBYTES; row++) {
90 if (A(row,row) != 1) {
91 continue;
92 }
93 varCount++;
94 for (col = 0; col < row; col++) {
95 if (A(row, col) != 0) {
96 varCount--;
97 break;
98 }
99 }
100 }
101 return varCount;
102
103 #undef A
104 }
105
106 /**
107 Computes the output of keccak for bits x and v of length SECRETBYTES and TWEAKBYTES
108 */
109 void computePoly(unsigned char x[], unsigned char v[], unsigned char *out) {
110 unsigned char input[SECRETBYTES+TWEAKBYTES];
111 memcpy(input, x, SECRETBYTES);
112 memcpy(&input[SECRETBYTES], v, TWEAKBYTES);
113 Keccak(576, 1024, input, SECRETBYTES+TWEAKBYTES, 0x06, out, OUTPUTBYTES);
114 }
115
116 //FIXME: This only works for TWEAKVARS up to 32
117 void sumCube(unsigned char x[], unsigned int C[], unsigned char *out) {
118 unsigned char v[TWEAKBYTES];
119 //memset(v,0,sizeof(v));
120 // Our cube consists of 2**TWEAKVARS combinations
121 for (int i = 0; i < (1<<TWEAKVARS); i++) {
122 //Next shift the bits in i to the correct positions in v
123 for (int j = 0; j < TWEAKVARS; j++) {
124 if (i & (1 << j)) {
125 SetBit(v, C[j]);
126 } else {
127 ClearBit(v, C[j]);
128 }
129 }
130 unsigned char newOutput[OUTPUTBYTES];
131 computePoly(x, v, newOutput);
132 for (int k = 0; k < OUTPUTBYTES; k++)
133 out[k] ^= newOutput[k];
134
135 }
```

```
136
137 }
138
139 int isLinear(unsigned int C[]) {
140 unsigned char zero[SECRETBYTES], x[SECRETBYTES], y[SECRETBYTES], xy[SECRETBYTES];
141 SetZero(zero);
142
143 for (int j =0; j< 50; j++) {
144
145 for (int i = 0; i< SECRETBYTES; i++) {
146 x[i] = (unsigned char) (rand() % 256);
147 y[i] = (unsigned char) (rand() % 256);
148 }
149 unsigned char sum1[OUTPUTBYTES];
150 unsigned char sum2[OUTPUTBYTES];
151 SetZero(sum1);
152 SetZero(sum2);
153 sumCube(zero, C, sum1);
154 sumCube(x, C, sum1);
155 sumCube(y, C, sum1);
156 for (int i = 0; i< SECRETBYTES; i++)
157 xy[i] = x[i] ^ y[i];
158 sumCube(xy, C, sum2);
159 if (strcmp(sum1, sum2)) {
160 printByteArray(sum1, OUTPUTBYTES);
161 printByteArray(sum2, OUTPUTBYTES);
162 }
163
164 }
165 return 1;
166 }
167
168 int cubeAttack(unsigned int C[], unsigned char secretBytes[], unsigned char *eqns,
    unsigned char *eqnsCopy, unsigned char *eqnsValues, int eqnsCount) {
169 // Offline phase
170 unsigned char x[SECRETBYTES];
171 SetZero(x);
172 unsigned char constants[OUTPUTBYTES];
173 SetZero(constants);
174 sumCube(x, C, constants);
175 unsigned char coefs[SECRETBYTES*8][OUTPUTBYTES];
176 for (int i = 0; i < SECRETBYTES*8; i++) {
177 SetZero(coefs[i]);
178 SetBit(x, i);
179 sumCube(x, C, coefs[i]);
180 ClearBit(x, i);
181 }
182
183 // Online phase
184 unsigned char outputBytes[OUTPUTBYTES];
185 SetZero(outputBytes);
186 sumCube(secretBytes, C, outputBytes);
187
188 // Construct equations using superpolys from offline phase and results from the online
    phase
189 for (int i = 0; i < OUTPUTBYTES*8; i++) {
190 int termCount = 0;
191 int hasConstant = 0;
192 int printedConstant = 0;
193 int sum = 0;
194 if (GetBit(constants, i)) {
195 hasConstant++;
196 sum++;
197 }
198
199 for (int j = 0; j < SECRETBYTES*8; j++) {
200 eqns[eqnsCount * 8 * SECRETBYTES + j] = 0;
201 eqnsCopy[eqnsCount * 8 * SECRETBYTES + j] = 0;
202 if ((hasConstant && !GetBit(coefs[j], i)) || (!hasConstant && GetBit(coefs[j], i))) {
203 #ifdef VERBOSE
204 if (hasConstant && !printedConstant) {
205 printedConstant = 1;
206 printf("1 ");
```

```
}
printf("+ x_%i ", j);

#endif
#ifdef TESTING
sum += GetBit(secretBytes, j);
#endif

eqns[eqnsCount * 8 * SECRETBYTES + j] = 1;
eqnsCopy[eqnsCount * 8 * SECRETBYTES + j] = 1;
termCount++;
}
}
#ifdef VERBOSE
// We are only interested in nonzero and nonconstant polynomials
if (termCount > 0) {
printf("= %i\n", GetBit(outputBytes, i));
}
#endif
#ifdef TESTING
if (sum % 2 != GetBit(outputBytes, i))
printf("Bad Polynomial!\n");
#endif

if (termCount > 0 && termCount < 8) {
eqnsValues[eqnsCount] = (GetBit(outputBytes, i) + hasConstant) % 2;
eqnsCount++;
}
}
return eqnsCount;
}

void genCube(unsigned int *C, int n)
{
unsigned int numbers[TWEAKBYTES*8];
for (int i = 0; i < TWEAKBYTES *8; i++) {
numbers[i]=i;
}
// Shuffle the array randomly
for (int k = 0; k < TWEAKBYTES*8 - 1; k++)
{
int j = k + rand() / (RAND_MAX / (TWEAKBYTES*8 - k) + 1);
int t = numbers[j];
numbers[j] = numbers[k];
numbers[k] = t;
}
// Fill C with the required amont of numbers
for (int i = 0; i < n; i++) {
C[i] = numbers[i];
}

}

void timestamp()
{
time_t ltime; /* calendar time */
ltime=time(NULL); /* get current cal time */
printf("%s",asctime( localtime(&ltime) ) );
}

int main(int argc, const char * argv[]) {
unsigned char secretBytes[SECRETBYTES] = "TopSecretMessage";
#ifdef VERBOSE
printByteArray(secretBytes, SECRETBYTES);
#endif
// Create a cube
unsigned int C[TWEAKVARS] = {24, 30, 101, 117, 49, 16, 75};

// Create array for storing equations
int eqnsCount = 0;
unsigned char eqns[SECRETBYTES * 8 * (EQUATIONS + 1)];
SetZero(eqns);
unsigned char eqnsCopy[SECRETBYTES * 8 * (EQUATIONS + 1)];
```

```
SetZero(eqnsCopy);
unsigned char eqnsValues[EQUATIONS + 1];
SetZero(eqnsValues);
int tmp = 0;
int varsFound = 0;
// Find some equations
timestamp();
while (varsFound < SECRETBYTES * 8)
{
eqnsCount = cubeAttack(C, secretBytes, eqns, eqnsCopy, eqnsValues, eqnsCount);
gauss_eliminate(eqns, eqnsCount);
if (eqnsCount > tmp)
{
tmp = eqnsCount;
printf("%i, %i\n", tmp, varsFound);
#ifdef VERBOSE
printArray(eqns, SECRETBYTES * 8 * SECRETBYTES * 8, 1);
#endif
}
genCube(C, TWEAKVARS);
varsFound = hasUniqueSolution(eqns,eqnsCount);
}
timestamp();
// Print results which we can feed in to sage for solving
printArray(eqnsCopy, SECRETBYTES * 8 * (eqnsCount), 0);
printf("VALUES:\n");
printArray(eqnsValues, eqnsCount, 0);
return 0;
}
```

B Cube attack equations

Below a subset of the linear equations used in the attack is given

```
  x_111 = 0
  x_113 = 1
  1 + x_120 = 0
  1 + x_88 = 0
  x_70 + x_99 = 1
  1 + x_108 + x_121 = 1
  1 + x_35 = 1
  x_96 = 1
  1 + x_120 = 0
  1 + x_121 = 1
  1 + x_127 = 1
  x_92 = 1
  x_117 = 1
  1 + x_35 + x_98 = 1
  1 + x_90 = 1
  1 + x_113 = 0
  1 + x_52 + x_115 = 0
  1 + x_95 + x_111 = 1
  1 + x_18 + x_111 = 1
  1 + x_74 = 0
  x_102 = 1
  1 + x_78 = 0
  1 + x_13 + x_77 = 0
  1 + x_23 = 1
  x_121 = 0
  1 + x_91 = 1
  x_75 + x_97 = 0
  1 + x_5 = 1
  x_18 = 0
  x_125 = 1
  1 + x_123 = 1
  x_114 = 1
  1 + x_69 = 0
  1 + x_117 = 0
  1 + x_67 = 1
  1 + x_45 = 0
  1 + x_100 = 0
  1 + x_73 = 1
```

```
  1 + x_3 + x_66 = 0
  x_29 = 0
  x_34 = 1
  x_54 = 1
  1 + x_61 + x_124 = 0
  1 + x_45 = 0
  1 + x_15 = 1
  x_54 = 1
  1 + x_86 = 0
  1 + x_41 = 0
  1 + x_109 = 0
  1 + x_35 = 1

  ...

  1 + x_122 = 0
  1 + x_63 + x_110 + x_127 = 0
```

C Source code - period finding

```
void int_to_state(uint64_t in, int *out)
{
uint64_t mask = 1U << (2*laneSize-1);
for (int i = 0; i < 25*laneSize; i++) {
out[i] = (in & mask) ? 1 : 0;
in <<= 1;
}
}

uint64_t state_to_int(int *in)
{
uint64_t out = 0;
int pow = 1;
for (int i = 0; i < laneSize * 25; i++) {
out += pow * in[laneSize * 25 - 1 - i];
pow <<= 1;
}
return out;
}

int main(int argc, const char * argv[]) {
uint64_t i;
uint64_t rounds = 33554432;
uint8_t *states_visited;
states_visited = (uint8_t *)malloc(sizeof(uint8_t)*rounds);
for(i = 0; i < rounds; i++) {
states_visited[i] = 0;
}
int state[25];
memset(state, 0, sizeof(state));
int visiting = 1;
uint64_t current_state = random() % rounds;
uint64_t start_state = current_state;
int_to_state(start_state, state);
int period = 0;
while (visiting) {
while (1) {
//    printf("%llu\n", current_state);
period++;
states_visited[current_state] = 1;
KeccakF25_StatePermute(state);
current_state = state_to_int(state);
if (current_state == start_state) {
for (int y = 0; y < 5; y++) {
for (int x = 0; x < 5; x++) {
printf("%x", readLane(x, y));
}
}
printf(" & ");

```

```
printf("%i\\\\\n", period);
break;
}
if (states_visited[current_state] == 1) {
printf("breaking, we've already been here\n");
break;
}
}
// Find next starting state
visiting = 0;
int j;
for (j = rand() % rounds; j < 2*rounds; j++) {
current_state = j % rounds;
if (states_visited[current_state] == 0) {
visiting = 1;
start_state = current_state;
int_to_state(current_state, state);
break;
}
}
printf("Moving on to state %llu\n", current_state);

}
return 0;
}
```