S C I E N C E ▪ P A S S I O N ▪ T E C H N O L O G Y

Christian Pasero

Efficiency of SAT Solver Attacks on NIST Lightweight Hash Candidates

BACHELOR'S THESIS

Bachelor's degree programme: Computer Science

Supervisor

Maria Eichlseder

Institute of Applied Information Processing and Communications
Graz University of Technology

Graz, July 2022

Contents

Abstract

NIST is currently running a competition since March 2019 for new lightweight cryptographic standards for hashing and authenticated encryption. Out of the 57 submissions, 10 were chosen to compete in the final round. ASCON and ROMULUS are two out of the ten finalists which follow a different approach for the hashing algorithm. While ASCON-XOF uses a sponge construction, ROMULUS-H uses a combination of Hirose's DBL compression function and Merkle-Damgård mode of operation.

A hash function should have two essential security properties: collision resistance and, the focus of this paper, pre-image resistance. The brute-force attack is the most intuitive attack on a hash function. However, it is still not clear if it is the fastest or the most efficient attack for every cipher. Therefore we compare a SAT-Solver attack to the brute-force attack. This paper focuses on the two hash functions, ASCON-XOF and ROMULUS-H. Additionally, we provide results on the other hash candidates in the NIST lightweight cryptography competition. If the SAT-Solver attack finds a pre-image in a shorter time than the brute force attack, we found a vulnerability.

The SAT-Solver was more efficient than the brute-force attack when reducing the permutation rounds. While for the brute-force attack everything above 32 bits was infeasible to solve in our implementation, the SAT-Solver was able to find a pre-image almost immeditely (when reducing the permutation rounds of ASCON-XOF to 2 out of 12 and for ROMULUS-H to 6 out of 40).

Keywords: hash function · SAT-Solver · pre-image resistance · brute-force

1 Introduction

Cryptographic hash functions play a significant role in security nowadays. Each hash function has a different approach and follows a unique digesting procedure. While some hash functions were already broken [WYY05], others appear to have a large security margin and are widely used [Dwo15]. Cryptographic hash functions are used in password-hashing (e.g. Argon2 [Wet16]), cryptographic digital signatures (e.g. SHA-3 [Dwo15]), mining bitcoin blocks (e.g. SHA-2 [Nat15]), and a lot more. The purpose of a cryptographic hash function is to map any input of arbitrary size to a fixed-length hash value. A very tiny change of the input changes the hash value completely. Therefore it is infeasible to change the input in a certain way such that the hash value remains unchanged. A cryptographic hash function is a one-way digest algorithm, so if the hash value gets leaked, there is no feasible way to recover the input, or the so called pre-image.

NIST is currently running a competition since March 2019 for new lightweight cryptographic standards for hashing and authenticated encryption [CN]. Out of the 57 submissions, 10 were chosen to compete in the final round. Ascon [DEMS21] and Romulus [GIK+21] are two out of the ten finalists which follow a different approach for the hashing algorithm. While Ascon-Xof uses a sponge construction [BDPA08], Romulus-H uses a combination of Hirose's DBL compression function [Hir06] and Merkle-Damgård [Mer79] mode of operation.

There are different ways to find a pre-image of a hash value. The most intuitive one is the brute-force attack. A vast amount of inputs in a specific range gets digested into its hash value and compared with the target hash value. These attacks are very unpractical because of the high bit-rate, but are still in use since no other attack has been faster yet. This research will clarify if a SAT-Solver attack is faster than the brute-force attack for Ascon-Xof and Romulus-H. Furthermore we will show to which extent the SAT-Solver attack is capable of finding a pre-image in a reasonable time by taking in consideration various factors, including permutation round reduction or shortening the hash length.

SAT-Solvers have achieved remarkable progress in the last years. Different competitions are running every year to find a new and better way of checking the satisfiability of a boolean equation. In this paper, we are using the z3 SAT-Solver from Microsoft Research [Nik13].

We use the attacks on Ascon-Xof and Romulus-H, two competitors of the NIST competition for a new lightweight cryptographic standard for hashing. After attacking the hash functions, we will compare the time consumption of the SAT-Solver attack for every tested input and illustrate its performance for several permutation rounds. We observe that Ascon-Xof is not secure up to 2 permutation rounds (when exploiting with a SAT-Solver attack). From 3 rounds up to 12 rounds the SAT-Solver stands no chance and Ascon-Xof turns out to be secure under our restrictions. For Romulus-H the critical round number is 6, which means that the permutation rounds are not as efficient as the permutation rounds of Ascon-Xof. This is compensated by the designers, by setting the permutation rounds of Romulus-H to 40 while Ascon-Xof only has 12.

Then we compare the brute-force attack to the SAT-Solver attack to check if the SAT-Solver attack finds a solution in a shorter time than the brute force attack. The SAT-Solver attack is more efficient when the number of rounds are low (for Ascon-Xof 3 rounds and for Romulus-H 8 rounds). On the other hand, the brute-force attack does not mainly depend on the permutation rounds, but on the hash length. So for both cryptographic hash algorithms the hash output is truncated for comparison to 32 bits.

Last but not least, we push the SAT-Solver to its limits. While the brute-force attack is not solvable in under 24 hours, with a pre-image length of 64 bits and a 32-bit hash output length, we can increase the hash output length of the SAT-Solver to 256 bits (for Ascon-Xof) or 320 bits (for Romulus-H). Ascon-Xof will be secure to up to 2 permutation rounds on 256 bits, but 3 rounds turn out to be infeasible to solve in under 24 hours. For Romulus-H we still find a preimage on 6 rounds with 320 bits. Everything above 6 rounds will take the SAT-Solver more than a day to solve.

Outline. Chapter 2 explains the background of the cryptographic hashing algorithms. In chapter 3, we present the implementation of our attack and how we evaluate the results. Chapter 4 describes the testing environment and which decisions we made on which assumptions. In chapter 5 we talk about the application of both attacks and visualize the results. Finally, in chapter 6, we discuss the results.

2 Background

2.1 Hash functions

A hash function maps a message of arbitrary length to a fixed-size value. The input message is the so-called pre-image, and the output is the hash value. Furthermore, a hash function should have two security properties: collision resistance and pre-image resistance. A hash function is collision resistant, if it is hard to find two inputs which map to the same output. A collision occurs when two different pre-images map to the same hash value. Since the hash value is a one-way function, one can not reverse a hash function and get the pre-image out of the hash value. The pre-image resistance is split into pre-image and second pre-image resistance. If a hash function is pre-image resistant, it is infeasible to find any input which hashes to a target output. If a hash function is second pre-image resistant it is infeasible to find a second input which maps to the same output as the target input.

2.2 Notations

In this paper, we will use the $\oplus$ symbol for the logical XOR gate. The $\odot$ symbol is used for the logical AND gate, and the $\ominus$ symbol for the logical NOR gate.

x_1	x_2	$\oplus$
0	0	0
0	1	1
1	0	1
1	1	0

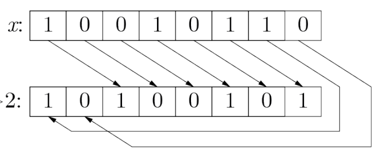
(a) XOR

x_1	x_2	$\odot$
0	0	0
0	1	0
1	0	0
1	1	1

(b) AND

x_1	x_2	$\ominus$
0	0	1
0	1	0
1	0	0
1	1	0

(c) NOR



(d) Right shift

Table 2.2: Visualisation of bit transformations

These permutations are not restricted to booleans, but can also be applied on word-oriented bitvectors. If we concatenate two values, we will use the || symbol where the second variable (on the right side of the symbol) gets attached to the end of the first variable. For arithmetic addition we will use the $\boxplus$ sign.

For circular shifting, we use the $\ggg$ symbol. Each bit gets rotated to the right. The last bit at the right gets placed at the first bit position on the left. If we want to shift the following 8 bits to the right by 2, we write $x \ggg 2$:

2.3 Ascon hash function

Ascon-Xof [DEMS21] is based on a sponge construction [BDPA08]. Three phases divide the sponge construction, starting with the initialization phase, where the predefined initialisation vector (IV) and the internal state (IS) are initialised. After the Initialization phase follows the absorb message phase, where the input message M is absorbed in blocks and put through a permutation. Finally, the squeezing phase produces the hash value.

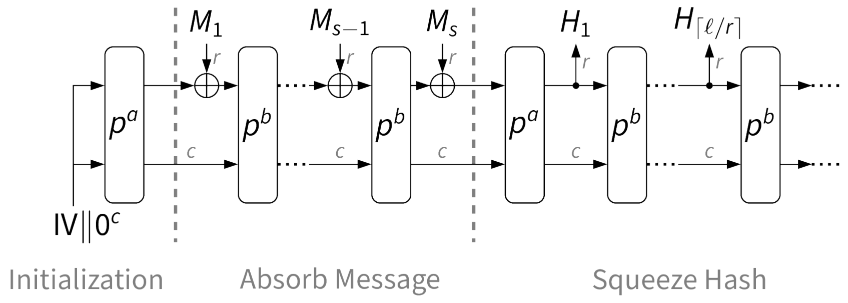


Figure 2.2: Ascon-Xof's sponge construction [BDPA08]

2.3.1 Initialization phase

Ascon uses a different IV for each hashing algorithm, namely Ascon-Hash, Ascon-Hasha, Ascon-Xof and Ascon-Xofa. The Ascon-Hash hashing algorithm produces a 256-bit hash with 12 permutation rounds for p^a and p^b. Ascon-Hasha reduces the permutation rounds for p^b from 12 to 8. The extandable output functions (XOF) Ascon-Xof and Ascon-Xofa produce a hash of arbitrary length and operate on 12 permutation rounds for p^a. For p^b Ascon-Xof uses 12 rounds and Ascon-Xofa uses 8 rounds. In our testing environment we use the Ascon-Xof algorithm. The IVs initialize the cipher's internal state as shown in Figure 2.2 and can be precomputed:

$$
\mathrm{S} \leftarrow \begin{cases}
\begin{array}{ll}
\texttt{ee9398aadb67f03d} \,|| & \\
\texttt{8bb21831c60f1002} \,|| & \\
\texttt{b48a92db98d5da62} \,|| & \textsc{Ascon-Hash}, \\
\texttt{43189921b8f8e3e8} \,|| & \\
\texttt{348fa5c9d525e140} &
\end{array}
&
\begin{array}{ll}
\texttt{b57e273b814cd416} \,|| & \\
\texttt{2b51042562ae2420} \,|| & \\
\texttt{66a3a7768ddf2218} \,|| & \textsc{Ascon-Xof} \\
\texttt{5aad0a7a8153650c} \,|| & \\
\texttt{4f3e0e32539493b6} &
\end{array}
\\[1em]
\begin{array}{ll}
\texttt{01470194fc6528a6} \,|| & \\
\texttt{738ec38ac0adffa7} \,|| & \\
\texttt{2ec8e3296c76384c} \,|| & \textsc{Ascon-Hasha}, \\
\texttt{d6f6a54d7f52377d} \,|| & \\
\texttt{a13c42a223be8d87} &
\end{array}
&
\begin{array}{ll}
\texttt{44906568b77b9832} \,|| & \\
\texttt{cd8d6cae53455532} \,|| & \\
\texttt{f7b5212756422129} \,|| & \textsc{Ascon-Xofa} \\
\texttt{246885e1de0d225b} \,|| & \\
\texttt{a8cb5ce33449973f} &
\end{array}
\end{cases}
$$

2.3.2 Message absorption phase

The message absorption phase consists of XORing the outer part r (rate) to the previous output with the M_i and creating together with the inner part c (capacity) a 320-bit block divided into five 64-bit words $S = x_0||x_1||x_2||x_3||x_4$. The last message block M_n is padded, by appending a single 1 and the smallest number of 0s such that the length is equal to 320 bits. After that, the round function p^b is applied:

$$x_0 \leftarrow x_0' \oplus M_x$$
$$x_1||x_2||x_3||x_4 \leftarrow x_1'||x_2'||x_3'||x_4'$$

2.3.3 Round function

The permutation rounds consists of three layers. The first layer is the so-called ConstantAdditionLayer, followed by the SubstitutionLayer and the LinearLayer.

ConstantAdditionLayer: We add a hardcoded constant c_r to x_2. The constant c_r depends on the current round r and is updated before the permutation:

$$x_2 \leftarrow x_2' \oplus c_r$$

SubstitutionLayer: In this layer we perform a 5-bit S-box. The S-box is defined in Figure 2.3.

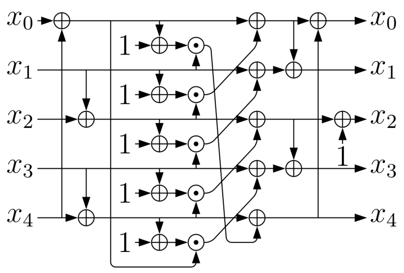


Figure 2.3: Ascon-Xof's S-box

LinearLayer: This layer applies a linear function for each row, defined in Figure 2.4.

$$\begin{aligned}
x_0 &\leftarrow x_0' \oplus (x_0' \ggg 19) \oplus (x_0' \ggg 28)\\
x_1 &\leftarrow x_1' \oplus (x_1' \ggg 61) \oplus (x_1' \ggg 39)\\
x_2 &\leftarrow x_2' \oplus (x_2' \ggg 1) \oplus (x_2' \ggg 6)\\
x_3 &\leftarrow x_3' \oplus (x_3' \ggg 10) \oplus (x_3' \ggg 17)\\
x_4 &\leftarrow x_4' \oplus (x_4' \ggg 7) \oplus (x_4' \ggg 41)
\end{aligned}$$

Figure 2.4: Ascon-Xof's linear layer p_l with 64-bit function

2.3.4 Squeezing phase

In the squeezing phase, the hash gets extracted in several rounds. Each consists of extracting a fixed amount of bits and performing a permutation once again. The first permutation of the the squeezing phase is a p^a permutation, the following are p^b permutations.

2.4 Romulus hash function

Romulus-H [GIK+21] is based on a Merkle-Damgård construction. Furthermore, it is part of the Skinny family [BJK+16] and follows the Tweakey framework [JNP14]. Three main sections divide Romulus-H: the initialization phase, the compression phase, and the finalizing phase.

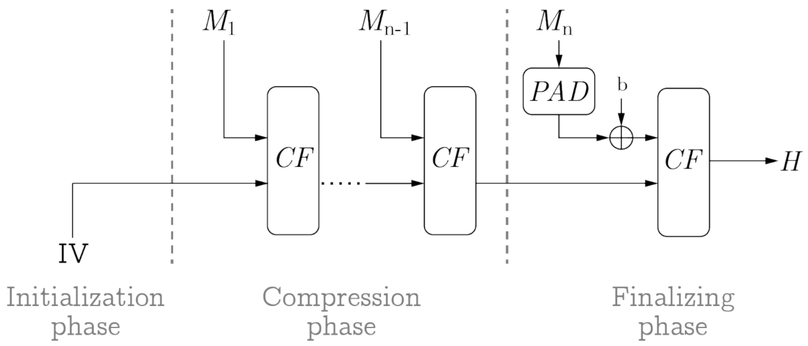


Figure 2.5: Romulus-H's Merkle-Damgård construction

2.4.1 Compression phase

The input of the compression function is 512 bits, composed of the padded message M (256 bits), L (128 bits) and R (128 bits). Then two separated round functions of 40 rounds are executed, producing two outputs, i.e., L' (128 bits) and R' (128 bits), which are concatenated and forwarded to the next compression function block. The compression function of the Compression Phase is defined in Figure 2.6.

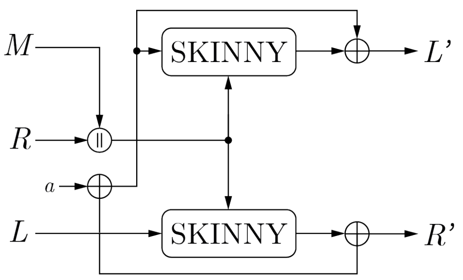


Figure 2.6: Romulus-H's compression function

2.4.2 Initialization phase

The Initialization Phase sets L and R to 0^n. The constants a and b denote to $0^7||1||0^{120}$ and $0^6||1||0^{120}$. The initialization of the cipher's internal state is performed by setting bytewise $IS_i = L_i$ for $0 \leq i \leq 15$. For the L' part, the IS is XORed to the constant a:

$$IS = \begin{bmatrix} L_0 & L_1 & L_2 & L_3 \\ L_4 & L_5 & L_6 & L_7 \\ L_8 & L_9 & L_{10} & L_{11} \\ L_{12} & L_{13} & L_{14} & L_{15} \end{bmatrix}$$

The three tweakeys are composed of the input R and M. TK_1 is initialized with the R vector by filling up $TK_{1_i} = R_i$. The second Tweakey gets filled with the first 128 bits of the padded message M and the third Tweakey with the first 128 bits.

$$TK_1 = \begin{bmatrix} R_0 & R_1 & R_2 & R_3 \\ R_4 & R_5 & R_6 & R_7 \\ R_8 & R_9 & R_{10} & R_{11} \\ R_{12} & R_{13} & R_{14} & R_{15} \end{bmatrix} TK_2 = \begin{bmatrix} M_{16} & M_{17} & M_{18} & M_{19} \\ M_{20} & M_{21} & M_{22} & M_{23} \\ M_{24} & M_{25} & M_{26} & M_{27} \\ M_{28} & M_{29} & M_{30} & M_{31} \end{bmatrix} TK_3 = \begin{bmatrix} M_0 & M_1 & M_2 & M_3 \\ M_4 & M_5 & M_6 & M_7 \\ M_8 & M_9 & M_{10} & M_{11} \\ M_{12} & M_{13} & M_{14} & M_{15} \end{bmatrix}$$

2.4.3 Round function

One round is composed of five operations, namely SubCells, AddConstants, AddRoundTweakey, ShiftRows and MixColumns. Each operation uses the cipher's internal state IS and the three Tweakeys TK_1, TK_2, TK_3. The output of the round is directly forwarded to the next round.

SubCells: an 8-bit S-box is applied on each cell of the cipher's internal state IS. Eight NOR and eight XOR gates describe the S-box, as depicted in Figure 2.7.

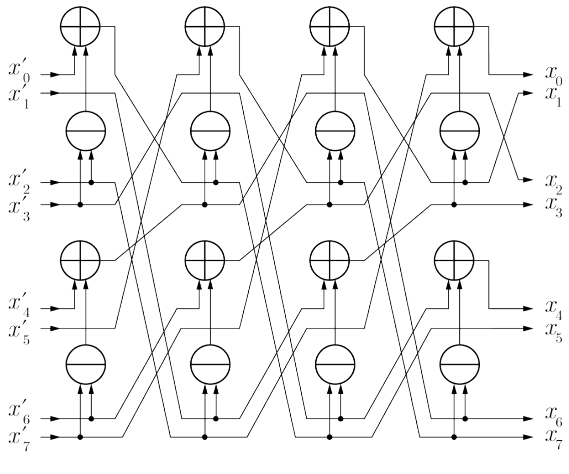


Figure 2.7: Romulus-H's S-box

AddConstants: In each round the round constanst rc is updated before its usage. Its update function is defined as:

$$(rc_5||rc_4||rc_3||rc_2||rc_1||rc_0) \leftarrow (rc'_4||rc'_3||rc'_2||rc'_1||rc'_0||rc'_5 \oplus rc'_4 \oplus 1)$$

We produce the three constants c_0, c_1, c_2 by extracting specific bits of the LFSR-output:

$$\begin{aligned} c_0 &= (\ 0\ ||\ 0\ ||\ 0\ ||\ 0\ ||rc_3||rc_2||rc_1||rc_0) \\ c_1 &= (\ 0\ ||\ 0\ ||\ 0\ ||\ 0\ ||\ 0\ ||\ 0\ ||rc_5||rc_4) \\ c_2 &= \texttt{0x2} \end{aligned}$$

The constants are XORed to the first column of the internal state:

$$\begin{bmatrix} m_0 \oplus c_0 & m_1 & m_2 & m_3 \\ m_4 \oplus c_1 & m_5 & m_6 & m_7 \\ m_8 \oplus c_2 & m_9 & m_{10} & m_{11} \\ m_{12} & m_{13} & m_{14} & m_{15} \end{bmatrix}$$

AddRoundTweakey: The first two rows of the internal State are XORed with the first two rows of the three tweakeys resulting in the new IS. Then the TK_2 and TK_3 cells are permuted:

$$\begin{bmatrix} C'_9 & C'_{15} & C'_8 & C'_{13} \\ C'_{10} & C'_{14} & C'_{12} & C'_{11} \\ C'_0 & C'_1 & C'_2 & C'_3 \\ C'_4 & C'_5 & C'_6 & C'_7 \end{bmatrix}$$

Finally, the cells of the first two rows get updated as follows:

$$\begin{aligned} TK_2 &= (\quad rc_6 \quad ||rc_5||rc_4||rc_3||rc_2||rc_1||rc_0||rc_7 \oplus rc_5) \\ TK_3 &= (rc_0 \oplus rc_6||rc_7||rc_6||rc_5||rc_4||rc_3||rc_2|| \quad rc_1 \quad) \end{aligned}$$

ShiftRows: Each row of the internal state is rotated to the right by its own row index (starting at 0), resulting in the following permutation:

$$\begin{bmatrix} C'_0 & C'_1 & C'_2 & C'_3 \\ C'_7 & C'_4 & C'_5 & C'_6 \\ C'_{10} & C'_{11} & C'_8 & C'_9 \\ C'_{13} & C'_{14} & C'_{15} & C'_{12} \end{bmatrix}$$

MixColumns: In the end, the internal state is multiplied with the multiplication matrix M.

$$M = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

2.5 Photon-Beetle hash function

PHOTON-Beetle-Hash [BCD+21] is based on a sponge construction and operates with 256-bit blocks. After initializing the internal state the round function is applied, which is composed of 12 rounds. The padding of the messages M_i is done by adding 0s to create a full r-bit block.

2.5.1 Initialization phase

The internal state IS is a 8×8 matrix, with a 4-bit value for each cell. The matrix is initialized by setting 4-bit wise $IS_i = m_i$ for $0 \leq i \leq 64$:

$$IS = \begin{bmatrix} m_0 & m_1 & \cdots & m_7 \\ m_8 & m_9 & \cdots & m_{15} \\ \vdots & \vdots & \ddots & \vdots \\ m_{56} & m_{57} & \cdots & m_{63} \end{bmatrix}$$

2.5.2 Round function

The round function is composed of 4 layers. The first layer is the so-called addKeyLayer, followed by the subCellsLayer, the shiftRowsLayer and finally the mixColumnsLayer.

addKeyLayer: A round r depending constant c_{rj} is XORed to the first column of the IS. The constants are defined in the PHOTON-Beetle-Hash paper [BCD+21]:

$$IS = \begin{bmatrix} m_0 \oplus rc_{r0} & m_1 & \cdots & m_7 \\ m_8 \oplus rc_{r1} & m_9 & \cdots & m_{15} \\ \vdots & \vdots & \ddots & \vdots \\ m_{56} \oplus rc_{r8} & m_{57} & \cdots & m_{63} \end{bmatrix}$$

subCellsLayer: In this layer we perform a 4-bit S-box. The S-box is defined in Table 2.3 as a lookup-table.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S-box	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

Table 2.3: PHOTON-Beetle-Hash S-box lookup-table

shiftRowsLayer: We shift the rows to the left accordingly to the row index starting from 0. We apply the following logical shift:

$$IS = \begin{bmatrix} m_0 & m_1 & \cdots & m_7 \\ m_9 & m_{10} & \cdots & m_8 \\ \vdots & \vdots & \ddots & \vdots \\ m_{63} & m_{56} & \cdots & m_{62} \end{bmatrix}$$

mixColumnsLayer: In this layer the columns are mixed independently using a serial matrix multiplication.

2.6 Esch hash function

Esch-256 [BBS+21] is based on a sponge construction and uses a fixed rate $r = 128$. The padding of the last message block is done by adding a single 1 and accordingly many 0s such that the length is equal to a multiple of 128. The number of permutation rounds for Esch-256 is 11 for the first permutation and 7 for the second permutation.

2.6.1 Initialization phase

The Sparkle permutation internally works with two 1 dimensional arrays x and y. Both contain six 32-bit values. The message block m_i is permuted with the Feistel algorithm and padded to a full r-bit block p_i which is than subsequently used for initializing x and y:

$$x = [p_0, p_1, p_2, p_3, p_4, p_5]$$
$$y = [p_6, p_7, p_8, p_9, p_{10}, p_{11}]$$

2.6.2 Sparkle permutation

Esch-256 [BBS+21] works with an Addition-Rotation-XOR-box (ARX-box) called Alzette and a DiffusionLayer.

ARX-box: Alzette is a 64-bit block cipher which can be divided into 4 rounds where the number of rotation rounds differ. This permutation is done pairwise for every 32-bit value in x and y, where for every cycle a different constant c is used. The ARX-box Alzette is depicted in Figure 2.8

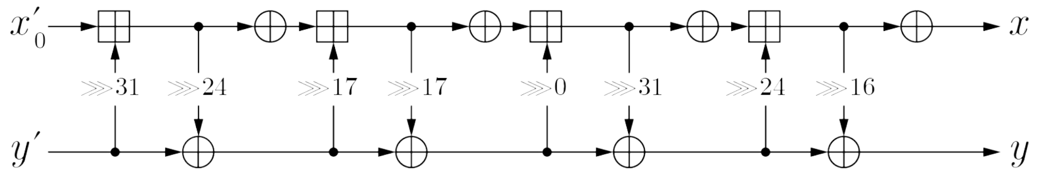


Figure 2.8: ESCH-256 ARX-box Alzette

DIFFUSIONLAYER: The diffusion layer applies the Feistel function on the x and y array which is then followed by a cell swap:

$$[p_4, p_5, p_3, p_0, p_1, p_2] \leftarrow [p'_0, p'_1, p'_2, p'_3, p'_4, p'_5]$$
$$[p_{10}, p_{11}, p_9, p_6, p_7, p_8] \leftarrow [p'_6, p'_7, p'_8, p'_9, p'_{10}, p'_{11}]$$

2.7 Keccak hash function

KECCAK [BDH+15] is also based on a sponge construction and uses the 10*1 padding rule, where a 1 is appended at the end of the input, then accordingly many 0s and another 1 to create a r-bit block. The number of permutation rounds for KECCAK is 24.

2.7.1 Initialization phase

The internal state IS is a 5×5 matrix with a 8-bit value in every cell IS_i. The IS is initialised bytewise with the padded message m:

$$IS = \begin{bmatrix} m_0 & m_1 & m_2 & m_3 & m_4 \\ m_5 & m_6 & m_7 & m_8 & m_9 \\ m_{10} & m_{11} & m_{12} & m_{13} & m_{14} \\ m_{15} & m_{16} & m_{17} & m_{18} & m_{19} \\ m_{20} & m_{21} & m_{22} & m_{23} & m_{24} \end{bmatrix}$$

2.7.2 Round function

The KECCAK permutation round function is divided into 5 different layer called after the Greek letters THETA, RHO, PI, CHI and IOTA.

THETA: The THETA layer calculates the value C_i for every row by XORing each IS_i of the same row. Then generates D_i by logically rotating the value of C_i to the left by 1. Now every D_i is XORed with the C_{i+3} to create E_i, which is then shifted by 1 index: $E_i = E_{i+1}$. Finally we XOR every cell of IS to the row value E_i: $IS_{i,j} = IS'_{i,j} \oplus E_i$.

RHO: In this layer we perform a logical shift of each cell to the left by cell dependent amount $C_{i,j}$. The IS is updated with: $IS_{i,j} = IS_{i,j} \lll C_{i,j}$, where C is defined as:

$$C = \begin{bmatrix} 0 & 1 & 6 & 4 & 3 \\ 4 & 4 & 6 & 7 & 4 \\ 3 & 2 & 3 & 1 & 7 \\ 1 & 5 & 7 & 5 & 0 \\ 2 & 2 & 5 & 0 & 6 \end{bmatrix}$$

PI: In the PI layer we mix the cells by iterating over the whole matrix and rearrange the cells accordingly $IS_{y,2*x+3*y} = IS'_{x,y}$

$$IS = \begin{bmatrix} IS'_0 & IS'_{16} & IS'_7 & IS'_{23} & IS'_{14} \\ IS'_{10} & IS'_1 & IS'_{17} & IS'_8 & IS'_{24} \\ IS'_{20} & IS'_{11} & IS'_2 & IS'_{18} & IS'_9 \\ IS'_5 & IS'_{21} & IS'_{12} & IS'_3 & IS'_{19} \\ IS'_{15} & IS'_6 & IS'_{22} & IS'_{13} & IS'_4 \end{bmatrix}$$

CHI: The effect of CHI is to XOR each bit with two other bits of the same row. The permutation in Figure 2.9 is performed on every row of IS:

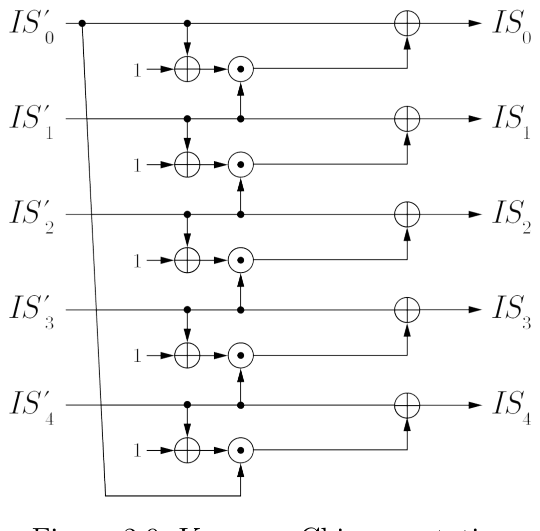


Figure 2.9: KECCAK Chi permutation

IOTA: In the last permutation layer we XOR the round constant c_r to the IS_0 cell:

$$IS = \begin{bmatrix} IS'_0 \oplus c_r & IS'_1 & IS'_2 & IS'_3 & IS'_4 \\ IS'_5 & IS'_6 & IS'_7 & IS'_8 & IS'_9 \\ IS'_{10} & IS'_{11} & IS'_{12} & IS'_{13} & IS'_{14} \\ IS'_{15} & IS'_{16} & IS'_{17} & IS'_{18} & IS'_{19} \\ IS'_{20} & IS'_{21} & IS'_{22} & IS'_{23} & IS'_{24} \end{bmatrix}$$

2.8 Spongent hash function

SPONGENT [BKL+13] is based on a sponge construction and uses the 10^* padding scheme, where a single 1 is appended at the end of the input and accordingly many 0s to create a r-bit block. The number of permutation rounds for SPONGENT is 80.

2.8.1 Initialization phase

We can visualize the internal state IS with a 1×20 array which is filled up with the padded messages p 8-bit wise: $IS_i = p_i$. The initialisation vector IV is set to `0x75`.

2.8.2 Round function

The permutation round is split into 3 layers, the LCOUNTERLAYER, the SBOXLAYER and the LINEARLAYER.

LCOUNTERLAYER: In this layer we XOR IV to the first IS_0 cell and the inverted (flipped) IV is XORed to the last IS_{19} cell. After applying the IV to the IS, we update the IV with a LFSR.

SBOXLAYER: In this layer we perform a 8-bit S-box. The S-box is defined in Table 2.4 as a lookup-table.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S-box	E	D	B	0	2	1	4	F	7	A	8	5	9	C	3	6

Table 2.4: SPONGENT S-box lookup-table

LINEARLAYER: This layer moves a bit j of IS to another bit position $P_b(j)$, where

$$P_b(j) = \begin{cases} j \cdot 40 \mod 159, & \text{if } j \in 0, ..., 158 \\ 159, & \text{if } j = 159 \end{cases}$$

3 Implementation

In this section, we introduce the basics of our implementation of both attacks. The first attack is the so-called brute-force attack and the second attack is called the SAT-Solver attack. Both try to find a pre-image of a target hash by following a different approach.

3.1 Brute-force setup

A brute-force attack follows the trial and error principle, where it guesses an input, computes the corresponding output and checks if it satisfies the constraint. This is realised by increasing a 64-bit integer and calculating its hash. If the hash corresponds to the target hash, we can terminate and found a compatible pre-image. All the running threads execute the function defined in Algorithm 1 to split up the workload. Each thread gets assigned a space of variables to check, where $\max_{int64}$ corresponds to the highest possible 64-bit integer value (`0xffffffff`) and i to the index of the thread.

$$\begin{aligned} start &= V_{\max_{int64}}/N_{\text{thread}} \cdot i \\ end &= V_{\max_{int64}}/N_{\text{thread}} \cdot (i+1) \end{aligned}$$

Each loop consists of increasing the counter by one and recalculating the hash value of the new obtained message. Then the calculated hash is compared with the predefined hash, and if they match, we can terminate. The function is synchronized because each thread has its own searching space, and no value will get checked twice or not at all.

Algorithm 1 Brute Force

```
1: function BruteForceThreadFunction(initPosition, hashLength, preImgLength)
2:     counter = initPosition
3:     while counter++ $\leq$ initPosition + threadSpaceField do
4:         for i in range preImgLength do
5:             msg[i] = (initPosition $\gg$ (8 $*$ i)) & 0xFF
6:         end for
7:         digest = HashFunction(msg)
8:         if Truncate(digest, hashLength) == searchedHash then
9:             terminate()
10:        end if
11:    end while
12: end function
```

3.2 SAT-Solver setup

The primary purpose of a SAT-Solver is to return the satisfiability of a logical boolean formula. In this paper, we use the z3SAT solver from Microsoft Research [Bjø19]. As a programming language, we chose Python. First, we need to set up the solver and the variables. Then the hash function is provided to the solver. The SAT-Solver will continue until it finds the correct hash value.

We can compare a SAT-Solver attack with a box model, where we provide an input-Bitvector, some mapping function, and an output-Bitvector. The mapping function gets divided into smaller Boxes, each with an input, mapping function, and output. The mapping function is defined by the corresponding hash function. The Ascon-Xof Box Model is defined in Figure 3.1 and the Romulus-H in Figure 3.2

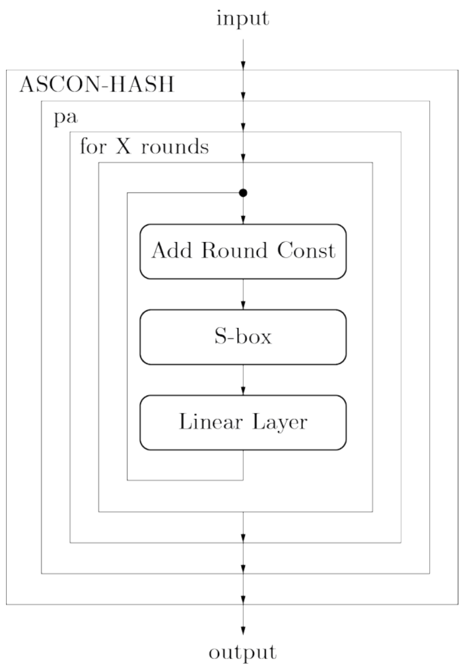


Figure 3.1: ASCON-Hash's box model

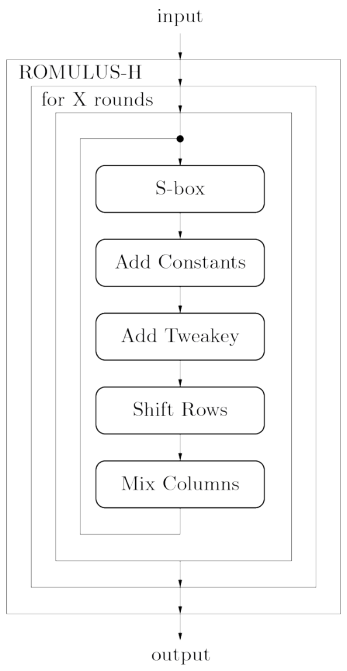


Figure 3.2: ROMULUS-H's box model

3.3 Evaluation

To determine the range of rounds the attacks are executed on, we run both attacks with 2 different target hashes for every number of rounds. We then create a monomial approximation of both attacks and take a closer look at where they cross each other.

For each round that falls into the boundaries, we choose randomly 10 different hash outputs and measure the execution time of the attack. The data is then visualized in a logarithmic time scale on the y-axis and the hash index (from 0 to 9) on the x-axis.

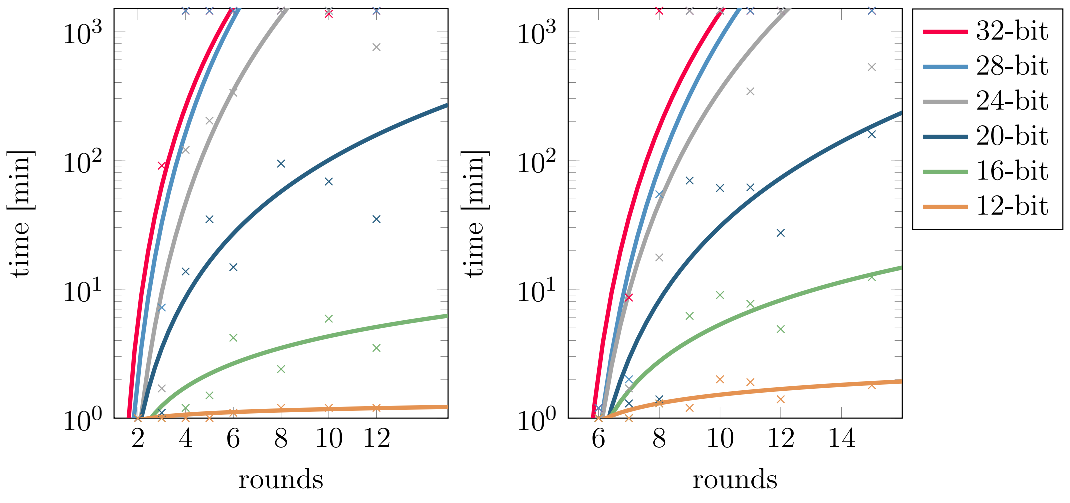
4 Testing Environment

Both attacks are executed on the IAIK-Cluster, more precisely on the xeon192g0 node. This node uses the Intel Xeon CPU E5-4669 v4 with 88 logical cores [Int18]. The upper limit of the attack runtime is 24 hours.

We run tests with different hash sizes and round numbers to obtain the critical area, where the SAT-Solver attack is as fast as the brute-force attack. With the gathered data, we can define an approach line as depicted in Figure 4.1a for Ascon-Xof and in Figure 4.1b for Romulus-H.

4.1 SAT-Solver attack projection

The execution time of the SAT-Solver depends on how many constraints it has to manage among others. By increasing the number of rounds, the number of rules increases and, therefore, the execution time. Another essential factor is the hash length. In Figure 4.1 the relation between the number of rounds and the time passed is shown. Each graph represents a certain hash length and is a projection of the mean from two different test-runs, where each test-run has a randomly chosen target hash.

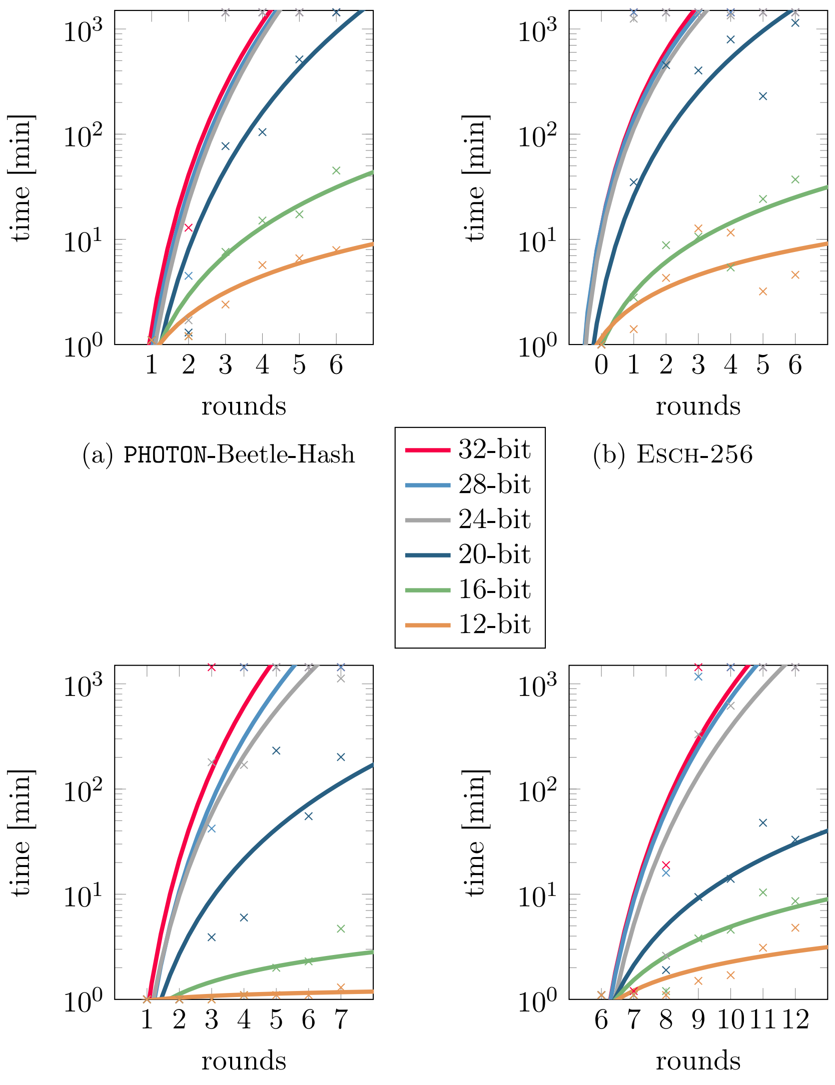


(a) Ascon-Xof (b) Romulus-H

Figure 4.1: SAT-Solver attack projection with different hash lengths

Comparison with other NIST competitors

For comparison reasons, we execute the SAT-Solver attack on other 4 NIST competitors: PHOTON-Beetle-Hash, Esch-256, Keccak and Spongent. The SAT-Solver attack projections are depicted in Figure 4.4. Esch-256 uses 2 different permutation rounds, since we are working with reduced hashing algorithms we set them equal for rounds below 5, while for rounds r greater than 4 we set the rounds to r and $r - 4$.

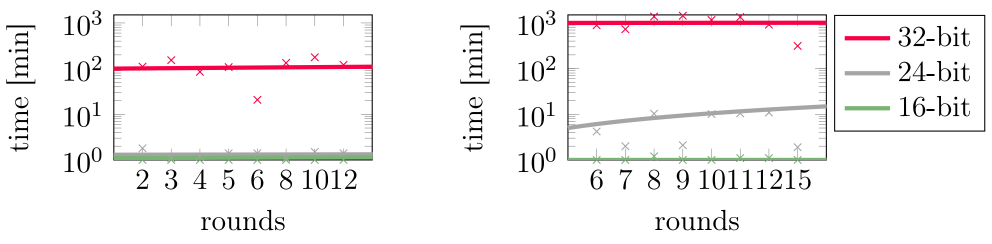


(a) PHOTON-Beetle-Hash (b) Esch-256 (c) Keccak (d) Spongent

Figure 4.2: SAT-Solver attack projection with different hash lengths

4.2 Brute-force attack projection

The execution time of the Brute-Force attack depends mainly on the hash length. By increasing the number of rounds, the execution time increases by a tiny amount δ. This is a constant number and depends solely on the execution time of a round. In Figure 4.3 the relation between the number of rounds and the time passed is shown. Each graph represents a certain hash length.

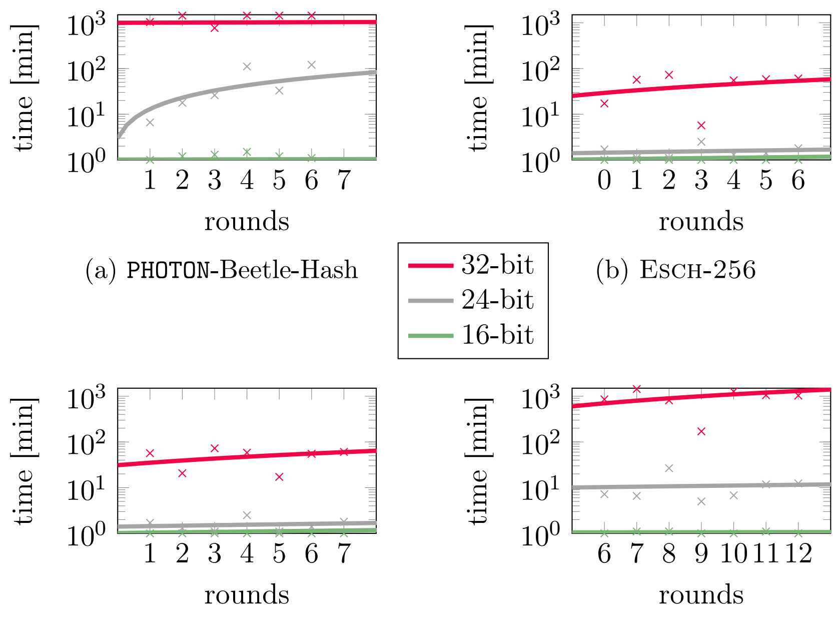


(a) Ascon-Xof (b) Romulus-H

Figure 4.3: Brute-force attack projection with different hash lengths

Comparison with other NIST competitors

The brute-force attack projections for `PHOTON`-Beetle-Hash, Esch-256, Keccak and Spongent are depicted in Figure 4.4.



(a) `PHOTON`-Beetle-Hash (b) Esch-256

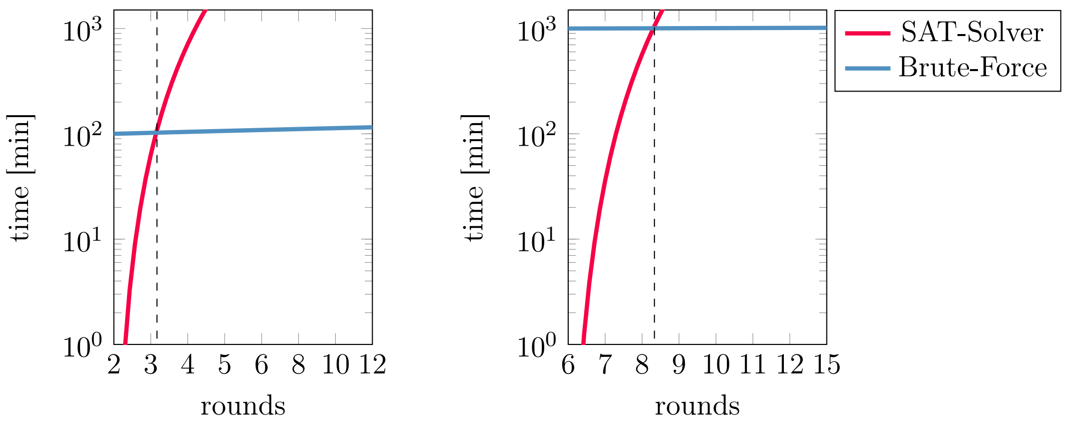
(c) Keccak (d) Spongent

Figure 4.4: Brute-force attack projection with different hash lengths

4.3 Selection of critical area

We determine a hash length of 32 bits for both hash functions because it is the highest possible value which is feasible to solve in 24 hours under our restrictions. Furthermore, we choose a lower- and upper bound of rounds for the critical area. Everything below the lower bound, the SAT-Solver attack will dominate the brute-force attack, and everything beyond the upper bound, the brute-force attack will be more efficient.

Therefore the critical area for Ascon-Xof lies between the rounds 2 and 4 as depicted in Figure 4.5a. The critical area for Romulus-H lies between the rounds 6 and 9 which can be inspected in Figure 4.5b.



(a) Ascon-Xof (b) Romulus-H

Figure 4.5: SAT-Solver and brute-force comparison 32 bits

Selection of critical area for other NIST competitors

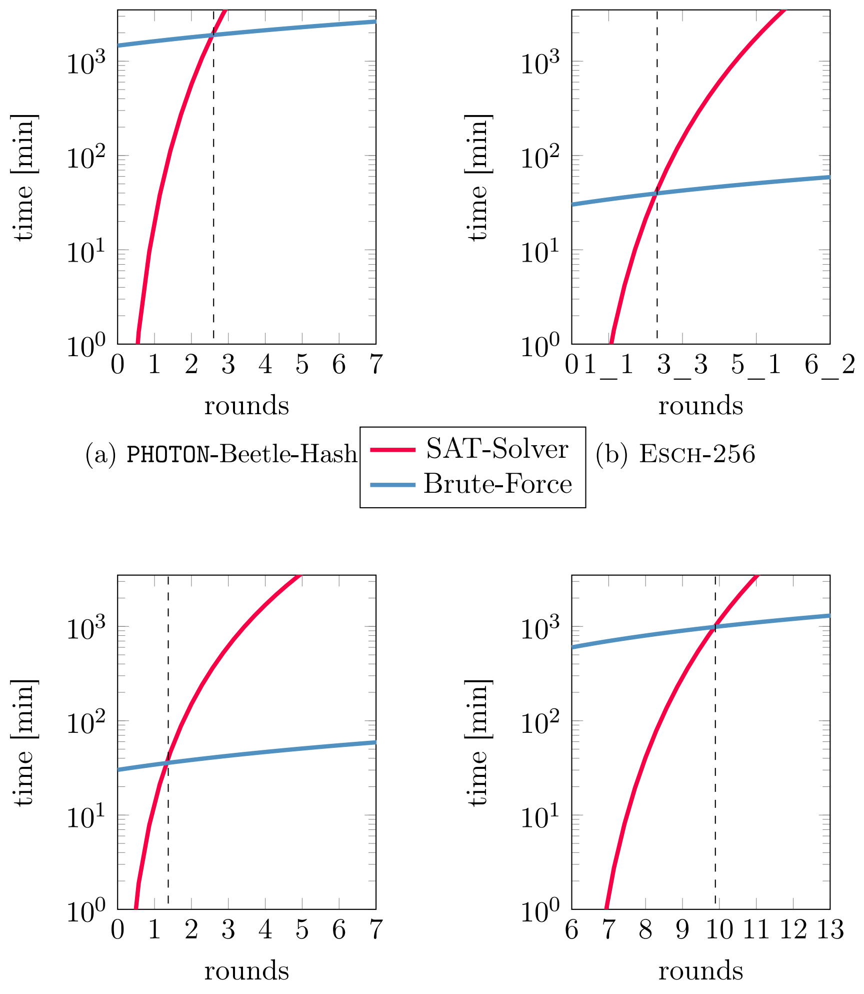
The SAT-Solver attack is compared to the brute-force attack with a hash length of 32 bits in Figure 4.6 to determine the critical area for each hashing algorithm.

PHOTON-Beetle-Hash: The comparison of the brute-force attack and the SAT-Solver attack is depicted in Figure 4.6a. The lower bound of the critical area is 1 and the upper bound is 3.

Esch-256: We set the lower bound of the critical area to 0 and the upper bound to 1. Esch-256 uses the permutation with two different step sizes. Since we reduced the permutation rounds, we set both step sizes to 0 for the lower bound and 1 for the upper bound. The comparison of the SAT-Solver attack and the brute-force attack is depicted in Figure 4.6b.

KECCAK: In Figure 4.6c we observe that the SAT-Solver attack can keep up with the brute-force attack when the permutation rounds are 1 or 2. Therefore we set the lower bound to 1 and the upper bound to 3.

SPONGENT: As visualized in Figure 4.6d the critical area is from 8 permutation rounds to 10 permutation rounds.



(a) PHOTON-Beetle-Hash (b) ESCH-256

(c) KECCAK (d) SPONGENT

Figure 4.6: SAT-Solver and brute-force comparison 32 bits

5 Application

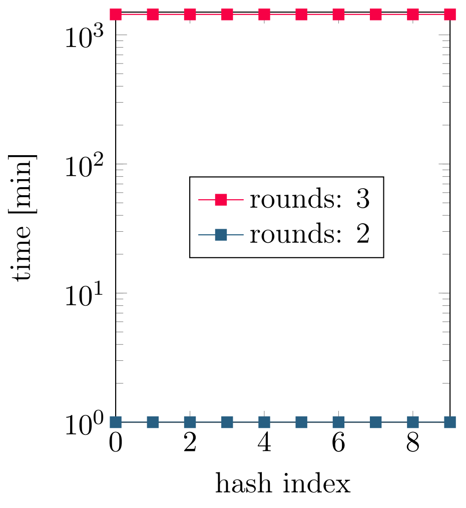
We will execute two attacks on Ascon-Xof and Romulus-H. The first attack will be the brute-force attack, and the second attack is with a SAT-Solver. Then we compare the runtime of each exploit and evaluated if the SAT-Solver attack is more efficient.

5.1 Ascon-XOF

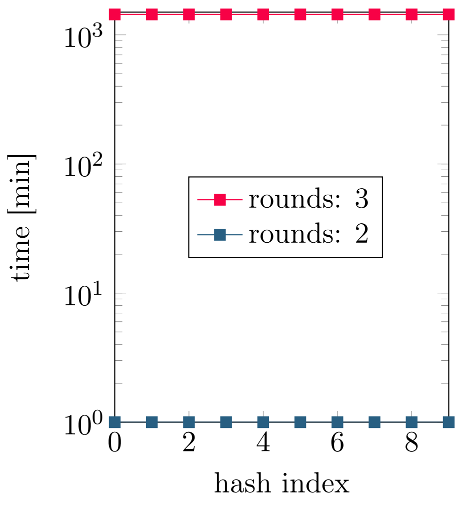
The lower bound of our test environment for Ascon-Xof is 2 rounds with 32 bits, the upper bound is 4 rounds with 32 bits. We run some more tests for putting the SAT-Solver to its limitation. When keeping the rounds to a maximum of 2, we can increase the hash length to 320 bits and still find a preimage almost immediately.

5.1.1 SAT-Solver attack limitations

We executed the SAT-Solver attack on bigger hash sizes and observed that it has no noteworthy impact on the execution time. On two permutation rounds, the SAT-Solver finds a pre-image almost immediately. When increasing the permutation rounds to 3, the attack turns out to be infeasible to solve in under 24 hours as shown in Figure 5.1.



(a) 64 bit

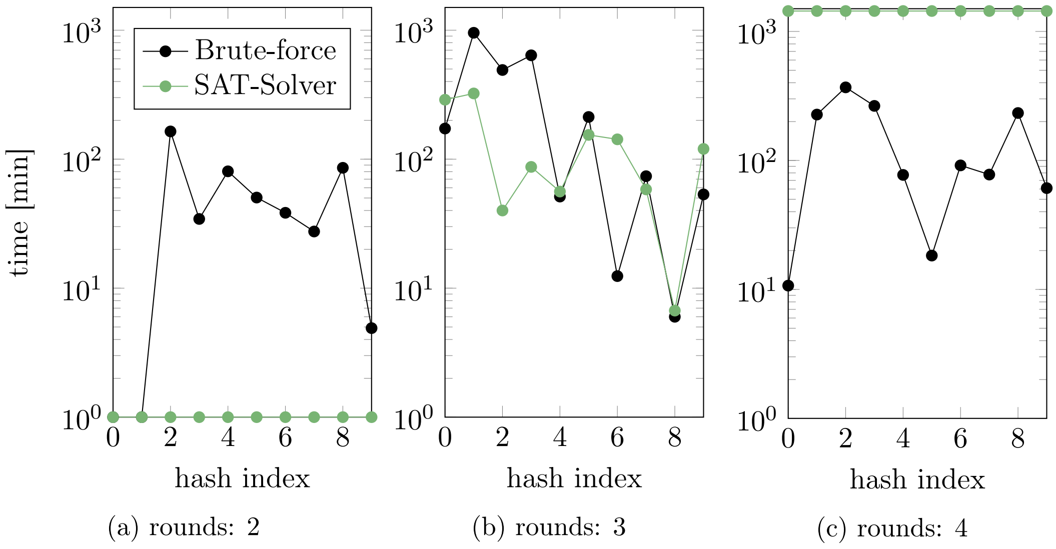


(b) 128 bit

Figure 5.1: SAT-sollver limit testing for Ascon-Xof

5.1.2 Comparison of brute-force and SAT-Solver attack

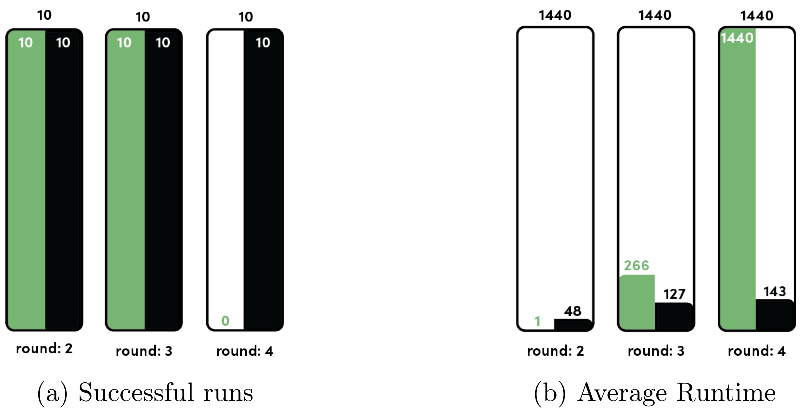
The attacks on our testing environment of Ascon-Xof are depicted in Figure 5.2. The SAT-Solver attack dominates the brute-force attack on 2 rounds, by finding a preimage of every test hash in under a minute. But when increasing the rounds to 4, the SAT-Solver attack does not find the pre-image in less than 24 hours. At the 3rd round, both attacks have a similar execution time and manage to find a preimage in under 24 hours. The brute-force attack dominates the SAT-Solver attack from the 4th round and above.



(a) rounds: 2 (b) rounds: 3 (c) rounds: 4

Figure 5.2: Test-runs for Ascon-Xof 32 bits

In Figure 5.3 we compare both attacks by their successful runs in Figure 5.3a, and their average runtime of successful and unsuccessful test-runs in Figure 5.3b. Both charts are in linear scale and follow the same color scheme as Figure 5.2.



(a) Successful runs (b) Average Runtime

Figure 5.3: Test-runs for Ascon-Xof 32 bits summary

5.2 Romulus-H

The lower bound of our test environment for Romulus-H is 6 rounds with 32 bits, the upper bound is 9 rounds with 32 bits. We run some more tests for putting the SAT-Solver to its limitation. When keeping the rounds to a maximum of 6, we can increase the hash length to 128 bits and still find a preimage almost immediately.

5.2.1 SAT-Solver attack limitations

By increasing the hash length to 64 bits, 8 rounds turn to be infeasible to solve. The execution time of round 6 and 7 stay almost the same. When we increase the hash length to 128 bits, the execution time of round 7 increases drastically and turns to be infeasible to solve in under 24 hours. The execution time of round 6 increases as well and the SAT-Solver needs approximately 10 minutes to find a preimage

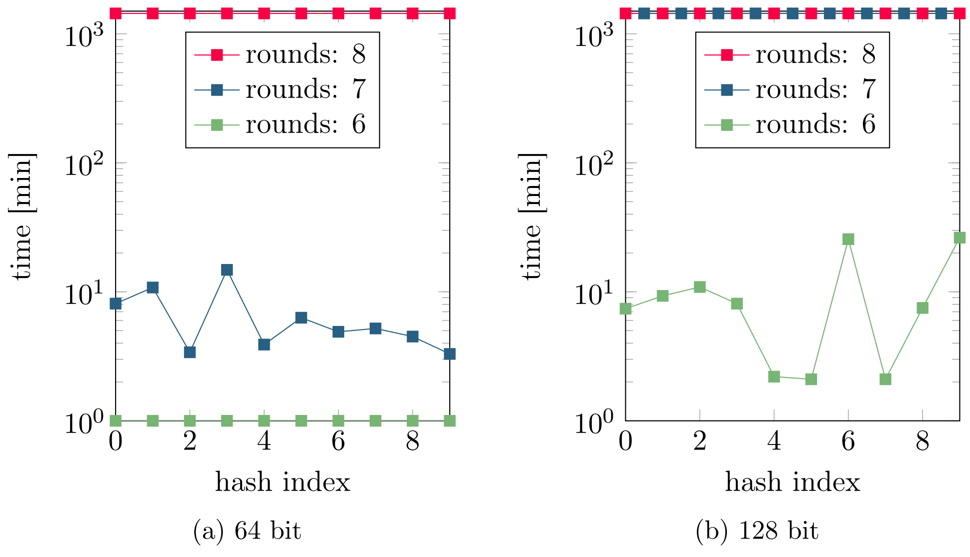
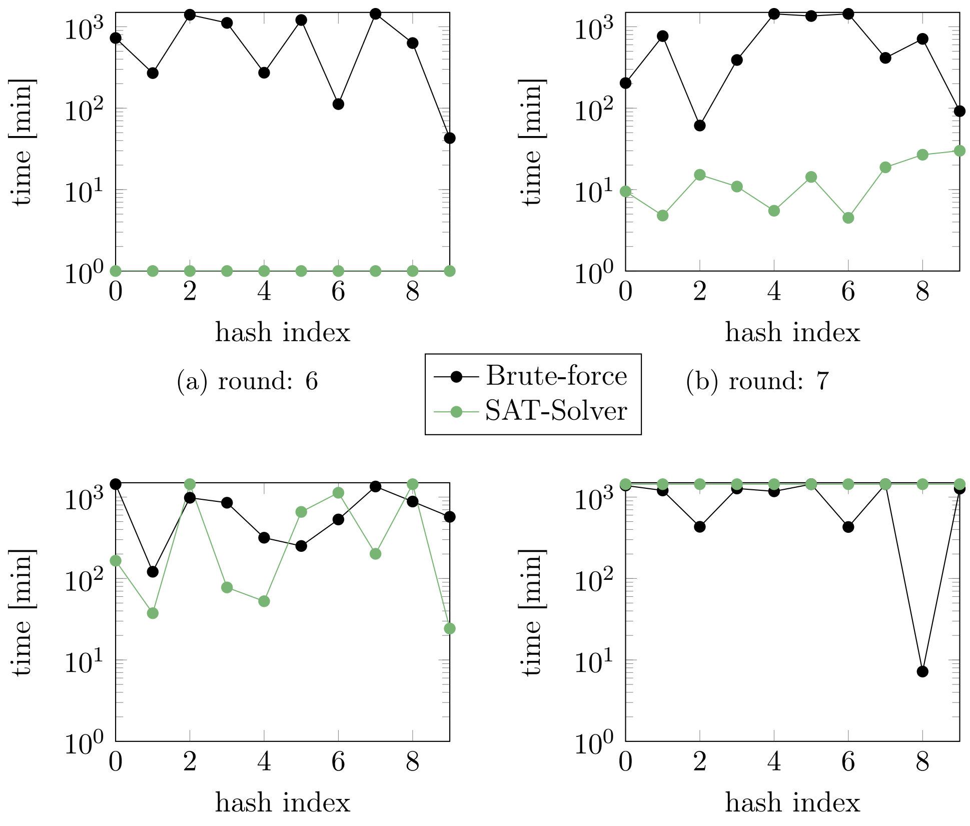


Figure 5.4: SAT-Solver limit testing for Romulus-H

5.2.2 Comparison of brute-force and SAT-Solver attack

The testing boundaries of Romulus-H are 6 rounds and 9 rounds at 32 bits. Up to round 6, the SAT-Solver finds a preimage almost immediately. On round 9 and above, the brute-force attack is more efficient as visualized in Figure 5.5

On round 7, the SAT-Solver attack does still have a better performance than the brute-force attack and finds a preimage at approximately 15 minutes. The brute-force attack execution time is around 700 minutes. By increasing the number of rounds by 1, the performance of brute-force attack and the SAT-Solver attack are even again, with a mean over all data points of 710 minutes:



(a) round: 6

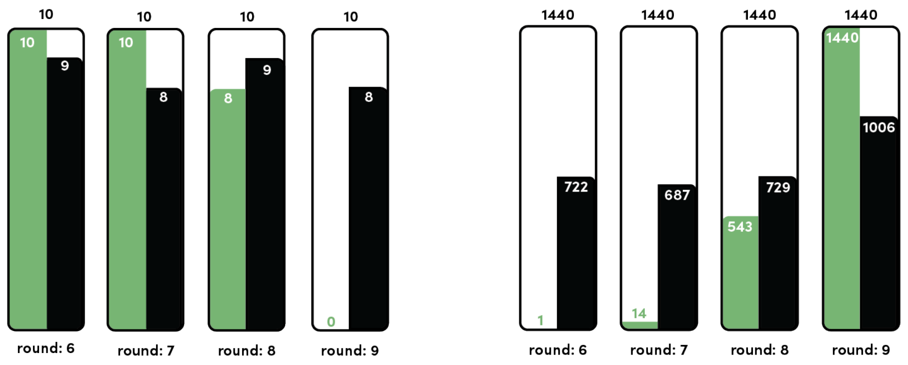
(b) round: 7

(c) round: 8

(d) round: 9

Figure 5.5: Test-runs for ROMULUS-H 32 bits

In Figure 5.6 we compare both attacks by their successful runs in Figure 5.6a, and their average runtime of successful and unsuccessful test-runs in Figure 5.6b. Both charts are in linear scale and follow the same color scheme as Figure 5.5.



(a) Successful runs

(b) Average Runtime

Figure 5.6: Test-runs for ROMULUS-H 32 bits summary

5.3 Photon-Beetle-Hash

The testing boundaries of PHOTON-Beetle-Hash are 1 round and 3 rounds at 32 bits. Above round 3, the SAT-Solver attack exceeds the 24 hours execution time. At 2 rounds the SAT-Solver attack is still better then the brute-force attack but stands no chance when increasing the permutation rounds to 3. The figures are visualized in Figure 5.7.

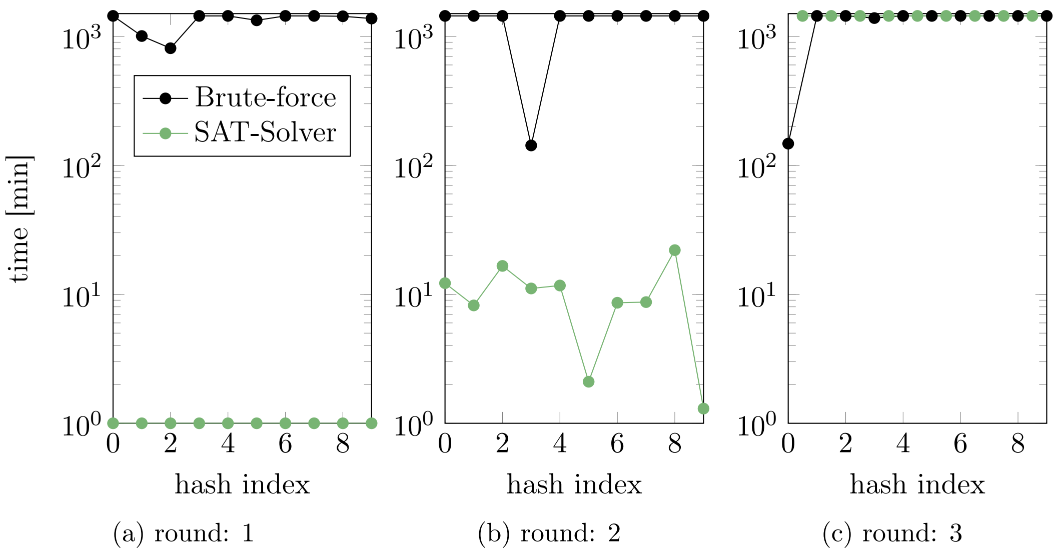


Figure 5.7: Test-runs for PHOTON-Beetle-Hash 32 bits

In Figure 5.8 we compare both attacks by their successful runs in Figure 5.8a, and their average runtime of successful and unsuccessful test-runs in Figure 5.8b. Both charts are in linear scale and follow the same color scheme as Figure 5.7.

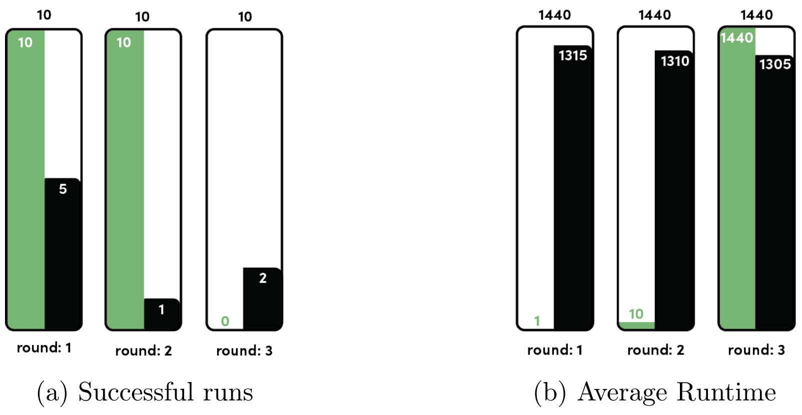
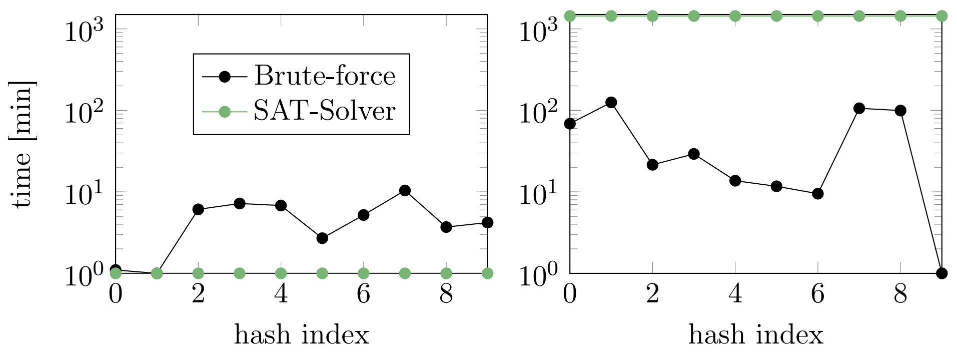


Figure 5.8: Test-runs for PHOTON-Beetle-Hash 32 bits summary

5.4 Esch

The Esch-256 hash function complexity grows fast with each round, therefore we run our tests with no round at all and 1 round. The unreduced Esch-256 uses 2 different permutation with different permutation round numbers. Since we reduced the hashing algorithm, we set them equal leading to round 0, where both permutation rounds are set to 0, and round 1, where both are set to 1.

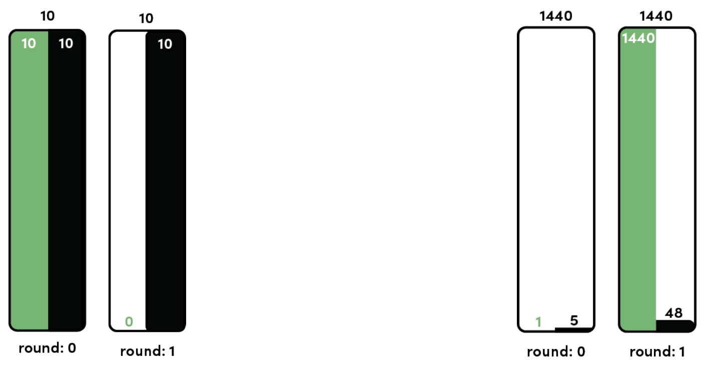
The SAT-Solver attack is infeasible to solve for 1 permutation round in under 24 hours for 32 bits hash length as visualized in Figure 5.9.



(a) round: 0   (b) round: 1

Figure 5.9: Test-runs for Esch-256 32 bits

In Figure 5.10 we compare both attacks by their successful runs in Figure 5.10a, and their average runtime of successful and unsuccessful test-runs in Figure 5.10b. Both charts are in linear scale and follow the same color scheme as Figure 5.9.

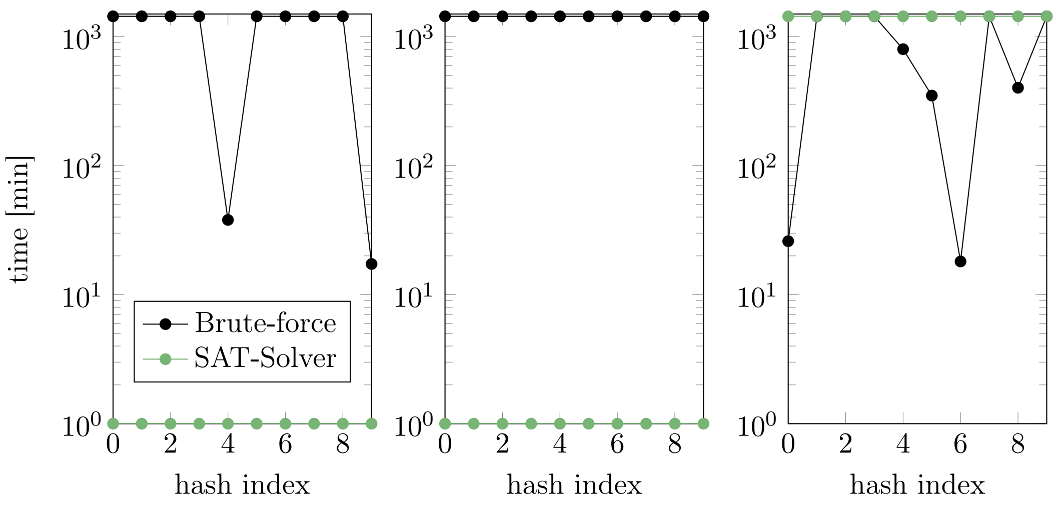


(a) Successful runs   (b) Average Runtime

Figure 5.10: Test-runs for Esch-256 32 bits summary

5.5 Keccak

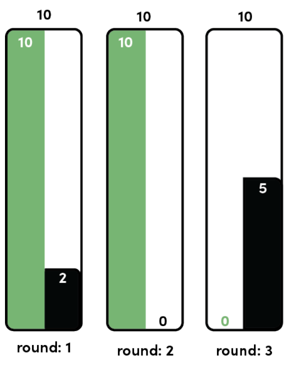
The boundaries for our testing environment are set to 1 round for the lower bound and 3 for the upper bound. The SAT-Solver attack was efficient up to the 2nd round but was infeasible to solve for the 3 rounds. The brute-force attack was not efficient at the 1st and 2nd round, but got some solutions at the 3rd round. This happened because the target hash was randomly chosen and for round 3 the pre-image happened to be a small 64 bit integer. The results are depicted in Figure 5.11



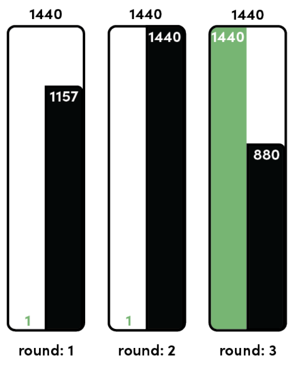
(a) round: 1 (b) round: 2 (c) round: 3

Figure 5.11: Test-runs for Keccak 32 bits

In Figure 5.12 we compare both attacks by their successful runs in Figure 5.12a, and their average runtime of successful and unsuccessful test-runs in Figure 5.12b. Both charts are in linear scale and follow the same color scheme as Figure 5.11.



(a) Successful runs



(b) Average Runtime

Figure 5.12: Test-runs for Keccak 32 bits summary

5.6 Spongent

The lower bound for SPONGENT is set to 7 rounds and the upper bound is set to 9 rounds. The SAT-Solver performs better than the brute-force attack up to 8 rounds. At the 9th round and above the execution time of the brute-force attack is lower then the execution time of the SAT-Solver. The test results are visualized in Figure 5.13.

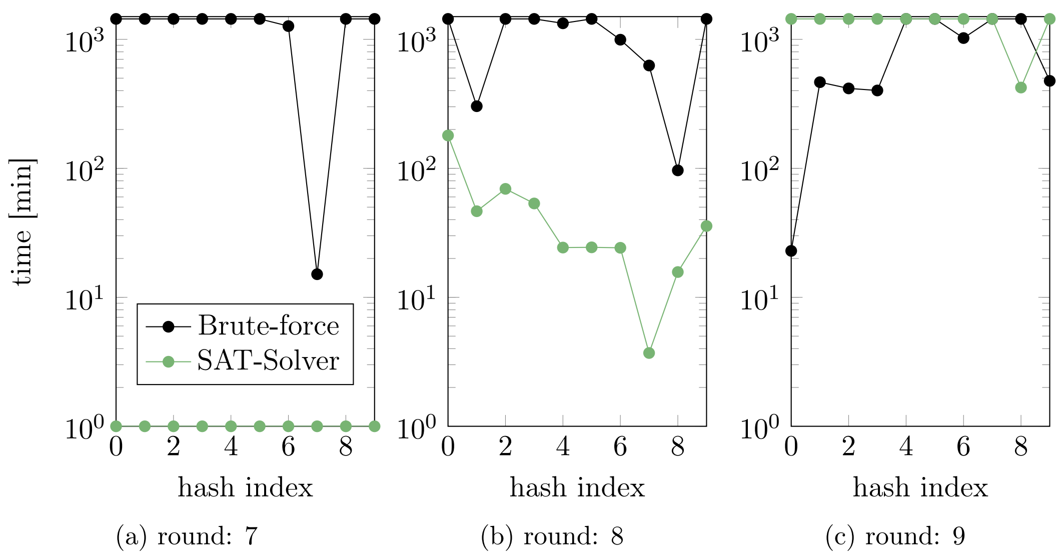


Figure 5.13: Test-runs for SPONGENT 32 bits

In Figure 5.14 we compare both attacks by their successful runs in Figure 5.14a, and their average runtime of successful and unsuccessful test-runs in Figure 5.14b. Both charts are in linear scale and follow the same color scheme as Figure 5.13.

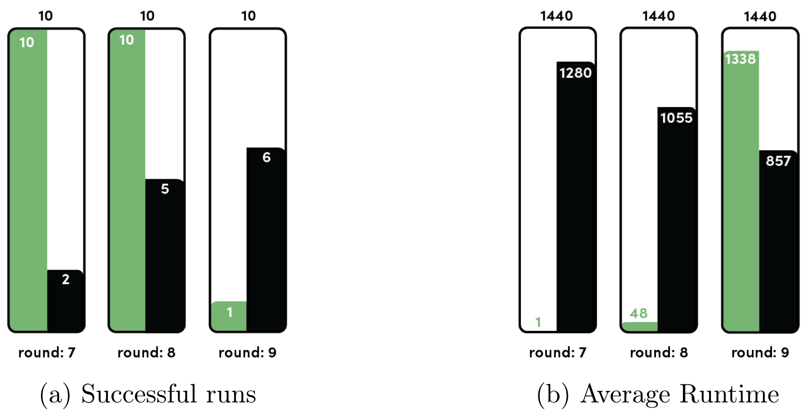


Figure 5.14: Test-runs for KECCAK 32 bits summary

5.7 Summary table

In this section we summarize the data we obtained. If an attack was not successful in our testing environment (exceeded execution time) we marked it with a − symbol.

Ascon-Xof					
Hash length	round	mean brute-force	mean SAT-Solver	better performance	
				Brute-force	SAT-Solver
32	2	48.8	1		●
	3	127.7	266.6	●	
	4	142.9	−	●	
64	2	−	1		●
	3	−	−		
128	2	−	1		●
	3	−	−		

Table 5.1: Ascon-Xof summary table

Romulus-H					
Hash length	round	mean brute-force	mean SAT-Solver	better performance	
				Brute-force	SAT-Solver
32	6	730.4	1		●
	7	729.9	14.0		●
	8	749.3	649.2		●
	9	1016.4	−	●	
64	6	−	1		●
	7	−	6.5		●
	8	−	−		
128	6	−	10.2		●
	7	−	−		

Table 5.2: Romulus-H summary table

In Table 5.3 we compare each NIST competitor by the permutation rounds and the execution time of the SAT-Solver attack at 32 bits hash length. The infeasible row defines up to which point the hash function is secure. The initial letter of the hash function is used as marker.

For better comparison, we visualized each candidate in proportion to its max permutation rounds in Figure 5.15. The red area marks the rounds at which the SAT-Solver attack finds a pre-image immediately. The yellow area visualizes the rounds which are still solvable for the SAT-Solver attack, but are more expensive. The red marked rounds are infeasible for the SAT-Solver attack in under 24 hours.

NIST competitors comparison								
round	mean SAT-Solver	Hash name	Infeasible					
			A	R	P	E	K	S
0	1	Esch-256						
1	–	Esch-256				●		
	1	PHOTON-Beetle-Hash				●		
	1	Keccak				●		
2	10.2	PHOTON-Beetle-Hash				●		
	1	Ascon-Xof				●		
	1	Keccak				●		
3	–	PHOTON-Beetle-Hash			●	●		
	–	Keccak			●	●	●	
	266.6	Ascon-Xof			●	●	●	
4	–	Ascon-Xof	●		●	●	●	
6	1	Romulus-H	●		●	●	●	
7	14.0	Romulus-H	●		●	●	●	
	1	Spongent	●		●	●	●	
8	649.15	Romulus-H	●		●	●	●	
	47.7	Spongent	●		●	●	●	
9	–	Romulus-H	●	●	●	●	●	
	2292.29	Spongent	●	●	●	●	●	●

[1] A = Ascon-Xof
[2] R = Romulus-H
[3] P = PHOTON-Beetle-Hash
[4] E = Esch-256
[5] K = Keccak
[6] S = Spongent

Table 5.3: Detailed Comparison of SAT-Solver attack

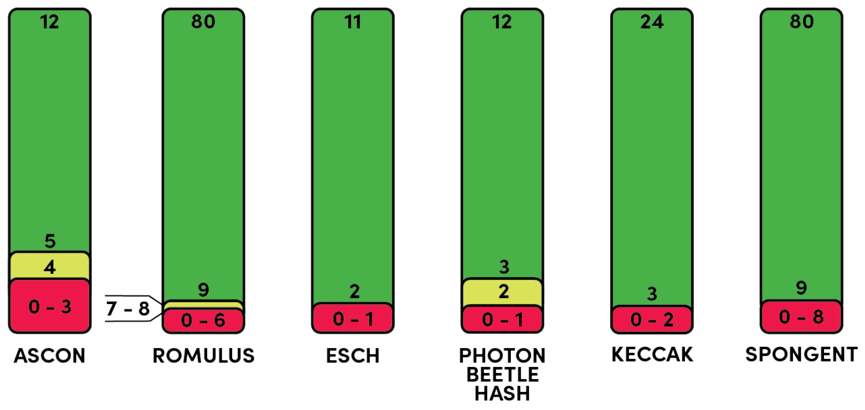


Figure 5.15: SAT-Solver vulnerability for every NIST competitor

6 Conclusion

In this paper we compared a SAT-Solver attack to the brute-force attack. The attacked hash functions were Ascon-Xof and Romulus-H on a range of reduced permutation rounds. We restricted our testing environment to 24 hours, 64-bit pre-image length and 32-bit target hash length. Furthermore we varied the target hash length from 64-bit up to 128-bit to investigate the impact to the SAT-Solver attack.

The SAT-Solver attack was very efficient at low number of permutation rounds, but stood no chance when increasing the rounds against the brute-force attack. To put this into real-world relation, Ascon-Xof consists of 12 rounds. The SAT-Solver attack was efficient to up to 2 rounds. When reaching round 3, the SAT-Solver was still able to find a pre-image but not as fast as the brute-force attack, but when reaching round 4, the execution time exceeded 24 hours. For the hash length, Ascon-Xof uses 256 bit, which has no noteworthy impact on the SAT-Solver attack. When setting the hash length to 64, 128, or 256 bits and the hash rounds to 2, we found a preimage almost immediately. The brute-force attack was constant when increasing the hash rounds, but turned to be infeasible when increasing the hash length. At a hash length of 64 bits, the brute-force attack exceeded an execution time of 24 hours.

Romulus-H consists of 40 rounds and a hash length of 256 bits. In our testing environment the SAT-Solver was efficient to up to round 7, when reaching round 8 it was slightly better then the brute-force attack, but when reaching round 9 the execution time exceeded 24 hours. By increasing the hash length of the SAT-Solver attack to 128 bits, the execution time increased for round 6 by a multiple of 10. The hash rounds 7, 8, and 9 turned to be infeasible to solve in under 24 hours for the SAT-Solver.

Another noteworthy observation was that Ascon-Xof had a better performance than Romulus-H. While Ascon-Xof only has 12 rounds and turns to be infeasible to solve for the SAT-Solver at round 4, Romulus-H has 40 rounds and can be exploited to round 8. This means that the rounds of Ascon-Xof provide a better and faster security level and therefore need less rounds to be secure.

While Moore's law is ending and the performance of the brute-force attacks will not increase that much anymore, SAT-Solvers are still at their beginning. In the next couple of years, SAT-Solver will be brought to multi-core, where tasks can be parallelized and thus, lead to lower time consumption. With this improvement, SAT-Solver have the potential to beat brute-force attacks on even a higher number of permutation rounds.

Bibliography

[BBS+21] Christof Beierle, Alex Biryukov, Luan Cardoso dos Santos, Johann Großschädl, Amir Moradi, Léo Perrin, Aein Rezaei Shahmirzadi, Aleksei Udovenko Vesselin Velichkov, and Qingju Wang. *Schwaemm and Esch: Lightweight Authenticated Encryption and Hashing using the Sparkle Permutation Family.* 2021. URL: `https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/sparkle-spec-final.pdf`.

[BCD+21] Zhenzhen Bao, Avik Chakraborti, Nilanjan Datta, Jian Guo, Mridul Nandi, Thomas Peyrin, and Kan Yasuda. *PHOTON-Beetle Authenticated Encryption and Hash Family.* 2021. URL: `https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/photon-beetle-spec-final.pdf`.

[BDH+15] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions". In: (2015). URL: `https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf`.

[BDPA08] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. *On the Indifferentiability of the Sponge Construction.* 2008. URL: `https://keccak.team/files/SpongeFunctions.pdf`.

[BJK+16] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. "The SKINNY Family of Block Ciphers and its Low-Latency Variant MANTIS". In: Berlin, Heidelberg: Springer Berlin Heidelberg, 2016. DOI: `10.1007/978-3-662-53008-5_5`.

[Bjø19] Nikolaj Bjørner. *The inner magic behind the Z3 theorem prover.* 2019. URL: `https://www.microsoft.com/en-us/research/blog/the-inner-magic-behind-the-z3-theorem-prover/` (visited on 04/06/2022).

[BKL+13] Andrey Bogdanov, Miroslav Knezevic, Gregor Leander, Deniz Toz, Kerem Varici, and Ingrid Verbauwhede. "SPONGENT: The Design Space of Lightweight Cryptographic Hashing". In: *IEEE Trans. Comput.* 62.10 (Oct. 2013). ISSN: 0018-9340. DOI: `10.1109/TC.2012.196`. URL: `https://doi.org/10.1109/TC.2012.196`.

[CN] CSRC and NIST. *Lightweight Cryptography.* URL: `https://csrc.nist.gov/Projects/lightweight-cryptography`.

[DEMS21] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. *Ascon.* 2021. URL: `https://csrc.nist.gov/CSRC/media/Projects/`

lightweight-cryptography/documents/finalist-round/updated-spec-doc/ascon-spec-final.pdf (visited on 07/21/2021).

[Dwo15] Morris Dworkin. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions.* en. 2015-08-04 2015. DOI: https://doi.org/10.6028/NIST.FIPS.202.

[GIK+21] Chun Guo, Tetsu Iwata, Mustafa Khairallah, Kazuhiko Minematsu, and Thomas Peyrin. *Romulus.* 2021. URL: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/romulus-spec-final.pdf (visited on 04/06/2022).

[Hir06] Shoichi Hirose. *How to Construct Double-Block-Length Hash Functions.* 2006. URL: https://csrc.nist.rip/groups/ST/hash/documents/HIROSE_article.pdf.

[Int18] Intel. *IntelXeon.* 2018. URL: https://ark.intel.com/content/www/us/en/ark/products/93805/intel-xeon-processor-e5-4669-v4-55m-cache-2-20-ghz.html (visited on 04/06/2022).

[JNP14] Jérémy Jean, Ivica Nikolić, and Thomas Peyrin. "Tweaks and Keys for Block Ciphers: The TWEAKEY Framework". In: *Advances in Cryptology – ASIACRYPT 2014.* Ed. by Palash Sarkar and Tetsu Iwata. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. ISBN: 978-3-662-45608-8.

[Mer79] Ralph Charles Merkle. *Secrecy, Authentication, and public key systems.* 1979. URL: http://www.merkle.com/papers/Thesis1979.pdf.

[Nat15] National Institute of Standards and Technology. *Secure Hash Standard.* 2015. URL: https://doi.org/10.6028/NIST.FIPS.180-4.

[Nik13] NikolajBjørner. *Z3Prover/z3.* 2013. URL: https://github.com/Z3Prover/z3.

[Wet16] Jos Wetzels. "Open Sesame: The Password Hashing Competition and Argon2". In: (Feb. 2016).

[WYY05] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. "Finding Collisions in the Full SHA-1". In: *Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings.* Ed. by Victor Shoup. Vol. 3621. Lecture Notes in Computer Science. Springer, 2005, pp. 17–36. DOI: 10.1007/11535218.

Now some words for my loved ones...

Abschließend möchte ich zu aller erst meiner Mutter danken. Vielen Dank für dein offenes Ohr, deine Ratschläge und vor allem die Ratschläge, die du mir schlussendlich nicht gegeben hast, damit ich daran wachsen kann. Du hast mir stets die Möglichkeit gegeben selbst an mir zu arbeiten und meinen eigenen Weg zu finden und dennoch mich in die richtige Richtung zu leiten. Vielen Dank für das Vertrauen und für alles was du je für mich gemacht hast.

Vorrei ringraziare anche mio padre. Attraverso te ho trovato la mia passione per l'informatica. Sará stata la prima tabella Excel che abbiamo creato, oppure le diverse ore trascorse insieme con giochi impegnativi? Non ne sono sicuro... Una cosa che peró so per certo, é che tu hai dato inizio alla mia scelta di intraprendere questa strada. Ti ringrazio molto per avermi finanziato e sostenuto in tutti questi anni e soprattutto di avermi sempre dato la sensazione di essere speciale e di essere in grado di affrontare ogni situazione.

Ohne meiner Schwester wäre die Familie nicht komplett und ich möchte mich auch bei dir bedanken. Vielen Dank für all die schweren Steine die du von meinem Weg beseitigt hast. Jede Last und jedes Problem das für andere auch noch so unbedeutend war, hast du stets wahrgenommen und bist mir beiseite gestanden. Danke, dass du mir mein Leben erleichtert hast.

Zu guter letzt möchte ich mich bei meiner Freundin Kathy bedanken. Danke für deine künstlerische Hilfe und Beratung. Danke, dass du stets hinter mir stehst und mich auch durch schwere Zeiten begleitest. Und Danke, dass du mich zu dem Menschen machst, der ich bin.