University of Technology, Graz

Master Thesis

Differential cryptanalysis with SAT solvers

Author:
Lukas Prokop

Supervisor:
Maria Eichlseder
Florian Mendel

A thesis submitted in fulfillment of the requirements for the master's degree in Computer Science

at the

Institute of Applied
Information Processing and
Communications

August 28, 2016

Lukas Prokop, BSc BSc

Differential cryptanalysis with SAT solvers

MASTER'S THESIS

to achieve the university degree of

Master of Science

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Dipl.-Ing. Dr.techn., Florian Mendel

Institute of Applied Information Processing and Communications

Second advisor: Maria Eichlseder

Graz, June 2016

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis dissertation.

Date

Signature

Abstract

Hash functions are ubiquitous in the modern information age. They provide preimage, second preimage and collision resistance which are needed in a wide range of applications.

In August 2006, Wang et al. showed efficient attacks against several hash function designs including MD4, MD5, HAVAL-128 and RIPEMD. With these results differential cryptanalysis has been shown useful to break collision resistance in hash functions. Over the years, advanced attacks based on those differential approaches have been developed.

To find collisions like Wang et al., a cryptanalyst needs to specify a differential characteristic as starting point, whose differences cancel out in the output. This starting point has a huge impact on the runtime. Once such a differential characteristic was discovered, in a second step the actual values for those differences are found yielding an actual hash collision.

Evaluating differential characteristics can be a cumbersome and tedious task. Dedicated tools can implement search heuristics. Performance shortcuts can be made by studying the hash algorithm's differential behavior, etc.

SAT solvers inherently implement a search heuristic and should derive those shortcuts on their own. In this thesis, we use SAT solvers to solve differential cryptanalysis problems. We show that SAT solvers are, in general, not able to derive those shortcuts on their own, but on the other hand we discuss approaches which significantly improve the runtime. SAT encoding remains an important topic to improve SAT solving runtimes.

We wrote a library generating CNFs for differential characteristics and optionally modify the CNF for certain SAT encodings. Those modifications allowed a significant runtime improvement helping us to solve full-rounds hash collisions in MD4 and 24-rounds hash collisions in SHA-256. Finally, we also provide a small CNF analysis library to compare encoded problems with each other.

Keywords: hash function, differential cryptanalysis, differential characteristic, MD4, SHA-256, collision resistance, satisfiability, SAT solver

Acknowledgements

First of all I would like to thank my academic advisor for his continuous support during this project. Many hours of debugging were involved in writing this master thesis project, but thanks to Florian Mendel, this project came to a release with nice results. Also thanks for continuously reviewing this document.

I would also like to thank Maria Eichlseder for her great support. Her unique way to ask questions brought me back on track several times. Mate Soos supported me during my bachelor thesis with SAT related issues and his support continued with this master thesis in private conversations.

Also thanks to Roderick Bloem and Armin Biere who organized a meeting one year before submitting this work defining the main approaches involved in this thesis. Armin Biere released custom lingeling versions for us, which allowed us to influence the guessing strategy in lingeling. He also shared his thoughts about our testcases with us.

And finally I am grateful for the support by Martina, who also supported me during good and bad days with this thesis, and my parents which provided a prosperous environment to me to be able to stand where I am today.

Thank you.
どもありがとうございました。

All source codes are available at lukas-prokop.at/proj/megosat and published under terms and conditions of Free/Libre Open Source Software. This document was printed with LuaLaTeX in the Linux Libertine typeface.

Contents

I

Chapter 1

Introduction

1.1 Overview

Hash functions are used as cryptographic primitives in many applications and protocols. They take an arbitrary input message and provide a hash value. Input message and hash value are considered as byte strings in a particular encoding. The hash value is of fixed length and satisfies several properties which make it useful in a variety of applications.

In this thesis, we consider the hash algorithms MD4 and SHA-256. Our goal is to find hash collisions using differential cryptanalysis. We define differences between two messages and determine actual bits such that the two messages result in the same hash value.
This whole equation system will be modelled as a satisfiability problem. A SAT solver reports satisfiability if and only if the particular differences can be resolved and an actual hash collision is found. We introduce a bit condition notation which allows us to visualize such differential states. Verification is done by several SAT solvers and we compare their runtime. Because the Boolean functions modelled in a CNF have a major influence on the runtime, we investigate several approaches and compare them.

Based on experience with these kind of problems with previous heuristic search tools we aim to apply best practices to a satisfiability setting. We will discuss, which SAT techniques lead to best performance characteristics for our MD4 and SHA-256 testcases.

1.2 Thesis Outline

This thesis is organized as follows:

In Chapter 1, we briefly introduce basic subjects of this thesis. We explain our high-level goal involving hash functions and SAT solvers.

In Chapter 2, we introduce the MD4 and SHA-256 hash functions. Certain design decisions imply certain properties which can be used in differential cryptanalysis. We discuss those decisions in this chapter after a formal definition of the function itself. Beginning with this chapter we develop a theoretical notion of our tools.

In Chapter 3, we discuss approaches of differential cryptanalysis. We start with work done by Wang, et al. and followingly introduce differential notation to simplify representation of differential states. This way we can easily dump hash collisions.

In Chapter 4, we discuss SAT solving techniques. We discuss how the problem needs to be encoded and give a brief overview over used SAT solvers. This includes a customized lingeling version by Armin Biere for our purposes.

In Chapter 5, we define SAT features which help us to classify SAT problems. This is a small subproject we did to look at properties of resulting DIMACS CNF files.

In Chapter 6, we discuss how we represent a problem (i.e. the hash function and a differential characteristic) as SAT problem. This ultimatively allows us to solve the problem using a SAT solver.

In Chapter 7, we present the result of our work. Runtimes are the main part of this chapter, but also results of Chapter 5 are presented.

In Chapter 8, we conclude and discuss future work based on our results.

Chapter 2

Hash algorithms

In this chapter, we will define hash functions and their desired security properties. In the following, we look at SHA-256 and MD4 as established hash functions. MD4 unlike SHA-256 is practically broken, but has a comparably small internal state. It therefore allows a good starting point to devise our attacks. In a next step, we scaled up to SHA-256 which has an internal state size at least twice as large. In Chapter 6, we will represent them with Boolean algebra to allow to reason about states in those hash functions using SAT solvers.

2.1 Preliminaries Redux

Definition 2.1 (*Hash function*)

A *hash function* is a mapping $h : X \to Y$ with $X = \{0,1\}^*$ and $Y = \{0,1\}^n$ for some fixed $n \in \mathbb{N}_{\geqslant 1}$.

- Let $x \in X$, then $h(x)$ is called *hash value of* x.
- Let $h(x) = y \in Y$, then x is called *preimage of* y.

Hash functions are considered as cryptographic primitives used as building blocks in cryptographic protocols. A hash function has to satisfy the following security requirements:

Definition 2.2 (*Preimage resistance*)

Given $y \in Y$, a hash function h is *preimage resistant* iff it is computationally infeasible to find $x \in X$ such that $h(x) = y$.

Definition 2.3 (*Second-preimage resistance*)

Given $x \in X$, a hash function h is *second-preimage resistant* iff it is computationally infeasible to find $x_2 \in X$ with $x \neq x_2$ such that $h(x) = h(x_2)$. x_2 is called *second preimage*.

Definition 2.4 (*Collision resistance*)

A hash function h is *collision resistant* iff it is computationally infeasible to find any two $x \in X$ and $x_2 \in X$ with $x \neq x_2$ such that $h(x) = h(x_2)$. Tuple (x, x_2) is called *collision*.

As far as hash functions accept input strings of arbitrary length, but return a fixed size output string, existence of collisions is unavoidable [31]. However, good hash functions make it very difficult to find collisions or preimages.

Any digital data can be hashed (i.e. used as input to a hash function) by considering it in binary representation. The format or encoding is not part of the hash function's specification.

2.1.1 Merkle-Damgård designs

The Merkle-Damgård design is a particular design of hash functions providing the following security guarantee:

Definition 2.5 (*Collision resistance inheritance*)

Let F_0 be a collision resistant compression function. A hash function in Merkle-Damgård design is collision resistant if F_0 is collision resistant.

This motivates thorough research of collisions in compression functions. The design was found independently by Ralph C. Merkle and Ivan B. Damgård. It was described by Merkle in his PhD thesis [17, p. 13–15] and used in popular hash functions such as MD4, MD5 and the SHA2 hash function family.

The single-pipe design works as follows:

1. Split the input into blocks of uniform block size. If necessary, apply padding to the last block to achieve full block size.
2. Compression function F_0 is applied iteratively using the output y_{i-1} of the previous iteration and the next input block x_i, denoted $y_i = F_0(y_{i-1}, x_i)$.
3. An optional postprocessing function is applied.

2.1.2 Padding and length extension attacks

Hash functions of single-piped Merkle-Damgård design inherently suffer from length extension attacks. MD4 and SHA-256 apply padding to their input to normalize their input size to a multiple of its block size. The compression function is applied afterwards. This design is vulnerable to length extensions.

Consider some collision (x_0, x_1) with $F_0(x_0) = y = F_0(x_1)$ where x_0 and x_1 have a size of one block. Let p be a suffix with size of one block. Then also $(x_0 \,\|\, p, x_1 \,\|\, p)$ (where $\|$ denotes concatenation) represents a collision in single-piped Merkle-Damgård designs, because it holds that:

$$F_0(F_0(x_0), p) = F_0(F_0(x_1), p) \iff F_0(y, p) = F_0(y, p)$$

Hence $(x_0 \,\|\, p, x_1 \,\|\, p)$ is a collision as well. As far as F_0 is applied recursively to every block, p can be of arbitrary size and (x_0, x_1) can be of arbitrary uniform size.

Because of this vulnerability, cryptanalysts focus on finding a collision in compression functions. In our tests will only consider input of one block and padding will be neglected due to this vulnerability.

2.1.3 Example usage

The following list describes application examples:

Digital signatures. Digital signatures are schemes to guarantee data and origin integrity. They are used to verify authenticity of PDF documents, encrypted emails or software distributions. A digital signature can be verified, showing that a particular user was involved in the signature process. No third party can forge a digital signature. Because the document itself might be too large, only its hash value is signed.

Storing passwords. User account data such as passwords are commonly stored in databases. Those databases are subject to theft if network breaches take place. Hashing and salting those passwords with a hash algorithm before storing them in the database is a working countermeasure.

Commitment schemes. Commitment schemes like Zero-Knowledge Proofs use hash algorithms to ensure the original message is possessed by a certain party without revealing the document itself. Coin-flipping or secure computation also use hash algorithms as cryptographic building blocks.

2.2 MD4

MD4 is a cryptographic hash function originally described in RFC 1186 [26], updated in RFC 1320 [27] and declared obsolete by RFC 6150 [34]. It was invented by Ronald Rivest in 1990 with properties given in Table 2.1. In 1995 [5], successful full-round attacks have been found to break collision resistance. Three years later, preimage and second-preimage resistance in MD4 got broken as well. Some of those attacks are described in [28] and [20]. We derived a Python 3 implementation based on a Python 2 implementation and made it available on github [24].

block size	512 bits	namely variable `block` in RFC 1320 [27]
digest size	128 bits	as per Section 3.5 in RFC 1320 [27]
internal state size	128 bits	namely variables A, B, C and D
word size	32 bits	as per Section 2 in RFC 1320 [27]

Table 2.1: MD4 hash algorithm properties

MD4 uses three auxiliary Boolean functions:

Definition 2.6

The Boolean `IF` function is defined as follows: If the first argument is true, the second argument is returned. Otherwise the third argument is returned.

The Boolean `MAJ` function returns true if the number of arguments with Boolean value true is in the majority. The Boolean `XOR` function returns true if the number of arguments with Boolean value true is odd.

Using the logical operators $\land$ (`AND`), $\lor$ (`OR`) and $\neg$ (`NEG`) we can define them as (see Section 4.1 for a thorough discussion of these operators):

$$\texttt{IF}(X, Y, Z) \coloneqq (X \land Y) \lor (\neg X \land Z) \tag{2.1}$$

$$\texttt{MAJ}(X, Y, Z) \coloneqq (X \land Y) \lor (X \land Z) \lor (Y \land Z) \tag{2.2}$$

$$\begin{aligned} \texttt{XOR}(X, Y, Z) &\coloneqq (X \land \neg Y \land \neg Z) \lor (\neg X \land Y \land \neg Z) \\ &\lor (\neg X \land \neg Y \land Z) \lor (X \land Y \land Z) \\ &\coloneqq (X \oplus Y \oplus Z) \end{aligned} \tag{2.3}$$

In the following, a brief overview of MD4 is given.

Padding and length extension. First of all, padding is applied. A single bit 1 is appended to the input. As long as the input does not reach a length congruent 448 modulo 512, bit 0 is appended. Afterwards, length appending takes place. Append the first 64 less significant bits of the input length (without the previous modifications) represented in binary.

Initialization. The message is split into 512-bit blocks (i.e. 16 32-bit words). State variables A_i with $-4 \leqslant i < 0$ are initialized with these hexadecimal values:

$$A_{-4} = \texttt{01234567} \quad A_{-1} = \texttt{89abcdef} \quad A_{-2} = \texttt{fedcba98} \quad A_{-3} = \texttt{76543210}$$

Round function and state variable updates. The round function is applied in 3 rounds with 16 iterations. In every iteration, values A_{-1}, A_{-2} and A_{-3} are taken as arguments to function F. Function F is `IF` in round 1, followed by `MAJ` for round 2 and `XOR` for the final round 3. The resulting value is added to A_{-1}, current message block M and constant X. Finally, the 32-bit sum will be left-rotated by p positions. Left rotation is formally defined in Definition 2.7. We define X and p:

Let i be the iteration counter between 1 and 16 and r the round between 1 and 3. Then X takes the value of the i-th column and r-th row of matrix C. p takes the value of row r and column i *mod* 4 of matrix P.

$$C = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ 0 & 4 & 8 & 12 & 1 & 5 & 9 & 13 & 2 & 6 & 10 & 14 & 3 & 7 & 11 & 15 \\ 0 & 8 & 4 & 12 & 2 & 10 & 6 & 14 & 1 & 9 & 5 & 13 & 3 & 11 & 7 & 15 \end{pmatrix}$$

$$P = \begin{pmatrix} 3 & 7 & 9 & 11 \\ 3 & 5 & 9 & 13 \\ 3 & 9 & 11 & 15 \end{pmatrix}$$

This round function design is visualized in Figure 2.1.

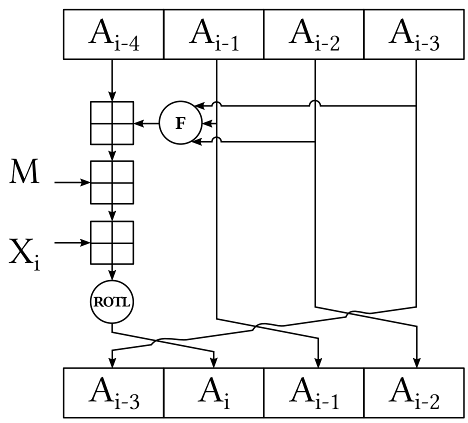


Figure 2.1: MD4 round function updating state variables

2.3 SHA-256

SHA-256 is a hash function from the SHA-2 family designed by the National Security Agency (NSA) and published as FIPS PUB 180-4 (originally 2001) [9]. It uses a Merkle-Damgård construction with a Davies-Meyer compression function. The best known preimage attack was found in 2011 and breaks preimage resistance for 52 rounds [11]. The best known collision attack breaks collision resistance for 31 rounds of SHA-256 [15] and pseudo-collision resistance for 46 rounds [12]. The best practical attack is a pseudo-collision for 38 steps [16].

block size	512 bits	as per Section 1 of the standard [9]
digest size	256 bits	mentioned as Message Digest size [9]
internal state size	256 bits	as per Section 1 of the standard [9]
word size	32 bits	as per Section 1 of the standard [9]

TABLE 2.2: SHA-256 hash algorithm properties

Definition 2.7 (*Shifts, rotations and a notational remark*)

Consider a 32-bit word X with 32 binary values b_i with $0 \leqslant i \leqslant 31$. b_0 refers to the least significant bit. Shifting ($\ll$ and $\gg$) and rotation ($\lll$ and $\ggg$) creates a new 32-bit word Y with 32 binary values a_i. We define the following notations:

$$\begin{aligned}
Y := X \ll n &\iff a_i := b_{i-n} \text{ if } 0 \leqslant i - n < 32 \text{ and } 0 \text{ otherwise} \\
Y := X \gg n &\iff a_i := b_{i+n} \text{ if } 0 \leqslant i + n < 32 \text{ and } 0 \text{ otherwise} \\
Y := X \lll n &\iff a_i := b_{i-n \ mod \ 32} \text{ as used in MD4} \\
Y := X \ggg n &\iff a_i := b_{i+n \ mod \ 32}
\end{aligned}$$

Besides MD4's `MAJ` and `IF`, another four auxiliary functions are defined. Recognize that $\oplus$ denotes the `XOR` function whereas $\boxplus$ denotes 32-bit addition.

$$\begin{array}{rcccccc}
\Sigma_0(X) & := & (X \ggg 2) & \oplus & (X \ggg 13) & \oplus & (X \ggg 22) \\
\Sigma_1(X) & := & (X \ggg 6) & \oplus & (X \ggg 11) & \oplus & (X \ggg 25) \\
\sigma_0(X) & := & (X \ggg 7) & \oplus & (X \ggg 18) & \oplus & (X \gg 3) \\
\sigma_1(X) & := & (X \ggg 17) & \oplus & (X \ggg 19) & \oplus & (X \gg 10)
\end{array}$$

Padding and length extension. The padding and length extension scheme of MD4 is used also in SHA-256. Append bit 1, followed by a sequence of bit 0 until the message reaches a length of 448 modulo 512 bits. Afterwards the first 64 bits of the binary representation of the original input are appended.

Initialization. In a similar manner to MD4, initialization of internal state variables (called "working variables" in [9, Section 6.2.2]) takes place before running the round function. The eight state variables are initialized with the following hexadecimal values:

$$A_{-1} = \texttt{6a09e667} \quad A_{-2} = \texttt{bb67ae85} \quad A_{-3} = \texttt{3c6ef372} \quad A_{-4} = \texttt{a54ff53a}$$
$$E_{-1} = \texttt{510e527f} \quad E_{-2} = \texttt{9b05688c} \quad E_{-3} = \texttt{1f83d9ab} \quad E_{-4} = \texttt{5be0cd19}$$

Furthermore SHA-256 uses 64 constant values in its round function. We initialize step constants K_i for $0 \leqslant i < 64$ with the following hexadecimal values (which must be read left to right and top to bottom):

428a2f98	71374491	b5c0fbcf	e9b5dba5	3956c25b	59f111f1
923f82a4	ab1c5ed5	d807aa98	12835b01	243185be	550c7dc3
72be5d74	80deb1fe	9bdc06a7	c19bf174	e49b69c1	efbe4786
0fc19dc6	240ca1cc	2de92c6f	4a7484aa	5cb0a9dc	76f988da
983e5152	a831c66d	b00327c8	bf597fc7	c6e00bf3	d5a79147
06ca6351	14292967	27b70a85	2e1b2138	4d2c6dfc	53380d13
650a7354	766a0abb	81c2c92e	92722c85	a2bfe8a1	a81a664b
c24b8b70	c76c51a3	d192e819	d6990624	f40e3585	106aa070
19a4c116	1e376c08	2748774c	34b0bcb5	391c0cb3	4ed8aa4a
5b9cca4f	682e6ff3	748f82ee	78a5636f	84c87814	8cc70208
90befffa	a4506ceb	bef9a3f7	c67178f2		

Precomputation of W. Let W_i for $0 \leqslant i < 16$ be the sixteen 32-bit words of the padded input message. Then compute W_i for $16 \leqslant i < 64$ the following way:

$$W_i := \sigma_1(W_{i-2}) + W_{i-7} + \sigma_0(W_{i-15}) + W_{i-16}$$

Round function. For every block of 512 bits, the round function is applied. The eight state variables are updated iteratively for i from 0 to 63. W_i and K_i refer to the previously initialized values.

$$E_i := A_{i-4} + E_{i-4} + \Sigma_1(E_{i-1}) + \texttt{IF}(E_{i-1}, E_{i-2}, E_{i-3}) + K_i + W_i$$
$$A_i := E_i - A_{i-4} + \Sigma_0(A_{i-1}) + \texttt{MAJ}(A_{i-1}, A_{i-2}, A_{i-3})$$

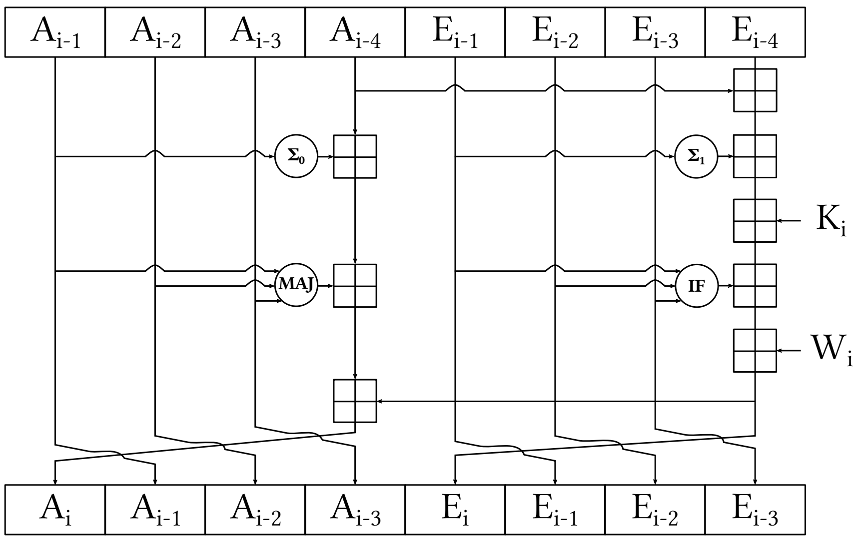
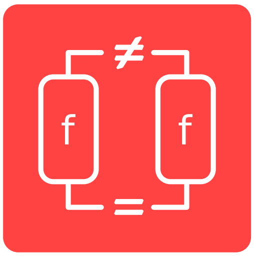


Figure 2.2: SHA-256 round function [6]



"Just because it's automatic doesn't mean it works."
—Daniel J. Bernstein

Chapter 3

Differential cryptanalysis

In Chapter 2, we defined two hash functions. In this chapter, we consider such functions from a differential perspective. Differential cryptanalysis will turn out to yield successful collision attacks on hash functions. We introduce a notation to easily represent differential characteristics.

3.1 Motivation

In August 2004, Wang et al. published results at Crypto'04 [35] which revealed that MD4, MD5, HAVAL-128 and RIPEMD can be broken practically using differential cryptanalysis. Their work is based on preliminary work by Hans Dobbertin [5]. On an IBM P690 machine, an MD5 collision can be computed in about one hour using this approach. Collisions for HAVAL-128, MD4 and RIPEMD were found as well. Patrick Stach's `md4coll.c` program [33] implements Wang's approach and can find MD4 collisions in few seconds on my Thinkpad x220 setup specified in Appendix A.

Let n denote the digest size, i.e. the size of the hash value $h(x)$ in bits. Due to the birthday paradox, a collision attack has a generic complexity of $2^{n/2}$ whereas preimage and second preimage attacks have generic complexities of 2^n. In other words it is computationally easier to find any two colliding hash values than the preimage or second preimage for a given hash value.

Following results by Wang et al., differential cryptanalysis was shown as powerful tool for cryptanalysis of hash algorithms. This thesis applies those ideas to satisfiability approaches.

Message 1			
4d7a9c83	**d6cb927a**	**29d5a578**	57a7a5ee
de748a3c	dcc366b3	b683a020	3b2a5d9f
c69d71b3	f9e99198	d79f805e	a63bb2e8
45dc8e31	97e31fe5	2794bf08	b9e8c3e9
Message 2			
4d7a9c83	**56cb927a**	**b9d5a578**	57a7a5ee
de748a3c	dcc366b3	b683a020	3b2a5d9f
c69d71b3	f9e99198	d79f805e	a63bb2e8
45dd8e31	97e31fe5	2794bf08	b9e8c3e9
Hash value of Message 1 and Message 2			
5f5c1a0d	71b36046	1b5435da	9b0d807a

TABLE 3.1: One of two MD4 hash collisions provided in [35]. A message represents one block of 512 bits. Values are given in hexadecimal, message words are enumerated from left to right, top to bottom. Differences are highlighted in bold for illustration purposes. For comparison the first bits of Message 1 are `11000001`… and the last bits are …`10011101`.

3.2 Fundamentals

Definition 3.1 (*Hash collision*)

Given a hash function h, a hash collision is a pair (x_1, x_2) with $x_1 \neq x_2$ such that $h(x_1) = h(x_2)$.

Pseudo-collisions are also often considered when attacking hash functions. A *pseudo-collision* is given if a hash collision can be found for a given hash function, but the initial vectors (IV) can be chosen for each message.

Hash algorithms consume input values as blocks of bits. As far as the length of the input must not conform to the block size, padding is applied. Now consider such a block of input values and another copy of it. We use those two blocks as inputs for two hash algorithm instances, but provide slight modifications in few bits. Differential cryptanalysis is based on the idea to consider those execution states and trace those differences to learn about the propagation of message differences. Compare this setup with Figure 3.1.

At the very beginning only the few defined differences are given. But as the hash algorithm progresses in computation, differences are propagated to more and more bits. Most likely the final value will differ in many bits, because of a desirable hash algorithm property called *avalanche effect.* A small difference in the input should lead to a significant difference in the output (i.e. visually recognizable).

Visualizing those differences helps the cryptanalyst to find modifications yielding a small number of differences in the evaluation state. Empirical results in

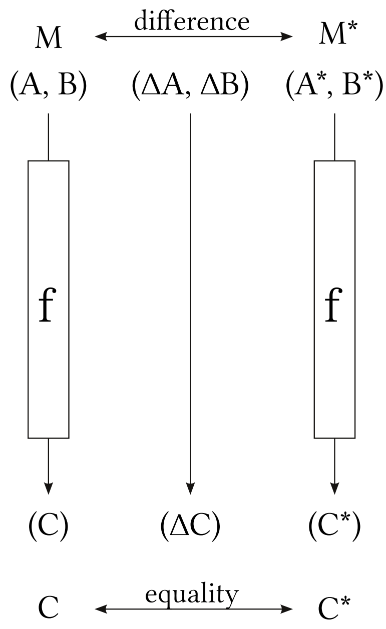


Figure 3.1: Common attack setting for a collision attack: Hash function f is applied to two inputs M and M^* which differ by some predefined bits. ΔM describes the difference between these values. A hash collision is given if and only if output values C and C^* show the same value. In differential cryptanalysis we observe the differences between two instances applying function f to inputs M and M^*.

differential cryptanalysis indicate that sparse characteristics are desirable, because it is easier to cancel out few differences in the output compared to many differences. The cryptanalyst consecutively modifies the input values to eventually receive a collision in the output value (i.e. $\Delta C = 0 \iff C = C^*$).

Definition 3.2

The differential state during a computation is called *differential characteristic* (also *differential path*).

Theorem 3.1

Assuming the number of differences in a differential characteristic is small, this characteristic is expected to result in a hash collision with higher probability.

3.3 Differential notation

Differential notation helps us to visualize differential characteristics by defining so-called *generalized bit conditions*. It was introduced by Christophe de Cannière and Christian Rechberger in 2006 [2, Section 3.2], inspired by *signed differences* by Wang et al. and is shown in Table 3.2.

(x_i, x_i^*)	$(0,0)$	$(1,0)$	$(0,1)$	$(1,1)$	(x_i, x_i^*)	$(0,0)$	$(1,0)$	$(0,1)$	$(1,1)$
?	✓	✓	✓	✓	3	✓	✓		
-	✓			✓	5	✓		✓	
x		✓	✓		7	✓	✓	✓	
0	✓				A		✓		✓
u		✓			B	✓	✓		✓
n			✓		C			✓	✓
1				✓	D	✓		✓	✓
#					E		✓	✓	✓

Table 3.2: Differential notation as introduced in [2]. The left-most column specifies a symbol called “bit condition” and right-side columns indicate which bit configurations are possible for two given bits x_i and x_i^*.

Consider two hash algorithm instances. Let x_i be some bit from the first instance and let x_i^* be the corresponding bit from the second instance. Differences are computed using a XOR and commonly denoted as $\Delta x = x_i \oplus x_i^*$. Bit conditions allow us to encode possible relations between bits x_i and x_i^*.

For example, let us take a look at the original Wang et al. hash collision in MD4 provided in Table 3.1. We extract all values with differences and represent them using differential notation. This gives us Table 3.3.

bit	hexadecimal	binary representation / differential notation
x_0	d6cb927a	11010110110010111001001001111010
x_1	29d5a578	00101001110101011010010101111000
x_2	45dc8e31	01000101110111001000111000110001
x_0^*	56cb927a	01010110110010111001001001111010
x_1^*	b9d5a578	10111001110101011010010101111000
x_2^*	45dd8e31	01000101110111011000111000110001
Δx		u1010110110010111001001001111010
		n01n1001110101011010010101111000
		010001011101110n1000111000110001

Table 3.3: The three words different between Message 1 and Message 2 of the original MD4 hash collision by Wang et al. The last three lines show how differences can be written down using bit conditions. As far as 4 symbols are not from the set $\{0, 1\}$ it holds that the messages differ by 4 bits.

The following properties hold for bit conditions:

- If $x_i = x_i^*$ holds and some value is known, $\{0, 1\}$ contains its bit condition.
- If $x_i \neq x_i^*$ holds and some value is known, $\{u, n\}$ contains its bit condition.
- If $x_i = x_i^*$ holds and the values are unknown, its bit condition is `-`.
- If $x_i \neq x_i^*$ holds and the values are unknown, its bit condition is `x`.

Applying this notation to hash collisions means that arbitrary bit conditions (except for #) can be specified for the input values. In one of the intermediate iterations, we enforce a difference using one of the bit conditions $\{u, n, x\}$. This excludes trivial solutions with no differences from the set of possible solutions. And the final values need to lack differences thus are represented using a dash `-`.

Δx	conjunctive normal form	Δx	conjunctive normal form
#	$(x) \wedge (\neg x)$	1	$(x) \wedge (x^*)$
0	$(\neg x) \wedge (\neg x^*)$	-	$\neg(x \oplus x^*)$
u	$(x) \wedge (\neg x^*)$	A	(x)
3	$(\neg x^*)$	B	$(x \vee \neg x^*)$
n	$(\neg x) \wedge (x^*)$	C	(x^*)
5	$(\neg x)$	D	$(\neg x \vee x^*)$
x	$(x \oplus x^*)$	E	$(x \vee x^*)$
7	$(\neg x \vee \neg x^*)$	?	

Table 3.4: All bit conditions represented as CNF using two Boolean variables x and x^* to represent two bits.

3.4 A simple addition example

Using this notation, we can now reason about the behavior of functions on differential values. We start with 1-bit addition as basic exercise to the reader. Consider a matrix with two input rows and one output row. The values of the first two rows are added such that the bit difference at the third row is created.

-		00	00	00	00	11	11	11	11
-	$\Rightarrow$	00	00	11	11	00	00	11	11
-		00	11	00	11	00	11	00	11

Figure 3.2: A simple 1-bit addition example: On the left the differential characteristic is given. Two dashes, by definition, denote a missing difference in both arguments. The result of the addition must never show a difference. This yields eight possible bit configurations where two values close to each other denote (M, M^*) of Figure 3.1. Due to the behavior of addition, we know that configurations 2, 3, 5 and 8 (from left to right) are invalid.

Figure 3.2 illustrates this example. Remember that symbols such as `-` and `0` underlie semantics defined in Table 3.2. It is also interesting to see how propagation of values can work. In Figure 3.3 we see how an underspecified value `?` can be strengthened once we have checked which values can be taken. Recognize that the system is constrained by the function in use and the definition of the differential symbols.

$$\begin{array}{c} \texttt{-} \\ \texttt{-} \\ \hline \texttt{?} \end{array} \quad \Rightarrow \quad \begin{array}{cccc} \texttt{00} & \texttt{00} & \texttt{11} & \texttt{11} \\ \texttt{00} & \texttt{11} & \texttt{00} & \texttt{11} \\ \hline \texttt{00} & \texttt{11} & \texttt{11} & \texttt{00} \end{array}$$

Figure 3.3: Like Figure 3.2, but any difference value for the result bit is possible. As such we consider any possible bit configuration, but eventually recognize that only four bit configurations are consistent with the behavior of addition. Because all resulting configurations show no bit difference in the output bit, we can strengthen `?` by replacing it with `-`. This illustrates how knowledge about differential states can be propagated.

Finally, we can extend our testcases to 4 bits and retrieve testcases such as Figures 3.4 and 3.5.

A:	`0011`	A:	`---x`	A:	`---x`	A:	`---x`
B:	`0101`	B:	`---x`	B:	`---x`	B:	`---x`
S:	`1000`	S:	`????`	S:	`???-`	S:	`x???`

A:	`0011`	A:	`---x`	A:	`----`
B:	`0101`	B:	`---x`	B:	`---x`
S:	`0000`	S:	`???x`	S:	`x-??`

Figure 3.4: Testcases for 4-bit addition: The upper line shows valid differential characteristics for 4-bit addition whereas the lower line show invalid ones for 4-bit addition. The rows are conventionally named using capital letters.

A:	`----`	A:	`7C-3`	A:	`0uCD`
S:	`0000`	S:	`-3u?`	S:	`ADC7`

A:	`---x`	A:	`xxxx`
S:	`0000`	S:	`0000`

Figure 3.5: Differential characteristics for the SHA-2 Sigma function. The upper line shows valid states. The lower line shows invalid ones.

3.5 Differential characteristics in action

In the previous section, we illustrated how propagation with differential values works and how differential characteristics are written down. It is always important to keep in mind which function the characteristic illustrates, because this is not documented with the characteristic.

Now consider MD4 as defined in Section 2.2. MD4 takes some input message (in our case limited to size of one block), the state variables are initialized and iteratively new A_i are computed.

Similarly, SHA-256 takes a message block M and initializes eight variables with an initial vector (IV). The remaining W_i are computed and iteratively, values A_i and E_i are computed.

Those values are structured in differential characteristics illustrated in Figure 3.6. Those layouts are used to specify our hash collisions we want to evaluate. Table 3.5 also gives an application of the layout.

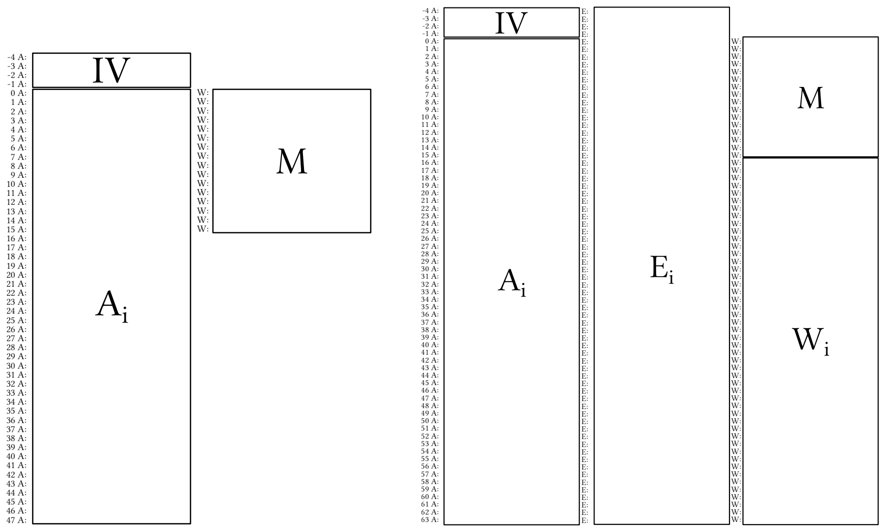


Figure 3.6: Layout of MD4 (left) and SHA-256 (right) differential characteristics

i		$\nabla S_{i,0}$	$\nabla S_{i,1}$	$\nabla S_{i,2}$
-4	A:	01100111010001010010001100000001		
-3	A:	00010000001100100101010001110110		
-2	A:	10011000101110101101110011111110		
-1	A:	11101111110011011010101110001001		
0	A:	01101011110101001110010000010010	W:	01001101011110101001110010000011
1	A:	0111011001001111111011100u110001	W:	u1010110110010111001001001111010
2	A:	101010110100000001110u01n1110010	W:	n01n1001110101011010010101111000
3	A:	101011u1001111010101001001010001	W:	01010111101001111010010111101110
4	A:	00101100011000110101010111110010	W:	11011110011101001000101000111100
5	A:	000110100110001010u1101000000001	W:	11011100110000110110011010110011
6	A:	0001101100unuu110001000001111010	W:	????????????????????????????????
7	A:	00101011100000010unn011001010000	W:	00111011001010100101110110011111
8	A:	????????????????????????????????	W:	11000110100111010111000110110011
9	A:	????????????????????????????????	W:	11111001111010011001000110011000
10	A:	10n0010010000101?????????????110	W:	11010111100111111000000001011110
11	A:	u100011010110110?????????????111	W:	10100110001110111011001011101000
12	A:	001011u00u101011?????????????011	W:	010001011101110n1000111000110001
13	A:	10un1n0100110001?????????????101	W:	10010111111000110001111111100101
14	A:	0000101001010001-------------110	W:	00100111100101001011111100001000
15	A:	0001111010101u01-------------100	W:	10111001111010001100001111101001
16	A:	n00n0un011010010-------------111		
17	A:	0001111100111010-------------110		
18	A:	0101011100001101-------------100		
19	A:	u1n1000000010111-------------100		
20	A:	n1un1001111111011101000000110100		
21	A:	11110011101100000101111111010100		
22	A:	01011101110011010011001100111010		
23	A:	01010000111011101100011110001111		
24	A:	00000010000100100011011100011010		
25	A:	10110000100101100001010011101010		
26	A:	00001010100010010111011101000001		
27	A:	00000110111011110101101010110011		
28	A:	10110110010111010110110000100101		
29	A:	10100010000011010100100001101001		
30	A:	00101001110101111100011101100011		
31	A:	11111100100100101101011110110110		
32	A:	01001111110100100110100000101111		
33	A:	00111000001111010110111011100100		
34	A:	00100000011101011110100000010101		
35	A:	n0100000001100110000010001110010		
36	A:	n0000111111010111101111001011001		
37	A:	11001000000110100100001100001100		
38	A:	10110000011001111110100110101100		
39	A:	00010010000010100001101100011100		
40	A:	11000000010010000111000110000101		
41	A:	00000110100001101111010100100110		
42	A:	01001110110111011111111010000110		
43	A:	01010000011000111101000001101101		
44	A:	11111000000101101111011100001100		
45	A:	10001010110110110010110000000100		
46	A:	10000010100110010101100011011100		
47	A:	10000001111001011011010010111101		

⇒

i		$\nabla S_{i,0}$	$\nabla S_{i,1}$	$\nabla S_{i,2}$
-4	A:	01100111010001010010001100000001		
-3	A:	00010000001100100101010001110110		
-2	A:	10011000101110101101110011111110		
-1	A:	11101111110011011010101110001001		
0	A:	01101011110101001110010000010010	W:	01001101011110101001110010000011
1	A:	0111011001001111111011100u110001	W:	u1010110110010111001001001111010
2	A:	101010110100000001110u01n1110010	W:	n01n1001110101011010010101111000
3	A:	101011u1001111010101001001010001	W:	01010111101001111010010111101110
4	A:	00101100011000110101010111110010	W:	11011110011101001000101000111100
5	A:	000110100110001010u1101000000001	W:	11011100110000110110011010110011
6	A:	0001101100unuu110001000001111010	W:	10110110100000111010000000100000
7	A:	00101011100000010unn011001010000	W:	00111011001010100101110110011111
8	A:	011100110010001u1111111110110000	W:	11000110100111010111000110110011
9	A:	101011n01unnu0001111100110011111	W:	11111001111010011001000110011000
10	A:	10n00100100001010100000010101110	W:	11010111100111111000000001011110
11	A:	u1000110101101100100101011111111	W:	10100110001110111011001011101000
12	A:	001011u00u1010111111110001111011	W:	010001011101110n1000111000110001
13	A:	10un1n01001100010100000111100101	W:	10010111111000110001111111100101
14	A:	00001010010100011000100011010110	W:	00100111100101001011111100001000
15	A:	0001111010101u010110011011010100	W:	10111001111010001100001111101001
16	A:	n00n0un0110100101001101101011111		
17	A:	00011111001110100001001000011110		
18	A:	01010111000011010000000010010100		
19	A:	u1n10000000101111001101011000100		
20	A:	n1un1001111111011101000000110100		
21	A:	11110011101100000101111111010100		
22	A:	01011101110011010011001100111010		
23	A:	01010000111011101100011110001111		
24	A:	00000010000100100011011100011010		
25	A:	10110000100101100001010011101010		
26	A:	00001010100010010111011101000001		
27	A:	00000110111011110101101010110011		
28	A:	10110110010111010110110000100101		
29	A:	10100010000011010100100001101001		
30	A:	00101001110101111100011101100011		
31	A:	11111100100100101101011110110110		
32	A:	01001111110100100110100000101111		
33	A:	00111000001111010110111011100100		
34	A:	00100000011101011110100000010101		
35	A:	n0100000001100110000010001110010		
36	A:	n0000111111010111101111001011001		
37	A:	11001000000110100100001100001100		
38	A:	10110000011001111110100110101100		
39	A:	00010010000010100001101100011100		
40	A:	11000000010010000111000110000101		
41	A:	00000110100001101111010100100110		
42	A:	01001110110111011111111010000110		
43	A:	01010000011000111101000001101101		
44	A:	11111000000101101111011100001100		
45	A:	10001010110110110010110000000100		
46	A:	10000010100110010101100011011100		
47	A:	10000001111001011011010010111101		

TABLE 3.5: One of the original MD4 collisions by Wang et al, with a few bits underspecified (left) and propagated values (right). The question marks indicate that any bit configuration for the two bits are possible. Dashes indicate that the bits have the same configuration in both instances, but the value itself is unknown. However, it turns out the description with missing values in iteration 6 (message) and iterations 8–19 is complete enough such that missing values can be deduced by other values and the description of the algorithm. A collision is given in the last 4 rounds, because no differences are left as they cancel out after round 36.

"What idiot called them logic errors rather than bool shit?"
—Unknown

Chapter 4

Satisfiability

Boolean algebra allows us to describe functions over two-valued variables. Satisfiability is the question for an assignment such that a function evaluates to true. Satisfiability problems are solved by SAT solvers. We discuss the basic theory behind satisfiability. Because any computation can be represented as satisfiability problem, we are able to verify whether an algorithm can reach a certain state. In Chapter 6, we will represent a differential cryptanalysis problem such that it is solvable iff the corresponding SAT problem is satisfiable.

4.1 Basic notation and definitions

Definition 4.1 (*Boolean function*)

A *Boolean function* is a mapping $h : X \to Y$ with $X = \{0, 1\}^n$ for $n \in \mathbb{N}_{\geqslant 1}$ and $Y = \{0, 1\}$.

Definition 4.2 (*Assignment*)

A *k-assignment* is an element of $\{0, 1\}^k$.

Let f be some k-ary Boolean function. An *assignment for function f* is any k-assignment.

Definition 4.3 (*Truth table*)

Let f be some k-ary Boolean function. The *truth table of Boolean function f* assigns truth value 0 or 1 to any assignment of f.

x_1	x_2	$f(x_1, x_2)$
1	1	1
1	0	0
0	1	0
0	0	0

(A) AND

x_1	x_2	$f(x_1, x_2)$
1	1	1
1	0	1
0	1	1
0	0	0

(B) OR

x	$f(x)$
1	0
0	1

(C) NOT

TABLE 4.1: Truth tables for AND, OR and NOT

Boolean functions are characterized by their corresponding truth table.

Table 4.1 shows example truth tables for the Boolean AND, OR and NOT functions. A different definition of the three functions is given the following way:

Definition 4.4

Let AND, OR and NOT be three Boolean functions.

- AND maps $X = \{0, 1\}^2$ to 1 if all values of X are 1.
- OR maps $X = \{0, 1\}^2$ to 1 if any value of X is 1.
- NOT maps $X = \{0, 1\}^1$ to 1 if the single value of X is 0.

All functions return 0 in the other case. Those functions are denoted $a_0 \wedge a_1$, $a_0 \vee a_1$ and $\neg a_0$ respectively, for input parameters a_0 and a_1.

It is interesting to observe, that any Boolean function can be represented using only these three operators. This can be proven by complete induction over the number of arguments k of the function.

Let $k = 1$. Then we consider any possible 2-assignment for one input variable x_1 and one value of $f(x_1)$. Then four truth tables are possible listed in Table 4.2. The description shows the corresponding definition of f using AND, OR and NOT only.

Now let g be some k-ary function. Let $(a_0, a_1, \dots, a_k)$ be the k input arguments to g and $x_1 := g(a_0, a_1, \dots, a_k)$. Then we can again look at Table 4.2 to discover that 4 cases are possible: 2 cases where the return value of our new $(k + 1)$-ary function depends on value x_1 and 2 cases where the return value is constant.

This completes our proof.

x_1	$f(x_1)$
1	1
0	1

(A) $f : x \mapsto 1$

x_1	$f(x_1)$
1	1
0	0

(B) $f : x \mapsto x$

x_1	$f(x_1)$
1	0
0	1

(C) $f : x \mapsto \neg x$

x_1	$f(x_1)$
1	0
0	0

(D) $f : x \mapsto 0$

TABLE 4.2: Unary f and its four possible cases

Boolean functions have an important property which is described in the following definition:

Definition 4.5

A Boolean function f is *satisfiable* iff there exists at least one input $x \in X$ such that $f(x) = 1$. Every input $x \in X$ satisfying this property is called *model*.

The corresponding tool to determine satisfiability is defined as follows:

Definition 4.6

A *SAT solver* is a tool to determine satisfiability (SAT or UNSAT) of a Boolean function. If satisfiability is given, it returns some model.

4.1.1 Computational considerations

The generic complexity of SAT determination is given by 2^n for n Boolean variables.

Let n be the number of variables of a Boolean function. No known algorithm exists to determine satisfiability in polynomial runtime. This means no algorithm solves the SAT problem with runtime behavior which depends polynomially on the growth of n. However, SAT solvers can take advantage of the problem's description. For example, consider function f:

$$f(x_0, x_1, x_2) = x_0 \wedge (\neg x_1 \vee x_2) \tag{4.1}$$

Instead of trying all possible 8 cases for 3 Boolean variables, we can immediately see that x_0 is required to be 1. So we don't need to test $x_0 = 0$ and can skip 4 cases. This particular strategy is called *unit propagation*.

4.1.2 SAT competitions

SAT research is heavily concerned with finding fast heuristics determining (un)satisfiability. Biyearly, SAT competitions [30] take place to challenge SAT solvers in a set of benchmarks. The committee evaluates the most successful SAT solvers defined by solving the most problems within a given time frame.

In 2014, lingeling by Armin Biere has won first prize in the Application benchmarks track and second prize in the Hard Combinatorial benchmarks track for SAT and UNSAT instances, respectively. Its parallelized sibling plingeling and Cube & Conquer sibling treengeling have won prizes in parallel settings. And in the most recent 2016 competition lingeling has won bronze in the Main track for SAT+UNSAT instances.

In Chapter 7, we will discuss runtime results shown by (but not limited to) those SAT solvers.

4.2 The DIMACS de-facto standard

Definition 4.7

A SAT problem is given in *Conjunctive Normal Form* (CNF) if the problem is defined as conjunction of disjunctions of literals.

A *conjunction* is a sequence of Boolean functions combined using a logical `AND`. A *disjunction* is a sequence of Boolean functions combined using a logical `OR`. A *literal* is a Boolean variable (*positive*) or its negation (*negative*).

A simple example for a SAT problem in CNF is the exclusive `OR` (`XOR`). It takes two Boolean values a and b as arguments and returns true if and only if the two arguments differ:

$$(a \vee b) \wedge (\neg a \vee \neg b) \tag{4.2}$$

It consists of one conjunction (denoted $\wedge$) of two disjunctions (denoted $\vee$) of literals (denoted a and b where prefix $\neg$ represents negation). This structure constitutes a CNF.

Analogously, we define a *Disjunctive Normal Form* (DNF) as disjunction of conjunctions of literals. The negation of a CNF is in DNF, because literals are negated and conjunctions become disjunctions, vice versa.

Theorem 4.1

Every Boolean function can be represented as CNF.

Theorem 4.1 is easy to prove. Consider the truth table of an arbitrary Boolean function f with k input arguments and j rows of output value false. We represent f as CNF.

Consider Boolean variables $b_{i,l}$ with $0 \leqslant i \leqslant j$ and $0 \leqslant l \leqslant k$. For every row i of the truth table with assignment (r_i), add one disjunction to the CNF. This disjunction contains $b_{i,l}$ if $r_{i,l}$ is false. The disjunction contains $b_{i,l}$ if $r_{i,l}$ is true.

As far as f is an arbitrary k-ary Boolean function, we have proven that any Boolean function can be represented as CNF.

SAT problems are usually represented in the DIMACS de-facto standard. Consider a SAT problem in CNF with *nbclauses* clauses and enumerate all variables from 1 to *nbvars*. A DIMACS file is an ASCII text file. Lines starting with “`c`” are skipped (comment lines). The first remaining line has to begin with “`p cnf`” followed by *nbclauses* and *nbvars* separated by spaces (header line). All following non-comment lines are space-separated indices of Boolean variables optionally prefixed by a hyphen. Then one line represents one clause and must be terminated with a zero character after a space. All lines are conjuncted to form a CNF.

Variations of the DIMACS de-facto standard also allow multiline clauses (the zero character constitutes the end of a clause) or arbitrary whitespace instead of spaces. Another variant terminates DIMACS files once it encounters a single percent sign on a line. The syntactical details are individually published on a per competition basis.

LISTING 4.1: CNF of the XOR in Display (4.2)

```
p cnf 2 2
a b
-a -b
```

4.3 Terminology

Given a conjunctive structure of disjunctions, we can define terms related to this structure. Those terms will be used in the SAT features we present in Section 5.4.

Definition 4.8

A *clause* is a disjunction of literals. A *k-clause* is a clause consisting of exactly k literals. A *unit clause* is a 1-clause.

A *Horn clause* is a clause with at most one positive literal. A *definite clause* is a clause with exactly one positive literal. A *goal clause* is a clause with no positive literal.

Definition 4.9

Given a literal, its *negated literal* is the literal with its sign negated. A literal is *positive*, if its sign is positive. A literal is *negative* if its sign is negative.

An *existential literal* is a literal which occurs exactly once and its negation does not occur. A *used variable* is a variable which occurs at least once in the CNF.

The *literal frequency* is the number of occurences of a literal in the CNF divided by the number of clauses declared. Equivalently *variable frequency* defines the number of variable occurences divided by the number of clauses declared.

Definition 4.10

The *clause length of a clause* is the number of literals contained. A clause is called *tautological* if a literal and its negated literal occurs in it.

A few basic properties hold in terms of satisfiability. For example, existential literals are interesting, because they can be set to true and make one clause immediately satisfied without influencing other clauses.

4.4 Basic SAT solving techniques

Definition 4.11

Given two CNFs A and B, they are called *equisatisfiable* iff A is satisfiable iff B.

4.4.1 Boolean constraint propagation (BCP)

One of the most basic techniques to SAT solving is *Boolean Constraint Propagation*, also called *unit propagation*. It is so fundamental that SATzilla, introduced in Section 5.2, applies it immediately before looking at SAT features.

Let l be the literal of a unit clause in a CNF. Remove any clause containing l and replace any occurences of $-l$ from the CNF. It is easy to see, that the resulting CNF is equisatisfiable, because due to the unit clause l must be true. So any clause containing l is satisfied and $-l$ yields false, where ($A \vee$ false) is equivalent to (A) for any Boolean function A.

4.4.2 Watched Literals

Watched Literals are another fundamental concept in SAT solving. It is very expensive to check satisfiability of all clauses for every assigned value of a literal. Watched Literals is a neat technique to reduce the number of checks.

In each clause two unassigned literals are declared to be "watched". Structurally it is implemented the other way around: A clauses watch list is maintained per literal. Now as long as at least two literals are unassigned, the clause cannot become false (recognize that a clause is false iff all literals are false). Therefore the clause does not need to be visited as long as at least one unassigned literal exist. This implies the following decision procedure:

- If all but one literal is false, propagate the remaining literal to be true.
- If all literals are false, report UNSAT.
- If any literal becomes true, watched literals do not change.
- Else replace the literal on the watch list with a remaining unassigned literal.

This empirical approach was established with the Chaff and zChaff SAT solvers [19] and has proven useful in various variants.

4.4.3 Remark

The previous two techniques illustrate basic approaches, but actual SAT solving research requires decades of development to tune individual SAT solvers. Memory models and concurrency strategies lead to fundamentally different runtime behaviors of SAT solvers.

As such, an initial idea to initiate an individual SAT solver specifically designed for solving problems in differential cryptanalysis was dropped, because development time is expected too long for a master thesis to be fruitful. As such we focused on popular and established SAT solvers of the SAT community.

4.5 SAT solvers in use

In this thesis, we considered several SAT solvers. They have been selected either by their popularity or their good results at previous SAT competitions:

- MiniSat 2.2.0
- CryptoMiniSat versions 4.5.3 and 5
- treengeling, lingeling and plingeling, in versions:
  - lingeling ats1
  - lingeling ats101
  - lingeling ats102
  - lingeling ats104
  - lingeling baz
- glucose version 4.0 and glucose syrup version 4.0

This means the hash collision attacks we implemented have run with these SAT solvers. The results are discussed in Chapter 7 and a more comprehensive list is provided in Appendix C.

MiniSat is known as "Swiss army knife of SAT solving" meaning that it includes many well-established techniques that can be built upon. SAT competitions 2009, 2011, 2013 and 2014 included a special MiniSat "hack track" where participants are asked to modify MiniSat to prove the best performance with as little change to the MiniSat codebase as possible. Even though is not one of the fastest SAT solvers today, it provides a nice codebase to experiment with.

CryptoMiniSat is a derivative of MiniSat, which was originally modified for cryptographic problems. It features XOR clauses meaning that binary clauses of structure $a \oplus b$ could be added and will be resolved using Gaussian elimination. Please recognize that our encoding introduced in Section 6.2 uses equivalence to model assignment and as such only clauses of structure $r = a \oplus b$ emerge rendering this feature impractical to use.

Glucose was the gold winner 2011 in the SAT+UNSAT application track. Modifications of glucose also ranked high throughout the years of SAT competition. Glucose is a sequential SAT solver whereas glucose syrup is its parallelized version.

Lingeling is SAT solver developed by Armin Biere. Lingeling has been the winner of several tracks in the SAT competitions 2011 to 2016. For example, it has won gold in the SAT+UNSAT application track in 2014. Lingeling has two siblings: plingeling and treengeling. plingeling is a parallelized version of lingeling. As such it executes in multiple threads and shares units and equivalences between those instances. treengeling is a Cube & Conquer solver meaning it partitions the problem into many subproblems and solves them individually.

Lingeling releases ats101, ats102 and ats104 are non-public, experimental releases of lingeling. They have been developed in private communication with Armin Biere. Our main goal was to achieve a separation between two sets of variables. First, all variables of the first need to be assigned in the best possible way. Afterwards, the second set of variables is considered. Specifically variables modelling the differences between the two hash algorithm instances should constitute the first set as discussed in Chapter 6.

Lingeling ats101 implements the strategy to guess difference variables first (with Boolean value false) and usual heuristics apply for all other variables. Our intermediate results with incomplete CNF files showed a high number of restarts. Therefore ats102 disables backjumping and therefore skips decisions for important variables. Finally ats104 is not expected to distinguish from ats102. It only provides further debugging information.

The SAT solvers have generally been run without any special options and several times, except for

- MiniSat was run with `pre=once` as it is generally recommended to run with the builtin preprocessor.
- Lingeling has been generally run with `phase=0` per default and `phase=-1` to prefer false as initial assignment to literals. However, lingeling ats101 implements this with a more forceful strategy.

Preprocessing is a difficult topic on its own. Sometimes preprocessing can provide a speedup, before actually solving the problem, but mostly SAT solvers implement preprocessing strategies themselves and run them repeatedly when solving the problem. Chapter 7 presents runtime results for that issue.

> “To be usable effectively [...] these features must be related to instance hardness and relatively cheap to compute”
> —SATzilla

Chapter 5

SAT features

At the very beginning, I was very intrigued by the question “What is an ‘average’ SAT problem?”. Answers to this question can help to optimize SAT solver memory layouts and find distinctive properties of CNFs. Specifically for this thesis, I wanted to find out whether our problems distinguish from “average” problems in any way such that we can use this distinction for runtime optimization.

I came up with 8 questions related to basic properties of SAT problems we will discuss in depth in this section. We will characterize an average SAT problem in Section 5.7:

1. Given an arbitrary literal. What is the percentage it is positive?
2. What is the clauses / variables ratio?
3. How many literals occur only once either positive or negative?
4. What is the average and longest clause length among CNF benchmarks?
5. How many Horn clauses exist in a CNF?
6. Are there any tautological clauses?
7. Are there any CNF files with more than one connected variable component?
8. How many variables of a CNF are covered by unit clauses?

We will now define the terms used in those questions.

5.1 Definition

Definition 5.1 (*SAT feature*)

A *SAT feature* is a statistical value (named *feature value*) retrievable from some given SAT problem.

The most basic example of a SAT feature is the number of variables and clauses of a given SAT problem. This SAT feature is stored in the CNF header of a SAT problem encoded in the DIMACS format.

The general goal is to write a tool which evaluates several SAT features at the same time and retrieves them for comparison with other problems. Therefore it should be computationally easy to evaluate SAT features of a given SAT problem. A suggested computational limit is given with polynomial complexity in terms of number of variables and number of clauses for memory as well as runtime. Sticking to this convention implies that evaluation of satisfiability must not be necessary to evaluate a SAT feature as long as no polynomial algorithm to determine satisfiability can be found. Hence the number of valid models cannot be a SAT feature as far as satisfiability needs to be determined. But no actual hard computational limit is defined.

5.2 Related work

The most similar resource I found—looking at SAT features—was the SATzilla project [22, 36] in 2012. The authors used 91 SAT features categorized in 9 groups, originally described by Nudelman, et al. [23]. Some features are only evaluated if they can be evaluated within a given time frame (e.g. 20 seconds).

The following list provides an excerpt of the features:

nvarsOrig number of variables defined in the CNF header

nvars number of active variables

reducedVars nvarsOrig reduced by nvars, divided by nvars

vars-clauses-ratio nvars divided by the number of active clauses

POSNEG-RATIO-CLAUSE-mean clause mean of $2 \cdot \|0.5 - \text{pos}/\text{length}\|$ where pos is the number of positive literals and length its clause length

POSNEG-RATIO-CLAUSE-entropy like POSNEG-RATIO-CLAUSE-mean but its Shannon entropy

TRINARY+ number of clauses with clause length 1, 2 or 3 divided by number of active clauses

HORNY-VAR-min minimum number of times a variable occurs in a Horn clause

cluster-coeff-mean let neighbors of a clause be all clauses containing any literal negated and let clauses c_1 and c_2 be conflicting if c_1 contains literal l and c_2 contains $-l$, then return the mean of two times the number of conflicting neighbors of a clause c divided by the number of unordered pairs of neighbors; returned iff computable within 20 seconds for all clauses

Please recognize that active clauses are the unsatisfied clauses after BCP has been applied. Equivalently active variables are remaining variables after application of BCP.

The SAT solvers we use (Section 4.5), also compute features they use when computing a solution. For example, CryptoMiniSat 4.5.3 prints lines such as:

```
c [features] numVars 56118, numClauses 358991, var_cl_ratio 0.156,
binary 0.019, trinary 0.520, horn 0.387, horn_mean 0.000, horn_std
0.000, horn_min 0.000, horn_max 0.000, horn_spread 0.000,
vcg_var_mean 0.000, vcg_var_std 0.902, vcg_var_min 0.000,
vcg_var_max 0.000, vcg_var_spread -0.000, vcg_cls_mean 0.000,
...
```

Even though we will partially use equivalent features (like Horn clauses), many are actually related to the current state of evaluation like decisions per conflicts. We consider this as a property of the evaluation and not the SAT problem itself.

Many SAT solvers collect feature values to improve algorithm selection, restart strategies and estimate problem sizes. Recent trends to apply Machine Learning to SAT solving imply feature evaluation. SAT features and the resulting satisfiability runtimes are used as training data for Machine Learning. Another SAT solver using SAT features heavily for algorithm selection besides SATzilla is ASlib [1].

5.3 Statistical properties

For our SAT features we need to define some basic statistical terminology. Let $x_1, x_2, \dots, x_n$ be a sequence of real numbers ($n \in \mathbb{N}$).

Arithmetic mean (or mean for short) is defined as

$$\overline{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

Standard deviation (or sd for short) with mean $\overline{x}$ is defined as

$$\sigma(x) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})}$$

Median with $x_1 \leqslant x_2 \leqslant \ldots \leqslant x_n$ (i.e. sorted ascendingly) is defined as

$$m = \begin{cases} x_{|\text{mid}|+1} & \text{if } n \text{ odd} \\ \frac{x_{\text{mid}}+x_{\text{mid}+1}}{2} & \text{if } n \text{ even} \end{cases} \quad \text{with mid} = \frac{n}{2}$$

and often considered more "robust" than the arithmetic mean.

Entropy is defined according to Claude Shannon's information theory:

$$H(x) = -\sum_{i=1}^{n} x_i \cdot log_2(x_i)$$

where $0 \cdot log_2(0) := 0$.

Furthermore *count* refers to the number of elements n, *largest* refers to the maximum element $max_{1 \leqslant i \leqslant n}(x_i)$ and *smallest* refers to the minimum element $min_{1 \leqslant i \leqslant n}(x_i)$.

5.4 Suggested SAT features

We wrote a tool called cnf-analysis. The evaluated features are partially inspired by SATzilla and lingeling. The latter prints basic statistics for every CNF it evaluates.

A summary of our suggested SAT features is given:

clause_variables_sd_mean
mean of sd of variables in a clause

clauses_length_(largest, smallest, mean, median, sd)
statistics related to the clause length

connected_(literal, variable)_components_count
two literals (variables) are connected if they occur in some clause together, count the number of connected components

definite_clauses_count
number of definite clauses in the CNF

existential_literals_count
number of existential literals in the CNF

existential_positive_literals_count
number of positive, existential literals in the CNF

(false, true)_trivial
is the CNF satisfied if all variables are claimed to be false (true)?

goal_clauses_count
number of goal clauses in the CNF

literals_count
number of literals in the CNF (i.e. sum of clause lengths)

literals_frequency_k_to_$k+5$
let n_l be the literal frequency of literal l, count the number of n_l satisfying $\frac{k}{100} \leqslant n_l < \frac{k+5}{100}$ where k is a variable in $\{0, 5, 10, \dots, 90, 95\}$ and $k = 95$ counts $\frac{k}{100} \leqslant n_l$.

literals_frequency_(largest, smallest, mean, median, sd)_entropy
statistics related to literal frequencies

literals_occurence_one_count
number of literals with occurence 1

nbclauses, nbvars number of clauses (variables) as defined in the CNF header

negative_literals_in_clause_(smallest, largest, mean)
statistics related to number of negative literals in clauses

(positive, negative)_unit_clause_count
number of unit clauses with a positive (negative) literal

positive_literals_count
number of positive literals in CNF

positive_literals_in_clause_(largest, smallest, mean, median, sd)
statistics related to number of positive literals in clauses

positive_negative_literals_in_clause_ratio_(mean, entropy)
let r_c be the number of positive literals divided by clause length of clause c, mean and related of all r_c

positive_negative_literals_in_clause_ratio_mean
mean of all r_c

tautological_literals_count
number of clauses which contain a tautological literal

two_literals_clause_count
number of clauses with two literals

variables_frequency_k_to_$k + 5$
same as literals_frequency_k_to_$k + 5$ but for variables

variables_frequency_(largest, smallest, mean, median, sd, entropy)
same as literals_frequency but for variables

variables_used_count
number of variables with occurence greater 0

5.5 Evaluation efficiency

The resource requirements of those features have been classified:

Type 1 read the files as bytestring, a DIMACS CNF parser is not necessary, constant memory is used

Type 2 features understand what a clause is, but still need constant memory

Type 3 subquadratic runtime and linear memory

Type 4 unrestricted

Memory and runtime is always considered in comparison with the filesize.

This classification should support computational considerations regarding feature evaluation tools. The suggested SAT features above have been explicitly selected to avoid Type 4 implementations to limit the time to compute features. Furthermore tools evaluating only a subset of features (like Type 2 features) can achieve better performance characteristics than general-purpose tools. For example, we wrote a dedicated tool to evaluate the maximum clause length of CNF files, which was much faster (175 GB, 1.5 hours).

The Python implementation triggered MemoryErrors on a computer with 4 GB RAM for a 770 MB CNF file. Followingly a much more efficient Go implementation was implemented which requires much less memory and is much faster. `bench_573.smt2.cnf` (1.6 MB filesize) took 1 second in Go instead of 2 minutes in Python. However, the data evaluated is less accurate compared to Python in terms of floating point precision, because Python unlike Go provides a nice implementation of statistical tools in the standard library. Go algorithms were written on our own.

In the following section, we define the dataset we consider.

5.6 CNF dataset

To evaluate CNF features of a representative set of CNF files, it was necessary to identify equivalent CNF files in the best possible way. Therefore we defined a hashing algorithm standardizing the CNF input provided to a SHA1 instance. Every CNF file is identifiable by its "cnfhash 2.0.0" hash value.

In the next step a complete set of CNF files of previous SAT competitions was collected. The following CNF file collections have been considered:

- SAT Race 2008
- SAT09 Competition
- SAT-Race 2010
- SAT11 Competition
- SAT Challenge 2012
- SAT Competition 2013
- SAT Competition 2014
- SAT-Race 2015
- SAT Competition 2016
- SATlib [8]

The benchmarks are mostly contributed by the participants of the associated conferences. Others are reused from previous years. Individual projects allow to generate CNF files for specific problems in a selectable problem size; such as CNFgen [13] by Massimo Lauria.

Some files turned out to be problematic. In SATlib, 3 gzipped files couldn't be decompressed and several files contain empty clauses. Empty clauses are assumed to immediately falsify the CNF. We removed empty clauses and evaluated the remaining CNF. I removed trailing zeros in CNFs. Variants of the DIMACS standard also expect lines with a percent symbol to terminate the CNF. Beside those minor issues documented as part of the cnf-analysis project, 175 gigabytes of CNF files have been evaluated with a total of 68,069 CNF files (62,251 unique CNF files).

5.7 The average SAT problem

Claim 5.1

> The set of public benchmarks in SAT competitions between 2008 and 2015 represent average SAT problems

It is important to point out that public benchmark files are specifically chosen to be evaluated before a conference is held. Hence they are expected to terminate within a given time frame and are therefore not oversized. On the one hand this ensures that the problems are feasible, but on the other hand they might be a biased selection. At this point no better data set is available and therefore we proceeded with this dataset.

According to my results, an average SAT problem consists of:

- 83,542 clauses in average ranging from 21 to 53,616,734 (median = 430, σ = 848,388)
- The longest clause we found had 61,473 literals, but the longest clause of an average CNF covers 20 literals.
- The total number of literals in a CNF ranges from 60 up to 150,609,758.
- The clause-variables ratio lies between 1.22 and 27,720 with mean = 9.54 and σ = 139.
- The average length of a clause is expected to be 3. The largest clause length mean we found was 19.58 compared to 2.83 as the smallest clause length mean.
- Surprisingly, in average a CNF file has 205 connected literal components and 53 connected variable components. However both corresponding medians are 1 meaning that at least have of the problems still have only one component. Component sizes have not been evaluated.
- In average, 32,787 clauses are definite and 35,094 clauses are goal clauses.
- In average, a literal occurs in 1.4 % of the clauses of the CNF.
- 47 % of literals in a clause are positive.
- The arithmetic mean tells 124 unit clauses per CNF file can be expected, but the median tells it is mostly 0.
- The largest variable found was 13,842,706 and 13,829,558 variables were used at most.
- Exactly one CNF file was true-trivial (namely `dubois/dubois100.cnf` of SATlib) whereas 13 CNF files were false-trivial (of SAT competition 2014 and SAT-Race 2015).

“There is concensus that encoding techniques usually have a dramatic impact on the efficiency of the SAT solver”
—Magnus Björk

Chapter 6

Problem encoding

In Chapter 4, we already discussed how SAT solvers work and which input they take. We also sketched how hash algorithm properties got broken using differential cryptanalysis (Chapter 3). In this chapter, we combine those subjects and describe how we run SAT solvers to find hash collisions.

We developed a basic prototype with the STP SMT solver. In the following, we wanted to tweak the CNF used by the SAT solver and wrote our own library *algotocnf* to generate CNFs modelling variable differences and their logic; as illustrated in Section 3. In the referred section, we distinguished 5 different approach. We evaluate the performance of those approaches in Chapter 7.

Every section represents a major approach whereas subsections represent derivatives of this approach with minor changes.

6.1 Basic approach

Our first approach started with Simple Theorem Prover (STP) [7] initially written by Vijay Ganesh and David L. Dill. It is currently maintained by Mate Soos.

STP is an SMT solver which allows to declare bitvectors. A bitvector is an array of Boolean variables providing high-level constructs such as additions or right-shift through an interface. Writing all clauses individually to model a hash algorithm is too cumbersome to be done in practice and STP simplifies this process. STP is recommendable as a tool to model arithmetic and bitwise functions.

First we wrote an implementation using the CVC language to model the MD4 hash algorithm. We provide a bitvector to the hash algorithm instance. When applying the corresponding bitwise operations we generate expressions such as `ASSERT(y = 0bin00000101)` to model the assignment of a constant. Here the desired constant is assigned variable `y`, because of equivalence. `y` is required to have the constant after this expression as value. Whereas we generate the `ASSERT` statement, it is STP's task to generate the CNF formula and solve it with a SAT solver.

We take two hash algorithm instances and an additional bitvector `diff` for every pair of bitvectors (`bv1`, `bv2`) where `bv2` represents the corresponding bitvector of the second hash algorithm instance to `bv1`. We claim `ASSERT(diff = BVXOR(bv1, bv2))` to ensure that `diff` represents the difference between `bv1` and `bv2`. Given the bitconditions for a bitvector from a differential characteristic, we require `diff` to enforce those particular differences. This corresponds to the idea of differential cryptanalysis introduced in Chapter 3.

It is now trivial to consider a differential characteristic of MD4 such as Testcase A (see Section B.1), where all differences are set, but the individual values in both instances need to be assigned. We generated the corresponding CVC input for STP. STP solves this particular problem within a second. Testcase B (compare with Section B.2) already provides a more complex example taking 40 minutes to solve, because not all differences are set. We used minisat as SAT solver in the backend, even though STP allows to replace it for CryptoMiniSat which is a more modern and versatile SAT solver.

Even though STP allows to come to useful results pretty quickly, it seems cumbersome to model all hash algorithms in the CVC language. STP provides a python interface meaning that pure Python implementations of hash algorithms can be taken with little modifications to model the hash algorithm itself. We add code to declare the difference bitvectors `diff` and finally add the constraints resulting from the differential characteristic.

This interface switch introduces no significant performance difference.

As a next step, we wanted to improve the evaluation performance to tackle more difficult problems such as SHA-256. We considered this design as a working prototype of a basic approach to be improved upon.

STP seems not suited for our next goal, because we wanted to modify the particular CNF generated for the SAT solver and needed good control over the SAT encoding which we expected to have a major influence on the performance.

6.2 algotocnf

We implemented our own library *algotocnf* to achieve greater flexibility in our SAT encoding.

6.2.1 Two instances and its difference

Similar to STP, algotocnf generates a CNF for a given hash algorithm implementation. Besides modelling bitvectors, it also implements differential bitvectors which inherently handle the difference bitvectors `diff` which contain difference variables. It can also directly takes differential characteristics (such as Table 3.5, specified in Chapter 3) as input. Similarly to STP, it implements arithmetic and bitwise operations.

We think *algotocnf* mainly differs from other SMT tools like STP, because of its implementation of differential logic.

To model our CNF *algotocnf* implements the following strategy:

1. Take a differential characteristic and the hash algorithm as input.
2. Every bit gets represented as a Boolean variable. If you apply addition, operator overloading in python will ensure that clauses are generated to describe the addition consisting of `XOR`s and `MAJ`s. Every operation is modelled as assignment. Hence, an operation using a few Boolean variables is equivalent to a single variable which represents the result. Similarly other operations related to integers are implemented as well.
3. Constants used in the implementation are automatically converted to bitvectors with unit clauses.
4. After running the hash algorithm with bitvectors per instance, all constraints related to the hash algorithm are added.
5. Afterwards, the differential characteristic is read. Values such as A_i represent intermediate states of bitvectors. Therefore the corresponding bitvectors are looked up and equivalences with temporary bitvectors are added. Those temporary bitvectors are initialized with all constraints resulting from the bit conditions of this bitvector (please refer to Chapters 2 and 3 for details). In conclusion, all constraints resulting from the differential characteristic are added.
6. Finally, the SAT solver is called. The CNF was mostly solved on a cluster specified in Appendix A.

7. Afterwards the program is run again to create the exact same problem instance and the solver's solution replaces symbolic values with actual Boolean values. The resulting differential characteristic is parsed backed and printed as differential characteristic where all bits have been determined (i.e. a hash collision has been found).

When adding clauses resulting from the differential characteristic as constraints, the question arises how those bit conditions are encoded. Essentially, we have only Boolean values available, but bit conditions tell constraints such as "a difference is given, but the actual value is unknown".

It seemed trivial to add a *difference variable* for every pair of Boolean values representing a bit in the two instances. Furthermore, the difference variable Δx is connected by a `XOR` with the variables of the pair (x, x').

$$\Delta x = x \oplus x'$$

Therefore, it is trivial for a preprocessor to simplify the formula appropriately. Hence, we don't expect runtime differences for the larger amount of variables.

And finally we expect the CNF to inherit a property of hash functions. Inputs are provided into the hash algorithm and strongly intermingled with other values. This results in a high diffusion and almost every variable is expected to share a clause with another variable.

The difference variables design corresponds to `diff` bit vectors in the STP and therefore models the difference variables described in Chapter 3. The design decisions of this encoding are fundamental to the resulting runtime as discussed in Chapter 7.

6.2.2 Adding the differential description

Using the approach in the previous section, we were able to find actual MD4 collisions using a SAT solver (please refer to Section 7.2.1). We used a reused our implementation of MD4 for SHA-256 and replaced the hash algorithm implementation. This implementation obviously presented worse runtime results, because the internal state of SHA-256 is much larger (by a factor of at least 2). Can we further improve the runtime of the SAT solver?

Since we work with bitvectors and apply high-level operations like `MAJ` or addition, we can additionally implement how differences in those operations propagate. Magnus Daum's thesis on "Cryptanalysis of Hash Functions of the MD4-Family" [4, Table 4.4] discusses how differences propagate in Boolean functions. Trivially, `XOR`s propagate differences the way they are[1]. Another example is `IF`:

[1] A difference in the arguments of two `XOR` instances remains the same difference after applying `XOR` to each instance

Let a, b and c be difference variables and `IF` is applied to both corresponding hash algorithm instances. r is the difference variable of the result. Then its differential behavior states that

$$(0,1,1) \implies 1 \qquad (0,0,0) \implies 0$$

where (IN) $\implies$ OUT denotes an input-output relation and in all other cases, the difference can be either 1 or 0. Because of this behavior we add clauses to explicitly describe this behavior:

$$(\neg a \land b \land c) \implies r \quad \iff \quad a \lor \neg b \lor \neg c \lor r$$

We also model the second behavior:

$$(\neg a \land \neg b \land \neg c) \implies \neg r \quad \iff \quad a \lor b \lor c \lor \neg r$$

This approach explicitly models differentiable behavior, which should be deducible by the SAT solver itself based on the clauses we added before. However, this lead to a major speedup which can be observed in the runtime results of Chapter 7.

6.2.3 Difference variables first

In this approach we reduce the number of evaluated differences by guessing Boolean value false first for difference variables.

Claim 6.1

> Deriving difference values first, followed by actual bit values for the two instances, leads to a speedup.

This proposed principle is fundamental to differential cryptanalysis. A previous tool at IAIK (TU Graz) implements propagation of hash algorithm values without a SAT solver and this strategy is essential to good performance. This strategy was introduced in the very early days of differential cryptanalysis and was also used by Wang et al. [35] to find their hash collisions.

For our SAT solver, we want to establish the following strategy: Take some CNF which includes difference variables. We assign a Boolean value to every difference variable unless a contradiction is found. For all the remaining variables we try to find a satisfiable assignment. If none can be found, we consecutively toggle the Boolean value of difference variables to cover all possible assignments and find a satisfiable one.

It is important to point out that DIMACS does not specify a way to annotate Boolean variables. As such that SAT solver cannot distinguish between difference variables and variables of the instances. Therefore, implementing this approach requires a custom SAT solver which is given with lingeling ats101.

Another claim is important for this approach:

Claim 6.2

Guessing difference values false first, followed by true, should solve hash collision problems faster.

This claim is justified by the desire to find sparse characteristics with few differences in intermediate variables to increase the probability of values cancelling each other out in the later rounds.

6.2.4 A lightweight approach

In this approach we made a step back and considered the ideas of the previous section, but neglected the differential description. This approach was interesting to quantify the effect introduced by adding the differential description.

6.2.5 Influencing the evaluation order

To take the idea to influence the evaluation order to the next level, we enforced the evaluation order even stronger by applying the following SAT design:

Let Δx be the difference variable of pair (x, x'). We introduce a new Boolean variable x^* called *preference variable*. We add clause

$$x^* = (\Delta x \wedge x)$$

and explicitly tell the SAT solver to guess on x^* before guessing on Δx, x or x'.

The SAT solver will assign $x' = 0$ first, because of the evaluation order. So either Δx or x must be false. Δx is assigned false, because as difference variable it has a higher priority over x. Equivalently for $x' = 1$, Δx needs to be true. So we actually achieve an early guess on the difference variable.

In Chapter 7 we evaluated the performance of this approach.

Chapter 7

Results

In Chapter 4, we discussed Boolean algebra; in particular we looked at satisfiability which is practically covered by SAT solvers. SAT solvers take Boolean functions in Conjunctive Normal Form and determine satisfiability. In Chapter 3, we discussed how we can analyze algorithms by observing progression of differences between algorithm instances. In particular, we looked at hash algorithms introduced in Chapter 2.

With this background, we designed an attack setting in Chapter 6 which enables us to verify and also find a hash collision given a differential characteristic as starting point. Our goal is to find hash collisions in practical time which we define by 1 day (86,400 seconds). Therefore, we designed several approaches to improve our runtime results.

In this section, we will evaluate those approaches. Furthermore, we briefly discuss claims we made about average SAT problems. In Section 5, we defined SAT features which to some extent characterize a SAT problem.

7.1 Evaluating SAT features

In Chapter 5, we posed 8 questions. In the following, we want to answer them with the data provided by the cnf-analysis project.

Given an arbitrary literal. What is the percentage it is positive? We look at every clause and determine the ratio of positive to the total number of literals. We determine the mean per CNF file and the mean among all CNF files and retrieve a value of 0.48 meaning that 48 % of the literals are positive.

What is the clauses / variables ratio? In average a CNF file has 12,219 variables and 89,541 clauses. Its clauses-variables ratio is 7.328.

How many literals occur only once either positive or negative? In average there are 36 existential literals per CNF file, but its standard deviation of 967 is very large.

What is the average and longest clause length among CNF benchmarks? The average clause length is 3.04 with a standard deviation of 0.99 and the longest clause length found was 61,473. Long clauses are typically outliers excluding specific assignments.

How many Horn clauses exist in a CNF? In average 29,994 goal clauses and 31,315 definite clauses exist with an average number of 83,649 clauses in a CNF file.

Are there any tautological clauses? In one file, 1679 tautological literals have been found. However, its mean is 0.07 with a standard deviation of 9.63 meaning that tautological clauses are very rare.

Are there any CNF files with more than one connected variable component? Indeed, an average CNF file contains 67.07 connected variable components. However, its median is 1 implying that at least half of the CNF files have only 1 connected variable component.

How many variables of a CNF are covered by unit clauses? In average 124 variables are covered by unit clauses. This is an insignificant number compared to 12,219 variables in an average CNF.

The clauses/variables ratio was thoroughly studied by the SAT community [23]. A strong correlation between the instance's hardness and the ratio of number of clauses to number of variables exists [32] though it is important to point out that this result holds for randomly generated SAT instances, which our testcases are not classified as.

Existential literals are interesting to discover, because they allow to remove a clause immediately. Consider a clause with literals $(l_1, l_2, \dots, l_n)$. If a guarantee exists such that the variable of any literal l_i does not occur in any other clause, we can claim l_i true rendering the clause satisfied.

Tautological clauses trivially also render clauses satisfied.

Connected variable components are interesting, because they split the SAT problem into several small independent subproblems which can be solved in parallel. Consider two sets of variables A and B. Now consider some clauses using only variables of A and some clauses using only variables of B. The overall CNF is satisfiable iff both clause sets are satisfiable. The overall CNF is falsifiable iff any clause set is falsifiable. Hence, if we know the connected variable components, we could easily create two parallel SAT solver instances and solve the problems independently. 4,607 out of 62,251 CNF files contained more than one connected variable component.

These features represent very fundamental properties of the SAT problem. But for us the question arises whether we can distinguish our cryptoproblems from average problems?

- We looked at 36 files classified as cryptographic problems. They are considered cryptographic, because their file or folder name indicated they are related to hash functions or general cryptographic applications like AES. The specific selection can be identified by the crypto tag annotated to these CNF files as part of the cnf-analysis project.
- In average these problems have 116,398 clauses and 27,407 variables. The average clauses-variables ratio is 5.58.
- The 36 cryptographic SAT instances give a standard deviation of 0.7 for clause length meaning that most clause lengths are close to the mean 3.4.
- The number of definite clauses is twice its value for general problems (62,457 versus 31,315) and the number of goal clauses is 26 % of its value for general problems (7,761 versus 29,994).
- The number of connected variable components is 2,236 in average (σ = 10,060), but the median is 1 again.

No other value has been found to be significantly different from average problems (or its difference follows immediately by the non-uniform clause length).

The number of connected variable components seems interesting in cryptographic problem, because it might indicate diffusion in cryptographic problems. Diffusion means that variables strongly interact with many different variables due to the repetitive structure of cryptographic primitives. And finally the other differences can be explained by a certain SAT design which reoccurs in these testcases, because 36 is an exceptionally small number compared to 62,251 unique CNF problems. It is expected the cryptographic problems were designed by a small set of authors.

Comparing our average problem with cryptographic problems did not draw any useful conclusions. Particularly a more thorough discussion of the SAT designs might be more valuable than our high-level features. We now specifically look at a SAT design we are familiar with: Do average SAT problems distinguish from *our* CNF testcases?

- For all MD4 testcases we have the same number of variables, because the internal state of the hash algorithm instances are always the same size. However, adding the differential description as described in Section 6.2.2 increases the number of clauses by about 47 % ($\sigma = 0.0005$) for MD4 instances and by about 43 % ($\sigma = 0.0008$) for SHA-256 instances. For SHA-256 problems, this is illustrated in Table 7.5. The preference variable introduced in Section 6.2.5 increases the number of variables by about 80 % and the number of clauses by factor 2.

  Compared to 83,542 clauses and 12,219 variables for our average SAT problem, we consider our testcases to be noticeably large. However, it is important to point out that the problem size does not necessarily correlate with the hardness of the SAT problem.

- The variables of clauses of average SAT problems have a standard deviation of 3,337 in average (σ =1,261, median =3,643). Our average SAT problem has a standard deviation of 1,004 in average (σ =13,992, median = 22). Hence variables which got created at every point during the CNF generation are shared within one clause. The general statement, that variable enumeration is arbitrary and therefore this standard deviation has no meaning holds, but we need to consider that practically speaking variables created close to each other share close variable indices. Under these assumptions a large σ indicates variables are reused. We assume this is another indicator for high diffusion in cryptographic algorithms. Values are intermingled over and over throughout the repetitive structure of hash algorithms.

- Connected variables components are 129 for MD4 problems and 2 for SHA-256 problems. For SHA-256 problems, a unit clause is given as existential literal and for MD4 problems, all components except one are of size 3. We did not investigate further, because this number is constant with an increasing problem size and all other variables are strongly correlated due to a high diffusion.

- An average literal frequency of $3.5 \cdot 10^{-5}$ for our testcases is much lower than 0.014 for average problems. We explain this with the larger problem size. Literal frequency is divided by the number of clauses of the CNF and is therefore smaller, the larger the problem is.

In general, we were not able to identify features allowing us to solve differential cryptanalysis problems more efficiently compared to general-purpose SAT problems. We concluded writing your own SAT solver dedicated to solving differential cryptanalysis problems is not worth the effort.

7.2 Finding hash collisions

In this section, we look at our runtime results of testcases provided in Appendix B. We make various claims and substantiate them with runtime results. Runtimes are always provided in seconds. Therefore, smaller runtimes are better. $\top$ denotes a timeout (solving took more than 1 day) and — denotes unavailable data.

We considered MD4 testcases A, B and C (listed in Table 7.1) and generated the corresponding CNF files. The SAT solvers mentioned in Section 4.5 were used to evaluate whether the problem is solvable within the time limit.

algorithm	testcase	rounds	diff. characteristic	clauses	variables
MD4	A	48	Appendix B.1	254,656	48,704
MD4	B	48	Appendix B.2	254,210	48,704
MD4	C	48	Appendix B.3	253,984	48,704

TABLE 7.1: MD4 testcases considered

7.2.1 Attacking MD4

Claim 7.1

Testcase A in the encoding described in Section 6.2.1 can be solved within one minute by all considered SAT solvers.

In our attack setting we started off with Testcase A. It serves rather as a toy example to verify correctness of our implementation than as an actual benchmark. Be aware that invalid implementations either result in unsatisfiability for satisfiable testcases or runtime results are unexpected because the SAT solver could not take advantage of our SAT design improvements. This particular testcase can be solved easily with all major SAT solvers as can be seen in Table 7.2. We end up with the result, that the hash collision given in Testcase C can be solved by the majority of modern SAT solvers. Of course the cryptanalyst needs to figure out good starting points for the hash collision and encode them in the differential characteristic, but this task is still considered practical, because it can be easily automated.

7.2.2 Evaluating simplification

As a next approach, we looked at CNF simplifiers. Those simplifiers consume a CNF file and transform the CNF file to an equisatisfiable CNF file. Those simplified CNF files might be subject to performance improvements.

solver	version	propagations	decisions	restarts	runtime
MiniSat	2.2.0	3,813,726	250,759	—	3
CryptoMiniSat	4.5.3	140,000	2,441,566	539	26
	5	32,163,801	2,178,965	598	29
Lingeling	ats1	6,586,770	436,621	980	23
Plingeling	ats1	452,630,440	3,275,498	—	88
Treengeling	ats1	18,629,811	1,640,289	—	64
Glucose	4.0	14,727,839	990,491	272	8
Glucose Syrup	4.0	37,021,496	629,363	201	14

Table 7.2: Testcase A can be solved within 1 minute by all SAT solvers

Claim 7.2

Simplification reduces the problem size (number of variables and clauses).

Consider for example Testcase C (Appendix B.3) in the basic encoding introduced in Section 6.2.1. Then simplification will reduce the problem size down to 42.9 % or more of its original size (as illustrated in Table 7.3). We verified these data for all simplified files and got similar results. Therefore, the claim holds considering the problem size gets reduced to approximately half of its size.

simplification	variables	percent of none	clauses	percent of none
none	48,704	100 %	253,984	100 %
cmsat	24,503	50.31 %	111,931	44.07 %
lingeling	48,704	100 %	106,626	41.98 %
minisat	20,895	42.90 %	118,236	46.55 %
satelite	27,495	56.45 %	153,262	60.34 %

Table 7.3: Problem sizes of Testcase C in the encoding of Section 6.2.1 after simplification. lingeling maintains the same number of variables according to the CNF header.

Claim 7.3

Simplification as preprocessing step does not significantly improve the runtime of SAT solvers.

We look at Testcase C which is a more difficult MD4 problem compared to Testcase A. Simplification runtime results depend on the SAT solver, which applies certain simplifications while trying to solve the CNF, and the simplifier used. A small number of variables or clauses does not necessarily lead to better performance. But an equisatisfiable encoding of the same problem is worth considering.

Table 7.4 lists runtimes depending on the simplification used.

none refers to the unsimplified CNF

cmsat refers to simplification applied with CryptoMiniSat version 5:
```
./cryptominisat5 -p1 file.cnf simplified.cnf
```

lingeling refers to simplification with lingeling version ats1:
```
./lingeling -s file.cnf -o simplified.cnf
```

minisat also simplifies CNF file with the following command line:
```
./minisat file.cnf -dimacs=simplified.cnf
```

satelite is specifically designed to simplify CNF files:
```
./satelite file.cnf simplified.cnf
```

It is worth pointing out that simplification time is not part of the runtime listed. Simplification can take very long. Especially, simplifications with lingeling have sometimes taken several hours without result.

In conclusion, simplification leads to a slight improvement of the runtime, but in general we cannot recommend simplifying every CNF file. Because technically speaking, SAT solvers internally apply simplification algorithms on their own.

solver	version	none	cmsat	lingeling	minisat	satelite
MiniSat	2.2.0	4,519	7,649	1,337	1,476	1,293
CryptoMiniSat	5	1,064	973	1,201	4,470	3,920
Lingeling	ats1	1,492	906	356	860	1,297
Treengeling	ats1	1,281	13,401	20,903	13,790	10,840
Plingeling	ats1	2,310	1,232	955	1,384	2,030

Table 7.4: Runtimes of Testcase C after CNF files have been simplified

7.2.3 Attacking SHA-256

While the basic approach works well for MD4, hash algorithm SHA-256 encompasses a much larger state making the problem significantly more difficult for the SAT solver. Consider Testcases 18, 21, 23 and 24. Those testcases describe round-reduced hash collisions on SHA-256 (the testcase number gives the number of rounds). Our next approach is called differential description as originally described in Section 6.2.2.

Claim 7.4

> A differential description encoding (Section 6.2.2) improves the runtime compared to a missing differential description.

testcase	clauses / variables	testcase	clauses / variables
18:	590,953 / 107,839	18 diff-desc:	846,487 / 107,839
21:	636,838 / 116,800	21 diff-desc:	911,629 / 116,800
23:	667,438 / 122,774	23 diff-desc:	955,067 / 122,774
24:	682,722 / 125,761	24 diff-desc:	976,770 / 125,761

Table 7.5: Problem sizes of our SHA-256 testcases (clauses / variables)

To testing differential description, we looked at MD4's Testcase C and compared it with out SHA-256 testcases. Those testcases are described in detail in Appendices B.4, B.5 and B.6.

Recall that differential description explicitly encodes how differences in arithmetic and bitwise operations propagate in the CNF. We discussed `XOR` and `IF` in Section 6.2.2. These clauses should be deducible by the SAT solver itself and do not narrow the search space. Therefore we expected equivalent runtime results for both cases (with or without differential description). However, the resulting data indicates the opposite.

In Table 7.6 we picked two SAT solvers lingeling and CryptoMiniSat and we can clearly see a significant improvement of the runtimes.

	CryptoMiniSat 5		lingeling-ats1	
testcase	w/o dd	w/ dd	w/o dd	w/ dd
MD4, C	1,064	231	798	53
SHA-256, 18	37	37	31	160
SHA-256, 21	$\top$	7,855	28,621	5,513
SHA-256, 23	$\top$	26,212	76,196	1,450
SHA-256, 24	$\top$	37,194	78,017	1,235

Table 7.6: Runtimes for various testcases with or without differential description with CryptoMiniSat and lingeling. Testcase C has been added for reference. We need to point out that the timeouts, unlike other testcases, were determined on Thinkpad x220 (compare Appendix A), because the processes consistently died on our cluster.

We continued by modifying the guessing strategy to reflect differential cryptanalysis, which generally use the assumption that difference variables are assigned first. This strategy requires customization of the SAT solver and therefore we only considered lingeling, which was adapted for our purposes.

7.2.4 Modifying the guessing strategy

In differential cryptanalysis the general assumption is made that differences should be guessed first. Once they are assigned, we can look at the Boolean values in the two hash algorithm instances. To model this behavior, we looked at options provided by SAT solvers.

Claim 7.5

> Lingeling option `--phase=-1` improves its runtime for our testcases.

Option `--phase=-1` of lingeling is described as "default phase" set to -1 (negative), 0 (Jeroslow-Wang strategy [10]) or 1 (positive). Per default a strategy engineered by Jeroslow-Wang [10] is used, but considering Claim 6.2 at page 40 we expect `--phase=-1` to provide better runtime results.

Indeed our results consistently indicate a small improvement. This can be recognized in Table 7.7.

testcase	18		21		23		24	
phase	0	-1	0	-1	0	-1	0	-1
runtime	31	22	28,621	19,717	76,196	71,677	85,774	70,259

TABLE 7.7: lingeling-ats1 results for SHA-256 comparing `--phase=-1` with `--phase=0`

7.2.5 Evaluating the lightweight approach

Though the results of `--phase=-1` was recognizable, we wanted to push it further. We got in contact with Armin Biere who provided us an extended lingeling implementation which distinguishes two sets of variables; namely a set of differences variables which needs to be assigned first.

Claim 7.6

> Evaluating difference variables first and with Boolean value false improves the runtime.

The lightweight approach mentioned in Section 6.2.4 evaluates difference variables first with Boolean value false, but does not add a differential description. Hence, differential behavior is not modelled explicitly. This approach is justified by the assumption that a low number of differences, leading to a sparse differential path, is more likely to cancel out differences ending in a hash collision.

Table 7.8 reveals a nice improvement (the runtime becomes 0.85 %) of its original runtime in average.

testcase	**C**	**18**	**21**	**23**	**24**
basic approach (ats1)	798	31	28,621	76,196	85,774
diff-first-false (ats101)	652	29	27,599	59,312	66,052

TABLE 7.8: lingeling-ats101 and lingeling-ats1 results comparing a difference variables (with Boolean value false) first approach with the basic approach

7.2.6 Using preference variables

Our last approach uses *preference variables* mentioned in Section 6.2.5. Under the assumption that preference variables x^* and difference variables Δx are assigned first, an additional clauses provides a decision tree which assigns difference variables first and once they are all set, values for the two hash algorithm instances are assigned.

Claim 7.7

Adding preference variables dramatically worsens performance.

Section 6.2.5 introduces preference variables which enforce the idea that difference variables are evaluated first. Preference variables only add additional clauses, but do not provide a runtime improvement per se. The larger number of variables and clauses make the problem potentially harder.

However, evaluating them with false first makes sure that a low number of differences is propagated. Otherwise the SAT solver would spend much time in fruitless branches and the number of restarts would be comparably high.

Given an assigned difference variable, differential description ensures that the value is propagated quickly to other parts of the equation system. This justifies why our encoding with preference variables should be compared to an instance with differential description and difference variables first.

Table 7.9 shows results for MD4 and SHA-256 testcases. The data indicates that for very small runtimes, the runtime improved. Unfortunately, for the SHA-256 testcases runtimes have worsened extraordinary.

testcase	**A**	**B**	**C**	**18**	**21**	**23**	**24**
CNF with diff-desc	11	133	155	49	2,282	1,314	2,632
preference variables added	8	50	62	$\top$	$\top$	$\top$	$\top$

TABLE 7.9: lingeling-ats101 testcases comparing an approach with differential description with additional preference variables

7.3 Summary

In this section we looked at various improvements to improve the runtime, namely

1. CNF simplification
2. differential description
3. Lingeling's `--phase=-1` option
4. Difference variables first with Boolean false
5. Preference variables

We evaluated these approaches with several SAT solvers and found some significiant runtime improvements. We successfully found hash collisions for MD4 and SHA-256, where the latter has been reduced to 18, 21, 23 and 24 steps.

Chapter 8

Summary and Future Work

8.1 Related work

Already in my bachelor thesis [25] we tried to integrate a SAT solver into a dedicated, automated tool finding collisions in hash functions using differential cryptanalysis. This approach was not very successful as restarts between hash algorithm rounds implied that intermediate results by the SAT solver got lost. This was one motivation for this master thesis: We represent all details in CNF.

Research was already done by Ilya Mironov and Lintao Zhang [18] to apply SAT solvers to differential cryptanalysis specifically to find hash collisions in MD4. Their approach corresponds to our basic approach applied to Testcase A (compare with Section B.1), where all differences are assigned. Therefore, our results in Table 7.2 within one minute are comparable with their evaluations within ten minutes. We can clearly see how SAT technology and CPU performance has progressed over 10 years since publication of this paper.

Finding hash collisions in Testcases B, C, 21, 23 and 24 seems to be a novel result of this master thesis and has not been found in related work.

8.2 Results

We successfully found full-round hash collisions for MD4 using SAT solvers mentioned in Section 4.5. We modified the lingeling SAT solver to improve our runtime results further and found 24-round hash collisions for SHA-256. Our attack starting points for MD4 — Testcases B.1, B.2 and B.3 — are based on the work by Yusuke Naito, Yu Sasaki, Noboru Kunihiro and Kazuo Ohta [29]. Our starting points for SHA-256 — Testcases B.4, B.5, B.6 and B.7 — are based on the work by Ivica Nikolić and Alex Biryukov [21].

8.3 Contributions

To encourage future work, the source code and data resulting from this thesis is available online. It allows the reader to run the experiments again and verify our claims. We did our best to describe our hardware setup as accurately as possible. At the following website, any results part of this project are collected:

`http://lukas-prokop.at/proj/megosat/`

8.4 Future work

Future work might want to consider our design decisions made in Chapter 6.

In general, it would be interesting to generate our testcase for a broader set of differential characteristics. Probably we can come up with empirical results showing a relation between the size of unspecified areas in the differential characteristic and the evaluated runtimes.

Furthermore, many SAT-related effort could be put to thoroughly discuss why differential description provides such a significant performance improvement. This necessarily means the SAT solver is generally not capable of deriving the useful clauses, differential description provides. The resulting numbers of restarts could also be subject of further research.

Lingeling ats101 was experimentally modified to define a separate set of variables to be evaluated first. This approach seemed promising and should be subject to future SAT research.

As far as cnf-analysis is concerned, the project aims to extend to a larger set of SAT features and feedback by SAT solver developers is appreciated. Our main contribution is a search interface to search for SAT features given the cnfhash of a CNF file. We hope to get in touch with new SAT feature ideas and SAT benchmark files.

Appendices

Appendix A

Hardware setup

We introduce two hardwarde setups used in this master thesis. Thinkpad x220 was used to simplify CNF files and evaluate SAT features. The cluster was used for running SAT solvers and retrieving runtime results.

Model type	Thinkpad Lenovo x220 tablet, 4299-2P6
Processor	Intel i5-2520M, 2.50 GHz, dual-core, Hyperthreaded
RAM	16 GB (extension to common retail setup)
L1–L3 cache sizes	32 KB / 256 KB / 3,072 KB

TABLE A.1: Thinkpad x220 Tablet specification [14]

Cluster node nehalem192go specification [3]	
Processor	Intel Xeon X5690, 3.47 GHz, 6 cores, Hyperthreaded
RAM	192 GB
L1–L3 cache sizes	32 KB / 256 KB / 12,288 KB
Cluster node nehalem72go specification	
Processor	Intel Xeon X5550, 2.67 GHz, 4 cores, Hyperthreaded
RAM	72 GB
L1–L3 cache sizes	32 KB / 256 KB / 8,192 KB
Cluster node xeon64g specification*	
Processor	Intel Xeon E5430, 2.66 GHz, 4 cores, Hyperthreaded
RAM	72 GB
L1–L3 cache sizes	32 KB / 6144 KB

TABLE A.2: One node nehalem192go, one node nehalem72go and four nodes xeon64g* were used for evaluating runtimes

Appendix B

Testcases

B.1 MD4 testcase A

Please compare with Figure B.1.
We can clearly see that all difference variables are defined. Either they are true (bit condition `x`) or false (bit condition `-`), but no variable has an undeciable state like ?. At the top we can see bit conditions `0` and `1` encoding the MD4 initial vector defined by the hash algorithm. The differences `x` introduce the hash collision and with round 47 being set of `-` only, the output is forced to be equal between both hash algorithm instances. This testcase is a trivial version of the differential characteristic described in [29].

B.2 MD4 testcase B

Please compare with Figure B.2.
In this testcase we have less knowledge about the state than in testcase A because many values are encoded with ? meaning that neither their difference nor their actual values are known. However, of course some `x` exists to introduce a hash collision and the last round only consists of dashes to assert no difference in the output. So unlike testcase A, the SAT solver needs to figure out the difference variables in rounds 0–11 increasing its overall runtime in all SAT solver implementations.

B.3 MD4 testcase C

Please compare with Figure B.3.
This testcases introduces a hash collision which is expected to cancel out at round

32. At the same time no information is provided about the intermediate state of the hash algorithm in rounds 0–20.

B.4 SHA-256 testcase 18 rounds

Please compare with Figure B.4.
All SHA-256 testcases got derived from a paper by Ivica Nikolić and Alex Biryukov [21]. Recall that message input of one block only fills W0 to W15. W16 or later are generated based on the previous message words. This differential characteristic depicts a hash collision introduced in round 3. When solving this differential characteristic differences will especially occur in the most-significant bit of words with unspecified difference. Differences cancel out after round 7.

B.5 SHA-256 testcase 21 rounds

Please compare with Figure B.5.
Unlike Testcase B.4 word A5 is underspecified with question marks. This certainly makes the problem harder for a SAT solver. At the same time differences of the message words W9 and W10 are specified. This makes the problem easier. At the same time the whole collision covers 21 rounds, unlike Testcase B.4 with 18 rounds.

B.6 SHA-256 testcase 23 rounds

Please compare with Figure B.6.
Message word W9 specifies no difference, but the collision is extended to 23 rounds.

B.7 SHA-256 testcase 24 rounds

Please compare with Figure B.7.
No measures were taken to simplify the problem for the SAT solver, but the hash collision needs to be found for 24 rounds, which makes the problem hard for the SAT solver because of its increased state size.

i		A		W
-4	A:	01100111010001010010001100000001		
-3	A:	00010000001100100101010001110110		
-2	A:	10011000101110101101110011111110		
-1	A:	11101111111001101101010111000100 1		
0	A:	x-------------------------------	W:	---x----------------------------
1	A:	--------------------------------	W:	--------------------------------
2	A:	---------------------x----------	W:	x-------------------------------
3	A:	xxx-----------------------------	W:	--------------------------------
4	A:	------------------------------xx	W:	x-------------------------------
5	A:	-------xxxxxxxxxxxxxxxx-x-------	W:	--------------------------------
6	A:	x--------x-----------x-x-xxxx--x	W:	--------------------------------
7	A:	---------x-----x----------------	W:	--------------------------------
8	A:	---------x-------x-x-x----------	W:	x-------------------------------
9	A:	-----------------x---x----------	W:	--------------------------------
10	A:	------x-----x--xxx-xxx----------	W:	--------------------------------
11	A:	x----------------xxx-x----------	W:	--------------------------------
12	A:	---x--x-------------------------	W:	x-------------------------------
13	A:	--------------------------------	W:	--------------------------------
14	A:	-x------------------------------	W:	--------------------------------
15	A:	x-x----------x------------------	W:	--------------------------------
16	A:	-xxx----------------------------		
17	A:	--------------------------------		
18	A:	--------------------------------		
19	A:	x-------------------------------		
20	A:	x-------------------------------		
21	A:	--------------------------------		
22	A:	--------------------------------		
23	A:	--------------------------------		
24	A:	--------------------------------		
25	A:	--------------------------------		
26	A:	--------------------------------		
27	A:	--------------------------------		
28	A:	--------------------------------		
29	A:	--------------------------------		
30	A:	--------------------------------		
31	A:	--------------------------------		
32	A:	x-------------------------------		
33	A:	--------------------------------		
34	A:	--------------------------------		
35	A:	--------------------------------		
36	A:	--------------------------------		
37	A:	--------------------------------		
38	A:	--------------------------------		
39	A:	--------------------------------		
40	A:	--------------------------------		
41	A:	--------------------------------		
42	A:	--------------------------------		
43	A:	--------------------------------		
44	A:	--------------------------------		
45	A:	--------------------------------		
46	A:	--------------------------------		
47	A:	--------------------------------		

Figure B.1: Differential characteristic of MD4 testcase A

i		A		W
-4	A:	01100111010001010010001100000001		
-3	A:	00010000001100100101010001110110		
-2	A:	10011000101110101101110011111110		
-1	A:	11101111110011011010101110001001		
0	A:	????????????????????????????????	W:	---x----------------------------
1	A:	????????????????????????????????	W:	--------------------------------
2	A:	????????????????????????????????	W:	x-------------------------------
3	A:	????????????????????????????????	W:	--------------------------------
4	A:	????????????????????????????????	W:	x-------------------------------
5	A:	????????????????????????????????	W:	--------------------------------
6	A:	????????????????????????????????	W:	--------------------------------
7	A:	????????????????????????????????	W:	--------------------------------
8	A:	????????????????????????????????	W:	x-------------------------------
9	A:	????????????????????????????????	W:	--------------------------------
10	A:	????????????????????????????????	W:	--------------------------------
11	A:	????????????????????????????????	W:	--------------------------------
12	A:	??????????????------------------	W:	x-------------------------------
13	A:	??????????????------------------	W:	--------------------------------
14	A:	??????????????------------------	W:	--------------------------------
15	A:	??????????????------------------	W:	--------------------------------
16	A:	???x----------------------------		
17	A:	?-------------------------------		
18	A:	?-------------------------------		
19	A:	?-------------------------------		
20	A:	x-------------------------------		
21	A:	--------------------------------		
22	A:	--------------------------------		
23	A:	--------------------------------		
24	A:	--------------------------------		
25	A:	--------------------------------		
26	A:	--------------------------------		
27	A:	--------------------------------		
28	A:	--------------------------------		
29	A:	--------------------------------		
30	A:	--------------------------------		
31	A:	--------------------------------		
32	A:	x-------------------------------		
33	A:	--------------------------------		
34	A:	--------------------------------		
35	A:	--------------------------------		
36	A:	--------------------------------		
37	A:	--------------------------------		
38	A:	--------------------------------		
39	A:	--------------------------------		
40	A:	--------------------------------		
41	A:	--------------------------------		
42	A:	--------------------------------		
43	A:	--------------------------------		
44	A:	--------------------------------		
45	A:	--------------------------------		
46	A:	--------------------------------		
47	A:	--------------------------------		

Figure B.2: Differential characteristic of MD4 testcase B

i		A		W
-4	A:	01100111010001010010001100000001		
-3	A:	00010000001100100101010001110110		
-2	A:	10011000101110101101110011111110		
-1	A:	11101111110011011010101110001001		
0	A:	????????????????????????????????	W:	---x----------------------------
1	A:	????????????????????????????????	W:	--------------------------------
2	A:	????????????????????????????????	W:	x-------------------------------
3	A:	????????????????????????????????	W:	--------------------------------
4	A:	????????????????????????????????	W:	x-------------------------------
5	A:	????????????????????????????????	W:	--------------------------------
6	A:	????????????????????????????????	W:	--------------------------------
7	A:	????????????????????????????????	W:	--------------------------------
8	A:	????????????????????????????????	W:	x-------------------------------
9	A:	????????????????????????????????	W:	--------------------------------
10	A:	????????????????????????????????	W:	--------------------------------
11	A:	????????????????????????????????	W:	--------------------------------
12	A:	????????????????????????????????	W:	x-------------------------------
13	A:	????????????????????????????????	W:	--------------------------------
14	A:	????????????????????????????????	W:	--------------------------------
15	A:	????????????????????????????????	W:	--------------------------------
16	A:	????????????????????????????????		
17	A:	????????????????????????????????		
18	A:	????????????????????????????????		
19	A:	????????????????????????????????		
20	A:	????????????????????????????????		
21	A:	--------------------------------		
22	A:	--------------------------------		
23	A:	--------------------------------		
24	A:	--------------------------------		
25	A:	--------------------------------		
26	A:	--------------------------------		
27	A:	--------------------------------		
28	A:	--------------------------------		
29	A:	--------------------------------		
30	A:	--------------------------------		
31	A:	--------------------------------		
32	A:	x-------------------------------		
33	A:	--------------------------------		
34	A:	--------------------------------		
35	A:	--------------------------------		
36	A:	--------------------------------		
37	A:	--------------------------------		
38	A:	--------------------------------		
39	A:	--------------------------------		
40	A:	--------------------------------		
41	A:	--------------------------------		
42	A:	--------------------------------		
43	A:	--------------------------------		
44	A:	--------------------------------		
45	A:	--------------------------------		
46	A:	--------------------------------		
47	A:	--------------------------------		

Figure B.3: Differential characteristic of MD4 testcase C

i		A		E		W
-4	A:	--------------------------------	E:	--------------------------------		
-3	A:	--------------------------------	E:	--------------------------------		
-2	A:	--------------------------------	E:	--------------------------------		
-1	A:	--------------------------------	E:	--------------------------------		
0	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
1	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
2	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
3	A:	x-------------------------------	E:	????????????????????????????????	W:	????????????????????????????????
4	A:	--------------------------------	E:	????????????????????????????????	W:	????????????????????????????????
5	A:	--------------------------------	E:	????????????????????????????????	W:	????????????????????????????????
6	A:	--------------------------------	E:	????????????????????????????????	W:	????????????????????????????????
7	A:	--------------------------------	E:	????????????????????????????????	W:	????????????????????????????????
8	A:	--------------------------------	E:	--------------------------------	W:	????????????????????????????????
9	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
10	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
11	A:	--------------------------------	E:	--------------------------------	W:	????????????????????????????????
12	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
13	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
14	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
15	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
16	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
17	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------

Figure B.4: SHA256 hash collision over 18 rounds

i		A		E		W
-4	A:	--------------------------------	E:	--------------------------------		
-3	A:	--------------------------------	E:	--------------------------------		
-2	A:	--------------------------------	E:	--------------------------------		
-1	A:	--------------------------------	E:	--------------------------------		
0	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
1	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
2	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
3	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
4	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
5	A:	x???????????????????????????????	E:	????????????????????????????????	W:	????????????????????????????????
6	A:	--------------------------------	E:	????????????????????????????????	W:	????????????????????????????????
7	A:	--------------------------------	E:	????????????????????????????????	W:	????????????????????????????????
8	A:	--------------------------------	E:	????????????????????????????????	W:	????????????????????????????????
9	A:	--------------------------------	E:	????????????????????????????????	W:	--------------------------------
10	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
11	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
12	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
13	A:	--------------------------------	E:	--------------------------------	W:	????????????????????????????????
14	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
15	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
16	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
17	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
18	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
19	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
20	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------

Figure B.5: SHA-256 hash collision over 21 rounds

i		A		E		W
-4	A:	--------------------------------	E:	--------------------------------		
-3	A:	--------------------------------	E:	--------------------------------		
-2	A:	--------------------------------	E:	--------------------------------		
-1	A:	--------------------------------	E:	--------------------------------		
0	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
1	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
2	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
3	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
4	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
5	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
6	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
7	A:	x???????????????????????????????	E:	????????????????????????????????	W:	????????????????????????????????
8	A:	--------------------------------	E:	????????????????????????????????	W:	????????????????????????????????
9	A:	--------------------------------	E:	????????????????????????????????	W:	--------------------------------
10	A:	--------------------------------	E:	????????????????????????????????	W:	????????????????????????????????
11	A:	--------------------------------	E:	????????????????????????????????	W:	--------------------------------
12	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
13	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
14	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
15	A:	--------------------------------	E:	--------------------------------	W:	????????????????????????????????
16	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
17	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
18	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
19	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
20	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
21	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
22	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------

Figure B.6: SHA-256 hash collision over 23 rounds

i		A		E		W
-4	A:	--------------------------------	E:	--------------------------------		
-3	A:	--------------------------------	E:	--------------------------------		
-2	A:	--------------------------------	E:	--------------------------------		
-1	A:	--------------------------------	E:	--------------------------------		
0	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
1	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
2	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
3	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
4	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
5	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
6	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
7	A:	x???????????????????????????????	E:	????????????????????????????????	W:	????????????????????????????????
8	A:	--------------------------------	E:	????????????????????????????????	W:	????????????????????????????????
9	A:	--------------------------------	E:	????????????????????????????????	W:	--------------------------------
10	A:	--------------------------------	E:	????????????????????????????????	W:	????????????????????????????????
11	A:	--------------------------------	E:	????????????????????????????????	W:	--------------------------------
12	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
13	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
14	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
15	A:	--------------------------------	E:	--------------------------------	W:	????????????????????????????????
16	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
17	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
18	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
19	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
20	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
21	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
22	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------
23	A:	--------------------------------	E:	--------------------------------	W:	--------------------------------

Figure B.7: SHA-256 hash collision over 24 rounds

Appendix C

Runtimes retrieved

In this appendix, we present runtime results for Testcase 21. This selection gives a reasonable amount of testcases to present and other results are available in the exhaustive list referred to at

> `http://lukas-prokop.at/proj/megosat`

Runtimes have been retrieved for various CNF files with various SAT solver configurations. All following results have been determined using the cluster setup described in Appendix A. Testcases were run at most 1 day. Other testcases have also been run, but some have failed (not included in the table) during the run.

solver & version SAT solver name and version number

testcase Testcase identifier, always 21, because this is the complete of results for Testcase B.5

simplified CNF simplification used beforehand (or "none")

diff-desc Has the differential description been added?

phase Has Lingeling option `--phase=-1` been set?

pref Have preference variables been added?

ocnf Have difference variables been declared as set of variables to be evaluated first?

nbvars The number of variables considered according to the SAT solver

nbclauses The number of clauses considered according to the SAT solver

runtime The evaluated runtime in seconds

solver	version	testcase	simplified	diff-desc	phase	pref	ocnf	nbvars	nbclauses	runtime
cmsat	4.5.3	21	none	yes	no	no	no	116,800	911,629	29,424
cmsat	4.5.3	21	cmsat	yes	no	no	no	59,528	318,049	27,359
cmsat	4.5.3	21	lingeling	yes	no	no	no	116,800	404,126	25,272
cmsat	4.5.3	21	minisat	yes	no	no	no	81,650	534,965	47,023
cmsat	4.5.3	21	satelite	yes	no	no	no	90,236	720,755	20,296
cmsat	5.0.0	21	none	yes	no	no	no	116,800	911,629	7,855
cmsat	5.0.0	21	cmsat	yes	no	no	no	59,528	318,049	4,457
cmsat	5.0.0	21	lingeling	yes	no	no	no	116,800	404,126	7,257
cmsat	5.0.0	21	minisat	yes	no	no	no	81,650	534,965	3,327
cmsat	5.0.0	21	satelite	yes	no	no	no	90,236	720,755	909
glucose	4.0	21	cmsat	yes	no	no	no	59,528	318,049	17,967
glucose	4.0	21	lingeling	yes	no	no	no	104,860	404,126	10,388
glucose	4.0	21	minisat	yes	no	no	no	81,650	534,965	19,210
glucose	4.0	21	satelite	yes	no	no	no	90,236	720,755	16,278
glucose-syrup	4.0	21	none	yes	no	no	no	116,800	894,072	25,909
glucose-syrup	4.0	21	cmsat	yes	no	no	no	59,528	318,049	13,281
glucose-syrup	4.0	21	lingeling	yes	no	no	no	104,860	404,126	12,123
glucose-syrup	4.0	21	minisat	yes	no	no	no	81,650	534,965	15,501
glucose-syrup	4.0	21	satelite	yes	no	no	no	90,236	720,755	27,215
lingeling	ats1	21	none	no	no		no	116,800	636,838	28,621
lingeling	ats1	21	none	no	yes		no	116,800	636,838	19,717
lingeling	ats1	21	none	yes	no	no	no	116,800	911,629	5,513
lingeling	ats1	21	none	yes	yes	no	no	116,800	911,629	1,140
lingeling	ats1	21	minisat	yes	no	no	no	81,650	534,965	1,110
lingeling	ats1	21	minisat	yes	yes	no	no	81,650	534,965	1,145

lingeling	ats101	21	none	no	no		no	116,800	636,838	27,599
lingeling	ats101	21	none	no	yes		no	116,800	636,838	19,247
lingeling	ats101	21	cmsat	no	no		no	63,555	358,308	32,099
lingeling	ats101	21	cmsat	no	yes		no	63,555	358,308	30,244
lingeling	ats101	21	minisat	no	no		no	65,572	414,470	30,205
lingeling	ats101	21	minisat	no	yes		no	65,572	414,470	20,193
lingeling	ats101	21	satelite	no	no		no	74,562	486,714	44,494
lingeling	ats101	21	satelite	no	yes		no	74,562	486,714	29,234
lingeling	ats101	21	none	yes	no	no	no	116,800	911,629	2,282
lingeling	ats101	21	none	yes	yes	no	no	116,800	911,629	1,172
lingeling	ats101	21	cmsat	yes	no	no	no	59,528	318,049	892
lingeling	ats101	21	cmsat	yes	yes	no	no	59,528	318,049	537
lingeling	ats101	21	lingeling	yes	no	no	no	104,860	404,126	1,502
lingeling	ats101	21	lingeling	yes	yes	no	no	104,860	404,126	1,231
lingeling	ats101	21	minisat	yes	no	no	no	81,650	534,965	596
lingeling	ats101	21	minisat	yes	yes	no	no	81,650	534,965	1,240
lingeling	ats101	21	satelite	yes	no	no	no	90,236	720,755	791
lingeling	ats101	21	satelite	yes	yes	no	no	90,236	720,755	1,124
lingeling	ats102	21	none	no	no		no	116,800	636,838	35,305
lingeling	ats102	21	satelite	no	no		no	74,562	486,714	69,722
lingeling	ats102	21	lingeling	yes	no	no	no	104,860	404,126	3,323
lingeling	ats102	21	satelite	yes	no	no	no	90,236	720,755	3,678
lingeling	ats104	21	cmsat	no	no		no	63,555	358,308	45,547
lingeling	ats104	21	satelite	no	no		no	74,562	486,714	55,627
lingeling	ats104	21	none	yes	no	no	no	116,800	911,629	3,110
lingeling	ats104	21	cmsat	yes	no	no	no	59,528	318,049	1,533

lingeling	ats104	21	lingeling	yes	no	no	no	104,860	404,126	3,631
minisat	4.0	21	none	yes	no	no	no	116,800	894,072	62,327
minisat	4.0	21	cmsat	yes	no	no	no	59,528	318,049	2,288
minisat	4.0	21	lingeling	yes	no	no	no	104,860	404,126	7,096
minisat	4.0	21	minisat	yes	no	no	no	81,650	534,965	17,990
minisat	4.0	21	satelite	yes	no	no	no	90,236	720,755	23,399
minisat-core	4.0	21	cmsat	no	no		no	63,555	358,308	60,458
minisat-core	4.0	21	none	yes	no	no	no	116,800	894,072	38,376
minisat-core	4.0	21	cmsat	yes	no	no	no	59,528	318,049	10,776
minisat-core	4.0	21	lingeling	yes	no	no	no	104,860	404,126	14,440
minisat-core	4.0	21	minisat	yes	no	no	no	81,650	534,965	18,914
minisat-core	4.0	21	satelite	yes	no	no	no	90,236	720,755	9,626
plingeling	ats101	21	none	no	no		no	116,800	636,838	42,025
plingeling	ats101	21	minisat	no	no		no	65,572	414,470	67,220
plingeling	ats101	21	satelite	no	no		no	74,562	486,714	47,064
plingeling	ats101	21	none	yes	no	no	no	116,800	911,629	2,545
plingeling	ats101	21	cmsat	yes	no	no	no	59,528	318,049	1,928
plingeling	ats101	21	minisat	yes	no	no	no	81,650	534,965	2,553
plingeling	ats101	21	satelite	yes	no	no	no	90,236	720,755	3,204
treengeling	ats101	21	none	yes	no	no	no	116,800	911,629	6,419
treengeling	ats101	21	cmsat	yes	no	no	no	59,528	318,049	5,118
treengeling	ats101	21	minisat	yes	no	no	no	81,650	534,965	27,567

Bibliography

[1] Bernd Bischl et al. "ASlib: A benchmark library for algorithm selection". In: *Artificial Intelligence* 237 (2016), pp. 41 –58. ISSN: 0004-3702. DOI: http://dx.doi.org/10.1016/j.artint.2016.04.003. URL: http://www.sciencedirect.com/science/article/pii/S0004370216300388 (visited on 07/23/2016).

[2] Christophe De Cannière and Christian Rechberger. "Finding SHA-1 Characteristics: General Results and Applications". In: *ASIACRYPT*. Ed. by Xuejia Lai and Kefei Chen. Vol. 4284. LNCS. Springer, 2006, pp. 1–20. ISBN: 3-540-49475-8. URL: http://dx.doi.org/10.1007/11935230_1 (visited on 08/23/2016).

[3] Intel Corporation. *Intel Xeon Processor X5690 (12M Cache, 3.46 GHz, 6.40 GT/s Intel QPI) Specifications*. URL: http://ark.intel.com/products/52576/Intel-Xeon-Processor-X5690-12M-Cache-3_46-GHz-6_40-GTs-Intel-QPI (visited on 04/05/2016).

[4] Magnus Daum. "Cryptanalysis of Hash functions of the MD4-family". PhD thesis. Ruhr-Universität Bochum, Universitätsbibliothek, 2005.

[5] Hans Dobbertin. "Cryptanalysis of MD4". In: *Journal of Cryptology* 11.4 (1998), pp. 253–271. ISSN: 1432-1378. DOI: 10.1007/s001459900047. URL: http://dx.doi.org/10.1007/s001459900047 (visited on 03/15/2016).

[6] Christoph Dobraunig, Maria Eichlseder, and Florian Mendel. "Analysis of SHA-512/224 and SHA-512/256". In: *Advances in Cryptology–ASIACRYPT 2015*. Springer, 2014, pp. 612–630.

[7] Vijay Ganesh and David L Dill. "A decision procedure for bit-vectors and arrays". In: *International Conference on Computer Aided Verification*. Springer. 2007, pp. 519–531.

[8] hh. *SATLIB - The Satisfiability Library*. URL: http://www.satlib.org/ (visited on 08/27/2016).

[9] National Institute of Standards Information Technology Laboratory and Technology. "Federal Information Processing Standards Publication 180-4". In: *National Bureau of Standards, US Department of Commerce* (2015). URL: http://dx.doi.org/10.6028/NIST.FIPS.180-4 (visited on 05/10/2016).

[10] Robert G. Jeroslow and Jinchang Wang. "Solving Propositional Satisfiability Problems". In: *Annals of Mathematics and Artificial Intelligence* 1.1-4 (Sept. 1990), pp. 167–187. ISSN: 1012-2443. DOI: 10.1007/BF01531077. URL: http://dx.doi.org/10.1007/BF01531077.

[11] Dmitry Khovratovich, Christian Rechberger, and Alexandra Savelieva. "Bicliques for preimages: attacks on Skein-512 and the SHA-2 family". In: *Fast Software Encryption*. Springer. 2012, pp. 244–263.

[12] Mario Lamberger and Florian Mendel. "Higher-Order Differential Attack on Reduced SHA-256." In: *IACR Cryptology ePrint Archive* 2011 (2011), p. 37.

[13] Massimo Lauria. *CNFgen – Cool benchmarks for your SAT solver!* URL: https://massimolauria.github.io/cnfgen/ (visited on 08/0½016).

[14] Lenovo Group Ltd. *ThinkPad X220 Tablet (4299) - Onsite (2011)*. URL: http://www.lenovo.com/shop/americas/content/pdf/system_data/x220t_tech_specs.pdf (visited on 04/05/2016).

[15] Florian Mendel, Tomislav Nad, and Martin Schläffer. "Improving local collisions: new attacks on reduced SHA-256". In: *Advances in Cryptology–EUROCRYPT 2013*. Springer, 2013, pp. 262–278.

[16] Florian Mendel, Tomislav Nad, and Martin Schläffer. "Improving local collisions: new attacks on reduced SHA-256". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2013, pp. 262–278.

[17] RC Merkle. "Secrecy, Authentification, and Public Key Systems". PhD thesis. PhD thesis, Stanford University, Dpt of Electrical Engineering, 1979.

[18] Ilya Mironov and Lintao Zhang. "Applications of SAT solvers to cryptanalysis of hash functions". In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2006, pp. 102–115.

[19] Matthew W Moskewicz et al. "Chaff: Engineering an efficient SAT solver". In: *Proceedings of the 38th annual Design Automation Conference*. ACM. 2001, pp. 530–535.

[20] Yusuke Naito et al. "Improved Collision Attack on MD4". In: (2005), pp. 1–5. URL: http://eprint.iacr.org/ (visited on 03/15/2016).

[21] Ivica Nikolić and Alex Biryukov. "Collisions for step-reduced SHA-256". In: *International Workshop on Fast Software Encryption*. Springer. 2008, pp. 1–15.

[22] Eugene Nudelman et al. "Satzilla: An algorithm portfolio for SAT". In: *Solver description, SAT competition 2004* (2004).

[23] Eugene Nudelman et al. "Understanding random SAT: Beyond the clauses-to-variables ratio". In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2004, pp. 438–452.

[24] prokls. *MD4 in pure Python 3.4*. URL: https://gist.github.com/prokls/86b3c037df19a8c957fe (visited on 04/10/2015).

[25] Lukas Prokop. *Using SAT Solvers to Detect Contradictions in Differential Characteristics*. URL: http://lukas-prokop.at/proj/bakk_iaik/thesis.pdf (visited on 08/23/2016).

[26] Ronald Rivest. *The MD4 Message Digest Algorithm*. RFC 1186. The Internet Engineering Task Force, 1990, pp. 1–18. URL: https://tools.ietf.org/html/rfc1186 (visited on 03/15/2016).

[27] Ronald Rivest. *The MD4 Message-Digest Algorithm*. RFC 1320. The Internet Engineering Task Force, 1992, pp. 1–20. URL: https://tools.ietf.org/html/rfc1320 (visited on 03/15/2016).

[28] Yu Sasaki et al. "New Message Difference for MD4". In: (2007), pp. 1–20. URL: http://www.iacr.org/archive/fse2007/45930331/45930331.pdf (visited on 03/15/2016).

[29] Yu Sasaki et al. "New message difference for MD4". In: *International Workshop on Fast Software Encryption*. Springer. 2007, pp. 329–348.

[30] SATcompetition@satlive.org. *SAT competitions*. URL: http://satcompetition.org/ (visited on 08/26/2016).

[31] Martin Schläffer and Elisabeth Oswald. "Searching for differential paths in MD4". In: *Fast Software Encryption*. Springer. 2006, pp. 242–261.

[32] Bart Selman, David G Mitchell, and Hector J Levesque. "Generating hard satisfiability problems". In: *Artificial intelligence* 81.1 (1996), pp. 17–29.

[33] Patrick Stach. *MD4 collision generator*. URL: http://crppit.epfl.ch/documentation/Hash_Function/Fastcoll_MD4/md4coll.c (visited on 04/05/2016).

[34] S. Turner and L. Chen. *The MD4 Message Digest Algorithm*. RFC 6150. The Internet Engineering Task Force, 2011, pp. 1–10. URL: https://tools.ietf.org/html/rfc6150 (visited on 03/15/2016).

[35] Xiaoyun Wang et al. "Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD." In: *IACR Cryptology ePrint Archive* 2004 (2004), p. 199.

[36] Lin Xu et al. "SATzilla: portfolio-based algorithm selection for SAT". In: *Journal of Artificial Intelligence Research* (2008), pp. 565–606.