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 16, 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. Looking at the differential behavior of the underlying operations of the hash algorithm shows how differential values propagate in the algorithm. The goal is to find a differential characteristic whose differences cancel out in the output. Once such a differential characteristic was discovered, in a second step the actual values for those differences are defined yielding an actual hash collision.

Finding a differential characteristic can be a cumbersome and tedious task. Whereas propagation can be automated using dedicated tools, finding an initial differential characteristic is a difficult task as it can be specified with arbitrary levels of granularity.

SAT solvers inherently implement both tasks. They consecutively propagate values which narrow the search space. The probability to find a satisfiable assignment increases if the narrowed search space has many satisfiable assignments. And finally the assignment reveals initial values. On the other hand, SAT solvers have no notion of differential values and therefore problem encoding becomes an important topic.

In this thesis we look at differential characteristics and encode them as SAT problem. A SAT solver tells us whether a differential characteristic can represent a hash collision or not. We implemented a framework which allows us to verify differential behavior for integer operations. We then looked at the encoded problems in details and tried to change the encoding to improve the runtime of the SAT solver. We also provide a small CNF analysis library to compare an encoded problem with others.

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.

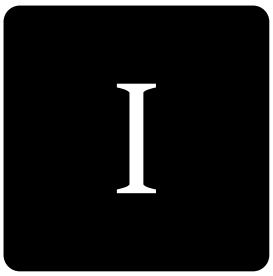
Also thanks to Roderick Bloem and Armin Biere who organized a meeting one year before submitting this work defining the main approaches involved in this thesis. Armin Biere released custom lingeling versions for us, e.g. featuring "more important" clauses in lingeling. He also provided further analysis for our testcases.

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 code is 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 and Linux Libertine Font.

Contents



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 will consider the hash algorithms MD4 and SHA-256. They use basic arithmetic functions like addition and bit-level functions such as XOR to transform an input to a hash value. We use a bit vector as input to this implementation and all operations applied to this bit vector will be represented as clauses of a SAT problem. Additionally we represent differential characteristics of hash collisions as SAT problem. If and only if satisfiability is given, the particular differential state is achievable using two different inputs leading to the same output. As far as SAT solvers return an actual model satisfying that state, we get an actual hash collision which can be verified and visualized. If the internal state of the hash algorithm is too large, the attack can be computationally simplified by modelling only a subset of steps of the hash algorithm or changing the modelled differential path.

Based on experience with these kind of problems with previous non-SAT-based 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 off 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.

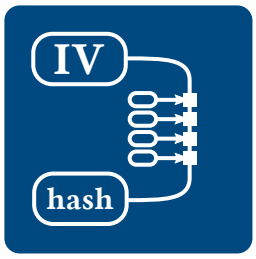
In Chapter ?? we discuss SAT solving. We give a brief overview over used SAT solvers and discuss how we can speed up SAT solvers for cryptographic problems.

In Chapter ?? 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 ?? 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 ?? we show data as result of our work. Runtimes are the main part of this chapter, but also results of the SAT features project are presented.

In Chapter ?? we suggest future work based on our results.



Chapter 2

Hash algorithms

In this chapter we will define hash functions and their desired security properties. Followingly we look at SHA256 and MD4 as established hash functions. We will represent them with Boolean algebra (in chapter **??**) to make it possible 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 [16]. 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 [10, p. 13–15] and followingly 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 SHA256 apply padding to their input to nor-

malize 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 only need to find 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

One example showing the use of hash functions as primitives are JSON Web Tokens (JWT) specified in RFC 7519 [8]. Its application allows web developers to represent claims to be transferred between two parties.

Section 8 defines implementation requirements and refers to RFC 7518 [5], which specifies cryptographic algorithms such as "HMAC SHA-256" to be implemented. It is (besides "none") the only required signature and MAC algorithm.

2.2 MD4

MD4 is a cryptographic hash function originally described in RFC 1186 [13], updated in RFC 1320 [14] and declared obsolete by RFC 6150 [18]. It was invented by Ronald Rivest in 1990 with properties given in Table 2.1. In 1995 [2] successful full-round attacks have been found to break collision resistance. Followingly preimage and second-preimage resistance in MD4 have been broken as well. Some of those attacks are described in [15] and [11]. We derived a Python 3 implementation based on a Python 2 implementation and made it available on github [12].

block size	512 bits	namely variable `block` in RFC 1320 [14]
digest size	128 bits	as per Section 3.5 in RFC 1320 [14]
internal state size	128 bits	namely variables A, B, C and D
word size	32 bits	as per Section 2 in RFC 1320 [14]

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 Boolean values true in arguments is at least 2. The Boolean `XOR` function returns true if the number of Boolean values true in arguments is odd.

Using the logical operators $\wedge$ (AND), $\vee$ (OR) and $\neg$ (NEG) we can define them as (see section **??** for a thorough discussion of these operators):

$$\texttt{IF}(X,Y,Z) := (X \wedge Y) \vee (\neg X \wedge Z) \tag{2.1}$$

$$\texttt{MAJ}(X,Y,Z) := (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z) \tag{2.2}$$

$$\begin{aligned}\texttt{XOR}(X,Y,Z) &:= (X \wedge \neg Y \wedge \neg Z) \vee (\neg X \wedge Y \wedge \neg Z) \\ &\quad \vee (\neg X \wedge \neg Y \wedge Z) \vee (X \wedge Y \wedge Z) \\ &:= (X \oplus Y \oplus Z)\end{aligned} \tag{2.3}$$

In the following a brief overview over MD4's design 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. Followingly, length appending takes place. Represent the length of the input (without the previous modifications) in binary and take its first 64 less significant bits. Append those 64 bits to the input.

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

$[\texttt{A}_{-4}]$ `01234567` $[\texttt{A}_{-1}]$ `89abcdef` $[\texttt{A}_{-2}]$ `fedcba98` $[\texttt{A}_{-3}]$ `76543210`

Round function with state variable updates. The round function is applied in three 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. The values of X and p are defined as follows:

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.

2.3 SHA-256

SHA-256 is a hash function from the SHA-2 family designed by the National Security Agency (NSA) and published in 2001 [4]. 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 [6]. The best known collision attack breaks collision resistance for 31 rounds of SHA-256 [9] and pseudo-collision resistance for 46 rounds [7].

block size	512 bits	as per Section 1 of the standard [4]
digest size	256 bits	mentioned as Message Digest size [4]
internal state size	256 bits	as per Section 1 of the standard [4]
word size	32 bits	as per Section 1 of the standard [4]

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:

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

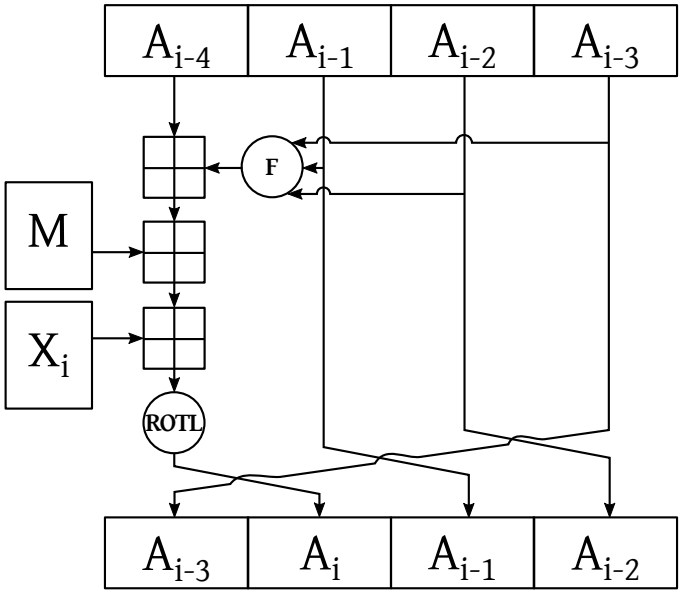


Figure 2.1: MD4 round function updating state variables

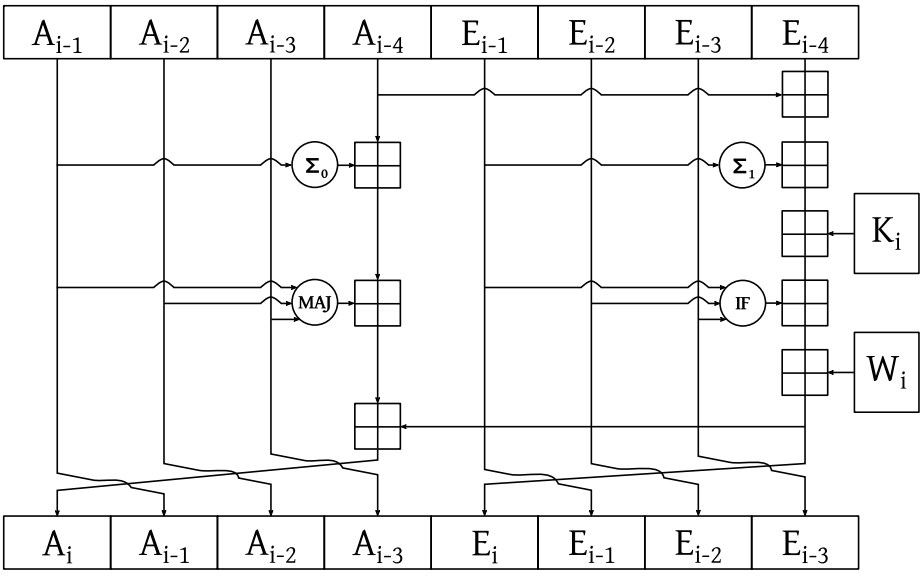


Figure 2.2: SHA-256 round function as characterized in [3]

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.

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

Padding and length extension. The padding and length extension scheme of MD4 is used also in SHA-256. Append bit 1 and 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 [4, Section 6.2.2]) takes place before running the round function. The eight state variables are initialized with the following hexadecimal values:

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

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.

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

W_i and K_i refer to the previously initialized values.

Computation of intermediate hash values. Intermediate hash values for the Davies-Meyer construction are initialized with the following values:

$$H_0^{(0)} := A_{-1} \qquad H_1^{(0)} := A_{-2} \qquad H_2^{(0)} := A_{-3} \qquad H_3^{(0)} := A_{-4}$$
$$H_4^{(i)} := E_{-1} \qquad H_5^{(i)} := E_{-2} \qquad H_6^{(i)} := E_{-3} \qquad H_7^{(i)} := E_{-4}$$

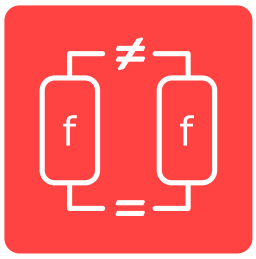
Every block creates its own E_i and A_i values for $60 \leqslant i < 64$. These are used to compute the next intermediate values:

$$H_0^{(j)} := A_{63} + H_0^{(i-1)} \qquad H_4^{(j)} := E_{63} + H_4^{(i-1)}$$
$$H_1^{(j)} := A_{62} + H_1^{(i-1)} \qquad H_5^{(j)} := E_{62} + H_5^{(i-1)}$$
$$H_2^{(j)} := A_{61} + H_2^{(i-1)} \qquad H_6^{(j)} := E_{61} + H_6^{(i-1)}$$
$$H_3^{(j)} := A_{60} + H_3^{(i-1)} \qquad H_7^{(j)} := E_{60} + H_7^{(i-1)}$$

Finalization. The final hash digest of size 256 bits is provided as

$$H_0^{(N)} \| H_1^{(N)} \| H_2^{(N)} \| H_3^{(N)} \| H_4^{(N)} \| H_5^{(N)} \| H_6^{(N)} \| H_7^{(N)}$$

where N denotes the index of the last block and operator $\|$ denotes concatenation. Hence $H_0^{(N)}$ are the four least significant bytes of the digest.



"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 considerations 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 [19] 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 [2]. 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 [17] implements Wang's approach and can find MD4 collisions in few seconds on my Thinkpad x220 setup specified in Appendix ??.

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 [19]. 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`. A message represents one block of 512 bits.

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 arbitrarily.

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 implementations, but provide slight modifications in few bits. Differential cryptanalysis is based on the idea to consider those execution states and tracing those difference 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 visually recognizable difference in the output.

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

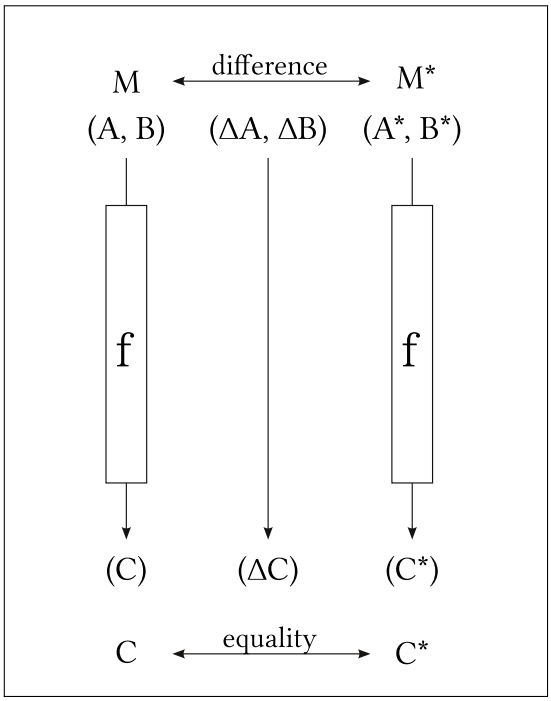


Figure 3.1: Typical attack setting for a collision attack: Hash function f is applied to two inputs M and M^* which differ by some predefined bits. M describes the difference between these values. A hash collision is given if and only if output values C and C^* show the same value. In differential cryptanalysis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

differences in a particular hash algorithm is called *differential path*. Empirical results in differential cryptanalysis indicate that spare paths 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.

Theorem 3.1

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

Definition 3.2

The propagation of differences in a particular function is called *differential path*. The complete differential state during a computation is called *differential characteristic*.

bit	binary	hexadecimal 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		u10101101100101110010010011110 10
		n01n10011101010110100101011110 00
		010001011101110n1000111000110001

TABLE 3.2: 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.

3.3 Differential notation

Differential notation helps us to visualize differential characteristics by defining so-called *generalized bit conditions*. It was introduced by Christian Rechberger and Christophe de Cannière in 2006 [1, Section 3.2], inspired by *signed differences* by Wang et al. and is shown in Table 3.3.

Consider two hash algorithm implementations. Let x_i be some bit from the first implementation and let x_i^* be the corresponding bit from the second implementation. 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 3.2.

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 (expect 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 -.

(x_i, x_i^*)	$(0,0)$	$(1,0)$	$(0,1)$	$(1,1)$
?	✓	✓	✓	✓
-	✓			✓
x		✓	✓	
0	✓			
u		✓		
n			✓	
1				✓
#				

(x_i, x_i^*)	$(0,0)$	$(1,0)$	$(0,1)$	$(1,1)$
3	✓	✓		
5	✓		✓	
7	✓	✓	✓	
A		✓		✓
B	✓	✓		✓
C			✓	✓
D	✓		✓	✓
E		✓	✓	✓

TABLE 3.3: Differential notation as introduced in [1]. 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^*.

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 most basic exercise to the reader. Consider a matrix with two input rows and one output row. The values of the first two lines are added to

TODO:

- illustrate how differences propagate by an addition example illustrated in differential notation
- reference to Magnus Daum's thesis (≠ signed bit diff desc)

3.5 Differential path

TODO:

- refer to some testcase which shows a differential path with many unresolved differences.
- Then show the corresponding testcase where `?` became `-` and `x`.
- Illustrate how MD4 and SHA-256 descriptions maps to matrix representation.

-		00 00 11 11
-	⇒	00 11 00 11
-		00 11 11 00

TABLE 3.5: A simple 1-bit addition example: On the left the differential characteristic is given

"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. We will learn that any computation can be represented as satisfiability problem. In Chapter **??** 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.

Boolean functions are characterized by their corresponding truth table.

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

v	$f(v)$
1	0
0	1

(C) NOT

TABLE 4.1: Truth tables for AND, OR and NOT

Table **??** 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 **??**. The description shows the corresponding definition of f using AND, OR and NOT only.

Now let g be some k-ary function. Let $(a_0, a_1, \ldots, a_k)$ be the k input arguments to g and $x_1 := g(a_0, a_1, \ldots, a_k)$. Then we can again look at Table **??** 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.

This is known as the famous $\mathscr{P} \overset{?}{\neq} \mathscr{NP}$ problem.

However, SAT solver can take advantage of the problem's description. For example consider function f in Display **??**.

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

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 good heuristics to find some model for a given SAT problem as fast as possible. Biyearly SAT competitions take place to challenge SAT solvers in a set of benchmarks. The committee evaluates the most successful SAT solvers solving the most problems within a given time frame.

SAT 2016 is currently ongoing, but 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.

In chapter **??** we will look at runtime results shown by (but not limited to) those SAT solvers.

4.2 The DIMACS de-facto standard

Definition 4.7

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

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

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.

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

Display **??** shows 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 **??** 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 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 minus symbol. Then one line represents one clause and must be terminated with a zero symbol after a space. All lines are conjuncted to form a CNF.

Variations of the DIMACS de-facto standard also allow multiline clauses (the zero symbol constitutes the end of a clause) or arbitrary whitespace instead of spaces. The syntactical details are individually published on a per competition basis.

Listing 4.1: Display ?? represented in DIMACS format

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

4.3 Terminology

Given a conjunctive structure of disjunctions, we can define define terms related to this structure:

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.

4.4 Basic SAT solving techniques

Definition 4.11

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

4.4.1 Boolean constraint propagation (BCP)

One of the most basic techniques to SAT solving is boolean constraint propagation. It is so fundamental that SATzilla, introduced in section **??**, applies it even when 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 $A \vee \bot$ 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 value of a literal. Watched Literals is a neat technique to reduce the number of checks.

TODO: finish

4.5 SAT solvers in use

TODO: discuss minisat, treengeling, lingeling, plingeling, cmsat 4.5, cmsat 5, glucose syrup static, glucose static, lingeling ats101, ats102, ats104

TODO: discuss how Armin modified lingeling

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

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. 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:

1. Given an arbitrary literal. What is the percentage it is positive?
2. What is the variables / clauses ratio?
3. How many literals occur only 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 SAT features and CNF analysis

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 retrieve 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 for evaluation algorithms. Sticking to this convention implies that evaluation of satisfiability must not be necessary to evaluate a SAT feature under the assumption that $\mathscr{P} \neq \mathscr{NP}$. 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 [**satzilla2004**, **satzilla2008**] in 2012. The authors systematically defined 138 SAT features categorized in 12 groups. The features themselves are not defined formally, but an implementation is provided bundled with example data. 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 - nvars, divided by nvars

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

POSNEG-RATIO-CLAUSE-mean mean of $2 \cdot \left\| 0.5 - \text{pos}_c/\text{length}_c \right\|$ where pos_c is the number of positive literals of clause c and length_c clause length of c

POSNEG-RATIO-CLAUSE-entropy like POSNEG-RATIO-CLAUSE-mean but 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 2 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.

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 runtime are used as training data for Machine Learning. One example using SAT features for algorithm selection is ASlib [**aslib**].

Previous work has shown that expensive algorithms can provide useful data in a small time frame if they are limited to a constant subproblem size.

5.3 Statistical features

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

Arithmetic mean (or short: mean) is defined as

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

Standard deviation (or short: sd) 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

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

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 $0 \cdot log_2(0) \coloneqq 0$.

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 $\{0, 5, 10, \ldots, 90, 95\}$ and $k = 95$ counts $\frac{k}{100} \leqslant n_l \leqslant \frac{k+5}{100}$.

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 o

In the following section we want to evaluate SAT features and compare test cases.

TODO: classify SAT features

5.5 CNF dataset

TODO: to retrieve a unique set of CNF files, it was necessary to devise a hash algorithm

TODO: list data set explicitly

TODO: Illustrate basic properties about those CNF files

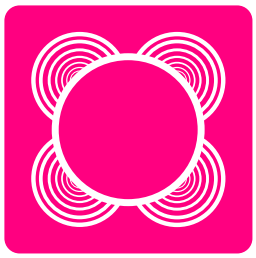
5.6 The average SAT problem

Proposition 5.1

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

TODO: what is an average SAT problem?

Given a set of clauses, return a subset of clauses satisfying given criterion	
`clauses_allLitsNeg`	all literals are negative
`clauses_oneLitNeg`	exactly one literal is negative
`clauses_geqOneLitNeg`	more than one literal is negative
`clauses_allLitsPos`	all literals are positive
`clauses_oneLitPos`	exactly one literal is positive
`clauses_geqOneLitPos`	more than one literal is positive
`clauses_length1`	clause contains exactly one literal ("unit clause")
`clauses_length2`	clause contains exactly two literals
`clauses_unique`	clause did not yet occur
`clauses_tautological`	clause contains some literal and its negation
Given a set of literals/variables, return Boolean property	
`literals_existential`	literal does not occur negated
`literals_unit`	literal occurs in clause of length 1
`literals_contradiction`	literal occurs with its negation on one clause
`literals_1occ`	literal occurs only in one clause once
`literals_2occs`	literal occurs two times in clauses
`literals_3occs`	literal occurs three times in clauses
`variables_unit`	variable occurs in clause of length 1
Given a set of clauses, return real number based on this clause	
`clauses_mapLength`	number of literals in clause
`clauses_mapRatioPosNeg`	number of positive literals divided by total number of literal
`clauses_mapNumPos`	number of positive literals in clause
Given one clause, return Boolean property	
`clauselits_someEx`	any is literal existential
`clauselits_allEx`	all literals are existential
`clauselits_someUnit`	contains unit variable
`clauselits_someContra`	contains contradiction variable
`clauselits_all1occ`	all variables occur only once in all clauses
`clauselits_all12occ`	all variables occur only once or twice in all clauses
Given all clauses, return the following property	
`concomp_variable`	number of connected components where two variables are in the same component iff they occur in at least one clause together
`concomp_literal`	number of connected components where two literals are in the same component iff they occur in at least one clause together
`xor2_count`	Number of clause pairs $(a \vee b, \neg a \vee b)$ for two variables a and b



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

Chapter 6

Problem encoding

We already discussed how SAT solvers work and which input they take. We also sketched how hash algorithm properties got broken using differential cryptanalysis. In this section we combine those subjects and describe how we designed an attack setting.

6.1 STP approach

TODO: describe approach

6.2 Two instances and its difference

TODO: we wrote a tool called algotocnf for our differential needs

6.3 Approach with a differential description

TODO: approach with differential desc

6.4 Influencing evaluation order

Chapter 7

Results

7.1 SAT features results

TODO: do cryptoproblems distinguish from other problems?

TODO: do our benchmark distinguish from other problems?

TODO: answer the 8 questions posed previously

7.2 Attack results

TODO: is simplification worth it?

TODO: discuss runtime and development, tuning by doing difference variables first, diff desc makes a difference

Appendix **??** provide a more exhaustive list of runtimes retrieved.

7.3 Related work

7.4 Conclusion

TODO: we attacked MD4 and SHA2, but can see the problems with

7.5 Contributions

To strengthen Reproducible Research, 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/`

Several subprojects are part of this master thesis:

algotocnf
: A python library implementing the encoding described in chapter **??**.

Python3 library and program: `https://github.com/prokls/algotocnf`

cnf-hash
: A standardized way to produce a unique hash for CNF files

Go implementation: `https://github.com/prokls/cnf-hash-go`
Python3 implementation: `https://github.com/prokls/cnf-hash-py`
Testsuite: `https://github.com/prokls/cnf-hash-tests2`

cnf-analysis
: Evaluate SAT features for a given CNF file.

Go implementation: `https://github.com/prokls/cnf-analysis-go`
Python3 implementation: `https://github.com/prokls/cnf-analysis-py`
Testsuite: `https://github.com/prokls/cnf-analysis-tests`

Chapter 8

Summary and Future Work

8.1 Summary of results

8.2 Future work

Appendices

Appendix A

Illustration

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:	01110110010011111111011100u110001	W:	u1010110110010111001001001111010
2	A:	10101011010000000011100u01n1110010	W:	n01n10011101010110100101011111000
3	A:	101011u10011110101010010010010001	W:	01010111101001111010010111101110
4	A:	00101100011000110101010111110010	W:	11011110011101001000101000111100
5	A:	00011010011000101 0u1101000000001	W:	11011100110000110110011010110011
6	A:	00011011 00unuu1100010000001111010	W:	10110110100000011101000000010000
7	A:	00101011100000010unn011001010000	W:	00111011001010100101110110011111
8	A:	01110011001000 1u1111111110110000	W:	11000110100111010111000110110011
9	A:	101011n01unnu00011111001100111111	W:	11111001111010011001000110011000
10	A:	10n00100100001010100000010101110	W:	11010111100111111000000001011110
11	A:	u10001101011011000100101011111111	W:	10100110001110111011001011101000
12	A:	001011u00u10101111111100011111011	W:	01000101110111 0n1000111000110001
13	A:	10un1n0100110001010000011110010 1	W:	10010111111000110001111111100101
14	A:	00001010010100011000100011010110	W:	00100111100101001011111000001000
15	A:	0001111010101u0101100111011010100	W:	10111001111010001100001111101001
16	A:	n00n0un01101001010011011011111		
17	A:	00011111001110100001001000011110		
18	A:	01010111000011010000000010010100		
19	A:	u1n1000000010111100110101100010 0		
20	A:	n1un10011111110111010000001101 00		
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:	01001111111010010011010000101111		
33	A:	00111000001111010110111011100100		
34	A:	00100000011101011110100000010101		
35	A:	n01000000011001100000100011100 10		
36	A:	n00001111110101111011111001011001		
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 A.1: One of the original MD4 collision given by Wang, et al.

Appendix B

Hardware setup

In the following we introduce two hardware setups which were used to run our testcases. The first setup is referred to as "Thinkpad x220" throughout the document whereas the second setup is referred to as "Cluster".

Type model	Thinkpad Lenovo x220 tablet, 4299-2P6
Processor	Intel i5-2520M, 2.50 GHz, dual-core, Hyperthreaded
RAM	16 GB (extension to common retail setup)
Memory	160 GB SSD
L3 cache size	3072 KB

Table B.1: Thinkpad x220 Tablet specification [**thinkpadx220**]

Processor	Intel Xeon X5690, 3.47 GHz, 6 cores, Hyperthreaded
RAM	192 GB
L3 cache size	12288 KB

Table B.2: Cluster node nehalem192g0 specification [**intelX5690**]

Appendix C

Testcases

Figures ??, ??, ?? and ?? show testcases used to test performance measures.

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:	x----------	W:	---x----------
1	A:	----------	W:	----------
2	A:	----------x----------	W:	x----------
3	A:	xxx----------	W:	----------
4	A:	----------xx	W:	x----------
5	A:	------xxxxxxxxxxxxxxx-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:	----------		

TABLE C.1: TODO description

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:	????????????????????????????????	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:	----------		

Table C.2: TODO description

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:	????????????????????????????????	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:	----------		

TABLE C.3: TODO description

i		$\nabla S_{i,0}$	$\nabla S_{i,1}$	$\nabla S_{i,2}$
−4	A:	01100111010001010010001100000001		
−3	A:	00010000001100100101010001110110		
−2	A:	10011000101110101101110011111110		
−1	A:	11101111110001101101010111000 1001		
0	A:	??????????????????????????????????	W:	??????????????????????????????????
1	A:	??????????????????????????????????	W:	??????????????????????????????????
2	A:	??????????????????????????????????	W:	??????????????????????????????????
3	A:	??????????????????????????????????	W:	??????????????????????????????????
4	A:	??????????????????????????????????	W:	??????????????????????????????????
5	A:	??????????????????????????????????	W:	??????????????????????????????????
6	A:	??????????????????????????????????	W:	??????????????????????????????????
7	A:	??????????????????????????????????	W:	??????????????????????????????????
8	A:	??????????????????????????????????	W:	??????????????????????????????????
9	A:	??????????????????????????????????	W:	??????????????????????????????????
10	A:	??????????????????????????????????	W:	??????????????????????????????????
11	A:	??????????????????????????????????	W:	??????????????????????????????????
12	A:	??????????????????????????????????	W:	??????????????????????????????????
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:	--------------------------------		

TABLE C.4: TODO description

Appendix D

Runtimes retrieved

Index

k-clause, 23

AND (Boolean function), 20
Assignment, 19, 21
Avalanche effect, 12

Bit condition, 14
Boolean function, 19

Clause, 23
Clause length, 23
Collision, 4
Collision resistance, 4
Conjunction, 22
Conjunctive Normal Form, 22

Definite clause, 23
Differential characteristic, 13
Differential notation, 14
Differential path, 13
Disjunction, 22
Disjunctive Normal Form, 22

Equisatisfiability, 23
Existential literal, 23

Feature value, 26

Generalized bit condition, 14

Hash collision, 12
Hash function, 3
Hash value, 3
Horn clause, 23

Left-rotation, 7
Left-shift, 7
lingeling, 24
Literal, 22

MD4, 5
Model, 21

Negated literal, 23
Negative literal, 22
NOT (Boolean function), 20

OR (Boolean function), 20

Positive literal, 22
Preimage, 3
Preimage resistance, 3
Pseudo collision, 12

Right-rotation, 7
Right-shift, 7

SAT feature, 26
SAT solver, 21
Satisfiability, 21
Second-preimage resistance, 4
SHA-256, 7

Tautological clause, 23
Truth table, 19

Unit clause, 23
Unit propagation, 21
Used variable, 23

Watched Literals, 24

Bibliography

[1] 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.

[2] 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.

[3] 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.

[4] 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).

[5] M. Jones. *JSON Web Algorithms (JWA)*. RFC 7518. The Internet Engineering Task Force, 2015, pp. 1–69. URL: https://tools.ietf.org/html/rfc7518 (visited on 05/09/2016).

[6] 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.

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

[8] N. Sakimura M. Jones J. Bradley. *JSON Web Token (JWT)*. RFC 7519. The Internet Engineering Task Force, 2015, pp. 16–16. URL: https://tools.ietf.org/html/rfc7519#section-8 (visited on 05/09/2016).

[9] 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.

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

[11] Yusuke Naito et al. “Improved Collision Attack on MD4”. In: (2005), pp. 1–5. URL: http://eprint.iacr.org/.

[12] prokls. *MD4 in pure Python 3.4*. URL: https://gist.github.com/prokls/86b3c037df19a8c957fe.

[13] 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.

[14] 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.

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

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

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

[18] 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).

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