

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

May 10, 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 in a wide range of applications. They are used as cryptographic primitives in protocols and 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 proven useful to break collision resistance in hash functions. Over the years advanced attacks based on those differential approaches have been developed.

In this thesis we encode differential attack settings as SAT problems. SAT solvers are utilized to solve the problem revealing actual message collisions for a defined message difference.

Keywords: hash function, differential cryptanalysis, MD4, 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.

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.

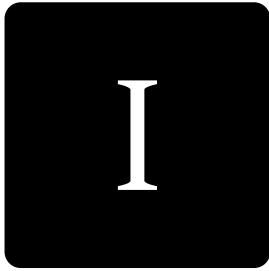
どもありがとうございました。

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 Lua^AT_EX and Linux Libertine Font.

Contents

1	Introduction	1
1.1	Cryptanalysis preliminaries	2
1.2	Cryptanalysis of Hash Functions	3
1.3	Differential cryptanalysis	4
1.4	Satisfiability	4
1.5	Satisfiability of hash algorithm states	6
1.6	Thesis Outline	7
2	Differential cryptanalysis	9
2.1	MD4	9
2.2	SHA-256	11
2.3	Differential notation	11
2.4	Addition example	11
2.5	Differential path	11
3	Satisfiability	13
3.1	The DIMACS de-facto standard	13
3.2	SAT features and CNF analysis	15
3.3	SAT features in comparison	16
3.4	Basic SAT solving techniques	16
3.5	SAT solvers in use	16
3.6	Encodings	16

3.6.1	STP approach	16
4	Results	19
4.1	Benchmark results	19
4.2	Related work	19
4.3	Conclusion	19
5	Summary and Future Work	21
5.1	Summary of results	21
5.2	Future work	21
	Appendices	23
A	Illustration	25
B	Testcases	27
C	Hardware setup	33



Chapter 1

Introduction

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 and 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 a complete hash collision which can be verified and visualized. If the internal state of the hash algorithm is too large, the attack can be easily computationally simplified by modelling only a subset of steps of the hash algorithm or changing the modelled differential path.

Gaining experience with these kind of problems with previous non-SAT-based tools we try 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.

1.1 Cryptanalysis preliminaries

Definition 1.1 (*Hash function*)

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

- 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 .

One example showing the use of hash functions as primitives are JSON Web Tokens (JWT) specified in RFC 7519 [7]. Section 8 defines implementation requirements and refers to RFC 7518 [5], which specifies cryptographic algorithms to be implemented. “HMAC SHA-256” (besides “none”) is the only signature and MAC algorithm required to be implemented. SHA-256 as hash algorithm is used as cryptographic primitive in this configuration.

A hash function has to satisfy the following security requirements:

Definition 1.2 (*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 1.3 (*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 1.4 (*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 $h(x) = h(x_2)$.

As far as hash functions accept input strings of arbitrary length, but return a fixed size output string, existence of collisions is unavoidable [14]. However, good hash functions make it very difficult to determine collisions or preimages.

The considered hash functions apply padding to their input to normalize their input size to a multiple of its block size. The round function follows afterwards. In the following we always consider input of block size meaning the round function is run only for one block and padding is always considered part of this input message. Padding is negligible, because once we have two colliding blocks, the collision will be reflected in the output in these single-pipe Merkle-Darmgård designs. This results in a length extension attack, making input padding negligible for cryptanalysis.

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	9bod807a

TABLE 1.1: One of two MD4 hash collisions provided in [17]. 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.

1.2 Cryptanalysis of Hash Functions

In August 2004, Wang et al. published results at Crypto'04 [17] 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 [4]. 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 [15] implements Wang's approach and can find MD4 collisions in few seconds on my Thinkpad x220 setup specified in [Appendix C](#).

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 . So it is cryptanalytically 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.

1.3 Differential cryptanalysis

Definition 1.5 (*Hash collision*)

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

Differential cryptanalysis is based on the idea to consider two execution states of hash algorithms for slightly different input messages. We trace those difference to learn about the propagation of message differences.

MD4 has a block size of 512 bits. Take some message in a particular encoding and provide this padded byte string as 512 bit input to the hash algorithm. Then use a copy of this byte string as input to a second hash algorithm implementation with slight modifications in few bits. MD4 has 48 round function applications in 3 rounds. Differential cryptanalysis considers the difference in the evaluation state between the two instances (compare with Figure 1.1).

Visualizing those differences helps the cryptanalyst to find modifications yielding a small number of differences in the evaluation state. The cryptanalyst consecutively modifies the input values to eventually receive a collision in the output value. If the number of differences in the evaluation state is small, this trail is expected to result in a hash collision with higher probability.

1.4 Satisfiability

Definition 1.6

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

The following definition gives three basic Boolean functions:

Definition 1.7

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.

Definition 1.8

A *truth table* unambiguously defines a Boolean function by enlisting the evaluated truth value for all possible sets of inputs.

Table 1.2 shows truth tables for AND, OR and NOT.

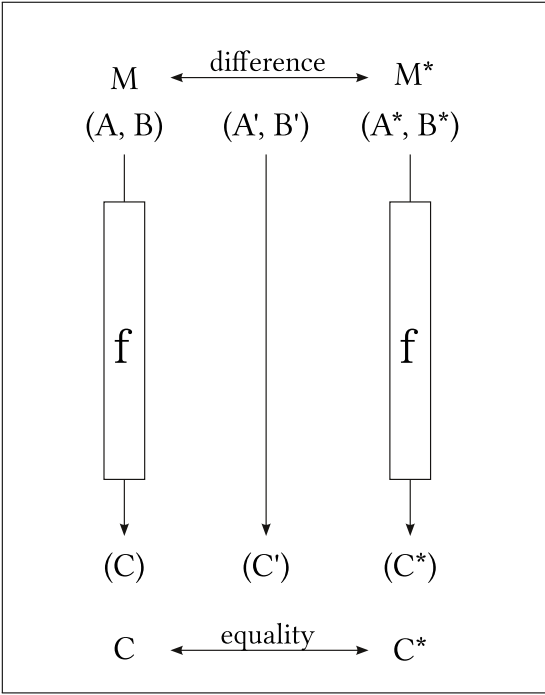


Figure 1.1: Typical attack setting for a collision attack: Hash function f is applied to two inputs M and M^* which differ by some predefined bits. A hash collision is given if and only if output values C and C^* show the same value. In differential cryptanalysis we observe the differences between two instances applying function f to inputs M and M^* .

v_1	v_2	$f(v_1, v_2)$	v_1	v_2	$f(v_1, v_2)$	v	$f(v)$
1	1	1	1	1	1	1	0
1	0	0	1	0	1	0	1
0	1	0	0	1	1	(c) NOT	
0	0	0	0	0	0		

(A) AND

(B) OR

TABLE 1.2: Truth tables for AND, OR and NOT

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

Definition 1.9

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*. Every element of X is called *assignment*.

The generic complexity of SAT determination is given by 2^n for n Boolean variables. The corresponding tool to determine satisfiability is defined as follows:

Definition 1.10

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

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.

1.5 Satisfiability of hash algorithm states

We discussed Boolean functions and satisfiability. At the same time we looked at basic properties of hash algorithms. But the question remains how we can link those areas together? This section is dedicated to this question.

Definition 1.11

An *algorithm* is a step-wise set of instructions to solve a problem. An *I/O algorithm* transforms given input values to output values.

Hash algorithms are one example of I/O algorithms.

I/O algorithms can be implemented as a sequence of instructions for computers. At the same time I/O algorithms can be represented as combination of Boolean functions. This claim is backed in more detail in Section 3.1 with Theorem 3.1. It follows immediately that we can represent I/O algorithms such as hash algorithms entirely as Boolean function.

Theorem 1.1

Every algorithm can be represented as Boolean function.

We consider 2bit addition as small example. Let a_i be the first argument where i denotes the binary position. If $i = 0$, the *least significant bit* (LSB) is considered. If $i = 1$, the *most significant bit* (MSB) is considered.

$$\begin{array}{rcl}
\begin{array}{rcl}
\text{1st arg:} & & a_1 \quad a_0 \\
\text{2nd arg:} & + & b_1 \quad b_0 \\
\hline
\text{carry:} & & c_0 \\
\text{sum:} & = & s_1 \quad s_0
\end{array} & \rightsquigarrow & \begin{array}{l}
s_0 = \text{XOR}(a_0, b_0) \\
c_0 = a_0 \wedge b_0 \\
s_1 = \text{XOR}(a_0, b_0, c_0)
\end{array}
\end{array}$$

Figure 1.2: Modelling 2bit addition (left) as Boolean function (right)

Let b_i be the second argument and s_i be the output value. Furthermore c_i is the carry bit, where c_1 is left out, because it is not used in 2bit addition. This model of 2bit addition as Boolean function can be seen in Figure 1.2.

1.6 Thesis Outline

This thesis is organized as follows:

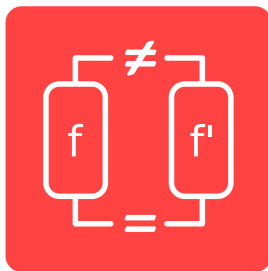
In Chapter 1 we discussed the basic properties and fundamentals of the tools in discussion including hash functions and SAT solvers.

In Chapter 2 we introduce the MD4 and SHA-256 hash functions and discuss possible approaches in differential cryptanalysis.

In Chapter 3 we discuss SAT solving and potential approaches to speed up SAT solvers for cryptographic problems.

In Chapter 4 we show results of our work and discuss its implications.

In Chapter 5 we suggest future work based on our results.



“JUST BECAUSE IT’S AUTOMATIC
DOESN’T MEAN IT WORKS.”
—Daniel J. Bernstein

Chapter 2

Differential cryptanalysis

2.1 MD4

MD4 is a cryptographic hash function originally described in RFC 1186 [11], updated in RFC 1320 [12] and declared obsolete by RFC 6150 [16]. It was invented by Ronald Rivest in 1990 with properties given in Table 2.1. Since 1995 [4] successful attacks have been found to break collisions, preimage and second-preimage resistance in MD4; including but not limited to [13] and [8]. A Python 3 implementation derived from a previous Python version is available at github [10].

block size	512 bits	namely variable block in RFC 1320 [12]
digest size	128 bits	as per Section 3.5 in RFC 1320 [12]
internal state size	128 bits	namely variables A , B , C and D
word size	32 bits	as per Section 2 in RFC 1320 [12]

TABLE 2.1: MD4 hash algorithm properties

In the following a quick overview over MD4’s design is given.

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 modifications of the previous step) in binary and take its first 64 bits.

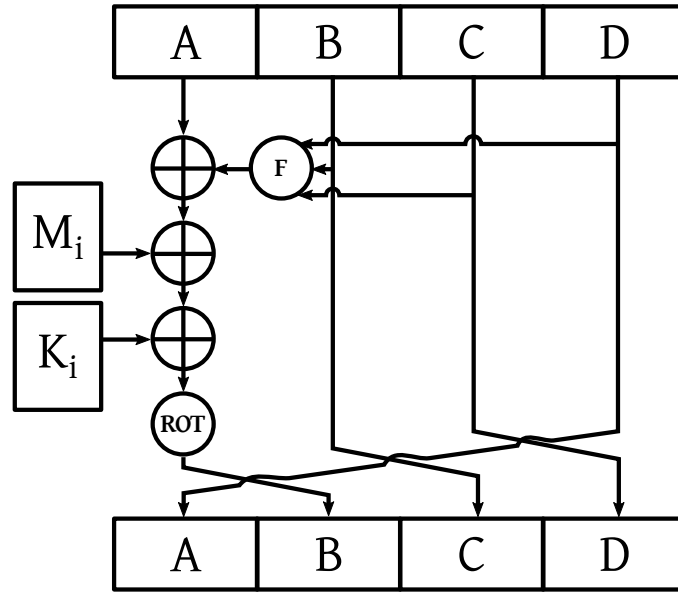


Figure 2.1: MD4 round function

Append those 64 bits to the input.

The message is split into 512-bit blocks (i.e. 16 32-bit words). Four state variables A , B , C and D are initialized with hexadecimal values:

$$[A] \text{ 01234567 } [B] \text{ 89abcdef } [C] \text{ fedcba98 } [D] \text{ 76543210 }$$

To process one block, three auxiliary Boolean functions are defined:

$$\text{IF}(X, Y, Z) = (X \wedge Y) \vee (\neg X \wedge Z) \quad (2.1)$$

$$\text{MAJ}(X, Y, Z) = (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z) \quad (2.2)$$

$$\text{XOR}(X, Y, Z) = X \oplus Y \oplus Z \quad (2.3)$$

Definition 2.1

The Boolean IF function behaves the following way: If the first argument is true, the second argument is returned. If the first argument is false, 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.

2.2 SHA-256

2.3 Differential notation

[2]

2.4 Addition example

2.5 Differential path

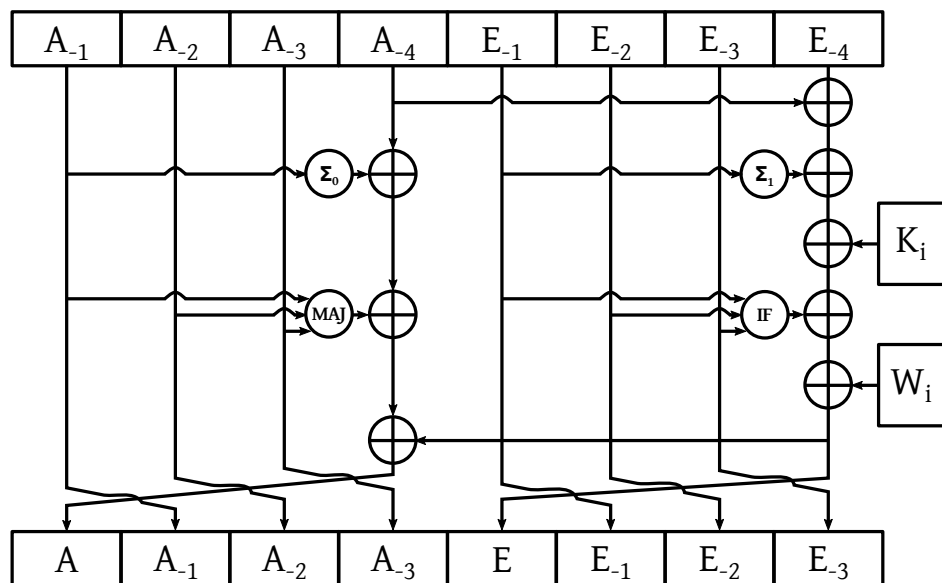


Figure 2.2: SHA-256 round function



“WHAT IDIOT CALLED THEM
LOGIC ERRORS RATHER THAN
BOOL SHIT?”
—Unknown

Chapter 3

Satisfiability

3.1 The DIMACS de-facto standard

Definition 3.1

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.

$$(a \vee b) \wedge (\neg a \vee \neg b) \quad (3.1)$$

Display 3.1 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 3.1

Every Boolean function can be represented as CNF.

Theorem 3.1 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 $0 \leq i \leq j$ and $0 \leq l \leq k$. 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 $\neg b_{i,l}$ 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 3.1: Display 3.1 represented in DIMACS format

```
p cnf 2 2
a b
-a -b
0
```

Definition 3.2

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.

3.2 SAT features and CNF analysis

At the very beginning I was very intrigued with the question “What is an ‘average’ SAT problem?”. Answers to this question can help to optimize SAT solver memory layouts. But originally I was wondering whether our differential cryptanalysis SAT problems distinguish from “average” SAT problems in some very basic properties. First of all, we need to elaborate on the question itself.

Definition 3.3 (SAT feature)

A SAT *feature* is a statistical value (named *feature value*) retrievable from some given SAT problem in some well-defined encoding.

A SAT feature is called *performance-driven* if the runtime of any computation contributes to the feature value.

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.

It should be computationally easy to evaluate SAT features of a given SAT problem. 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. A SAT feature is expected to be computable in linear time and memory with the number of variables and number of clauses. But a suggested limit is only given with polynomial complexity 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 $P \neq NP$. Hence the number of valid models cannot be a SAT feature as far as satisfiability needs to be determined. But no actual hard boundary for runtime requirements is given. 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.

The most similar resource I found looking at SAT features was the SATzilla project [9, 18] 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.

POSNEG_RATIO_CLAUSE_mean ratio of positive to negative clauses, mean

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 [1].

However, most of these SAT features are performance-driven. Examples for performance-driven SAT features include the number of restarts within a certain time frame or evaluation of local minima.

POSNEG-RATIO-CLAUSE-mean

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

3.3 SAT features in comparison

Proposition 3.1

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

Define a large set of SAT features. Present data. Categorize data.

3.4 Basic SAT solving techniques

3.5 SAT solvers in use

3.6 Encodings

3.6.1 STP approach

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



Chapter 4

Results

4.1 Benchmark results

4.2 Related work

4.3 Conclusion

Chapter 5

Summary and Future Work

5.1 Summary of results

5.2 Future work

Appendices

Appendix A

Illustration

i		$VS_{i,0}$	$VS_{i,1}$	$VS_{i,2}$
-4	A:	01100111010001010010001100000001		
-3	A:	00010000001100100101010001110110		
-2	A:	1001100010111010101110011111110		
-1	A:	111011111001101010101110001001		
0	A:	01101011110101001110010000010010	W:	01001101011110101001110010000011
1	A:	0111011001001111111011100 <u>u</u> 110001	W:	<u>u</u> 1010110110010111001001001111010
2	A:	101010110100000001110 <u>u</u> 01n110010	W:	n01n100111010101101001010111000
3	A:	101011 <u>u</u> 100111010101001001010001	W:	0101011110100111101001011110110
4	A:	00101100011000110101010111110010	W:	1101110011101001000101000111100
5	A:	000110100110001010 <u>u</u> 1101000000001	W:	1101100110000110110011010110011
6	A:	0001101100 <u>unuu</u> 110001000001111010	W:	101101101000011101000000100000
7	A:	00101011100000010 <u>unn</u> 011001010000	W:	00111011001010100101110110011111
8	A:	011100110010001 <u>u</u> 11111111110110000	W:	11000110100111010111000110110011
9	A:	101011n01 <u>unnu</u> 0001111100110011111	W:	11111001111010011001000110011000
10	A:	10n00100100001010100000010101110	W:	1101011110011111100000001011110
11	A:	<u>u</u> 1000110101101100100101011111111	W:	10100110001110111011001011101000
12	A:	001011 <u>u</u> 00 <u>u</u> 101011111110001111011	W:	010001011101110n1000111000110001
13	A:	10un1n01001100010100000111100101	W:	10010111111000110001111111100101
14	A:	00001010010100011000100011010110	W:	00100111100101001011111100001000
15	A:	0001111010101 <u>u</u> 010110011011010100	W:	10111001111010001100001111101001
16	A:	n00n0un0110100101001101101011111		
17	A:	00011111001110100001001000011110		
18	A:	01010111000011010000000010010100		
19	A:	<u>u</u> 1n10000000101111001101011000100		
20	A:	n1un1001111111011101000000110100		
21	A:	1111001110110000010111111010100		
22	A:	01011101110011010011001100111010		
23	A:	01010000111011101100011110001111		
24	A:	00000010000100100011011100011010		
25	A:	10110000100101100001010011101010		
26	A:	00001010100010010111011101000001		
27	A:	000001101110111101011010110011		
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 A.1: One of the original MD₄ collision given by Wang, et al.

Appendix B

Testcases

Figures B.1, B.2, B.3 and B.4 show testcases used to test performance measures.

i		$VS_{i,0}$	$VS_{i,1}$	$VS_{i,2}$
-4	A:	01100111010001010010001100000001		
-3	A:	00010000001100100101010001110110		
-2	A:	100110001011101010110011111110		
-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:	-----XXXXXXXXXXXXX-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-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 B.1: TODO description

i		$VS_{i,0}$	$VS_{i,1}$	$VS_{i,2}$
-4	A:	01100111010001010010001100000001		
-3	A:	00010000001100100101010001110110		
-2	A:	100110001011101010110011111110		
-1	A:	1110111110011011010101110001001		
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 B.2: TODO description

i		$VS_{i,0}$	$VS_{i,1}$	$VS_{i,2}$
-4	A:	01100111010001010010001100000001		
-3	A:	00010000001100100101010001110110		
-2	A:	100110001011101010110011111110		
-1	A:	1110111110011011010101110001001		
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 B.3: TODO description

i		$VS_{i,0}$	$VS_{i,1}$	$VS_{i,2}$
-4	A:	01100111010001010010001100000001		
-3	A:	00010000001100100101010001110110		
-2	A:	100110001011101010110011111110		
-1	A:	1110111110011011010101110001001		
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 B.4: TODO description

Appendix C

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”.

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

TABLE C.1: Thinkpad x220 Tablet specification [6]

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

TABLE C.2: Cluster node nehalem192go specification [3]

Index

- k -clause, 14
- Algorithm, 6
- AND (Boolean function), 4
- Assignment, 6
- Boolean function, 4
- Clause, 14
- Collision resistance, 2
- Conjunction, 13
- Conjunctive Normal Form, 13
- Definite clause, 14
- Disjunction, 13
- Disjunctive Normal Form, 13
- Feature value, 15
- Hash collision, 4
- Hash function, 2
- Hash value, 2
- Horn clause, 14
- I/O Algorithm, 6
- Least significant bit, 6
- Literal, 13
- Model, 6
- Most significant bit, 6
- Negative literal, 13
- NOT (Boolean function), 4
- OR (Boolean function), 4
- Positive literal, 13
- Preimage, 2
- Preimage resistance, 2
- SAT feature, 15
- SAT solver, 6
- Satisfiability, 6
- Second-preimage resistance, 2
- Truth table, 4
- Unit clause, 14

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>.
- [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.
- [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] Hans Dobbertin. “Cryptanalysis of MD4”. In: *Journal of Cryptology* 11.4 (1998), pp. 253–271. ISSN: 1432-1378. DOI: [10.1007/s001459900047](http://dx.doi.org/10.1007/s001459900047). URL: <http://dx.doi.org/10.1007/s001459900047>.
- [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] 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).
- [7] 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).
- [8] Yusuke Naito et al. “Improved Collision Attack on MD4”. In: (2005), pp. 1–5. URL: <http://eprint.iacr.org/>.
- [9] Eugene Nudelman et al. “Satzilla: An algorithm portfolio for SAT”. In: *Solver description, SAT competition 2004* (2004).
- [10] prokls. *MD4 in pure Python 3.4*. URL: <https://gist.github.com/prokls/86b3c037df19a8c957fe>.

- [11] 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>.
- [12] 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>.
- [13] Yu Sasaki et al. “New Message Difference for MD4”. In: (2007), pp. 1–20. URL: <http://www.iacr.org/archive/fse2007/45930331/45930331.pdf>.
- [14] Martin Schl  ffer and Elisabeth Oswald. “Searching for differential paths in MD4”. In: *Fast Software Encryption*. Springer. 2006, pp. 242–261.
- [15] Patrick Stach. *MD4 collision generator*. URL: http://crppit.epfl.ch/documentation/Hash_Function/Fastcoll_MD4/md4coll.c (visited on 04/05/2016).
- [16] 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).
- [17] Xiaoyun Wang et al. “Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD.” In: *IACR Cryptology ePrint Archive 2004* (2004), p. 199.
- [18] Lin Xu et al. “SATzilla: portfolio-based algorithm selection for SAT”. In: *Journal of Artificial Intelligence Research* (2008), pp. 565–606.