Journal of Artificial Intelligence Research 81 (2024) 359-399 Submitted 07/2023; published 10/2024



# Inverting Cryptographic Hash Functions via Cube-and-Conquer

**Oleg Zaikin** oleg.zaikin@icc.ru
*Novosibirsk State University*
*Novosibirsk, 630090, Russia*
*Matrosov Institute for System Dynamics and Control Theory SB RAS*
*Irkutsk, 664033, Russia*

## Abstract

MD4 and MD5 are fundamental cryptographic hash functions proposed in the early 1990s. MD4 consists of 48 steps and produces a 128-bit hash given a message of arbitrary finite size. MD5 is a more secure 64-step extension of MD4. Both MD4 and MD5 are vulnerable to practical collision attacks, yet it is still not realistic to invert them, i.e., to find a message given a hash. In 2007, the 39-step version of MD4 was inverted by reducing to SAT and applying a CDCL solver along with the so-called Dobbertin's constraints. As for MD5, in 2012 its 28-step version was inverted via a CDCL solver for one specified hash without adding any extra constraints. In this study, Cube-and-Conquer (a combination of CDCL and lookahead) is applied to invert step-reduced versions of MD4 and MD5. For this purpose, two algorithms are proposed. The first one generates inverse problems for MD4 by gradually modifying the Dobbertin's constraints. The second algorithm tries the cubing phase of Cube-and-Conquer with different cutoff thresholds to find the one with the minimum runtime estimate of the conquer phase. This algorithm operates in two modes: (i) estimating the hardness of a given propositional Boolean formula; (ii) incomplete SAT solving of a given satisfiable propositional Boolean formula. While the first algorithm is focused on inverting step-reduced MD4, the second one is not area-specific and is therefore applicable to a variety of classes of hard SAT instances. In this study, 40-, 41-, 42-, and 43-step MD4 are inverted for the first time via the first algorithm and the estimating mode of the second algorithm. Also, 28-step MD5 is inverted for four hashes via the incomplete SAT solving mode of the second algorithm. For three hashes out of them, it is done for the first time.

## 1. Introduction

A cryptographic hash function maps a message of arbitrary finite size to a hash of fixed size. Consider the following properties: (i) preimage resistance; (ii) second preimage resistance; (iii) collision resistance (Menezes, van Oorschot, & Vanstone, 1996). The first property indicates that it is computationally infeasible to invert the cryptographic hash function, i.e., to find any message that matches a given hash. According to the second property, given a message and its hash, it is computationally infeasible to find another message with the same hash. The third property indicates that it is computationally infeasible to find two different messages with the same hash. A proper cryptographic hash function must have all three properties. Cryptographic hash functions are ubiquitous in the modern digital world. Examples of their applications include the verification of data integrity, passwords, and signatures.

It is well known that the resistance of a cryptographic hash function can be analyzed by algorithms for solving the Boolean satisfiability problem (SAT) (Bard, 2009). SAT in its decision form is to determine whether a given propositional Boolean formula is satisfiable or not (Biere, Heule, van Maaren, & Walsh, 2021b). This is one of the most well-studied NP-complete problems (Cook, 1971; Garey & Johnson, 1979). Over the last 25 years, numerous crucial scientific and industrial problems have been successfully solved by SAT. In almost all these cases, CDCL solvers, i.e., those based on the Conflict-Driven Clause Learning algorithm (Marques-Silva & Sakallah, 1999), were used.

Cube-and-Conquer is an approach for solving extremely hard SAT instances (Heule, Kullmann, Wieringa, & Biere, 2011) for which CDCL solvers alone are not enough. According to this approach, a given problem is split into subproblems in the cubing phase via a lookahead solver (Heule & van Maaren, 2021). In the conquer phase, the subproblems are solved via a CDCL solver until a satisfying assignment is found. Several hard mathematical problems from number theory and combinatorial geometry have been solved by Cube-and-Conquer recently, e.g., the Boolean Pythagorean Triples problem (Heule, Kullmann, & Marek, 2016). However, the authors of this study are not aware of any successful application of this approach to cryptanalysis problems. This study aims to fill this gap by analyzing the preimage resistance of the cryptographic hash functions MD4 and MD5 via Cube-and-Conquer.

MD4 was proposed in 1990 (Rivest, 1990). It consists of 48 steps and produces a 128-bit hash given a message of arbitrary finite size. In 1995, it was shown that MD4 is not collision resistant (Dobbertin, 1996). Since MD4 still remains preimage resistant and second preimage resistant, its step-reduced versions have been studied in this context recently. In 1998, the Dobbertin's constraints for intermediate states of MD4 registers were proposed, which reduce the number of preimages, but at the same time significantly simplify the inversion (Dobbertin, 1998). This breakthrough approach made it possible to easily invert 32-step MD4. In 2007, SAT encodings of slightly modified Dobbertin's constraints were constructed, and as a result 39-step MD4 was inverted via a CDCL solver (De, Kumarasubramanian, & Venkatesan, 2007) for one very regular hash (128 1s). When the preimage resistance is studied, it is a common practice to invert very regular hashes such as all 1s or all 0s. Since 2007, several unsuccessful attempts have been made to invert 40-step MD4.

MD5 is a more secure 64-step version of MD4, which was proposed in 1992 (Rivest, 1992). Thanks to their elegant yet efficient designs, MD4 and MD5 have become one of the most influential cryptographic functions with several notable successors, such as RIPEMD and SHA-1. Since 2005, MD5 has been known to be not collision resistant (Wang & Yu, 2005). Because of the more secure design, the Dobbertin's constraints are not applicable to MD5 (Aoki & Sasaki, 2008). 26-step MD5 was inverted in 2007 (De et al., 2007), while for 27- and 28-step MD5 it was done for the first time in 2012 (Legendre, Dequen, & Krajecki, 2012). In both papers, CDCL solvers were applied, but no extra constraints were added. In (Legendre et al., 2012), 28-step MD5 was inverted for only one hash `0x01234567 0x89abcdef 0xfedcba98 0x76543210`. This hash is a regular binary sequence (it is symmetric, and the numbers of 0s and 1s are equal), but at the same time it is less regular than 128 1s mentioned above. The same result was presented later in two papers by the same authors. Unfortunately, none of these three papers explained the non-existence of results for 128 1s and 128 0s. Since 2012, no further progress in inverting step-reduced MD5 has been made.

This paper proposes Dobbertin-like constraints as a generalization of Dobbertin's constraints. In addition, two algorithms are proposed. The first one generates Dobbertin-like constraints until preimages of a step-reduced MD4 are found by a complete algorithm. In the paper, SAT solving algorithms are applied for this purpose. The second algorithm does sampling to find a cutoff threshold for the cubing phase of Cube-and-Conquer with the minimum runtime estimate of the conquer phase. The algorithm operates in two modes: (i) estimating the hardness of a given formula; (ii) incomplete SAT solving of a given formula. Since the estimating mode is general, it can be applied to arbitrary SAT instances including unsatisfiable ones, while the incomplete SAT solving mode is oriented only on satisfiable SAT instances, preferably with many solutions.

Using the first algorithm and the estimating mode of the second algorithm, 40-, 41-, 42-, and 43-step MD4 are inverted for four hashes: 128 1s, 128 0s, the one from (Legendre et al., 2012), and a random hash. When the best cutoff threshold and the corresponding runtime estimate are found in the estimating mode, in the conquer phase cubes are produced, and all corresponding subproblems are solved via a CDCL SAT solver. This differs from the typical conquer phase of Cube-and-Conquer, which stops when a satisfying assignment of any subproblem is found. It was done (i) to compare the total real runtime of all subproblems with the estimated runtime and (ii) to investigate how many preimages exist in the considered inverse problems.

It does not make sense to apply Dobbertin's constraints to MD5 because of its more secure design compared to MD4. Therefore, the first algorithm is not applicable to inverse problems for a step-reduced MD5. The estimating mode of the second algorithm is not well suited for inverting step-reduced MD5 because for an unconstrained inverse problem the cubing phase produces subproblems that are too hard. In this study, only the incomplete SAT solving mode of the second algorithm is applied to MD5. In particular, 28-step MD5 is inverted for the same four hashes as for MD4. All the experiments are run on a personal computer.

In summary, the contributions of the paper are:

- Dobbertin-like constraints as a generalization of Dobbertin's constraints.
- An algorithm that generates Dobbertin-like constraints and the corresponding inverse problems to find preimages of a step-reduced MD4.
- A general algorithm for finding a cutoff threshold with the minimum runtime estimate of the conquer phase of Cube-and-Conquer.
- For the first time, 40-, 41-, 42-, and 43-step MD4 are inverted.
- For the first time, 28-step MD5 is inverted for the two most regular hashes (128 1s and 128 0s) and a random non-regular hash.

The paper is organized as follows. Preliminaries on SAT, cryptographic hash functions, and Dobbertin's constraints are given in Section 2. Section 3 proposes Dobbertin-like constraints and the algorithm aimed at inverting step-reduced MD4. The Cube-and-Conquer-based algorithm is proposed in Section 4. The considered inverse problems for step-reduced MD4 and MD5, as well as their SAT encodings, are described in Section 5. Experimental results on inverting step-reduced MD4 and MD5 are presented in Sections 6, 7, and 8. Section 9 outlines related work. Finally, conclusions are drawn.

This paper builds on an earlier work (Zaikin, 2022), but extends it significantly in several directions. First, the algorithm for generating Dobbertin-like constraints for MD4 is improved by discarding impossible values of the last bits in the modified constraint. As a result, in most cases the considered step-reduced versions of MD4 are inverted approximately two times faster than in (Zaikin, 2022). Second, the incomplete SAT solving mode of the Cube-and-Conquer-based algorithm is proposed. Third, all considered step-reduced versions of MD4 are inverted for four hashes compared to two hashes in (Zaikin, 2022). Finally, 28-step MD5 is inverted, while in (Zaikin, 2022) only step-reduced MD4 was studied.

## 2. Preliminaries

This section gives preliminaries on SAT, Cube-and-Conquer, cryptographic hash functions, MD4, Dobbertin's constraints, and MD5.

### 2.1 Boolean Satisfiability

*Boolean satisfiability problem* (SAT) (Biere et al., 2021b) is to determine whether a given propositional Boolean formula is satisfiable or not. A formula is *satisfiable* if there exists a truth assignment that satisfies it; otherwise it is *unsatisfiable*. SAT is historically the first NP-complete problem (Cook, 1971). A propositional Boolean formula is in *Conjunctive Normal Form* (CNF), if it is a conjunction of clauses. A *clause* is a disjunction of literals, where a *literal* is a Boolean variable or its negation.

The *Davis–Putnam–Logemann–Loveland* (DPLL) algorithm is a complete backtracking SAT solving algorithm (Davis, Logemann, & Loveland, 1962). It forms a decision tree, where each internal node corresponds to a decision variable, while edges correspond to variables' values. When a decision variable is assigned, *Unit Propagation* (UP) reduces the tree (Dowling & Gallier, 1984). UP iteratively applies the *unit clause rule*: if there is only one remaining unassigned literal in a clause and all other literals are assigned to False, then the literal is assigned to True. If an unsatisfied clause is encountered, a *conflict* is declared, and chronological backtracking is performed.

Lookahead is another complete SAT solving algorithm (Heule & van Maaren, 2021). It improves DPLL by the following heuristic. When a decision variable should be chosen, each unassigned variable is assigned to True followed by UP, the reduction is measured, and the same is done for False assignment. *Failed literal* denotes a literal for which a conflict is found during UP. If both literals of a variable are failed, the unsatisfiability of the formula is proven. If there is exactly one failed literal $l$ for some variable, then $l$ is assigned to False. This rule is known as *failed literal elimination*. If for a variable both literals are not failed, the reduction measure for this variable is calculated as a combination of literal-measures. A variable with the largest reduction measure is picked as a decision variable. Thus lookahead allows one to choose good decision variables and simplifies the formula by the described reasoning. Lookahead SAT solvers are strong on random k-SAT formulae.

In contrast to DPLL, in *Conflict-Driven Clause Learning* (CDCL), when a conflict occurs, the reason is found and *non-chronological backtracking* is performed (Marques-Silva & Sakallah, 1999). To forbid the conflict, a *conflict clause* is formed and added to the formula. Conflict clauses are used to limit the exploration of the decision tree and to choose proper decision variables. This complete algorithm is much more efficient than DPLL. Also, it is stronger than lookahead on non-random instances. That is why most modern complete SAT solvers are based on CDCL. Recently, CDCL was improved by local search (Cai, Zhang, Fleury, & Biere, 2022). Although local search algorithms are incomplete because they cannot prove unsatisfiability, the mentioned combination of CDCL and local search is complete.

Problems from the following areas can be efficiently reduced to SAT: model checking, planning, cryptanalysis, combinatorics, and bioinformatics (Biere et al., 2021b). When a cryptanalysis problem is reduced to SAT and solved by SAT solvers, it is called *SAT-based cryptanalysis* or *logical cryptanalysis* (Cook & Mitchell, 1996; Massacci & Marraro, 2000). This is a special type of algebraic cryptanalysis (Bard, 2009). In the last two decades, SAT-based cryptanalysis has been successfully applied to stream ciphers, block ciphers, and cryptographic hash functions.

### 2.2 Cube-and-Conquer

If a given SAT instance is too hard for a sequential SAT solver, it makes sense to solve it in parallel (Balyo & Sinz, 2018). If only complete algorithms are considered, there are two main approaches to parallel SAT solving: *portfolio* (Hamadi, Jabbour, & Sais, 2009) and *divide-and-conquer* (Böhm & Speckenmeyer, 1996). According to the portfolio approach, many different sequential SAT solvers (or maybe different configurations of the same solver) solve the same problem simultaneously. In the divide-and-conquer approach, the problem is decomposed into a family of simpler subproblems that are solved by sequential solvers.

*Cube-and-conquer* (Heule et al., 2011; Heule, Kullmann, & Biere, 2018) is a divide-and-conquer SAT solving approach that combines lookahead with CDCL. In the *cubing phase*, a modified lookahead solver splits a given formula into *cubes*. In the *conquer phase*, by joining each cube with the original formula a subformula is formed. Finally a CDCL solver is run on the subformulas. If the original formula is unsatisfiable, then all the subformulas are unsatisfiable. Otherwise, at least one subformula is satisfiable. If a satisfying assignment of any subformula is found, the conquer phase stops. Since cubes can be processed independently, the conquer phase can be easily parallelized. In (Heule et al., 2011) it was proposed to process the subformulas via an incremental CDCL solver.

As mentioned in the previous subsection, lookahead is a complete algorithm. When used in the cubing phase of Cube-and-Conquer, a lookahead solver is forced to cut off some branches, thus producing cubes. Therefore, such a solver produces a decision tree, where leaves are either *refuted* ones (with no possible solutions) or cubes. There are two main cutoff heuristics that decide when a branch becomes a cube. In the first one, a branch is cut off after a given number of decisions (Hyvärinen, Junttila, & Niemelä, 2010). According to the second one, it occurs when the number of variables in the corresponding subproblem drops below a given threshold (Heule et al., 2011). In the present study, the second cutoff heuristic is used because it usually shows better results on hard instances (Heule et al., 2018).

### 2.3 Cryptographic Hash Functions

A hash function $h$ is a function with the following properties (Menezes et al., 1996).

1. *Compression*: $h$ maps an input $x$ of arbitrary finite size to an output $h(x)$ of fixed size.

2. *Ease of computation*: for any given input $x$, $h(x)$ is easy to compute.

An unkeyed cryptographic hash function $h$ is a hash function that has the following properties (Menezes et al., 1996).

1. *Collision resistance*: it is computationally infeasible to find any two inputs $x$ and $x'$ such that $x \neq x', h(x) = h(x')$.

2. *Preimage resistance*: for any given output $y$, it is computationally infeasible to find any of its preimages, i.e., any such input $x'$ that $h(x') = y$.

3. *Second-preimage resistance*: for any given input $x$, it is computationally infeasible to find $x'$ such that $x' \neq x, h(x) = h(x')$.

Inputs of cryptographic hash functions are usually called *messages*, while outputs are called *hash values* or just *hashes*. Hereinafter only unkeyed cryptographic hash functions are considered.

Methods for disproving the mentioned three properties are called *collision attacks*, *preimage attacks*, and *second preimage attacks*, respectively. If an attack is computationally feasible, then it is called *practical*. Usually it is much easier to propose a practical collision attack than practical attacks of two other types. This study is focused on practical preimage attacks on step-reduced cryptographic hash functions MD4 and MD5. In the rest of the paper, a practical inversion of a cryptographic hash function implies a practical preimage attack and vise versa.

### 2.4 MD4

The Message Digest 4 (MD4) cryptographic hash function was proposed by Ronald Rivest in 1990 (Rivest, 1990). Given a message of arbitrary finite size, *padding* is applied to obtain a message that can be divided into 512-bit blocks. Then a 128-bit hash is produced by iteratively applying the MD4 compression function to the blocks in accordance to the Merkle-Damgard construction (Merkle, 1989; Damgård, 1989).

Consider the compression function in more detail. Given a 512-bit input, it produces a 128-bit output. The function consists of three rounds, sixteen steps each, and operates by transforming data in four 32-bit registers $A, B, C, D$. If a message block is the first one, then the registers are initialized with the following constants, respectively: `0x67452301`; `0xefcdab89`; `0x98badcfe`; `0x10325476`. Otherwise, the registers are initialized with an output produced by the compression function on the previous message block. The message block $M$ is divided into sixteen 32-bit words. In every step, one register is updated by mixing one message word with the values of all four registers and an additive constant. This transformation is partially performed by a round-specific function. Additive constants are also round-specific. As a result, every round all sixteen words take part in such updates. Finally, registers are incremented by the values they had after the current block initialization, and the output is produced as a concatenation of $A, B, C, D$. The round functions and additive constants are presented in Table 1.

Table 1: Characteristics of MD4 rounds.

| Round | Round function | Additive constant |
|---|---|---|
| 1 | $F(x, y, z) = (x \wedge y) \vee (\neg x \wedge z)$ | `0x0` |
| 2 | $G(x, y, z) = (x \wedge y) \vee (x \wedge z) \vee (y \wedge z)$ | `0x5a827999` |
| 3 | $H(x, y, z) = x \oplus y \oplus z$ | `0x6ed9eba1` |

Algorithm 1 presents the compression function when it processes the first message block. The function $[abcd\ k\ s]_F$ stands for $a = (a + F(b, c, d) + M[k]) \lll s$, where $\lll s$ is the circular shifting to the left by $s$ bits position, while $+$ is the addition modulo $2^{32}$. The functions $[abcd\ k\ s]_G$ and $[abcd\ k\ s]_H$ stand for $a = (a + G(b, c, d) + M[k] + \texttt{0x5a827999}) \lll s$ and $a = (a + H(b, c, d) + X[k] + \texttt{0x6ed9eba1}) \lll s$, respectively.

**Algorithm 1** MD4 compression function on the first 512-bit message block.

**Input:** 512-bit message block $M$.
**Output:** Updated values of registers $A$, $B$, $C$, $D$.

$AA \leftarrow A \leftarrow$ `0x67452301`
$BB \leftarrow B \leftarrow$ `0xefcdab89`
$CC \leftarrow C \leftarrow$ `0x98badcfe`
$DD \leftarrow D \leftarrow$ `0x10325476`
$[\text{ABCD}\ 0\ 3]_F$ $[\text{DABC}\ 1\ 7]_F$ $[\text{CDAB}\ 2\ 11]_F$ $[\text{BCDA}\ 3\ 19]_F$ ▷ Steps 1-4
$[\text{ABCD}\ 4\ 3]_F$ $[\text{DABC}\ 5\ 7]_F$ $[\text{CDAB}\ 6\ 11]_F$ $[\text{BCDA}\ 7\ 19]_F$ ▷ Steps 5-8
$[\text{ABCD}\ 8\ 3]_F$ $[\text{DABC}\ 9\ 7]_F$ $[\text{CDAB}\ 10\ 11]_F$ $[\text{BCDA}\ 11\ 19]_F$ ▷ Steps 9-12
$[\text{ABCD}\ 12\ 3]_F$ $[\text{DABC}\ 13\ 7]_F$ $[\text{CDAB}\ 14\ 11]_F$ $[\text{BCDA}\ 15\ 19]_F$ ▷ Steps 13-16
$[\text{ABCD}\ 0\ 3]_G$ $[\text{DABC}\ 4\ 5]_G$ $[\text{CDAB}\ 8\ 9]_G$ $[\text{BCDA}\ 12\ 13]_G$ ▷ Steps 17-20
$[\text{ABCD}\ 1\ 3]_G$ $[\text{DABC}\ 5\ 5]_G$ $[\text{CDAB}\ 9\ 9]_G$ $[\text{BCDA}\ 13\ 13]_G$ ▷ Steps 21-24
$[\text{ABCD}\ 2\ 3]_G$ $[\text{DABC}\ 6\ 5]_G$ $[\text{CDAB}\ 10\ 9]_G$ $[\text{BCDA}\ 14\ 13]_G$ ▷ Steps 25-28
$[\text{ABCD}\ 3\ 3]_G$ $[\text{DABC}\ 7\ 5]_G$ $[\text{CDAB}\ 11\ 9]_G$ $[\text{BCDA}\ 15\ 13]_G$ ▷ Steps 29-32
$[\text{ABCD}\ 0\ 3]_H$ $[\text{DABC}\ 8\ 9]_H$ $[\text{CDAB}\ 4\ 11]_H$ $[\text{BCDA}\ 12\ 15]_H$ ▷ Steps 33-36
$[\text{ABCD}\ 2\ 3]_H$ $[\text{DABC}\ 10\ 9]_H$ $[\text{CDAB}\ 6\ 11]_H$ $[\text{BCDA}\ 14\ 15]_H$ ▷ Steps 37-40
$[\text{ABCD}\ 1\ 3]_H$ $[\text{DABC}\ 9\ 9]_H$ $[\text{CDAB}\ 5\ 11]_H$ $[\text{BCDA}\ 13\ 15]_H$ ▷ Steps 41-44
$[\text{ABCD}\ 3\ 3]_H$ $[\text{DABC}\ 11\ 9]_H$ $[\text{CDAB}\ 7\ 11]_H$ $[\text{BCDA}\ 15\ 15]_H$ ▷ Steps 45-48
$A \leftarrow A + AA$ ▷ Increment A by the initial value
$B \leftarrow B + BB$ ▷ Increment B by the initial value
$C \leftarrow C + CC$ ▷ Increment C by the initial value
$D \leftarrow D + DD$ ▷ Increment D by the initial value

In 1995, a practical collision attack on MD4 was proposed (Dobbertin, 1996). In 2005, it was theoretically shown that on a very small fraction of messages MD4 is not second preimage resistant (Wang, Lai, Feng, Chen, & Yu, 2005). In 2008, a theoretical preimage attack on MD4 was proposed (Leurent, 2008).

### 2.5 Dobbertin's Constraints for MD4

Since MD4 is still preimage resistant and second preimage resistant from the practical point of view, its step-reduced versions have been studied recently. In 1998, Hans Dobbertin introduced additional constraints for MD4 (Dobbertin, 1998). Consider a constant 32-bit word $K$ and 32-step MD4. The constraints are as follows: $A = K$ in steps 13, 17, 21, 25; $D = K$ in steps 14, 18, 22, 26; $C = K$ in steps 15, 19, 23, 27 (numbering from 1). Further in the present paper these constraints are called *Dobbertin's constraints*. Algorithm 2 shows how the MD4 compression function is changed when Dobbertin's constraints are applied.

**Algorithm 2** MD4 compression function on the first 512-bit message block with applied Dobbertin's constraints.

**Input:** 512-bit message block $M$, constant word $K$.
**Output:** Updated values of registers $A$, $B$, $C$, $D$.

```
Initialize A, B, C, D as in Algorithm 1
Steps 1–12 as in Algorithm 1
[ABCD 12  3]_F  A ← K        ▷ Constrained step 13
[DABC 13  7]_F  D ← K        ▷ Constrained step 14
[CDAB 14 11]_F  C ← K        ▷ Constrained step 15
[BCDA 15 19]_F               ▷ Step 16
[ABCD  0  3]_G  A ← K        ▷ Constrained step 17
[DABC  4  5]_G  D ← K        ▷ Constrained step 18
[CDAB  8  9]_G  C ← K        ▷ Constrained step 19
[BCDA 12 13]_G               ▷ Step 20
[ABCD  1  3]_G  A ← K        ▷ Constrained step 21
[DABC  5  5]_G  D ← K        ▷ Constrained step 22
[CDAB  9  9]_G  C ← K        ▷ Constrained step 23
[BCDA 13 13]_G               ▷ Step 24
[ABCD  2  3]_G  A ← K        ▷ Constrained step 25
[DABC  6  5]_G  D ← K        ▷ Constrained step 26
[CDAB 10  9]_G  C ← K        ▷ Constrained step 27
[BCDA 14 13]_G               ▷ Step 28
Steps 29–48 as in Algorithm 1
Increment A, B, C, D as in Algorithm 1
```

Consider step 17. $A = C = D = K$ due to constrained steps 13, 14, and 15, while $B$ is unknown. Since $G$ is the majority function, $G(x, y, y) = y$ for arbitrary $x$ and $y$. Therefore, we have

$$K = (A + G(B,C,D) + M[0] + \texttt{0x5a827999}) \lll 3 =$$
$$(K + G(B,K,K) + M[0] + \texttt{0x5a827999}) \lll 3 =$$
$$(K + K + M[0] + \texttt{0x5a827999}) \lll 3.$$

Then it follows

$$K \ggg 3 = 2K + M[0] + \texttt{0x5a827999},$$

and finally

$$M[0] = (K \ggg 3) - 2K - \texttt{0x5a827999}.$$

Here $-$ stands for subtraction modulo $2^{32}$. For example, if $K = \texttt{0xffffffff}$, then

$$M[0] = \texttt{0xffffffff} - 2 \cdot \texttt{0xffffffff} - \texttt{0x5a827999} =$$
$$-\texttt{0xffffffff} - \texttt{0x5a827999} = \texttt{0xa57d8668}.$$

Thus, if $A$ is equal to a constant word in step 17, $M[0]$ becomes a constant as well. The same holds for $M[4]$, $M[8]$, $M[1]$, $M[5]$, $M[9]$, $M[2]$, $M[6]$, and $M[10]$ due to constrained steps 18, 19, 21, 22, 23, 25, 26, and 27, respectively. Finally, Dobbertin's constraints turn 9 message words out of 16 into constants. Therefore, the constrained compression function maps $\{0,1\}^{224}$ onto $\{0,1\}^{128}$ while the original one maps $\{0,1\}^{512}$ onto $\{0,1\}^{128}$. As a result, for any given hash and a randomly chosen $K$, the number of preimages (messages) is significantly reduced, maybe even to 0. Dobbertin's constraints are an example of streamlined constraints (Gomes & Sellmann, 2004). Such constraints are not implied by the formula, so they can remove some (or even all) solutions but have a good chance of leaving at least one solution.

Dobbertin's constraints were originally proposed for 32-step MD4, and they do not guarantee that for a certain pair (hash,$K$) at least one preimage remains. On the other hand, they guarantee that the corresponding system of equations becomes much smaller and easier to solve. It is clear that different $K$ can be tried until a preimage is found. The point is that even if a few such simplified problems are to be solved, it may be faster than solving the original problem. The same holds for more than 32 steps because the constraints are applied before the 32nd step. In other words, adding more unconstrained steps does not reduce the number of solutions.

In (Dobbertin, 1998), Dobbertin's constraints were used to invert 32-step MD4 by randomly choosing values of $K$ and $B$ (on step 28) until a consistent system was formed and a preimage was found. In the case of 32 steps, a constant value $B$ in addition to $K$ in step 28 implies values of the remaining 7 message words. This is not the case for more than 32 steps. In 2000, modified Dobbertin's constraints were applied to invert MD4 when the second round is omitted (Kuwakado & Tanaka, 2000). In 2007, a SAT-based implementation of slightly modified Dobbertin's constraints (where the constraint in step 13 is omitted) made it possible to invert 39-step MD4 (De et al., 2007). Since 2007, several unsuccessful attempts to invert 40-step MD4 have been made, see, e.g., (Legendre et al., 2012). The present study aims to invert 40-, 41-, 42-, and 43-step MD4.

### 2.6 MD5

MD5 was proposed in 1992 by Ronald Rivest as a slightly slower but more secure extension of MD4 (Rivest, 1992).

The main changes in MD5 compared to MD4 are as follows.

1. The fourth 16-step round with its own round function, so MD5 consists of 64 steps;
2. New function for the second round;
3. Usage of an unique additive constant in each of the 64 steps;
4. Addition of output from the previous step.

The round functions are as follows:

- Round 1. $F(x, y, z) = (x \wedge y) \vee (\neg x \wedge z)$.
- Round 2. $G(x, y, z) = (x \wedge z) \vee (y \wedge \neg z)$.
- Round 3. $H(x, y, z) = x \oplus y \oplus z$.
- Round 4. $I(x, y, z) = y \oplus (x \vee \neg z)$.

For the first time, a practical collision attack on MD5 was presented in 2005 (Wang & Yu, 2005). In 2009, a theoretical preimage attack was proposed (Sasaki & Aoki, 2009). It is known that Dobbertin's constraints are not efficient for MD5 because of changes 2-4 mentioned above (Aoki & Sasaki, 2008). In particular, when applying to MD5, these constraints remove all solutions (preimages), so simplicity of the obtained simplified problem does not help. In 2007, 26-step MD5 was inverted (De et al., 2007) while in 2012, it was done for 27-, and 28-step MD5 (Legendre et al., 2012). In both papers, CDCL solvers were applied, yet no additional constraints were added to the corresponding formulas.

Despite the described vulnerabilities, MD5 is still widely used to verify data integrity on operating systems of the Linux family[1].

## 3. Dobbertin-like Constraints for Inverting Step-reduced MD4

As mentioned in the previous section, the progress in inverting step-reduced MD4 was mainly due to Dobbertin's constraints. This section proposes Dobbertin-like constraints as their generalization. In addition, an algorithm for inverting step-reduced MD4 via Dobbertin-like constraints is proposed.

### 3.1 Dobbertin-like Constraints

Suppose that given a constant word $K$, only 11 of 12 Dobbertin's constraints hold as usual, while in the remaining constraint only $b, 0 \leq b \leq 32$ bits of the register are equal to the corresponding $b$ bits of $K$. At the same time, the remaining $32 - b$ bits in the register take the opposite values to those in $K$. We denote these constraints as *Dobbertin-like constraints.*

1. https://linux.die.net/man/1/md5sum

It is clear that Dobbertin's constraints are a special case of Dobbertin-like constraints when $b = 32$.

We denote an inverse problem for step-reduced MD4 with applied Dobbertin-like constraints as MD4inversion$(y, s, K, p, L)$, where

- $y$ is a given 128-bit hash;
- $s$ is the number of MD4 steps (starting from the first one);
- $K$ is a 32-bit constant word used in Dobbertin's constraints;
- $p \in \{13, 14, 15, 17, 18, 19, 21, 22, 23, 25, 26, 27\}$ is a specially constrained step;
- $L$ is a 32-bit word such that if $L_i = 0, 0 \leq i \leq 31$, then $i$-th bit of the register's value modified in step $p$ is equal to $K_i$. Otherwise, it is equal to $\sim K_i$.

In other words, the 32-bit word $L$ serves as a bit mask and controls how similar the specially constrained register and $K$ are to each other. To make this definition clearer, three examples are given below. Hereinafter `0hash` and `1hash` mean 128 0s and 128 1s (i.e., 4 words `0x00000000` and `0xffffffff`, respectively).

**Example 3.1** (MD4inversion(0hash, 32, 0x62c7Ec0c, 21, 0x00000000))**.** The problem is to invert `0hash` produced by 32-step MD4. Since $L =$ `0x00000000`, in step 21 the specially constrained register's value is $K$, so all 12 Dobbertin's constraints are applied as usual with $K =$ `0x62c7Ec0c`. A similar inverse problem (up to choice of $K$) was solved in (Dobbertin, 1998).

**Example 3.2** (MD4inversion(1hash, 39, 0xfff00000, 12, 0xffffffff))**.** The problem is to invert `1hash` produced by 39-step MD4. Since $L =$ `0xffffffff`, in step 12 the specially constrained register's value is $\sim K =$ `0x000fffff`. In the remaining 11 Dobbertin's steps the registers' values are $K =$ `0xfff00000`.

**Example 3.3** (MD4inversion(1hash, 40, 0xffffffff, 12, 0x00000003))**.** The problem is to invert `1hash` produced by 40-step MD4. Since $L =$ `0x00000003`, in step 12 the first 30 bits of the specially constrained register are equal to those in $K$, while the last two bits have values $\sim K_{30}$ and $\sim K_{31}$, respectively. Therefore, this register's value is `0xfffffffc`, while in the remaining 11 Dobbertin's steps the registers' values are $K =$ `0xffffffff`.

### 3.2 Inversion Algorithm

Dobbertin-like constraints can be used to find preimages of a step-reduced MD4 according to the following idea. For a given hash $y$, step $s$, and constant $K$, an inverse problem is formed with $L =$ `0x00000000`. Thus, all 12 Dobbertin's constraints are applied. The inverse problem is solved, and if a preimage is found, nothing else should be done. Otherwise, if it is proven that no preimages exist in the current inverse problem, a new one is formed with $L =$ `0x00000001`. In this case, the specially constrained register's value is just 1 bit shy of being $K$. The inverse problem is also solved. If still no preimage, $L$ is further modified: `0x00000002`, `0x00000003`, and so on. The intuition here is that Dobbertin's constraints lead to a system of equations that is either consistent with very few solutions or quite close to a consistent one. In the latter case, trying different values of $L$ helps to form a consistent system and find its solution.

Algorithm 3 follows the described idea. In the pseudocode, the function DecimalToBinary converts a decimal number to binary, while $\mathcal{A}$ is a complete algorithm, which for a formed inverse problem returns preimages if they exist. In the while loop, all possible values of the specially constrained register are varied, yet the first values are very close to $K$. Note that it is not guaranteed that Algorithm 3 finds any preimage for a given hash. However, as it will be shown in sections 6 and 7, Algorithm 3 is able to find preimages for step-reduced MD4. Moreover, it usually does it in just few iterations (from 1 to 3) of the while loop.

**Algorithm 3** Invert step-reduced MD4 via Dobbertin-like constraints.

**Input:** Hash $y$; the number of MD4 steps $s$; constant $K$; step $p$ with the specially constrained register; a complete algorithm $\mathcal{A}$.

**Output:** Preimages for hash $y$.

```
preimages ← {}
i ← 0
while i < 2^32 do
    L ← DecimalToBinary(i)
    preimages ← A(MD4inversion(y, s, K, p, L))
    if preimages is not empty then
        break
    i ← i + 1
return preimages
```

Complete algorithms of various types can be used to solve inverse problems formed in Algorithm 3. In particular, wide spectrum of constraint programming (Rossi, van Beek, & Walsh, 2006) solvers are potential candidates. In preliminary experiments, we used state-of-the-art sequential and parallel CDCL SAT solvers (Balyo & Sinz, 2018) to invert 40-step MD4, but even in the first iteration, where all 12 Dobbertin's constraints are added, SAT instances turned out to be too hard for them. That is why we decided to use Cube-and-Conquer SAT solvers, which are more suitable for extremely hard SAT instances (Heule et al., 2018). The next section describes how a given problem can be properly split into simpler subproblems in the cubing phase of Cube-and-Conquer.

## 4. Finding Cutoff Thresholds for Cube-and-Conquer

Recall (see Subsection 2.2) that in Cube-and-Conquer the following cutoff threshold is meant in the cubing phase: the number of variables in a subformula, formed by adding a cube to the original Boolean propositional formula and applying UP. It is crucial to properly choose this threshold. If it is too high, then the cubing phase is performed in no time, but very few extremely hard (for a CDCL solver) subformulas might be produced; if it is too low, then the cubing phase will be extremely time consuming, while there will be a huge number of subformulas.

Earlier two algorithms aimed at finding a cutoff threshold with the minimum estimated runtime of Cube-and-Conquer were proposed. Subsection 7.2 of the tutorial (Heule, 2018a) proposed Algorithm A as follows:

> Optimizing the heuristics requires selecting useful subproblems of the hard formula. This can be done as follows: First determine the depth for which the number of refuted nodes is at least 1000. Second, randomly pick about 100 subproblems (cubes) of the partition on that depth. Second, solve these 100 subproblems and select the 10 hardest ones for the optimization.

Later, Algorithm B was proposed in (Bright, Cheung, Stevens, Kotsireas, & Ganesh, 2021):

> The cut-off bound was experimentally chosen by randomly selecting up to several hundred instances from each case and determining a bound that minimizes the sum of the cubing and conquering times.

This section proposes a new algorithm inspired by algorithms A and B. The algorithm aims to find a cutoff threshold with the minimum runtime estimate of the conquer phase, so the runtime of the cubing phase is not taken into account because it is assumed that the latter is negligible.

First, it is needed to preselect promising thresholds. On the one hand, the number of refuted leaves should be quite significant since it may indicate that at least some subformulas have become simpler compared to the original formula. In addition, the total number of cubes should not be too large. An auxiliary Algorithm 4 follows this idea. Given a lookahead solver, a CNF, and a cutoff threshold, the function LookaheadWithCut runs the solver on the CNF with the cutoff threshold (see Subsection 2.2) and outputs cubes and the number of refuted leaves.

**Algorithm 4** Preselect promising thresholds for the cubing phase of Cube-and-Conquer.

**Input:** CNF $\mathcal{F}$; lookahead solver `ls`; starting threshold $n_{\texttt{start}}$; threshold decreasing step `nstep`; maximum number of cubes `maxc`; minimum number of refuted leaves `minr`.
**Output:** Stack of promising thresholds and corresponding cubes.

```
function PreselectThresholds(F, ls, n_start, nstep, maxc, minr)
    stack ← {}
    n ← n_start
    while n > 0 do
        ⟨c, r⟩ ← LookaheadWithCut(ls, F, n)      ▷ Get cubes and number of refuted.
        if Size(c) > maxc then
            break                                ▷ Break if too many cubes.
        if r ≥ minr then
            stack.push(⟨n, c⟩)                   ▷ Add threshold and cubes.
        n ← n − nstep                            ▷ Decrease threshold.
    return stack
```

Note that it is intended that the found cutoff threshold will be used to solve all corresponding subproblems by a CDCL SAT solver in the conquer phase. This setting differs from the typical conquer phase of Cube-and-Conquer, which stops upon finding a satisfying assignment of any subproblem (see Subsection 2.2). The setting was chosen (i) to compare the total real runtime of all subproblems with the estimated runtime and (ii) to investigate how many preimages exist in the considered inverse problems for step-reduced MD4. However, later in this section it will be discussed how the algorithm can be easily modified to fit the typical setting of the conquer phase.

When promising values of the threshold are preselected by, it is needed to estimate the hardness of the corresponding conquer phases. It can be done by choosing a fixed number of cubes by simple random sampling (Starnes, Yates, & Moore, 2010) among those produced in the cubing phase. If all corresponding subproblems from the sample are solved by a CDCL solver in a reasonable time, then an estimated total solving time for all subproblems can be easily calculated. This idea is implemented as Algorithm 5.

**Algorithm 5** Find a cutoff threshold with the minimum estimated runtime of the conquer phase.

**Input:** CNF $\mathcal{F}$; lookahead solver `ls`; threshold decreasing step `nstep`; maximum number of cubes `maxc`; minimum number of refuted leaves `minr`; sample size $N$; CDCL solver `cs`; CDCL solver time limit `maxcst`; number of CPU cores `cores`; operating mode `mode`.

**Output:** A threshold $n_{\texttt{best}}$ with the runtime estimate $e_{\texttt{best}}$ and cubes $c_{\texttt{best}}$; Boolean `isSAT` that indicates whether $\mathcal{F}$ is satisfiable.

```
isSAT ← Unknown
n_start ← VARNUM(F) − nstep
⟨n_best, e_best, c_best⟩ ← ⟨n_start, +∞, {}⟩
stack ← PRESELECTTHRESHOLDS(F, ls, n_start, nstep, maxc, minr)      ▷ First stage.
while stack is not empty do                              ▷ Second stage: estimate thresholds.
    ⟨n, c⟩ ← stack.pop()                                 ▷ Get a threshold and cubes.
    sample ← SIMPLERANDOMSAMPLE(c, N)                    ▷ Select N random cubes.
    runtimes ← {}
    for each cube from sample do
        ⟨t, st⟩ ← SOLVECUBE(cs, F, cube, maxcst)         ▷ Add a cube and solve.
        if t ≥ maxcst and mode = estimating then         ▷ If CDCL was interrupted
            break                                        ▷ in estimating mode, stop processing sample.
        else
            runtimes.add(t)                              ▷ Add proper runtime.
        if st = True then                                ▷ If SAT,
            isSAT ← True
            if mode = incomplete-solving then            ▷ and incomplete SAT solving mode,
                return ⟨n_best, e_best, c_best, isSAT⟩   ▷ return SAT immediately.
    if SIZE(runtimes) < N then                           ▷ If at least one interrupted in sample,
        break                                            ▷ stop main loop.
    e ← MEAN(runtimes) · SIZE(c)/cores                   ▷ Calculate a runtime estimate.
    if e < e_best then
        ⟨n_best, e_best, c_best⟩ ← ⟨n, e, c⟩             ▷ Update best threshold.
return ⟨n_best, e_best, c_best, isSAT⟩
```

In the first stage, promising thresholds are preselected by the function PreselectThresholds which is described in Algorithm 4, while in the second stage the one with the minimum runtime estimate of the conquer phase is chosen among them. Given a CDCL solver, a CNF, a cube, and a time limit in seconds, the function SolveCube adds the cube to the CNF as unit clauses, runs the CDCL solver with the time limit on the formed CNF, and returns the runtime in seconds and an answer whether the CNF is satisfiable or not.

The algorithm operates in two modes. In the *estimating* mode, the algorithm terminates upon reaching a time limit by the CDCL solver on any subproblem from random samples. In the *incomplete SAT solving* mode (Kautz, Sabharwal, & Selman, 2021), the algorithm terminates upon finding a satisfying assignment. The first mode is aimed at estimating the hardness of a given CNF, while the second one aims to find a satisfying assignment of a satisfiable CNF.

The proposed algorithm has the following features.

1. A stack is used to preselect promising cutoff thresholds in the first stage in order to start the second stage with solving the simplest subproblems. It allows obtaining some estimate quickly and then improve it.

2. In the *estimating* mode, if in the second stage a CDCL solver fails solving some subproblem within the time limit, the algorithm terminates. This is done because in this case it is impossible to calculate a meaningful estimate for the threshold. Another reason is that subproblems from next thresholds will likely be even harder.

3. It is possible that satisfying assignments are found when solving subproblems from random samples. Indeed, if a given CNF is satisfiable, then cubes which imply satisfying assignments might be chosen to samples.

4. In the *estimating* mode, even if a satisfying assignment is found when solving some subproblem from samples, the algorithm does not terminate because in this case the main goal is to calculate a runtime estimate.

5. In the *estimating* mode it is a general algorithm that is able to estimate the hardness of an arbitrary CNF.

6. In the *incomplete SAT solving* mode, a solution can be found only for a satisfiable CNF, and even in this case this is not guaranteed because of the time limit for the CDCL solver.

7. Algorithm 5 can be easily modified to be oriented on finding only one solution in the conquer phase. For this purpose it is required to remove `mode` and return SAT if $\texttt{st} = True$.

8. The proposed runtime estimation is a stochastic costly black-box objective function (Audet & Hare, 2017; Semenov, Zaikin, & Kochemazov, 2021). The algorithm minimizes this objective function.

Since all details of Algorithm 5 are given, it now can be compared with Algorithms A and B (see the beginning of this section). It is clear that the idea is the same in all three algorithms — for a certain value of the cutoff threshold, a sample of cubes is formed, the corresponding subproblems are solved, and finally a runtime estimate is calculated. However, there are several major differences which are listed below.

1. Algorithms A and B were described informally and briefly, while Algorithm 5 is presented formally and in detail.

2. In contrast to Algorithms A and B, Algorithm 5 takes into account the situation when some subproblems from a sample are so hard that they can not be solved in reasonable time by a CDCL solver.

3. Algorithms A and B assume that further in the conquer phase incremental solving is applied to subproblems (Heule et al., 2011), while Algorithm 5 assumes that every subproblem is solved by a non-incremental CDCL solver.

The main difference is the second one. This feature of Algorithm 5 is extremely important in application to cryptanalysis problems, which are considered in the rest of the present paper. The reason is that in this case subproblems in a sample usually differ by thousands and even millions of times in CDCL solver's runtime. A possible explanation why this feature was not taken into account in both Algorithms A and B is that they were applied to combinatorial and geometric problems, where subproblems' hardness in a sample is usually uniform. Importance of the third feature follows from the second one — incremental SAT solving is efficient in the case of the uniform hardness, otherwise it can significantly slow down the solving process.

When a cutoff threshold is found by Algorithm 5, the conquer phase operates as follows. First, subformulas are created by adding cubes to the original CNF in the form of unit clauses. Second, all subformulas are solved by the same CDCL solver that was used to find the threshold for the cubing phase. In contrast to Algorithm 5, here the CDCL solver's runtime is not limited. Finally, the conquer phase outputs a list of all found satisfying assignments (if any) or UNSAT if all subproblems turned out to be UNSAT.

## 5. Considered Inverse Problems and Their SAT Encodings

This section describes the considered inverse problems for step-reduced MD4 and MD5, as well as their SAT encodings.

Following all earlier attempts to invert step-reduced MD4 via SAT (De et al., 2007; Legendre et al., 2012; Lafitte, Nakahara, & Heule, 2014; Gribanova & Semenov, 2018), the padding is omitted (see Subsection 2.4) and only one 512-bit message block is considered. Therefore, a step-reduced MD4 compression function is considered when it operates on the first block, like it was shown in Algorithm 1. The final incrementing is also omitted since it should be done only after 48-th step. Note that these restrictions does not make inverse problems easier since the compression function is the main component of MD4 function from the resistance point view. Inversion of step-reduced MD5 compression function is considered in similar way. For the sake of simplicity, from now on, MD4 means the MD4 compression function, and the same for MD5.

### 5.1 Considered Hashes

The following four hashes are chosen for inversion:

1. `0x00000000 0x00000000 0x00000000 0x00000000`;
2. `0xffffffff 0xffffffff 0xffffffff 0xffffffff`;
3. `0x01234567 0x89abcdef 0xfedcba98 0x76543210`;
4. `0x62c7Ec0c 0x751e497c 0xd49a54c1 0x2b76cff8`.

Recall that `0hash` and `1hash` mean the first and the second hash from the list, respectively. These two hashes are chosen for inversion because it is a common practice in the cryptographic community. The reason is that inverting a hash that looks like a random word is suspicious. Indeed, one can take a random message, produce its hash and declare that this very hash is inverted. On the other hand, if a hash has a regular structure, this approach does not work. All 0s and all 1s are two hashes with the most regular structure, that is why they are usually chosen. For the first time 32-step MD4 was inverted for `0hash` (Dobbertin, 1998), while in 39-step case it was done for `1hash` (De et al., 2007), and later for `0hash` (Legendre et al., 2012). As for the cryptographic hash functions SHA-0 and SHA-1, their 23-step (out of 80) versions for the first time were inverted for `0hash` (Legendre et al., 2012).

The third hash from the list was used to invert 28-step MD5 in (Legendre et al., 2012). Hereinafter this hash is called `symmhash`. It is symmetrical — the last 64 bits are the first 64 bits in reverse order, but at the same time it is less regular than `1hash` or `0hash`. The same result for 28-step MD5 was described in two more papers of the same authors. Unfortunately, none of these three papers explain why `1hash` and `0hash` were not considered.

The fourth hash from the list is chosen randomly. The goal is to show that the proposed approach is applicable not only to hashes with regular structure. This hash is further called `randhash`.

### 5.2 Step-reduced MD4

In contrast to (De et al., 2007), where only $K =$ `0x00000000` was used as a constant in Dobbertin's constraints, in the present study $K =$ `0x00000000` and $K =$ `0xffffffff` are tried in Dobbertin-like constraints. The constraint in step 12 is chosen for the modification (so $p = 12$, see Subsection 3.1) since in (De et al., 2007) the constraint for this very step was entirely omitted. Eight step-reduced versions of MD4 from 40 to 47 steps, as well as the full MD4 are studied. Hence there are $9 \times 4 \times 2 = 72$ MD4-related inverse problems in total. None of these 72 inverse problems have been solved so far.

Consider 40-step MD4. Since $K$ has two values and $y$ has four values, Algorithm 3 should be run on eight inputs. As a result, according to the notation from Subsection 3.1, the following eight inverse problems are formed in the corresponding first iterations of Algorithm 3:

1. `MD4inversion`(`0hash`, 40, `0x00000000`, 12, `0x00000000`);
2. `MD4inversion`(`0hash`, 40, `0xffffffff`, 12, `0x00000000`);

3. $\texttt{MD4inversion}(\texttt{1hash}, 40, \texttt{0x00000000}, 12, \texttt{0x00000000})$;
4. $\texttt{MD4inversion}(\texttt{1hash}, 40, \texttt{0xffffffff}, 12, \texttt{0x00000000})$.
5. $\texttt{MD4inversion}(\texttt{symmhash}, 40, \texttt{0x00000000}, 12, \texttt{0x00000000})$;
6. $\texttt{MD4inversion}(\texttt{symmhash}, 40, \texttt{0xffffffff}, 12, \texttt{0x00000000})$;
7. $\texttt{MD4inversion}(\texttt{randhash}, 40, \texttt{0x00000000}, 12, \texttt{0x00000000})$;
8. $\texttt{MD4inversion}(\texttt{randhash}, 40, \texttt{0xffffffff}, 12, \texttt{0x00000000})$.

For illustrative purpose, consider the first case: invert `0hash` produced by 40-step MD4 with Dobbertin's constraints and $K = \texttt{0x00000000}$. If no preimage exists for this inverse problem, then in the second iteration of Algorithm 3 $L$ is increased by 1, so the inverse problem $\texttt{MD4inversion}(\texttt{0hash}, 40, \texttt{0x00000000}, 12, \texttt{0x00000001})$ is formed and so on.

### 5.3 Step-reduced MD5

In the present paper, inversion of 28-step MD5 compression function is considered for the four hashes presented above. Recall that in contrast to MD4, no extra constraints that reduce the number of preimages are added. Note that for all hashes but `symmhash` the inverse problems have not been solved earlier.

### 5.4 SAT Encodings

It is possible to construct CNFs that encode MD4 and MD5 by the following automatic tools: CBMC (Clarke, Kroening, & Lerda, 2004); SAW (Carter, Foltzer, Hendrix, Huffman, & Tomb, 2013); TRANSALG (Semenov, Otpuschennikov, Gribanova, Zaikin, & Kochemazov, 2020); CRYPTOSAT (Lafitte, 2018). In the present paper, the CNFs are constructed via TRANSALG[2] of version 1.1.5. This tool takes a description of an algorithm as an input and outputs a CNF that implements the algorithm. The description must be formulated in a domain specific C-like language called TA language. TA language supports the following basic constructions used in procedural languages: variable declarations; assignment operators; conditional operators; loops; function calls. In addition it supports various integer operations and bit operations including bit shifting and comparison that is quite handy when describing a cryptographic algorithm. A TA program is a list of functions in TA language. All the constructed CNFs and the corresponding TA programs are available online as a dataset (Zaikin, 2024). All these CNFs can be easily reconstructed by giving the TA programs to TRANSALG as inputs.

In a CNF that encodes step-reduced MD4, the first 512 variables correspond to a message, the last 128 variables correspond to a hash, while the remaining auxiliary variables are needed to encode how the hash is produced given the message. The first 512 variables are further called *message variables*, while the last 128 ones — *hash variables*. The Tseitin transformations are used in TRANSALG to introduce auxiliary variables (Tseitin, 1970).

Characteristics of the constructed CNFs are given in Table 2.

2. https://gitlab.com/transalg/transalg

Table 2: Characteristics of CNFs that encode the considered step-reduced MD4 and MD5.

| Function | Variables | Clauses | Literals |
|---|---|---|---|
| MD4-40 | 7025 | 70 809 | 317 307 |
| MD4-41 | 7211 | 73 158 | 329 330 |
| MD4-42 | 7397 | 75 507 | 341 353 |
| MD4-43 | 7583 | 77 856 | 353 376 |
| MD4-44 | 7769 | 80 205 | 365 399 |
| MD4-45 | 7955 | 82 554 | 377 422 |
| MD4-46 | 8141 | 84 903 | 389 445 |
| MD4-47 | 8327 | 87 252 | 401 468 |
| MD4-48 | 8513 | 89 601 | 413 491 |
| MD5-28 | 7471 | 54 672 | 216 362 |

The CNF that encodes 40-step MD4 has 7 025 variables and 70 809 clauses. Then every step adds 186 variables and 2 349 clauses, so as a result the CNF that encodes the full (48-step) MD4 has 8 513 variables and 89 601 clauses. Note that these CNFs encode the functions themselves, so all message and hash variables are unassigned. To obtain a CNF that encodes an inverse problem for a given 128-bit hash, corresponding 128 one-literal clauses are to be added, so all hash variables become assigned. The problem is to find values of the message variables. Dobbertin's constraints are added as another 384 unit clauses — 32 clauses for every constraint. As a result, the CNF that encodes the inversion of 40-step MD4 with all 12 Dobbertin's constraints has 7 025 variables and 71 321 clauses, while that for the 48-step version consists of 8 513 variables and 90 113 clauses. Note that Dobbertin-like constraints (see Subsection 3.1) are also added as 384 unit clauses; the only difference is in values of the corresponding 32 variables that encode the specially constrained register.

The CNF that encodes 28-step MD5 has 7 471 variables and 54 672 clauses. A CNF that encodes an inverse problem has 7 471 variables and 54 800 clauses since only 128 unit clauses for hash variables are added.

## 6. Inverting 40-step MD4 via Dobbertin-like Constraints

This section first discusses how the proposed algorithms can be combined. Then the experimental setup, simplification, and results for 40-step MD4 are described.

### 6.1 Possible Combinations of Proposed Algorithms

Recall that in every iteration of Algorithm 3 an inverse problem is formed by varying values of $L$ and is solved by some complete algorithm $\mathcal{A}$. Assume that a step-reduced MD4 is to be inverted given a hash. Then the following combinations of Algorithm 3 and the estimating mode of Algorithm 5 are possible:

1. In Algorithm 3, $\mathcal{A}$ is (i) encoding an inverse problem to a CNF, (ii) finding the best cutoff threshold for the CNF by the estimating mode of Algorithm 5, and (iii) running the conquer phase of Cube-and-Conquer with the found cutoff threshold on the CNF.

2. A CNF is formed for $L =$ `0x00000000` as in the first iteration of Algorithm 3, and the best cutoff threshold for it is found by the estimating mode of Algorithm 5. Then Algorithm 3 starts such that $\mathcal{A}$ is (i) encoding an inverse problem to a CNF and (ii) running the conquer phase of Cube-and-Conquer with the found cutoff threshold on the CNF.

In the first combination, its own cutoff threshold is found in every iteration, while in the second one, a cutoff threshold is found once and then it is used in every iteration. In this study, the second combination is used to reduce the number of Algorithm 5 calls. Note that for any value of $L$ the same amount of unit clauses is added to a CNF.

### 6.2 Experimental Setup

Algorithm 3 was implemented in Python, while Algorithm 5 and the conquer phase of Cube-and-Conquer were implemented in C++ as a parallel SAT solver Estimate-and-Cube-and-Conquer (EnCnC). The sources are available at github[3], while the sources, benchmarks, and solutions are available at Zenodo (Zaikin, 2024).

All experiments were executed on a personal computer equipped with the 12-core CPU AMD 3900X. The implementations are multithreaded, so all 12 CPU cores were employed. In case of Algorithm 5, values of a cutoff threshold and then subproblems from samples are processed in parallel. In case of the conquer phase, subproblems are processed in parallel.

The inputs of Algorithm 3 in case of 40-step MD4 were discussed in Section 5. As for Algorithm 5, the inputs are as follows:

- March_cu lookahead solver (Heule et al., 2011) since it has been recently successfully applied to several hard problems (Heule et al., 2016; Heule, 2018b).
- `nstep` = 5. It was chosen in preliminary experiments. If this parameter is equal to 1, then a better threshold usually can be found, but at the same time Algorithm 5 becomes quite time consuming. On the other hand, if *nstep* is quite large, e.g., 50, then as a rule almost all most promising thresholds are skipped.
- `maxc` = 2 000 000. On the considered CNFs, March_cu reaches 2 000 000 cubes in about 30 minutes, so this value of *maxc* looks reasonable. Higher values were also tried, but it did not give any improvement.
- `minr` = 1 000. If it is less then 1 000, then subproblems are too hard because they are not simplified enough by lookahead. At the same time, higher value of this parameter did not allow collecting enough amount of promising thresholds.
- $N = 1\ 000$. First $N = 100$ was tried, but it led to too optimistic estimates which were several times lower than real solving time. On the other hand, $N = 10\ 000$ is too time consuming and gives just modest improvement in accuracy compared to $N = 1\ 000$. The accuracy of obtained estimates is discussed later in Subsection 7.2.
- Kissat CDCL solver of version sc2021 (Biere, Fleury, & Heisinger, 2021a) because this solver and its modifications won the SAT Competition in 2020–2023.

3. https://github.com/olegzaikin/EnCnC

- `maxst` = 5 000 seconds. It is a standard time limit in SAT Competitions (Balyo, Froleyks, Heule, Iser, Järvisalo, & Suda, 2021), so modern CDCL solvers are designed to show all their power within this time.
- `cores` = 12.
- `mode` = `estimating`.

### 6.3 Simplification

In case of 40-step MD4, two parameters were varied for each of four considered hashes (see Section 5.1). The first one is the value of the Dobbertin's constant $K$ (see Section 3): `0x00000000` and `0xffffffff`. The second one is simplification applied to a CNF. A motivation behind varying the second parameter is as follows. First, it is crucial to simplify a CNF before giving it to a lookahead solver. Second, in preliminary experiments it was found out that the simplification type can significantly alter the effectiveness of Cube-and-Conquer on the considered problems.

The CDCL solver CaDiCaL of version 1.5.0 (Froleyks & Biere, 2021) was used to simplify the CNFs. This solver uses inprocessing, i.e., a given CNF is simplified during the CDCL search (Biere, 2011). The more conflicts have been generated by a CDCL solver so far, the more simplified (in terms of the number of variables) the CNF has been made. A natural simplification measure in this case is the number of generated conflicts. In the experiments related to 40-step MD4, the following limits on the number of generated conflicts were tried: 1, 10 thousand, 100 thousand, 1 million, 10 million. Note that 1 conflict as the limit in some cases gives the same result as UP (see Subsection 2.1), while in the remaining cases the corresponding CNF is slightly smaller.

For example, consider problem `MD4inversion`(`1hash`, 40, `0xffffffff`, 12, `0x00000000`). Table 3 presents characteristics of six CNFs which encode this problem. The original (unsimplified) CNF is described by the number of variables, clauses, and literals. For those simplified by CaDiCaL, also the runtime on 1 CPU core is given.

Table 3: CNFs that encode `MD4inversion`(`1hash`, 40, `0xffffffff`, 12, `0x00000000`). The best values are marked with bold.

| Simplification type | Variables | Clauses | Literals | Simplification runtime |
|---|---|---|---|---|
| no (original CNF) | 7025 | 71 321 | 317 819 | - |
| 1 conflict | 3824 | 33 371 | 138 820 | 0.02 sec |
| 10 thousand conflicts | 2969 | 27 355 | 116 618 | 0.31 sec |
| 100 thousand conflicts | 2803 | 23 121 | 94 250 | 4.29 sec |
| 1 million conflicts | 2756 | **22 391** | **90 412** | 1 min 19 sec |
| 10 million conflicts | **2054** | 24 729 | 110 267 | 33 min |

It is clear, that first the number of variables, clauses, and literals decrease, and then 10 million conflicts provides lower number of variables yet the number of clauses and literals is higher than that on 1 million conflicts. For other hashes and values of $L$ the picture is similar.

### 6.4 Experiments

Of course, more parameters can be varied for each hash in addition to $K$ and simplification type mentioned in the previous subsection. One of the most natural is a CDCL solver used in Cube-and-Conquer. For example, a cryptanalysis oriented solver can be chosen (Soos, Nohl, & Castelluccia, 2009; Nejati & Ganesh, 2019; Kochemazov, 2021). Moreover, internal parameters of the chosen CDCL solver can be varied as well.

Recall that there are 4 hashes and 5 simplification types, while $K$ has 2 values, so $4 \times 5 \times 2 = 40$ CNFs were constructed with fully applied Dobbertin's constraints ($L = \texttt{0x00000000}$) for MD4-40. On each of them the first iteration of Algorithm 3 was run. It turned out, that Algorithm 5 could not find any estimates for all 20 CNFs with $K = \texttt{0x00000000}$. The reason is that in all these cases Kissat was interrupted due to the time limit even for the simplest (lowest) values of the cutoff threshold. On the other hand, for $K = \texttt{0xffffffff}$ much more positive results were achieved. For `0hash`, `symmhash`, and `randhash`, estimates for all simplification types were successfully calculated, and the best type was 1 conflict in all these cases. On the other hand, for `1hash` no estimates were found for 1 conflict and 10 thousand conflicts, while the best estimate was gained for 1 million conflicts.

The results are presented in Table 4. For each pair (simplification type, hash), the best estimate for 12 CPU cores, the corresponding cutoff threshold, and the number of cubes are given. Here "-" means that no estimate was obtained because Kissat was interrupted on the simplest threshold. Runtimes of Algorithm 5 are not presented there, but on average it took about 2 hours for $K = \texttt{0x00000000}$ and about 3 hours for $K = \texttt{0xffffffff}$.

Figure 1 depicts how the objective function was minimized on the inverse problem for `0hash`. Here 10k stands for 10 thousand conflicts, 1m for 1 million conflicts and so on. The figures for the remaining three hashes can be found in Appendix A.

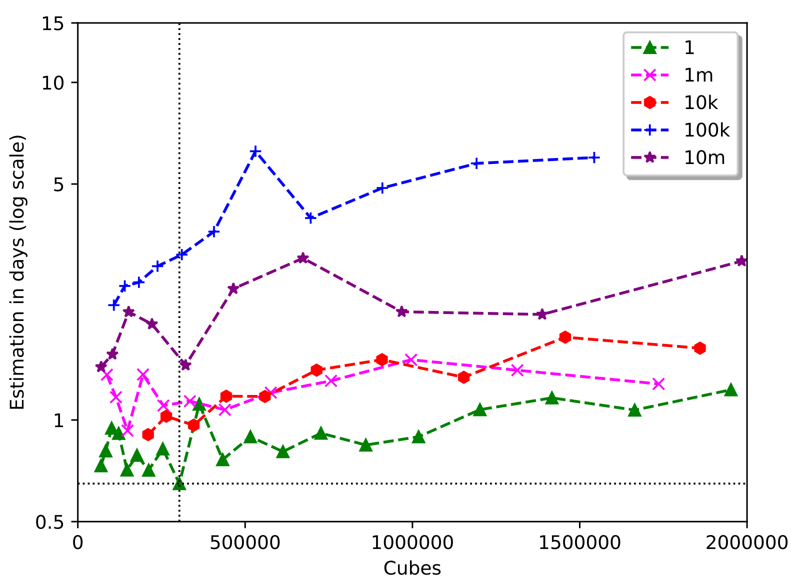



Figure 1: Minimization of the objective function on 40-step MD4, `0hash`. The intersection of two dotted lines shows the best estimate.

Table 4: Runtime estimates for 40-step MD4. The best estimates are marked with bold.

| Hash | Simplification conflicts | $e_{\texttt{best}}$ | $n_{\texttt{best}}$ | $\lvert c_{\texttt{best}}\rvert$ |
|---|---|---|---|---|
| `0` | **1** | **15 h 33 min** | **3290** | **303 494** |
| | 10 thousand | 21 h 43 min | 2530 | 210 008 |
| | 100 thousand | 52 h 32 min | 2485 | 107 657 |
| | 1 million | 22 h 19 min | 2400 | 148 518 |
| | 10 million | 34 h 27 min | 1895 | 69 605 |
| `1` | 1 | - | - | - |
| | 10 thousand | - | - | - |
| | 100 thousand | 81 h 31 min | 2535 | 362 429 |
| | **1 million** | **42 h 43 min** | **2510** | **182 724** |
| | 10 million | 991 h 12 min | 1890 | 1 671 849 |
| `symm` | **1** | **19 h 16 min** | **3395** | 80 491 |
| | 10 thousand | 29 h 47 min | 2725 | 181 267 |
| | 100 thousand | 22 h 44 min | 2615 | 60 403 |
| | 1 million | 21 h 11 min | 2530 | 151 567 |
| | 10 million | 59 h 28 min | 1945 | 189 744 |
| `rand` | **1** | **14 h 27 min** | **3400** | **75 823** |
| | 10 thousand | 227 h 54 min | 2660 | 1 098 970 |
| | 100 thousand | 20 h 22 min | 2540 | 159 942 |
| | 1 million | 17 h 33 min | 2455 | 225 854 |
| | 10 million | 81 h 3 min | 1915 | 242 700 |

As mentioned in Section 4, in the estimating mode of Algorithm 5 it is possible to find satisfying assignments of a given satisfiable CNF. That is exactly what happened for `symmhash` — a satisfying assignments was found for the CNF simplified by 100 thousand conflicts. It means that a preimage for `symmhash` generated by 40-step MD4 was found just in few hours during the search for good thresholds for the cubing phase. However, the goal was to solve all subproblems of the considered inverse problems (up to chosen value of $L$) to find more preimages. That is why using the cubes produced with the help of the best cutoff thresholds, the conquer phase was run on all four inverse problems: 1-conflict-based for `0hash`, `symmhash`, and `randhash`; 1-million-conflicts-based for `1hash`. As a result, all subproblems were solved successfully. The subproblems' runtimes in case of `0hash` are shown in Figure 2.

For `0hash` and `1hash`, no satisfying assignments were found, therefore the corresponding inverse problems have no solutions. On the other hand, satisfying assignments were found for `symmhash` and `randhash`. The found thresholds, estimates, and the real runtimes are presented in Table 5. In the header, `sol` stands for the number of solutions. Note that the best estimate $e_{best}$ was calculated only for $L = \texttt{0x00000000}$, so for other values of $L$ it is equal to "-". The right three columns present subproblems' statistics: mean runtime; maximum runtime; and standard deviation of runtimes (when they are in seconds). The minimum runtime is not reported since it was equal to 0.007 seconds in all cases.

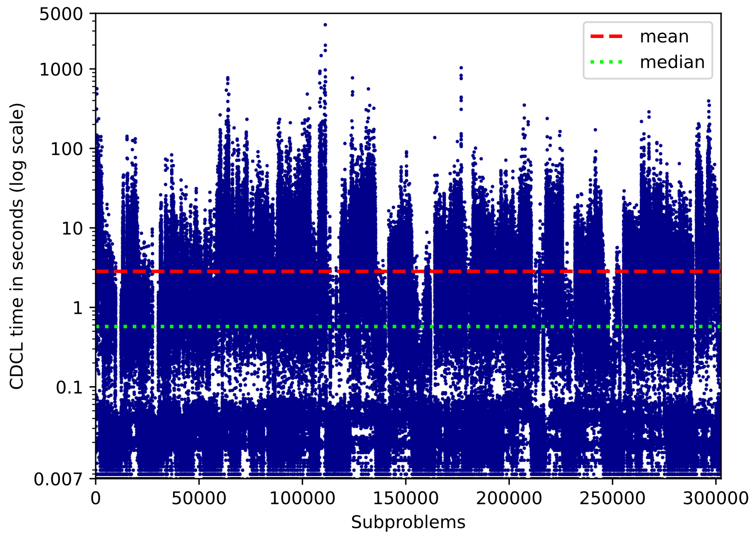



Figure 2: KISSAT runtimes on subproblems from the conquer phase applied to `MD4inversion`(`0hash`, 40, `0xffffffff`, 12, `0x00000000`)

.

Table 5: Estimated and real runtimes (on 12 CPU cores) of the conquer phase for inverse problems related to 40-step MD4. The best estimates from Table 4 are presented.

| Hash | $L$ | $e_{\texttt{best}}$ | real time | sol | mean | max | sd |
|---|---|---|---|---|---|---|---|
| 0 | 0x00000000 | 15 h 33 min | 20 h 9 min | 0 | 2.84 sec | 1 h | 13.68 |
| | 0x00000001 | - | 19 h 25 min | 0 | 2.61 sec | 29 min | 10.22 |
| | 0x00000002 | - | 34 h 27 min | 1 | 5.7 sec | 38 min | 20.62 |
| 1 | 0x00000000 | 42 h 43 min | 48 h 29 min | 0 | 11.5 sec | 26 min | 30.23 |
| | 0x00000001 | - | 59 h 7 min | 0 | 4.08 sec | 17 min | 11.4 |
| | 0x00000002 | - | 28 h 1 min | 1 | 7.7 sec | 17 min | 18.68 |
| symm | 0x00000000 | 19 h 16 min | 20 h 45 min | 2 | 11.24 sec | 18 min | 21.1 |
| rand | 0x00000000 | 14 h 27 min | 15 h 48 min | 1 | 9.08 sec | 38 min | 21.59 |

The next iteration of Algorithm 3 (with $L =$ `0x00000001`) was executed for `0hash` and `1hash`. Note that the same simplification and cutoff threshold as for $L =$ `0x00000000` were applied to the corresponding CNFs. The conquer phase again did not find any satisfying assignment. Finally, preimages for both hashes were found in the third iteration ($L =$ `0x00000002`), see Table 5. All found preimages are presented in Table 6. The obtained results will be discussed in the next section.

Table 6: Found preimages for 40-step MD4.

| Hash | Preimages |
|---|---|
| `0` | 0xe57d8668 0xa57d8668 0xa57d8668 0xbc8c857b 0xa57d8668 0xa57d8668 0xa57d8668 0xcb0a1178<br>0xa57d8668 0xa57d8668 0xa57d8668 0x307bc4e7 0xad02e703 0xe1516b23 0x981c2a75 0xc08ea9f7 |
| `1` | 0xe57d8668 0xa57d8668 0xa57d8668 0x1d236482 0xa57d8668 0xa57d8668 0xa57d8668 0x97a13204<br>0xa57d8668 0xa57d8668 0xa57d8668 0x991ede3 0x301e2ac3 0x5bed2a3d 0xe167a833 0x890d22f0 |
| `symm` | 0xa57d8668 0xa57d8668 0xa57d8668 0xc8cf2f7c 0xa57d8668 0xa57d8668 0xa57d8668 0x61915bc1<br>0xa57d8668 0xa57d8668 0xa57d8668 0x2c017cc4 0xda6acfa2 0x55e9f993 0x50d83f7b 0x2d7d47a6 |
| | 0xa57d8668 0xa57d8668 0xa57d8668 0x154f3b86 0xa57d8668 0xa57d8668 0xa57d8668 0x95b7616d<br>0xa57d8668 0xa57d8668 0xa57d8668 0xf3ca15df 0x7eb66f5e 0x446dc43f 0x7d8e2888 0xafe37a76 |
| `rand` | 0xa57d8668 0xa57d8668 0xa57d8668 0xbb809ab0 0xa57d8668 0xa57d8668 0xa57d8668 0xab67285f<br>0xa57d8668 0xa57d8668 0xa57d8668 0x85517639 0xc3eab3d 0x6edfba39 0xa1512693 0xaa686ac9 |

## 7. Inverting 41-, 42-, and 43-step MD4 via Dobbertin-like Constraints

This section presents results on inverting 41-, 42-, and 43-step MD4. Finally, all MD4-related results are discussed.

### 7.1 41-, 42-, and 43-step MD4

Recall that in the previous section on inverting 40-step MD4, Algorithm 3 was run on 40 CNFs: for each of 4 hashes, 2 values of $K$ and 5 simplification types were tried. Note that $K =$ `0x00000000` did not allow solving any 40-step-related problem. As for simplification types, for 3 hashes out of 4 the best estimates were obtained on 1-conflict-based CNFs, while for the remaining one 1 million conflicts was the best. Following these results, in this section only $K =$ `0xffffffff` and two mentioned simplification types are used. Therefore, only 8 CNFs were constructed for 41-step MD4, and the same for 42-, 43-, and 44-step MD4. Also it turned out that the best 40-step-related estimates were achieved when at most 303 494 cubes were produced, see Table 4. That is why in this section the value of $maxc$ is reduced from 2 000 000 to 500 000. The remaining input parameters of Algorithm 5 are the same.

The same approach was applied as in the previous section: for each pair (the number of steps, hash) first the best cutoff threshold was found via Algorithm 5 for a CNF with added Dobbertin's constraints ($L =$ `0x00000000`), and then Algorithm 3 used the found threshold to run the conquer phase of Cube-and-Conquer as a complete algorithm in each iteration. For 44 steps, no estimates were obtained. On the other hand, for 41, 42, and 43 steps estimates were successfully calculated and they turned out to be comparable to that for 40 steps. Moreover, Algorithm 5 found preimages for two problems: 41 step and `1hash`; 42 steps and `0hash`. In Section 4 it was discussed that such a situation is possible if a given CNF is satisfiable. The found estimates for 43-step MD4 are presented in Table 7. For all hashes, 1 conflict was the best. For 41 steps, 1 conflict was better on `0hash` and `1hash`, while on the remaining two hashes 1-million-conflicts based simplification was the winner. On 42-step MD4, 1 conflict was the best for all hashes except `1hash`.

Table 7: Runtime estimates for 43-step MD4. The best estimates are marked with bold.

| Hash | Simplif. conflicts | $e_{\texttt{best}}$ | $n_{\texttt{best}}$ | $\lvert c_{\texttt{best}} \rvert$ |
|---|---|---|---|---|
| `0` | **1** | **15 h 26 min** | **3 390** | **103 420** |
| | 1 million | - | - | - |
| `1` | **1** | **39 h 10 min** | **3 395** | **98 763** |
| | 1 million | 52 h 5 min | 2 575 | 121 969 |
| `symm` | **1** | **37 h 51 min** | **3 395** | **81 053** |
| | 1 million | 50 h 7 min | 2 555 | 253 489 |
| `rand` | **1** | **49 h 13 min** | **3 385** | **120 619** |
| | 1 million | 86 h 23 min | 2 565 | 246 972 |

Figure 3 depicts how the objective function was minimized on the inverse problem for `1hash` in case of 43 steps. Figures for the remaining three 43-steps-related inverse problems are presented in Appendix A.

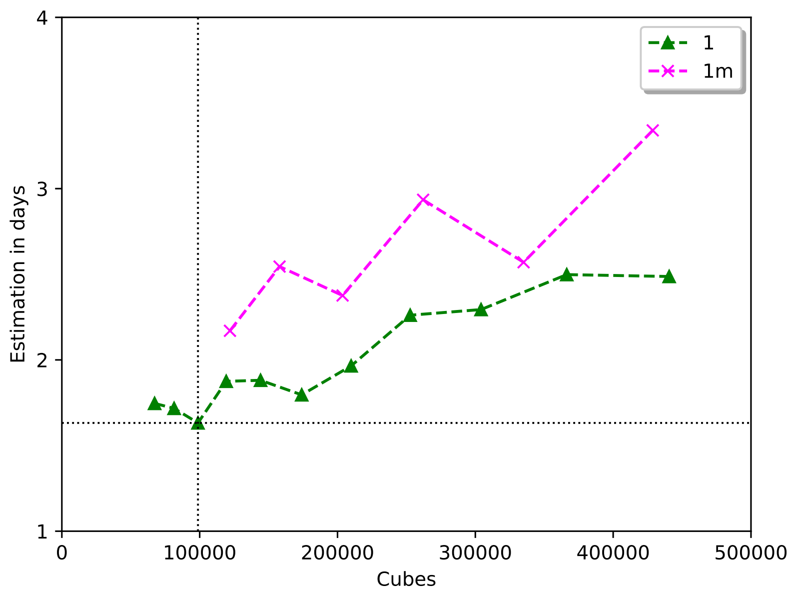



Figure 3: Minimization of the objective function on the inverse problem `MD4inversion`(`1hash`, 43, `0xffffffff`, 12, `0x00000000`). The intersection of two dotted lines shows the best estimate among all simplification types.

Using the found cutoff thresholds, Algorithm 3 was run on all inverse problems with $L =$ `0x00000000`, and for 43 steps preimages were found for all four hashes. For steps 41 and 42, preimages were found in the first or the second iteration of Algorithm 3. The results are presented in Table 8. Here values 0 and 1 of $L$ stand for `0x00000000` and `0x00000001`, respectively, while sd stands for standard deviation in seconds. It can be seen that at least some inverse problems turned out to be easier compared to that for 40-step MD4. This phenomenon is discussed in the next subsection.

Table 8: Estimated and real runtimes (on 12 CPU cores) of the conquer phase for inverse problems related to 41-, 42, and 43-step MD4.

| Steps | Hash | $L$ | $e_{\texttt{best}}$ | real time | sol | mean | max | `sd` |
|---|---|---|---|---|---|---|---|---|
| 41 | `0` | 0 | 8 h 40 min | 10 h 11 min | 0 | 6.4 sec | 17 min | 16.77 |
| | | 1 | - | 21 h 23 min | 1 | 12.41 sec | 14 h 23 min | 421.41 |
| | `1` | 0 | 37 h | 45 h 10 min | 3 | 9.78 sec | 52 min | 44.73 |
| | `symm` | 0 | 19 h 54 min | 20 h 10 min | 0 | 12.08 sec | 17 min | 24.28 |
| | | 1 | - | 20 h 15 min | 4 | 11.57 sec | 17 min | 23.66 |
| | `rand` | 0 | 16 h 6 min | 17 h 25 min | 1 | 10.05 sec | 43 min | 31.07 |
| 42 | `0` | 0 | 19 h 36 min | 22 h 32 min | 3 | 11.68 sec | 19 min | 25.51 |
| | `1` | 0 | 25 h 15 min | 29 h 19 min | 0 | 10.91 sec | 1 h 14 min | 45.61 |
| | | 1 | - | 39 h | 1 | 16.38 sec | 2 h 18 min | 86.32 |
| | `symm` | 0 | 28 h 20 min | 29 h 35 min | 1 | 12.25 sec | 32 min | 19.98 |
| | `rand` | 0 | 21 h 16 min | 21 h 30 min | 0 | 10.22 sec | 15 min | 18.51 |
| | | 1 | - | 20 h 35 min | 3 | 9.34 sec | 13 min | 16.71 |
| 43 | `0` | 0 | 15 h 26 min | 17 h 14 min | 2 | 7.23 sec | 16 min | 16.6 |
| | `1` | 0 | 39 h 10 min | 42 h 16 min | 1 | 18.64 sec | 39 min | 29.88 |
| | `symm` | 0 | 37 h 51 min | 41 h 55 min | 1 | 22.59 sec | 34 min | 46.44 |
| | `rand` | 0 | 49 h 13 min | 51 h 21 min | 1 | 18.51 sec | 46 min | 30.41 |

The subproblems' runtimes for 43 steps and `1hash` are shown in Figure 4. In Table 9, the found preimages for 43-step MD4 are presented. The corresponding tables for steps 41 and 42 are presented in Appendix B.

### 7.2 Discussion

**Correctness** The correctness of the found preimages was verified by the reference implementation from (Rivest, 1990). This verification can be easily reproduced since MD4 is hard to invert, but the direct computation is extremely fast. First, the additional actions (padding, incrementing, see Section 5), as well as the corresponding amount of the last steps should be deleted. Then the found preimages should be given as inputs to the compression function.

**Simplification** According to the estimates, in most cases the 1-conflict-based simplification is better than more advanced simplifications. On the other hand, if only this simplification type had been chosen, then the inverse problem for `1hash` produced by 40-step MD4 would have remained unsolved. The non-effectiveness of the advanced simplifications is an interesting phenomenon which is worth investigating in the future.

**Classes of subproblems** Figures 2 and 4 show that in the conquer phase about 25% of subproblems are extremely easy (runtime is less than 0.1 second) and there is a clear gap

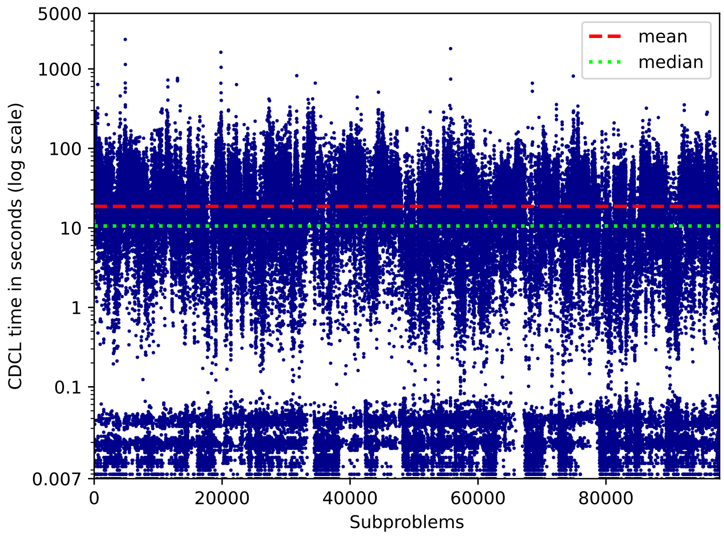



Figure 4: KISSAT runtimes on subproblems from the conquer phase applied to `MD4inversion`(`1hash`, 43, `0xffffffff`, 12, `0x00000000`).

Table 9: Found preimages for 43-step MD4.

| Hash | Preimages |
|---|---|
| `0` | 0xa57d8668 0xa57d8668 0xa57d8668 0xf48a97a3 0xa57d8668 0xa57d8668 0xa57d8668 0xd330e8ed<br>0xa57d8668 0xa57d8668 0xa57d8668 0x37c9ca21 0xe1df551f 0x7f49d66a 0x135a1c93 0x9e744bdb |
| | 0xa57d8668 0xa57d8668 0xa57d8668 0xb289afa0 0xa57d8668 0xa57d8668 0xa57d8668 0xaf2c850e<br>0xa57d8668 0xa57d8668 0xa57d8668 0x19c5ce09 0xcae6b29e 0xb2595b20 0xab3a433d 0xf6cdee42 |
| `1` | 0xa57d8668 0xa57d8668 0xa57d8668 0x82ef987a 0xa57d8668 0xa57d8668 0xa57d8668 0xe18fbc3b<br>0xa57d8668 0xa57d8668 0xa57d8668 0x558f3513 0xbf09004d 0x8fb490dd 0x502eca9 0xbd0e1a80 |
| `symm` | 0xa57d8668 0xa57d8668 0xa57d8668 0xd1c33d35 0xa57d8668 0xa57d8668 0xa57d8668 0xc8519181<br>0xa57d8668 0xa57d8668 0xa57d8668 0x8157aaf2 0xd7bdc37b 0xe52f3348 0xf17901d9 0x7e2de5a4 |
| `rand` | 0xa57d8668 0xa57d8668 0xa57d8668 0x24f0e099 0xa57d8668 0xa57d8668 0xa57d8668 0xe57e4c54<br>0xa57d8668 0xa57d8668 0xa57d8668 0x8fbbadcd 0xc0326ae6 0xe0e6a048 0x6217a3b9 0x15ee5a3b |

between these subproblems and the remaining ones. Since this gap is much lower than mean and median runtime, is seems promising to solve all extremely easy subproblems beforehand and apply the corresponding reasoning to the remaining subproblems.

**Estimation accuracy** The obtained estimates can be treated as accurate since they are close to the real solving times (see Tables 5 and 8). On average the real time on inverse problems with $L =$ `0x00000000` is 11 % higher than the estimated time, while in the worst case for 40-step MD4 and `0hash` the real time is 30 % higher. As for the real time on inverse problems with $L =$ `0x00000001` and $L =$ `0x00000002`, the situation is different. In some cases, the real time is still close to the estimated time for $L =$ `0x00000000`. However, for `MD4inversion`(`0hash`, 41, `0xffffffff`, 12, `0x00000001`) the real time is 2.5 times higher, while the standard deviation is also very high. It can be concluded that the heavy-tail behavior occurs in this case (Gomes & Sabharwal, 2021). These results might indicate that it is better to find its own cutoff threshold for each value of $L$, that corresponds to the first combination of Algorithm 3 and Algorithm 5 described at the beginning of Section 6. Note that for those problems where their own thresholds were used, i.e., when $L =$ `0x00000000`, the heavy-tail behavior does not occur.

**Hardness of inverse problems** It might seem counterintuitive that for 40-43 steps the hardness of the inverse problems in fact is more or less similar. Recall that when Dobbertin's constraints are applied, values of 9 message words of 16 with indices 0, 1, 2, 4, 5, 6, 8, 9, and 10 are derived automatically (in a CNF this is done by UP), so only 7 words remain unknown (see Subsection 2.5). It means that in the CNF 224 message bits are unknown compared to 512 message bits when Dobbertin's constraints are not added. It holds true for Dobbertin-like constraints as well. In the 40th step, the register's value is updated via a round function that takes as input an unknown word $M[14]$ along with registers' values. That is why the 40th step gives a leap in hardness compared to 39 steps. In the next 8 steps, message words with the following indices are used for updating: $1, 9, 5, 13, 3, 11, 7, 15$. It means that in steps 41, 42, and 43 the round function operates with known (constant) $M[1]$, $M[9]$, and $M[5]$, respectively. In MD4 compression function, the main hardness is added by mixing a message word with registers' values. Therefore, steps 41-43 do not add any hardness. Rather, additional connections between registers' values are added. As for the remaining steps 44-48, only unknown message words are used for updating, so each of these steps gives a new leap in hardness. That is why no estimates were calculated for 44 steps earlier in this section — those inverse problems are much harder. Apparently, the leap between steps 43 and 44 has the similar nature as that between steps 39 and 40. In this case, the usage of a powerful supercomputer can help inverting 44-step MD4.

**Partially constant preimages** In all found preimages for steps 40 and 43, 9 of 16 message words are equal to `0xa57d8668`. These words were automatically derived because of the known $K$. Recall that $K =$ `0xffffffff` was used in all cases. However, in some preimages for 41 and 42 steps $M[0] =$ `0x257d8668`, while all remaining 8 message words are equal to `0xa57d8668`. The reason is that in these cases the preimages were found not in the first iteration of Algorithm 3, so in the 13th step the constant was not $K$, but rather its slightly modified value.

**Preimage attacks and second preimage attacks** In this section and Section 6, practical preimage attacks (see Subsection 2.3) on 40-, 41-, 42-, and 43-step MD4 are proposed. Recall that the conquer phase aimed to solve all subproblems (see Section 4). As a result, 2 preimages were found for the hash `symm` produced by 40-step MD4, see Table 5. According to Table 8, more that one preimage was found for at least one hash in case of 41-, 42-, and 43-step MD4. Therefore, second preimage attacks are proposed on 40-, 41-, 42-, and 43-step MD4. It is possible that more preimages exist for the considered inverse problems. If an AllSAT solver had been applied to the subproblems instead of a SAT solver, then all preimages would have been found.

## 8. Inverting Unconstrained 28-step MD5

As mentioned in Subsection 2.6, Dobbertin's constraints are not efficient for MD5. That is why in this study 28-step MD5 is inverted without adding any extra constraints, like it was done in (Legendre et al., 2012). Recall that in this case for an arbitrary hash there are about $2^{384}$ preimages, but it is not easy to find any of them. Algorithm 5 in its estimating mode is not applicable to MD5 either because the cubing phase gives too hard subproblems for an unconstrained inverse problem, so no runtime estimate can be calculated in reasonable time. On the other hand, since the considered inverse problem has huge number of solutions, the incomplete SAT solving mode of Algorithm 5 suites well for it.

First a CNF that encodes 28-step MD5 was constructed based on the encoding from Subsection 5.3. The CNF has 7 471 variables and 54 672 clauses. The same four hashes were considered for inversion as for MD4: `0hash`, `1hash`, `symmhash`, `randhash`. Therefore, 4 CNFs were constructed by adding corresponding 128 unit clauses to the original CNF. Then these CNFs were simplified by CaDiCaL such that at most 1 conflict was generated. Characteristics of the simplified CNFs are presented in Table 10.

Table 10: Characteristics of the simplified CNFs that encode inverse problems for 28-step MD5.

| Hash | Variables | Clauses | Literals |
|---|---|---|---|
| `0` | 6 814 | 50 572 | 199 596 |
| `1` | 6 844 | 50 749 | 200 153 |
| `symm` | 6 842 | 50 737 | 200 114 |
| `rand` | 6 842 | 50 741 | 200 110 |

The SAT solver EnCnC (see the beginning of Subsection 6.2) was run on these CNFs in the incomplete SAT solving mode with the following inputs:

- March_cu.
- `nstep` = 10.
- `minr` = 0.
- $N = 1\ 000$.
- Kissat_sc2021.
- `maxst` = 5 000 seconds.
- `cores` = 12.
- `mode` = `incomplete-solving`.

The key parameter here is *maxc* (the maximum number of generated cubes), for which the following values were tried: 2 000 000; 1 000 000; 500 000; 250 000; 125 000; 60 000. Note that the default value of `maxc` in EnCnC is 1 000 000. Recall that in the incomplete

SAT solving mode, EnCnC stops if a satisfying assignment is found; if a CDCL solver is interrupted due to a time limit on some subproblem, EnCnC continues working. The corresponding 6 versions of EnCnC with different values of *maxc* were run on the CNFs with the wall-clock time limit of 1 day. In Table 11, the wall-clock runtimes are presented. Also, the same data is shown in Figure 5.

Table 11: Wall clock time for 28-step MD5 on a 12-core CPU. Here "-" means that the solver was interrupted due to the time limit of 1 day. The best results are marked with bold.

| Solver | 0hash | 1hash | symmhash | randhash |
|---|---|---|---|---|
| EnCnC-maxc=2m | 1 h 47 min | 1 h 41 min | 39 min | 1 h 36 min |
| EnCnC-maxc=1m | 42 min | 53 min | 13 min | 59 min |
| EnCnC-maxc=500k | 48 min | 32 min | 22 min | **15 min** |
| EnCnC-maxc=250k | 38 min | 4 min | 41 min | 37 min |
| EnCnC-maxc=125k | 16 min | 35 min | **6 min** | 20 min |
| EnCnC-maxc=60k | **4 min** | **3 min** | 14 min | 1 h 32 min |
| P-MCOMSPS | - | - | - | - |
| Treengeling | - | - | - | - |

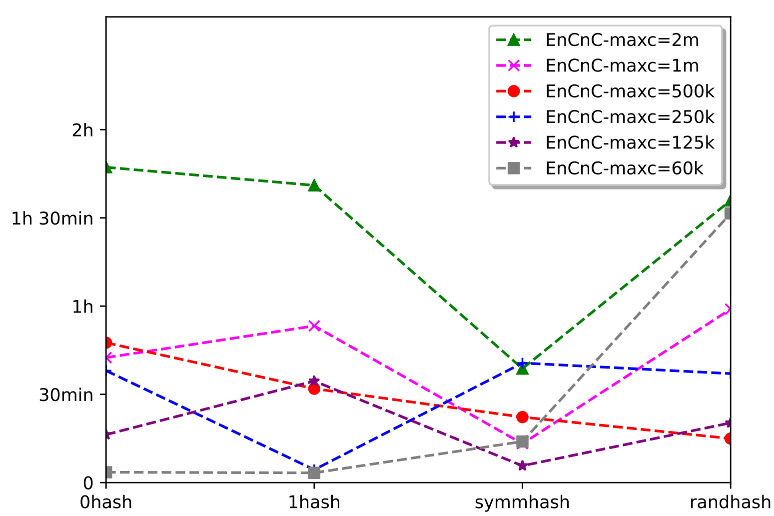



Figure 5: Runtimes of EnCnC in the incomplete SAT solving mode on four MD5-28-related inverse problems.

In addition, two complete parallel CDCL SAT solvers were tried. The first one, P-MCOMSPS, is the winner of the Parallel track in SAT Competition 2021 (Vallade, Frioux, Oanea, Baarir, Sopena, Kordon, Nejati, & Ganesh, 2021). It is a portfolio solver built upon

Table 12: Preimages found by ENCNC-MAXC=2M for 28-step MD5.

| Hash | Preimages |
|---|---|
| `0` | 0xd825e4fb 0xa73fcaa9 0x660cd53d 0xb9308515 0x4677d4e0 0xcadcee62 0x40722cb3 0xf41a4b12<br>0xac2fdec3 0x9cbcb4a3 0xffcca30f 0x9a0e2026 0x475763e5 0x30ce233b 0xbef0cd57 0x1a6b39d |
| `1` | 0xdfe6feeb 0xc4437a85 0x11af5182 0xe3b13f03 0x5103e1fc 0xea231da2 0xc3b513d1 0xb95fa9d7<br>0x7a2a331c 0x2ddf2607 0x699a2dae 0xc1827561 0xfe80aeed 0xcf45b09a 0x5b596c8f 0xd0265347 |
| `symm` | 0x54032182 0x2a1693f1 0x1053aef3 0x9f4d7c87 0x9f0d5ba1 0xb43a63f8 0x4310aa89 0x9df4e0d8<br>0xada73cbf 0x63fd55c2 0x49f1f4a0 0x5e05beff 0x6c149122 0x54a25f8e 0x12ef4bb0 0x78482fb4 |
| `rand` | 0x120686db 0xad5834c6 0x7d660963 0x71c408fe 0x17cf4511 0x75df78de 0x544ae232 0x13745ecc<br>0x9190f8a2 0x4878ab8d 0x43229cc7 0x5013f2de 0xd49b395a 0xa151b704 0x5f1dd4ec 0xc860dfb5 |

the widely-used PAINLESS framework (Frioux, Baarir, Sopena, & Kordon, 2017). The second one, TREENGELING (Biere, 2016), is a Cube-and-Conquer solver. It was chosen to compare ENCNC with a competitor built upon a similar strategy. Besides this, TREENGELING won several prizes in SAT Competitions and SAT Races.

Let us discuss the results. Based on average runtime, the best version of ENCNC is ENCNC-MAXC=125K, while the worst is ENCNC-MAXC=2M. Nevertheless, all versions managed to find satisfying assignments for all 4 CNFs within the time limit. At 23 runs out of 24, versions of ENCNC did it during solving the first 12 subproblems from the first random sample (for the lowest values of the cutoff threshold). It means that KISSAT did not reach the time limit of 5 000 seconds in these cases. The only exception is ENCNC-MAXC=60K on `randhash`, where on all 12 first subproblems KISSAT was interrupted due to the time limit, and then a satisfying assignment was found in one of the next 12 subproblems from the same sample. As for the competitors, they could not solve anything within the time limit. In Table 12, the preimages found by ENCNC-MAXC=2M are presented. It should be noted that preimages for `0hash`, `1hash`, and `randhash` have not been published so far.

The found preimages were verified by the reference implementation from (Rivest, 1992). It can be easily reproduced in the same way that was discussed in Subsection 7.2.

## 9. Related Work

Apparently, SAT-based cryptanalysis was first proposed in 1996 (Cook & Mitchell, 1996), but for the first time it was applied to solve a real cryptanalysis problem in 2000 (Massacci & Marraro, 2000). In particular, a reduced version of the block cipher DES was analyzed there via a SAT solver. Since that publication, SAT-based cryptanalysis has been successfully applied to analyze various block ciphers, stream ciphers, and cryptographic hash functions (Bard, 2009).

SAT-based cryptanalysis via CDCL solvers has been applied earlier to cryptographic hash functions of the MD family as follows. For the first time it was done in (Jovanovic & Janicic, 2005) to construct benchmarks with adjustable hardness. In (Mironov & Zhang, 2006), a practical collision attack on MD4 was performed. 39-step MD4 was inverted in (De et al., 2007; Legendre et al., 2012; Lafitte et al., 2014; Gribanova, Zaikin, Kochemazov, Otpuschennikov, & Semenov, 2017; Gribanova & Semenov, 2018). In (Gladush, Gribanova, Kondratiev, Pavlenko, & Semenov, 2022), the hardness of practical preimage attacks on 43-, 45-, and 47-step MD4 was estimated. In (Gribanova & Semenov, 2020), an MD4-based function was constructed and the full (48-step) version of this function was inverted. As for MD5, in (Mironov & Zhang, 2006) and later in (Gribanova et al., 2017), practical collision attacks on MD5 were performed. In (De et al., 2007), 26-step MD5 was inverted, while in (Legendre et al., 2012) it was done for 27- and 28-step MD5.

Also, CDCL solvers were applied to analyze cryptographic hash functions from the SHA family. Note that SHA-1 is an improved version of MD4. For the first time a collision for SHA-1 was found in (Stevens, Bursztein, Karpman, Albertini, & Markov, 2017) and it was done partially by a CDCL solver. Step-reduced versions of SHA-0, SHA-1, SHA-256, and SHA-3 were inverted in (Nossum, 2012; Legendre et al., 2012; Homsirikamol, Morawiecki, Rogawski, & Srebrny, 2012; Nejati, Liang, Gebotys, Czarnecki, & Ganesh, 2017). An algebraic fault attack on SHA-1 and SHA-2 was performed in (Nejati, Horácek, Gebotys, & Ganesh, 2018), while that on SHA-256 was done in (Nakamura, Hori, & Hirose, 2021).

The first theoretical preimage attack on MD4 with the complexity of $2^{102}$ was proposed in (Leurent, 2008). Later the complexity was reduced to $2^{99.7}$ (Guo, Ling, Rechberger, & Wang, 2010). As for MD5, the best theoretical attack has the complexity of $2^{123.4}$ (Sasaki & Aoki, 2009).

The following hard mathematical problems have been solved via Cube-and-Conquer: the Erdős discrepancy problem (Konev & Lisitsa, 2015); the Boolean Pythagorean Triples problem (Heule et al., 2016); Schur Number Five (Heule, 2018b); Lam's problem (Bright et al., 2021); Keller's Conjecture (Brakensiek, Heule, Mackey, & Narváez, 2022). In (Li, Bright, & Ganesh, 2024), the lower bound for the Minimum Kochen–Specker Problem was improved. In (Weaver & Heule, 2020), new minimal perfect hash functions were found. Note that these hash functions are not cryptographic ones and find their application in lookup tables. In the present paper, for the first time significant cryptanalysis problems were solved via Cube-and-Conquer.

The present paper presents a general Cube-and-Conquer-based algorithm for estimating hardness of SAT instances. Usually this is done by other approaches: the tree-like space complexity (Ansótegui, Bonet, Levy, & Manyà, 2008); supervised machine learning (Hutter, Xu, Hoos, & Leyton-Brown, 2014); the popularity–similarity model (Almagro-Blanco & Giráldez-Cru, 2022); backdoors (Williams, Gomes, & Selman, 2003).

Backdoors are closely connected with Cube-and-Conquer. Informally, backdoor is a subset of variables of a given formula, such that by varying all possible values of the backdoor's variables simpler subproblems are obtained which can be solved independently (Williams et al., 2003; Kilby, Slaney, Thiébaux, & Walsh, 2005; Dilkina, Gomes, & Sabharwal, 2007; Samer & Szeider, 2008). In fact, a set of backdoor's values can be considered a cube, while choosing a proper backdoor and varying all corresponding values is a special way to generate cubes in the cubing phase of Cube-and-Conquer. For a given SAT instance and a backdoor, hardness of the instance can be estimated by processing a (relatively small) sample of subproblems (Semenov, Zaikin, Bespalov, & Posypkin, 2011).

The search for a backdoor with the minimum hardness was reduced to minimization of pseudo-Boolean objective functions in application to SAT-based cryptanalysis in (Semenov, Zaikin, Otpuschennikov, Kochemazov, & Ignatiev, 2018; Kochemazov & Zaikin, 2018; Semenov, Chivilikhin, Pavlenko, Otpuschennikov, Ulyantsev, & Ignatiev, 2021). In (Zaikin & Kochemazov, 2021; Semenov et al., 2021) it was shown that these functions are costly, stochastic, and black-box. In the present paper, a pseudo-Boolean objective function with the same properties is minimized to find a cutoff threshold of the cubing phase of Cube-and-Conquer rather than a backdoor.

## 10. Conclusions and Future Work

This paper proposed two algorithms. Given a hash, the first algorithm gradually modifies one of twelve Dobbertin's constraints for MD4 until a preimage for a given hash is found. Any complete algorithm can be used to solve the corresponding intermediate inverse problems. The second proposed algorithm can operate with a given CNF in two modes. In the estimating mode, cutoff thresholds of the cubing phase of Cube-and-Conquer are varied, and the CNF's hardness for each threshold is estimated via sampling. The threshold with the best estimate can be naturally used to choose a proper computational platform and solve the SAT instance if the estimate is reasonable. This mode is general, so it can be applied to estimate the hardness and solve hard SAT instances from various classes. In the incomplete SAT solving mode, the second algorithm is a SAT solver oriented on satisfiable CNFs with many satisfying assignments.

The cryptographic hash function MD4 was analyzed by a combination of the first algorithm and the estimating mode of the second algorithm, which were implemented as a multithreaded program. As a result, the first practical preimage attacks and second preimage attacks on 40-, 41-, 42-, and 43-step MD4 were performed on a computer. In contrast to MD4, MD5 served as an example of a cryptographic hash function for which no problem-specific constraints were added. By applying the incomplete SAT solving mode of the second algorithm, preimages for two most regular hashes (128 1s and 128 0s) produced by 28-step MD5 were found for the first time.

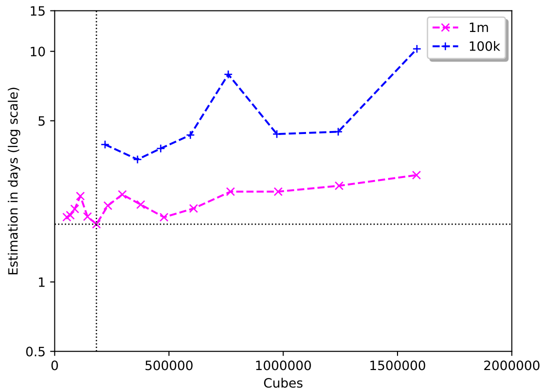
In the future we plan to apply the proposed algorithms to analyze other cryptographic hash functions: SHA-1, SHA-2, RIPEMD. Also we are going to investigate two MD4-related phenomena which were figured out during the experiments. The first one is the non-effectiveness (in most cases) of an advanced simplification in application to the constructed CNFs. The second one is an evident division of subproblems in the conquer phase to extremely simple ones and hard ones. Finally, we plan to compare the estimating mode of the second proposed algorithm with other approaches, which are usually used to estimate the hardness of a given SAT instance.

## Acknowledgments

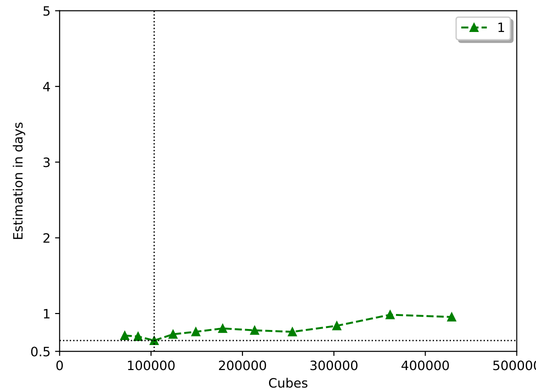
This study was financially supported by the Mathematical Center in Akademgorodok under the agreement No. 075-15-2022-282 with the Ministry of Science and Higher Education of the Russian Federation. The author thanks anonymous reviewers for valuable and thorough comments. The author is grateful to Stepan Kochemazov and Alexander Semenov for fruitful discussions.

## Appendix A. Estimates for Step-reduced MD4

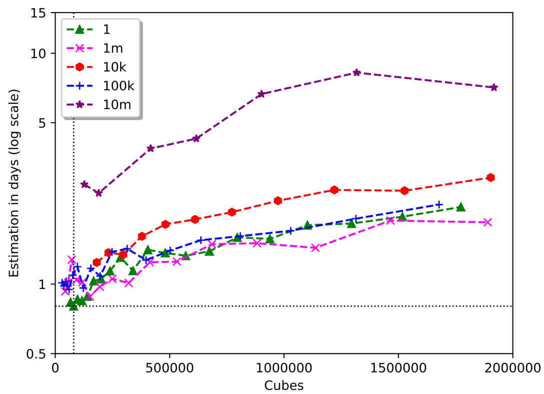
The following figures depict how the objective function was minimized on 40- and 43-step MD4.



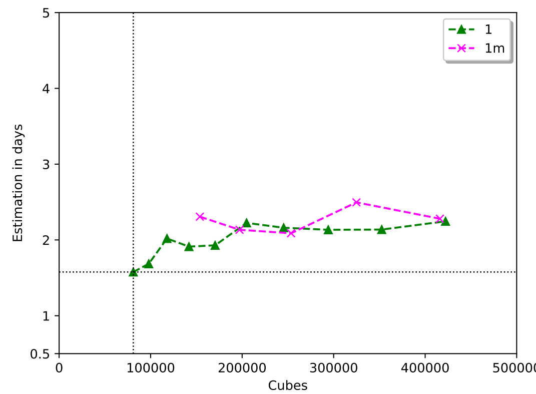

(a) *MD4-40,* 1hash



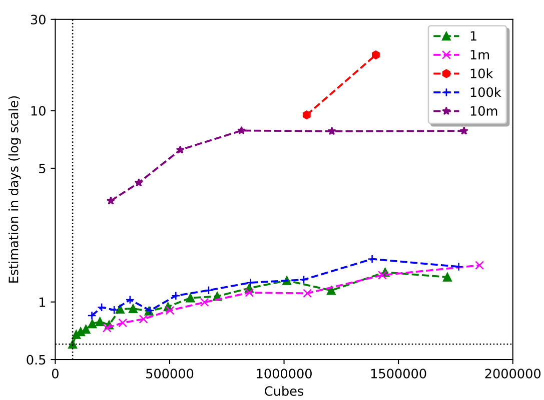

(b) *MD4-43,* 0hash



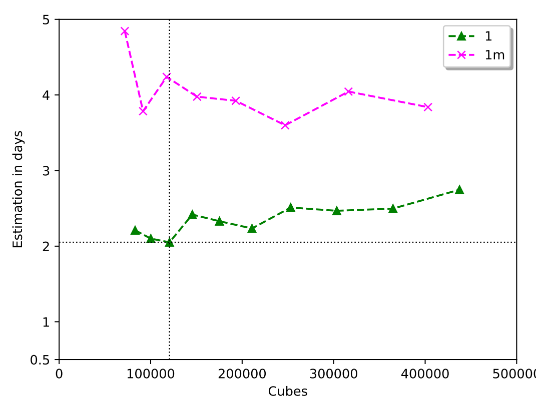

(c) *MD4-40,* symmhash




(d) *MD4-43,* symmhash




(e) *MD4-40,* randhash




(f) MD4-43, *randhash*

Figure 6: Minimization of the objective function on 40- and 43-step MD4.

## Appendix B. Found Preimages for Step-reduced MD4

Table 13: Found preimages for 41-step MD4.

| | |
|---|---|
| *0* | 0x257d8668 0xa57d8668 0xa57d8668 0xdafb914d 0xa57d8668 0xa57d8668 0xa57d8668 0x1edf9f78<br>0xa57d8668 0xa57d8668 0xa57d8668 0x12984195 0x97f0b6c 0xd9e5df17 0xabe482c7 0x23d98522 |
| *1* | 0xa57d8668 0xa57d8668 0xa57d8668 0x5c31dc3 0xa57d8668 0xa57d8668 0xa57d8668 0x52f59fb2<br>0xa57d8668 0xa57d8668 0xa57d8668 0x1e8a7cbb 0x3982e99f 0x812d980d 0x27b8d0b5 0xb81a00d1 |
| *symm* | 0x257d8668 0xa57d8668 0xa57d8668 0xeaaf86e 0xa57d8668 0xa57d8668 0xa57d8668 0xc3b97274<br>0xa57d8668 0xa57d8668 0xa57d8668 0x21b8d189 0x15fc5540 0xd283c2c4 0x7d27396b 0x7bb74632 |
| | 0x257d8668 0xa57d8668 0xa57d8668 0x5e8d818a 0xa57d8668 0xa57d8668 0xa57d8668 0x8fc29cce<br>0xa57d8668 0xa57d8668 0xa57d8668 0x8c6b49cc 0xe31a2c8d 0x9a5e1c5d 0x2dd896f5 0x1ed72fab |
| | 0x257d8668 0xa57d8668 0xa57d8668 0x9278c8f 0xa57d8668 0xa57d8668 0xa57d8668 0x4e3194eb<br>0xa57d8668 0xa57d8668 0xa57d8668 0x22efb603 0xe2b4a054 0xd74ec43 0xf09b0821 0xe4ca9fca |
| | 0x257d8668 0xa57d8668 0xa57d8668 0x6172bd01 0xa57d8668 0xa57d8668 0xa57d8668 0x8e35540f<br>0xa57d8668 0xa57d8668 0xa57d8668 0x4b8210a9 0xd5c0fedb 0x45c28d93 0x1b542bb8 0x74c28676 |
| *rand* | 0xa57d8668 0xa57d8668 0xa57d8668 0x4b11d0ca 0xa57d8668 0xa57d8668 0xa57d8668 0x4c195670<br>0xa57d8668 0xa57d8668 0xa57d8668 0x76529071 0x68d3862d 0xdd3779df 0x768ce847 0x77e1b04e |
| | 0xa57d8668 0xa57d8668 0xa57d8668 0xcfbf3444 0xa57d8668 0xa57d8668 0xa57d8668 0xaac69f2f<br>0xa57d8668 0xa57d8668 0xa57d8668 0xbdaf1de9 0xfb9496dc 0x537e7a8c 0xd083975f 0xf3a5fc76 |
| | 0xa57d8668 0xa57d8668 0xa57d8668 0xbfbf37eb 0xa57d8668 0xa57d8668 0xa57d8668 0xf3252a5c<br>0xa57d8668 0xa57d8668 0xa57d8668 0x3f829fe3 0x28c0fe6 0x27eadfa1 0xc87af86e 0x48fcd23d |

Table 14: Found preimages for 42-step MD4.

| | |
|---|---|
| *0* | 0xa57d8668 0xa57d8668 0xa57d8668 0xecdab667 0xa57d8668 0xa57d8668 0xa57d8668 0xe3844a01<br>0xa57d8668 0xa57d8668 0xa57d8668 0xa3205929 0xfad1ea59 0xd2cae4d2 0x52149d55 0xc82cffbf |
| | 0xa57d8668 0xa57d8668 0xa57d8668 0xae60af85 0xa57d8668 0xa57d8668 0xa57d8668 0x8bcd69e3<br>0xa57d8668 0xa57d8668 0xa57d8668 0x59b8bf6 0x7755a76 0xfbe0b515 0xf9a31765 0x14d516a6 |
| | 0xa57d8668 0xa57d8668 0xa57d8668 0xa9210d09 0xa57d8668 0xa57d8668 0xa57d8668 0xba9694ea<br>0xa57d8668 0xa57d8668 0xa57d8668 0x6a8157fe 0xd6566aae 0xbacb3d6c 0x1ec4854d 0x22357d65 |
| *1* | 0x257d8668 0xa57d8668 0xa57d8668 0xd8f77148 0xa57d8668 0xa57d8668 0xa57d8668 0x88275d15<br>0xa57d8668 0xa57d8668 0xa57d8668 0xcf6b92d0 0x4a8e498d 0x3beb0878 0xb55e027 0x87b4d62c |
| *symm* | 0xa57d8668 0xa57d8668 0xa57d8668 0xd1dce7ea 0xa57d8668 0xa57d8668 0xa57d8668 0xcbc2a90<br>0xa57d8668 0xa57d8668 0xa57d8668 0xd9834f6d 0x5267d5d6 0x41a9cf18 0x71469663 0xbd507731 |
| *rand* | 0x257d8668 0xa57d8668 0xa57d8668 0xbd7389e6 0xa57d8668 0xa57d8668 0xa57d8668 0x3eb8ae3a<br>0xa57d8668 0xa57d8668 0xa57d8668 0x162c323e 0xa4056a04 0x9da74aac 0xfee2c77 0x8b25de8e |
| | 0x257d8668 0xa57d8668 0xa57d8668 0xc1748842 0xa57d8668 0xa57d8668 0xa57d8668 0xd7e32a57<br>0xa57d8668 0xa57d8668 0xa57d8668 0x21c5baab 0x552a7372 0xa21b2963 0x2fe88ffb 0xadfddb3 |
| | 0x257d8668 0xa57d8668 0xa57d8668 0xc455558f 0xa57d8668 0xa57d8668 0xa57d8668 0xff87976a<br>0xa57d8668 0xa57d8668 0xa57d8668 0x3e82e858 0x46ad9cde 0x76f3b1d0 0x31aadb79 0x45cc1c91 |

## References

Almagro-Blanco, P., & Giráldez-Cru, J. (2022). Characterizing the temperature of SAT formulas. *Int. J. Comput. Intell. Syst.*, *15*(1), 69.

Ansótegui, C., Bonet, M. L., Levy, J., & Manyà, F. (2008). Measuring the hardness of SAT instances. In *AAAI*, pp. 222–228.

Aoki, K., & Sasaki, Y. (2008). Preimage attacks on one-block MD4, 63-step MD5 and more. In *SAC*, pp. 103–119.

Audet, C., & Hare, W. (2017). *Derivative-Free and Blackbox Optimization.* Springer Series in Operations Research and Financial Engineering. Springer International Publishing.

Balyo, T., Froleyks, N., Heule, M., Iser, M., Järvisalo, M., & Suda, M. (Eds.). (2021). *Proceedings of SAT Competition 2021: Solver and Benchmark Descriptions*. Department of Computer Science, University of Helsinki.

Balyo, T., & Sinz, C. (2018). Parallel Satisfiability. In *Handbook of Parallel Constraint Reasoning*, pp. 3–29. Springer.

Bard, G. V. (2009). *Algebraic Cryptanalysis* (1st edition). Springer Publishing Company, Incorporated.

Biere, A. (2011). Preprocessing and inprocessing techniques in SAT. In *HVC*, p. 1.

Biere, A. (2016). Splatz, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2016. In *Proc. of SAT Competition 2016 – Solver and Benchmark Descriptions*, pp. 44–45.

Biere, A., Fleury, M., & Heisinger, M. (2021a). CaDiCaL, Kissat, Paracooba entering the SAT Competition 2021. In *SAT Competition 2021 – Solver and Benchmark Descriptions*, pp. 10–13.

Biere, A., Heule, M., van Maaren, H., & Walsh, T. (Eds.). (2021b). *Handbook of Satisfiability - Second Edition*, Vol. 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press.

Böhm, M., & Speckenmeyer, E. (1996). A fast parallel SAT-solver - efficient workload balancing. *Ann. Math. Artif. Intell.*, *17*(3-4), 381–400.

Brakensiek, J., Heule, M., Mackey, J., & Narváez, D. E. (2022). The resolution of Keller's conjecture. *J. Autom. Reason.*, *66*(3), 277–300.

Bright, C., Cheung, K. K. H., Stevens, B., Kotsireas, I. S., & Ganesh, V. (2021). A SAT-based resolution of Lam's problem. In *AAAI*, pp. 3669–3676.

Cai, S., Zhang, X., Fleury, M., & Biere, A. (2022). Better decision heuristics in CDCL through local search and target phases. *J. Artif. Intell. Res.*, *74*, 1515–1563.

Carter, K., Foltzer, A., Hendrix, J., Huffman, B., & Tomb, A. (2013). SAW: the software analysis workbench. In *HILT*, pp. 15–18.

Clarke, E. M., Kroening, D., & Lerda, F. (2004). A tool for checking ANSI-C programs. In *TACAS*, pp. 168–176.

Cook, S. A. (1971). The complexity of theorem-proving procedures. In *STOC*, pp. 151–158. ACM.

Cook, S. A., & Mitchell, D. G. (1996). Finding hard instances of the satisfiability problem: A survey. In *Satisfiability Problem: Theory and Applications*, pp. 1–17.

Damgård, I. (1989). A design principle for hash functions. In *CRYPTO*, pp. 416–427.

Davis, M., Logemann, G., & Loveland, D. W. (1962). A machine program for theorem-proving. *Commun. ACM*, *5*(7), 394–397.

De, D., Kumarasubramanian, A., & Venkatesan, R. (2007). Inversion attacks on secure hash functions using SAT solvers. In *SAT*, pp. 377–382.

Dilkina, B., Gomes, C. P., & Sabharwal, A. (2007). Tradeoffs in the complexity of backdoor detection. In *CP*, pp. 256–270.

Dobbertin, H. (1996). Cryptanalysis of MD4. In *FSE*, pp. 53–69.

Dobbertin, H. (1998). The first two rounds of MD4 are not one-way. In *FSE*, pp. 284–292.

Dowling, W. F., & Gallier, J. H. (1984). Linear-time algorithms for testing the satisfiability of propositional horn formulae. *J. Log. Program.*, *1*(3), 267–284.

Frioux, L. L., Baarir, S., Sopena, J., & Kordon, F. (2017). PaInleSS: A framework for parallel SAT solving. In *SAT 2017*, pp. 233–250.

Froleyks, N., & Biere, A. (2021). Single clause assumption without activation literals to speed-up IC3. In *FMCAD*, pp. 72–76.

Garey, M. R., & Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman.

Gladush, A., Gribanova, I., Kondratiev, V., Pavlenko, A., & Semenov, A. (2022). Measuring the effectiveness of SAT-based guess-and-determine attacks in algebraic cryptanalysis. In *Parallel Computational Technologies*, pp. 143–157.

Gomes, C. P., & Sabharwal, A. (2021). Exploiting runtime variation in complete solvers. In *Handbook of Satisfiability - Second Edition*, Vol. 336 of *Frontiers in Artificial Intelligence and Applications*, pp. 463–480. IOS Press.

Gomes, C. P., & Sellmann, M. (2004). Streamlined constraint reasoning. In *CP*, pp. 274–289.

Gribanova, I., & Semenov, A. (2018). Using automatic generation of relaxation constraints to improve the preimage attack on 39-step MD4. In *MIPRO*, pp. 1174–1179.

Gribanova, I., & Semenov, A. (2020). Constructing a set of weak values for full-round MD4 hash function. In *MIPRO*, pp. 1212–1217.

Gribanova, I., Zaikin, O., Kochemazov, S., Otpuschennikov, I., & Semenov, A. (2017). The study of inversion problems of cryptographic hash functions from MD family using algorithms for solving Boolean satisfiability problem. In *Mathematical and Information Technologies*, pp. 98–113.

Guo, J., Ling, S., Rechberger, C., & Wang, H. (2010). Advanced meet-in-the-middle preimage attacks: First results on full tiger, and improved results on MD4 and SHA-2. In *ASIACRYPT*, pp. 56–75.

Hamadi, Y., Jabbour, S., & Sais, L. (2009). ManySAT: a parallel SAT solver. *J. Satisf. Boolean Model. Comput.*, *6*(4), 245–262.

Heule, M. (2018a). Cube-and-Conquer Tutorial. https://github.com/marijnheule/CnC/.

Heule, M. (2018b). Schur number five. In *AAAI*, pp. 6598–6606.

Heule, M., Kullmann, O., & Biere, A. (2018). Cube-and-conquer for satisfiability. In *Handbook of Parallel Constraint Reasoning*, pp. 31–59. Springer.

Heule, M., Kullmann, O., & Marek, V. W. (2016). Solving and verifying the Boolean Pythagorean triples problem via Cube-and-Conquer. In *SAT*, pp. 228–245.

Heule, M., Kullmann, O., Wieringa, S., & Biere, A. (2011). Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *HVC*, pp. 50–65.

Heule, M. J. H., & van Maaren, H. (2021). Look-ahead based SAT solvers. In *Handbook of Satisfiability - Second Edition*, Vol. 336 of *Frontiers in Artificial Intelligence and Applications*, pp. 183–212. IOS Press.

Homsirikamol, E., Morawiecki, P., Rogawski, M., & Srebrny, M. (2012). Security margin evaluation of SHA-3 contest finalists through SAT-based attacks. In *CISIM*, pp. 56–67.

Hutter, F., Xu, L., Hoos, H. H., & Leyton-Brown, K. (2014). Algorithm runtime prediction: Methods & evaluation. *Artificial Intelligence*, *206*, 79–111.

Hyvärinen, A. E. J., Junttila, T. A., & Niemelä, I. (2010). Partitioning SAT instances for distributed solving. In *LPAR*, pp. 372–386.

Jovanovic, D., & Janicic, P. (2005). Logical analysis of hash functions. In *FroCoS*, pp. 200–215.

Kautz, H. A., Sabharwal, A., & Selman, B. (2021). Incomplete algorithms. In *Handbook of Satisfiability - Second Edition*, Vol. 336 of *Frontiers in Artificial Intelligence and Applications*, pp. 213–232. IOS Press.

Kilby, P., Slaney, J. K., Thiébaux, S., & Walsh, T. (2005). Backbones and backdoors in satisfiability. In *AAAI*, pp. 1368–1373.

Kochemazov, S. (2021). Exploring the limits of problem-specific adaptations of SAT solvers in SAT-based cryptanalysis. In *Parallel Computational Technologies*, pp. 149–163.

Kochemazov, S., & Zaikin, O. (2018). ALIAS: A modular tool for finding backdoors for SAT. In *SAT*, pp. 419–427.

Konev, B., & Lisitsa, A. (2015). Computer-aided proof of Erdős discrepancy properties. *Artif. Intell.*, *224*, 103–118.

Kuwakado, H., & Tanaka, H. (2000). New algorithm for finding preimages in a reduced version of the md4 compression function. *IEICE TRANS, Fundamentals, A*, *83*(1), 97–100.

Lafitte, F. (2018). CryptoSAT: a tool for SAT-based cryptanalysis. *IET Information Security*, *12*(6), 463–474.

Lafitte, F., Nakahara, Jr., J., & Heule, D. V. (2014). Applications of SAT solvers in cryptanalysis: Finding weak keys and preimages. *J. Satisf. Boolean Model. Comput.*, *9*(1), 1–25.

Legendre, F., Dequen, G., & Krajecki, M. (2012). Encoding hash functions as a SAT problem. In *ICTAI*, pp. 916–921.

Leurent, G. (2008). MD4 is not one-way. In *FSE*, pp. 412–428.

Li, Z., Bright, C., & Ganesh, V. (2024). A SAT solver and computer algebra attack on the minimum Kochen-Specker problem (student abstract). In *AAAI*, pp. 23559–23560.

Marques-Silva, J. P., & Sakallah, K. A. (1999). GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, *48*(5), 506–521.

Massacci, F., & Marraro, L. (2000). Logical cryptanalysis as a SAT problem. *J. Autom. Reasoning*, *24*(1/2), 165–203.

Menezes, A., van Oorschot, P. C., & Vanstone, S. A. (1996). *Handbook of Applied Cryptography*. CRC Press.

Merkle, R. C. (1989). A certified digital signature. In *CRYPTO*, pp. 218–238. Springer.

Mironov, I., & Zhang, L. (2006). Applications of SAT solvers to cryptanalysis of hash functions. In *SAT*, pp. 102–115.

Nakamura, K., Hori, K., & Hirose, S. (2021). Algebraic fault analysis of SHA-256 compression function and its application. *Inf.*, *12*(10), 433.

Nejati, S., & Ganesh, V. (2019). CDCL(crypto) SAT solvers for cryptanalysis. In *CASCON*, pp. 311–316.

Nejati, S., Horácek, J., Gebotys, C. H., & Ganesh, V. (2018). Algebraic fault attack on SHA hash functions using programmatic SAT solvers. In *CP*, pp. 737–754.

Nejati, S., Liang, J. H., Gebotys, C. H., Czarnecki, K., & Ganesh, V. (2017). Adaptive restart and CEGAR-based solver for inverting cryptographic hash functions. In *VSTTE*, pp. 120–131.

Nossum, V. (2012). SAT-based preimage attacks on SHA-1. Master's thesis, University of Oslo, Department of Informatics.

Rivest, R. L. (1990). The MD4 message digest algorithm. In *CRYPTO*, pp. 303–311.

Rivest, R. L. (1992). The MD5 message-digest algorithm. *RFC*, *1321*, 1–21.

Rossi, F., van Beek, P., & Walsh, T. (Eds.). (2006). *Handbook of Constraint Programming*, Vol. 2 of *Foundations of Artificial Intelligence*. Elsevier.

Samer, M., & Szeider, S. (2008). Backdoor trees. In *AAAI*, pp. 363–368.

Sasaki, Y., & Aoki, K. (2009). Finding preimages in full MD5 faster than exhaustive search. In *EUROCRYPT*, pp. 134–152.

Semenov, A., Chivilikhin, D., Pavlenko, A., Otpuschennikov, I., Ulyantsev, V., & Ignatiev, A. (2021). Evaluating the hardness of SAT instances using evolutionary optimization algorithms. In *CP*, pp. 47:1–47:18.

Semenov, A., Otpuschennikov, I., Gribanova, I., Zaikin, O., & Kochemazov, S. (2020). Translation of algorithmic descriptions of discrete functions to SAT with applications to cryptanalysis problems. *Log. Methods Comput. Sci.*, *16*(1), 29:1–29:42.

Semenov, A., Zaikin, O., Bespalov, D., & Posypkin, M. (2011). Parallel logical cryptanalysis of the generator A5/1 in BNB-grid system. In *Parallel Computing Technologies*, pp. 473–483.

Semenov, A., Zaikin, O., & Kochemazov, S. (2021). Finding effective SAT partitionings via black-box optimization. In *Black Box Optimization, Machine Learning, and No-Free Lunch Theorems*, pp. 319–355. Springer International Publishing.

Semenov, A., Zaikin, O., Otpuschennikov, I., Kochemazov, S., & Ignatiev, A. (2018). On cryptographic attacks using backdoors for SAT. In *AAAI*, pp. 6641–6648.

Soos, M., Nohl, K., & Castelluccia, C. (2009). Extending SAT solvers to cryptographic problems. In *SAT*, pp. 244–257.

Starnes, D., Yates, D., & Moore, D. (2010). *The Practice of Statistics*. W. H. Freeman.

Stevens, M., Bursztein, E., Karpman, P., Albertini, A., & Markov, Y. (2017). The first collision for full SHA-1. In *CRYPTO*, pp. 570–596.

Tseitin, G. S. (1970). On the complexity of derivation in propositional calculus. In *Studies in constructive mathematics and mathematical logic, part II, Seminars in mathematics*, Vol. 8, pp. 115–125. V.A. Steklov Mathematical Institute, Leningrad.

Vallade, V., Frioux, L. L., Oanea, R., Baarir, S., Sopena, J., Kordon, F., Nejati, S., & Ganesh, V. (2021). New concurrent and distributed Painless solvers: P-MCOMSPS, P-MCOMSPS-COM, P-MCOMSPS-MPI, and P-MCOMSPS-COM-MPI. In *SAT Competition 2021 – Solver and Benchmark Descriptions*, pp. 40–41.

Wang, X., Lai, X., Feng, D., Chen, H., & Yu, X. (2005). Cryptanalysis of the hash functions MD4 and RIPEMD. In *EUROCRYPT*, pp. 1–18.

Wang, X., & Yu, H. (2005). How to break MD5 and other hash functions. In Cramer, R. (Ed.), *EUROCRYPT*, pp. 19–35.

Weaver, S. A., & Heule, M. (2020). Constructing minimal perfect hash functions using SAT technology. In *AAAI*, pp. 1668–1675.

Williams, R., Gomes, C. P., & Selman, B. (2003). Backdoors to typical case complexity. In *IJCAI*, pp. 1173–1178.

Zaikin, O. (2022). Inverting 43-step MD4 via Cube-and-Conquer. In *IJCAI-ECAI*, pp. 1894–1900.

Zaikin, O. (2024). Inverting Cryptographic Hash Functions via Cube-and-Conquer — Results and Source code, Dataset on Zenodo. https://doi.org/10.5281/zenodo.13766981.

Zaikin, O., & Kochemazov, S. (2021). On black-box optimization in divide-and-conquer SAT solving. *Optimization Methods and Software*, *36*(4), 672–696.