# Inverting 43-step MD4 via Cube-and-Conquer

**Oleg Zaikin**
Swansea University
o.s.zaikin@swansea.ac.uk

## Abstract

MD4 is a prominent cryptographic hash function proposed in 1990. The full version consists of 48 steps and produces a hash of size 128 bits given a message of an arbitrary finite size. In 2007, its truncated 39-step version was inverted via reducing to SAT and applying a CDCL solver. Since that time, several attempts have been made but the 40-step version still remains unbroken. In this study, 40-, 41-, 42-, and 43-step versions of MD4 are successfully inverted. The problems are reduced to SAT and solved via the Cube-and-Conquer approach. Two algorithms are proposed for this purpose. The first one generates inversion problems for MD4 by adding special constraints. The second one is aimed at finding a proper threshold for the cubing phase of Cube-and-Conquer. While the first algorithm is focused on inverting MD4 and similar cryptographic hash functions, the second one is not area specific and so is applicable to a variety of classes of hard SAT instances.

## 1 Introduction

A cryptographic hash function maps a message of arbitrary finite size to a hash of a fixed size. Such a computation must be very easy, but at the same time it should be computationally infeasible to invert it, i.e. to find a message given its hash. Cryptographic hash functions are really pervasive in the modern digital world. Examples of their applications include verification of passwords and signatures.

The MD4 hash function was proposed back in 1990 [Rivest, 1990]. Thanks to an elegant yet efficient design it has become one of the most influential cryptographic functions with numerous notable successors, such as MD5, SHA-1, and RIPEMD. MD4 consists of 48 steps. In [Dobbertin, 1998] the Dobbertin's constraints on intermediate states of MD4 registers were proposed, which significantly simplify the inversion. This breakthrough approach made it possible to invert the 32-bit truncated version of MD4.

It is well-known that the resistance of a cryptographic hash function can be studied by reduction to the Boolean satisfiability problem (SAT). SAT is to determine whether a given Boolean formula is satisfiable or not. Recently various scientific and industrial problems have been successfully solved via the SAT approach.

For the first time MD4 was analyzed by the SAT approach in [Jovanovic and Janicic, 2005] to construct benchmarks with adjustable hardness. In [Mironov and Zhang, 2006], collisions for MD4 were generated via Conflict-Driven Clause Learning (CDCL [Marques-Silva and Sakallah, 1999]) solvers. In 2007, SAT encodings of slightly modified Dobbertin's constraints were constructed, and as a result the 39-step truncated MD4 was inverted via a CDCL solver [De *et al.*, 2007].

Despite the found vulnerabilities, MD4 is still used to compute password-derived digests in some operating systems of the Windows family, including the modern ones. One of the reason is that inversion of the full MD4 still remains a computationally infeasible task. Since 2007, several unsuccessful attempts have been made to invert the truncated versions with 40+ steps. This study is aimed at filling this gap by inverting some of the truncated versions via Cube-and-Conquer.

Cube-and-Conquer is a SAT approach for solving extremely hard SAT problems [Heule *et al.*, 2011]. On the cubing phase, a given problem is split into subproblems via a lookahead solver [Heule and van Maaren, 2009], then on the conquer phase they are solved via a CDCL solver. Several hard combinatorial problems have been solved via this approach recently, e.g. the Boolean Pythagorean Triples problem [Heule *et al.*, 2016].

In this paper, two algorithms are proposed. The first one gradually relaxes Dobbertin's constraints until a preimage of MD4 (or a truncated version of MD4) is found. The second one is aimed at finding a good threshold for the cubing phase of Cube-and-Conquer. The latter algorithm is a general one, and can be applied to any hard SAT instance. With the help of these algorithms, the 40-, 41-, 42-, and 43-step versions of MD4 were inverted for two specified hashes.

The paper is organized as follows. Preliminaries on cryptographic hash functions and MD4 are given in Section 2. Section 3 proposes the algorithm for relaxing Dobbertin-like constraints. The considered truncated versions of MD4, as well as their SAT encodings, are described in Section 4. Section 5 proposes the algorithm for adjusting the cubing phase. Experimental results on inverting the truncated versions of MD4 are given in Section 6. Finally, conclusions are drawn.

## 2 Preliminaries

This section gives preliminaries on cryptographic hash functions and discusses the current status of MD4.

### 2.1 Cryptographic Hash Functions

Hereinafter only unkeyed cryptographic hash functions are considered, see [Menezes *et al.*, 1996]. Inputs of cryptographic hash functions are usually called *messages*, while outputs are called *hash values* or just *hashes*.

A cryptographic hash function $h$ ideally must have the following five properties.

1. *Compression*: $h$ maps a message $x$ of arbitrary finite size to a hash $h(x)$ of fixed size.
2. *Ease of computation*: for any given message $x$, $h(x)$ is easy to compute.
3. *Preimage resistance*: for any given hash $y$, it is computationally infeasible to find any of its preimages, i.e. any such message $x'$ that $h(x') = y$.
4. *Second-preimage resistance*: for any given message $x$, it is computationally infeasible to find $x'$ such that $x' \neq x, h(x) = h(x')$.
5. *Collision resistance*: it is computationally infeasible to find any two messages $x$ and $x'$ such that $x \neq x', h(x) = h(x')$.

The first two properties are obligatory, while the remaining three are potential and can be compromised. Usually at the time of publishing a cryptographic hash function all the five properties are valid. However, the subsequent publications may propose methods for abolishing one (or even all) of the three potential properties. These methods are called *preimage attacks*, *second preimage attacks*, and *collision attacks*, respectively. As a rule, the collision resistance is the weakest among the potential ones. If an attack is computationally feasible, then it is called *practical*.

### 2.2 MD4

The Message Digest 4 (MD4) cryptographic hash function was proposed by Ronald Rivest in 1990 [Rivest, 1990]. Given a message of an arbitrary 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 512-bit blocks.

Consider the compression function in more detail. Given a 512-bit message block, 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 constants specified in the standard. Otherwise, registers are initialized with an output produced by the compression function on the previous message block. The message block is divided into sixteen 32-bit words. In each step, one register's value is updated by mixing one word with values of all four registers. As a result, in each 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 hash is produced as a concatenation of $A, B, C, D$.

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 resistant to the second preimage attack [Wang *et al.*, 2005]. In 2008, a theoretical preimage attack on MD4 was proposed [Leurent, 2008]. Despite these vulnerabilities, MD4 still remains preimage resistant in practice. That is why truncated versions (with reduced number of steps) have been actively studied recently. In 1998, Hans Dobbertin showed that by adding additional constraints (further they are called *Dobbertin's constraints*) the 32-step MD4 can be easily inverted [Dobbertin, 1998]. In 2007, a SAT-based implementation of the attack from [Dobbertin, 1998] made it possible to invert the 39-step version [De *et al.*, 2007]. Since 2007, several unsuccessful attempts have been made to invert 40+ versions, see, e.g. [Legendre *et al.*, 2012].

## 3 Inverting MD4 via Dobbertin-like Constraints

This section describes the Dobbertin's constraints and proposes their generalization — Dobbertin-like constraints. Finally it proposes an algorithm for inverting MD4 via Dobbertin-like constraints.

### 3.1 Dobbertin-like Constraints

Hereinafter if a truncated version of MD4 is mentioned, at least 32 (first) steps are assumed. The *Dobbertin's constraints* for MD4 or its truncated version are as follows: the registers $A$, $D$, and $C$ are equal to a constant 32-bit word $K$ in steps 12, 16, 20, 24, steps 13, 17, 21, 25, and steps 14, 18, 22, 26, respectively (numbering from 0) [Dobbertin, 1998]. As a result, for any given hash and randomly chosen $K$ the number of preimages (messages) is significantly reduced, maybe even to 0. On the other hand, if at least one preimage exists, then it is usually much simpler to find it.

Suppose that given a constant $K$ all but one Dobbertin's constraints are applied as usual, while the remaining one might be relaxed or even entirely omitted. By relaxing it is meant that in the corresponding register only $b, 0 \leq b \leq 32$ most significant bits are equal to $b$ most significant bits of $K$, while the remaining $32 - b$ bits in the register may take arbitrary values. Denote these constraints as *Dobbertin-like constraints*. The Dobbertin's constraints is a special case of Dobbertin-like constraints when $b = 32$.

Denote an inversion problem with applied Dobbertin-like constraints as $\texttt{MD4inversion}(y, s, K, p, b)$, where $y$ is a given 128-bit hash, $s$ is the number of MD4 steps, $K$ is a 32-bit constant word used in the Dobbertin's constraints, $p$ is the partially constrained step, $b$ is the number of constant most significant bits in step $p$. Hereinafter $0^{128}$ and $1^{128}$ mean 16 words `0x00000000` and `0xffffffff` respectfully.

**Example 1** ($\texttt{MD4inversion}(1^{128}, 39, \texttt{0x00000000}, 12, 0)$)**.** *This inversion problem was solved in [De* et al.*, 2007]. In this case the Dobbertin's constraint in step 12 is not applied at all while inverting the hash* $1^{128}$ *produced by the 39-step truncated MD4. The remaining 11 Dobbertin's constraints are fully applied.*

**Algorithm 1** Algorithm for inverting MD4 or its truncated version via Dobbertin-like constraints

**Input**: Hash $y$, the number of MD4 steps $s$, constant $K$, step $p$ with partially constrained register, a complete algorithm $\mathcal{A}$
**Output**: Preimages for hash $y$ if they are found, $\emptyset$ otherwise.

1: $preimages \leftarrow \emptyset$
2: $b \leftarrow 32$
3: **while** $b \geq 0$ and $preimages = \emptyset$ **do**
4: $\quad preimages \leftarrow \mathcal{A}(\texttt{MD4inversion}(y, s, K, p, b))$
5: $\quad b \leftarrow b - 1$
6: **return** $preimages$

### 3.2 Inversion Algorithm

Dobbertin-like constraints applied to MD4 or its truncated version can be used for finding preimages according to the following idea. For a given hash and a random $K$ first all 12 Dobbertin's constraints are applied. The inversion problem is solved and, if a preimage is found, nothing else should be done. Otherwise, if it is proven that no preimages exist, then one Dobbertin's constraint is chosen and relaxed by letting 1 less significant bit in the corresponding register take any value. The modified inversion problem is solved. If a solution still does not exist, then the constraint is further relaxed by 1 more bit and so on. The intuition here is that the Dobbertin's constraints lead to a system that is either consistent (with very few solutions) or quite "close" to a consistent one. In the latter case the proposed relaxing may help finding such a consistent system.

Algorithm 1 follows the described idea. In the pseudocode a complete algorithm $\mathcal{A}$ is used, which for a formed inversion problem returns all possible preimages if they exist, and $\emptyset$ otherwise. Note that it is not guaranteed that Algorithm 1 finds preimages for a given hash. If it finishes with no found preimages, then the constraint in step $p$ can be excluded from consideration, while one of the remaining 11 Dobbertin's constraints might be additionally relaxed in a similar way.

Complete algorithms of various types can be used to solve inversion problems formed in Algorithm 1. In particular, wide spectrum of integer programming and constraint programming solvers are potential candidates. In the present study, a complete SAT algorithm is used for this purpose.

## 4 Problems and Their SAT Encodings

This section describes the considered inversion problems and their SAT encodings.

### 4.1 Considered Problems

Similarly to [De *et al.*, 2007], the goal in fact is to find preimages for the following two hashes: $0^{128}$ and $1^{128}$. While in [De *et al.*, 2007] only $K = \texttt{0x00000000}$ was used in the Dobbertin-like constraints, in the present study also $K = \texttt{0xffffffff}$ is tried. Step 12 is chosen for the relaxation (so $p = 12$) since in [De *et al.*, 2007] this step was entirely omitted. Eight truncated versions of MD4, from 40 to 47 steps, as well as the full MD4 are studied. Hence there are 36 inversion problems in total.

Consider the 40-step truncated MD4. Algorithm 1 should be run on four inputs because both $y$ and $K$ have two values. According to the notation from Subsection 3.1, the following four inversion problems are formed in the corresponding first iterations of Algorithm 1:

- $\texttt{MD4inversion}(0^{128}, 40, \texttt{0x00000000}, 12, 32)$;
- $\texttt{MD4inversion}(0^{128}, 40, \texttt{0xffffffff}, 12, 32)$;
- $\texttt{MD4inversion}(1^{128}, 40, \texttt{0x00000000}, 12, 32)$;
- $\texttt{MD4inversion}(1^{128}, 40, \texttt{0xffffffff}, 12, 32)$.

Following all earlier attempts to invert truncated MD4 via SAT (see, e.g. [De *et al.*, 2007; Legendre *et al.*, 2012]), the goal in fact is to invert truncated MD4 compression function (see Subsection 2.2), therefore the padding is omitted. The final incrementing is also omitted since it should be done only after all 48 steps.

### 4.2 SAT Encoding

The SAT encoding of MD4 was taken from [Semenov *et al.*, 2020], where it was constructed via the TRANSALG tool. In the Conjunctive Normal Form (CNF), the first 512 variables correspond to a message, the last 128 variables correspond to a hash, while the remaining auxiliary variables 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].

The 40-step version is encoded by a CNF with 7 025 variables and 70 809 clauses. Then every step adds 186 variables and 2 349 clauses, so as a result a CNF that encodes the full (48-step) MD4 compression function 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 inversion problem for a given 128-bit hash, 128 corresponding one literal clauses are to be added, so all hash variables become assigned. The problem is to find values of the message variables. The Dobbertin's constraints can be added as another 384 one literal clauses (12 clauses for each constraint). As a result a CNF that encodes the inversion of the 40-step MD4 with all 12 Dobbertin's constraints has 7 025 variables and 71 321 clauses, while that for the full 48-step version consists of 8 513 variables and 90 113 clauses.

Since SAT approach is used, the algorithm $\mathcal{A}$ in Algorithm 1 might call a complete SAT solver on the corresponding CNF. In preliminary experiments state-of-the art sequential (non-parallel) CDCL SAT solvers were tried to invert the 40-step truncated version, but even on the first iterations (where all 12 Dobbertin's constraints are added) CNFs turned out to be way too hard for them. That is why it was decided to use Cube-and-Conquer SAT solvers, which are more suitable for extremely hard SAT instances. The next section describes how a given problem can be properly split into simpler subproblems on the cubing phase of Cube-and-Conquer.

# 5 Finding Threshold in Cube-and-Conquer

This section first briefly describes Cube-and-Conquer, and then proposes an algorithm for finding a threshold for the cubing phase of Cube-and-Conquer.

## 5.1 Cube-and-Conquer

*Cube-and-conquer* [Heule *et al.*, 2011] is a SAT solving approach that combines lookahead [Heule and van Maaren, 2009] with CDCL [Marques-Silva and Sakallah, 1999]. On the *cubing phase* of Cube-and-Conquer, a lookahead solver splits a given CNF into *cubes*. For each cube by joining it with the CNF a subproblem is formed. On the *conquer phase*, the subproblems are solved via a CDCL solver. Since cubes can be processed independently, the conquer phase can be easily parallelized.

Originally, 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 binary search 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 *et al.*, 2010]. According to the second one, it happens 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 since it usually shows better results on hard instances.

## 5.2 Algorithm for Finding Cutoff Threshold

It is crucial to properly choose a cutoff threshold $n$ in the cubing phase. If it is too high, then very few extremely hard (for a CDCL solver) cubes will be produced; if it is too low, then there will be too huge number of cubes, and also the cubing phase will be extremely time consuming. To find a good threshold, first it is needed to properly choose promising values of $n$. On the one hand, the number of refuted leaves should be quite significant since it indicates that subformulas have become really simpler compared to the original CNF. On the other hand, the total number of cubes should not be too large.

When promising values of the threshold are chosen, then it is needed to estimate for them the hardness of the conquer phase. Such an estimation can be calculated by sampling: a fixed number of cubes is randomly chosen among those produced by the lookahead solver. If all cubes from the sample are solved by the CDCL solver in a reasonable time, then an estimated total solving time for all cubes may be easily calculated. Algorithm 2 follows the proposed idea.

**Algorithm 2** Finding a cutoff threshold with minimal estimated runtime of the conquer phase

**Input**: CNF $\mathcal{F}$, lookahead solver `lookahead`, CDCL solver `cdcl`, decreasing step $k$, maximal number of cubes $max\_cubes$, minimal number of refuted leaves $min\_refuted$, sample size $N$, CDCL solver time limit $max\_t$, the number of CPU cores $cpu\_cores$.
**Output**: A threshold $n_{best}$ with estimation $est_{best}$ and cubes $cubes_{best}$.

$n \leftarrow \texttt{varnum}(\mathcal{F}) - k$
$\langle n_{best}, est_{best}, cubes_{best}\rangle \leftarrow \langle n, +\infty, \emptyset\rangle$
$stack \leftarrow$ *empty stack*
**while** $n > 0$ **do**
  $\langle cubes, refuted\rangle \leftarrow \texttt{lookahead}(\mathcal{F}, n)$
  **if** $\texttt{size}(cubes) > max\_cubes$ **then**
    **break**
  **if** $refuted \geq min\_refuted$ **then**
    $stack.\texttt{push}(\langle n, cubes\rangle)$
  $n \leftarrow n - k$
**while** $stack$ is not empty **do**
  $\langle n, cubes\rangle \leftarrow stack.\texttt{pop}()$
  $sample \leftarrow \texttt{random\_sample}(cubes, N)$
  $runtimes \leftarrow \texttt{solve}(\texttt{cdcl}, \mathcal{F}, sample, max\_t)$
  **if** any time from $runtimes > max\_t$ **then**
    **break**
  $upd\_cubes \leftarrow cubes \setminus sample$
  $est \leftarrow \texttt{average}(runtimes) \cdot \texttt{size}(upd\_cubes)$
  $est \leftarrow est/cpu\_cores$
  **if** $est < est_{best}$ **then**
    $\langle n_{best}, est_{best}, cubes_{best}\rangle \leftarrow \langle n, est, upd\_cubes\rangle$
**return** $\langle n_{best}, est_{best}, cubes_{best}\rangle$

## 5.3 Algorithm Features

The proposed algorithm has several features. First, it does not estimate the runtime of the cubing phase because it is assumed that this is negligible compared to the conquer phase. In all further experiments it is really so. Second, a stack is used for collecting promising thresholds on the first stage of the algorithm in order to start the second stage with solving the simplest subproblems (with lowest $n$). It allows obtaining some estimation quite fast and then improve it. Third, if on the second stage for a current threshold a CDCL solver does not solve any subproblem from the random sample in time limit, the algorithm stops. This is done because in this case it is impossible to calculate a meaningful estimation for the threshold. Another reason is that subproblems from the next thresholds will likely be even harder. Forth, it is possible that satisfying assignments are found when solving subproblems from random samples. Indeed, cubes which imply satisfying assignments might be chosen to samples. In Section 6, such cases are discussed. For the sake of simplicity this feature is not reflected in the pseudocode. Note that the algorithm does not stop upon finding a satisfying assignment.

Cubes produced by Algorithm 2 are processed on the conquer phase. Using these cubes and the original CNF, subproblems are created and solved by the same CDCL solver that was used on the cubing phase. In the present study the goal is to find all solutions of a considered problem. That is why, given a subformula, the CDCL solver finds all its satisfying assignments. In opposite to the cubing phase, here the runtime of the CDCL solver is not limited.

The proposed algorithm is a general one and may be applied to any hard SAT instances. The next section describes its implementation, as well as its usage for solving some of the considered inversion problems.

# 6 Inverting 40-, 41-, 42-, and 42-step MD4

All truncated MD4 inversion problems described in Section 4 are studied experimentally by Algorithm 1. In this case a complete algorithm $\mathcal{A}$ is Algorithm 2 followed by the conquer phase, discussed in Subsection 5.3. This section describes the experimental setup and obtained results.

## 6.1 Experimental Setup

Algorithms 1 and 2, as well as the conquer phase of Cube-and-Conquer were implemented in C++ and Python[1]. All experiments were held on a personal computer equipped with the 12-core CPU AMD 3900X and 48 Gb of RAM. The implementation is multithreaded, so all 12 CPU cores were employed in all runs for both cubing and conquer phases. In case of Algorithm 2, values of $n$ and then subproblems from samples are processed in parallel.

Parameters of Algorithm 1 were discussed in Subsection 4.1. As for Algorithm 2, the following parameters were used: the MARCH_CU lookahead solver [Heule *et al.*, 2011]; the KISSAT CDCL solver of version sc2021 [Biere *et al.*, 2021]; $max\_t = 5\ 000$ seconds; $k = 10$; $max\_cubes = 1\ 000\ 000$; $min\_refuted = 500$; $N = 1\ 000$; $cpu\_cores = 12$.

MARCH_CU was chosen because it was recently successfully applied to several hard problems (see, e.g. [Heule *et al.*, 2016; Heule, 2018]), while KISSAT and its modifications won SAT Competitions 2020 and 2021. The time limit of 5 000 seconds is a standard cutoff in SAT Competitions, so modern CDCL solvers are designed to show all their power within this time. The decreasing step $k$ was chosen in preliminary experiments. If it is equal to 1, then a better threshold usually can be found, but at the same time Algorithm 2 requires too much time. On the other hand, if $k$ is quite large, e.g. 100, then usually almost all most promising thresholds are just skipped. On the considered CNFs, MARCH_CU reaches 1 000 000 cubes in about 15 minutes, so that value of $max\_cubes$ looks reasonable. If $min\_refuted$ is less then 500, then subproblems are too hard because they are not simplified enough by lookahead. At the same time, higher value of this parameter did not allowed collecting enough amount of promising thresholds. First $N = 100$ was tried, but it led to too optimistic estimations, while $N = 1\ 000$ provided accurate ones.

It should be noted that in both phases of Cube-and-Conquer subproblems were solved by KISSAT in the non-incremental mode, i.e. it solved them independently from each other.

## 6.2 Experiments

First, inversion of 40-step MD4 was studied. Two parameters were varied in this case: The first one is the value of the Dobbertin's constant $K$ (see Subsection 4.1): `0x00000000` and `0xffffffff`. The second one is the CNF minimisation algorithm. In this study, the following algorithms were tried: (1) preprocessing by Unit Propagation (UP) [Dowling and Gallier, 1984]; (2) preprocessing by SATELITE [Eén and Biere, 2005]; (3) inprocessing by CADICAL of version 1.4.1 [Biere *et al.*, 2021]. SATELITE was used in its default configuration, while in case of CADICAL, the limit of 1 million conflicts was used that corresponds to about 80 seconds of runtime.

[1]The implementation, as well as all the constructed CNFs are available online at https://github.com/olegzaikin/MD4-CnC.

| s | $y$ | $b$ | m | $n_{best}$ | $est_{best}$ | real time | sol |
|---|---|---|---|---|---|---|---|
| 40 | 0 | 32 | S | 3310 | 15h 25m | 17h 48m | 0 |
| | | 31 | S | 3330 | 23h 59m | 38h 33m | 0 |
| | | 30 | C | 2450 | 61h 18m | 81h 10m | 1 |
| | 1 | 32 | C | 2490 | 41h 58m | 53h 45m | 0 |
| | | 31 | C | 2500 | 93h 18m | 109h 39m | 0 |
| | | 30 | C | 2510 | 213h 35m | 244h 14m | 1 |
| 41 | 0 | 32 | S | 3370 | 9h 16m | 9h 40m | 0 |
| | | 31 | C | 2460 | 18h 39m | 21h 3m | 1 |
| | 1 | 32 | U | 3410 | 36h 22m | 38h 49m | 3 |
| 42 | 0 | 32 | U | 3400 | 20h 41m | 21h 54m | 3 |
| | 1 | 32 | C | 2580 | 27h 19m | 30h 51m | 0 |
| | | 31 | C | 2580 | 50h 39m | 67h 52m | 1 |
| 43 | 0 | 32 | S | 3380 | 14h 2m | 16h 7m | 2 |
| | 1 | 32 | U | 3410 | 37h 52m | 39h 3m | 1 |

Table 1: Estimated and real runtimes (on 12 CPU cores) for all solved inversion problems with $K =$ `0xffffffff`. In the header, $s$ stands for the number of MD4 steps, $y$ for hash, $b$ for the number of constant most significant bits in step 12, `m` for the best minimisation algorithm, `sol` for the number of solutions. For the $y$ column, values 0 and 1 mean $0^{128}$ and $1^{128}$ resp. In the `m` column,U stands for UP, S for SATELITE, and C for CADICAL.

A motivation behind varying the second parameter is as follows. First, it is crucial to minimise a CNF before launching a lookahead solver on it. Second, in preliminary experiments it was found out that the minimisation algorithm's type might significantly influence the effectiveness of Cube-and-Conquer on the considered problems.

Having two hashes to invert, in total 12 CNFs were constructed for the 40-step MD4. On each of them Algorithm 1 was run. It turned out, that on the first iteration (with $b = 32$), Algorithm 2 could not find any estimations for all 6 CNFs with $K =$ `0x00000000`. The reason was because in all these cases the CDCL solver was interrupted due to the time limit even for the simplest (lowest) values of the threshold $n$. On the other hand, for $K =$ `0xffffffff` much more positive results were achieved. Namely, for the hash $1^{128}$, on the CADICAL-based CNF estimations were successfully calculated for several thresholds, while on both SATELITE-based and UP-based CNFs no estimations were gained. For the hash $0^{128}$, estimations were calculated successfully for all 3 CNFs, but the best estimation was on the SATELITE-based CNF. Due to these results, $K =$ `0x00000000` was not tried anymore, so $K =$ `0xffffffff` was used in all other inversion problems.

Using the found thresholds, the conquer phase was run on two CNFs: the CADICAL-based one for $1^{128}$ and SATELITE-based for $0^{128}$. All subproblems were solved successfully, but no satisfying assignments were found for them. The found thresholds, estimations, and the real runtime are presented in Table 1. Runtimes of Algorithm 2 are not presented there, but on average in all experiments it took about 2 hours on a CNF.

The next iteration of Algorithm 1 (with $b = 31$) was executed for both hashes. Six CNFs were constructed, and Algorithm 2 was run on them. On the best thresholds

| s | $y$ | preimages |
|---|---|---|
| 40 | 0 | 0xe57d8668 0xa57d8668 0xa57d8668 0xbc8c857b |
| | | 0xa57d8668 0xa57d8668 0xa57d8668 0xcb0a1178 |
| | | 0xa57d8668 0xa57d8668 0xa57d8668 0x307bc4e7 |
| | | 0xad02e703 0xe1516b23 0x981c2a75 0xc08ea9f7 |
| | 1 | 0xe57d8668 0xa57d8668 0xa57d8668 0x1d236482 |
| | | 0xa57d8668 0xa57d8668 0xa57d8668 0x97a13204 |
| | | 0xa57d8668 0xa57d8668 0xa57d8668 0x991ede3 |
| | | 0x301e2ac3 0x5bed2a3d 0xe167a833 0x890d22f0 |
| 43 | 0 | 0xa57d8668 0xa57d8668 0xa57d8668 0xf48a97a3 |
| | | 0xa57d8668 0xa57d8668 0xa57d8668 0xd330e8ed |
| | | 0xa57d8668 0xa57d8668 0xa57d8668 0x37c9ca21 |
| | | 0xe1df551f 0x7f49d66a 0x135a1c93 0x9e744bdb |
| | | 0xa57d8668 0xa57d8668 0xa57d8668 0xb289afa0 |
| | | 0xa57d8668 0xa57d8668 0xa57d8668 0xaf2c850e |
| | | 0xa57d8668 0xa57d8668 0xa57d8668 0x19c5ce09 |
| | | 0xcae6b29e 0xb2595b20 0xab3a433d 0xf6cdee42 |
| | 1 | 0xa57d8668 0xa57d8668 0xa57d8668 0x82ef987a |
| | | 0xa57d8668 0xa57d8668 0xa57d8668 0xe18fbc3b |
| | | 0xa57d8668 0xa57d8668 0xa57d8668 0x558f3513 |
| | | 0xbf09004d 0x8fb490dd 0x502eca9 0xbd0e1a80 |

Table 2: Found preimages for 40- and 43-step MD4.

the conquer phase again did not find any solution, but this time it was more time consuming. Finally, preimages for both hashes were found on the third iteration ($b = 30$). Recall that on the conquer phase all solutions are found, so all possible preimages for the inversion problems $\texttt{MD4inversion}(0^{128}, 40, \texttt{0xffffffff}, 12, 30)$ and $\texttt{MD4inversion}(1^{128}, 40, \texttt{0xffffffff}, 12, 30)$ were enumerated (see Subsection 3.1).

On the next stage, Algorithm 2 was run on all remaining inversion problems with $b = 32$ (recall that only $K = \texttt{0xffffffff}$ was used). For 44+ steps, no estimations were obtained. It seems that these problems are way too hard for this approach. On the other hand, for 41, 42, and 43-steps estimations were successfully calculated and they were quite reasonable. Moreover, during the search for estimations, solutions were found quite fast for two problems: 41 step and the hash $1^{128}$; 42 steps and the hash $0^{128}$. Since the goal was to find all solutions, the conquer phase was anyway run on the best thresholds. The results are presented in Table 1. It can be seen that these steps turned out to be easier to solve compared to 40 step. This phenomenon is discussed in the next subsection. For the sake of compactness, preimages only for 40- and 43-step MD4 are presented in Table 2.

### 6.3 Discussion

The correctness of the found preimages was verified by the reference implementation mentioned in [Rivest, 1990]. This verification might be easily reproduced since MD4 is hard to invert, but the direct computation is extremely fast. First, the additional actions (padding, incrementing, see Subsection 4.1), as well as the corresponding amount of final steps should be deleted. Then the preimages from Table 2 should be given as inputs to the compressing function.

The obtained estimations can be treated as accurate ones since they are close to the real solving time (see Table 1). On average the real time is 18 % higher than the estimated time, while in the worst case (for $s = 40, b = 31, y = 0^{128}$) the real time is 61 % higher.

According to the results, on the simplest problems (with $b = 32$) usually either SATELITE or UP gives the best estimations, while on the hardest ones CADICAL is the winner. Note, that if only one minimisation algorithm had been chosen (any of three), then some of the problems would have remained unsolved. In fact, either SATELITE or UP could have been excluded, but not CADICAL. All in all, different minimisation algorithms complement each other on the considered problems.

It might seem counterintuitive that the hardest inversion problem is for 40 steps instead of 43. The same behavior was seen when inverting 32-39 steps via a CDCL solver: the 39-step truncated version is not the hardest problem among them. In short, the reason is that in MD4 with the Dobbertin-like constraints the hardness of inversion is increased not in all steps. Adding the 40th step gives a leap in hardness, and the next such leap happens when adding the 44th step.

As for 44+ steps, more meticulous and time consuming cubing phase might allow one to find estimations of the hardness via Algorithm 2. For this purpose both $min\_refuted$ and $max\_cubes$ should be significantly increased. It also makes sense to try another branching strategies on the cubing phase (see [Kullmann, 2009]).

## 7 Related Work

Truncated versions (up to 39 steps) of MD4 were inverted via CDCL solvers in the following papers: [De *et al.*, 2007; Legendre *et al.*, 2012; Lafitte *et al.*, 2014; Gribanova *et al.*, 2017; Gribanova and Semenov, 2018]. CDCL solvers were also applied to invert truncated versions of the MD5, SHA-0, SHA-1, SHA-256, and SHA-3 cryptographic hash functions [Nossum, 2012; Legendre *et al.*, 2012; Homsirikamol *et al.*, 2012; Nejati *et al.*, 2017].

The following hard combinatorial problems have been solved via Cube-and-Conquer: the Erdős discrepancy problem [Konev and Lisitsa, 2015]; the Boolean Pythagorean Triples problem [Heule *et al.*, 2016]; Schur number five [Heule, 2018]; Lam's problem [Bright *et al.*, 2021].

## 8 Conclusion

This paper proposed two algorithms. The first one is aimed at relaxing the Dobbertin's constraints for MD4 until the inversion problem is solved. The second algorithm finds a threshold for the cubing phase of Cube-and-Conquer with the best runtime estimation. The main result is a successful inversion of the 40-, 41, 42, and 43-step versions of MD4. In other words, a practical SAT-based preimage attack on these truncated versions of MD4 was proposed.

## Acknowledgments

The author thanks anonymous reviewers for valuable comments. The author is grateful to Stepan Kochemazov, Oliver Kullmann, and Alexander Semenov for fruitful discussions. This research was supported by EPSRC grant EP/S015523/1.

## References


[Biere *et al.*, 2021] Armin Biere, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba entering the SAT Competition 2021. In *SAT Competition 2021 – Solver and Benchmark Descriptions*, pages 10–13. University of Helsinki, 2021.

[Bright *et al.*, 2021] Curtis Bright, Kevin K. H. Cheung, Brett Stevens, Ilias S. Kotsireas, and Vijay Ganesh. A SAT-based resolution of Lam's problem. In *AAAI*, pages 3669–3676, 2021.

[De *et al.*, 2007] Debapratim De, Abishek Kumarasubramanian, and Ramarathnam Venkatesan. Inversion attacks on secure hash functions using SAT solvers. In *SAT*, pages 377–382, 2007.

[Dobbertin, 1996] Hans Dobbertin. Cryptanalysis of MD4. In *FSE*, pages 53–69, 1996.

[Dobbertin, 1998] Hans Dobbertin. The first two rounds of MD4 are not one-way. In *FSE*, pages 284–292, 1998.

[Dowling and Gallier, 1984] William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *J. Log. Program.*, 1(3):267–284, 1984.

[Eén and Biere, 2005] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *SAT*, pages 61–75. Springer, 2005.

[Gribanova and Semenov, 2018] Irina Gribanova and Alexander A. Semenov. Using automatic generation of relaxation constraints to improve the preimage attack on 39-step MD4. In *MIPRO*, pages 1174–1179, 2018.

[Gribanova *et al.*, 2017] Irina Gribanova, Oleg Zaikin, Stepan Kochemazov, Ilya Otpuschennikov, and Alexander Semenov. The study of inversion problems of cryptographic hash functions from MD family using algorithms for solving Boolean satisfiability problem. In *MIT*, pages 98–113, 2017.

[Heule and van Maaren, 2009] Marijn Heule and Hans van Maaren. Look-ahead based SAT solvers. In *Handbook of Satisfiability*, pages 155–184. IOS Press, 2009.

[Heule *et al.*, 2011] Marijn Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *HVC*, pages 50–65, 2011.

[Heule *et al.*, 2016] Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the Boolean Pythagorean triples problem via Cube-and-Conquer. In *SAT*, pages 228–245, 2016.

[Heule, 2018] Marijn J. H. Heule. Schur number five. In *AAAI*, pages 6598–6606, 2018.

[Homsirikamol *et al.*, 2012] Ekawat Homsirikamol, Pawel Morawiecki, Marcin Rogawski, and Marian Srebrny. Security margin evaluation of SHA-3 contest finalists through SAT-based attacks. In *CISIM*, pages 56–67, 2012.

[Hyvärinen *et al.*, 2010] Antti Eero Johannes Hyvärinen, Tommi A. Junttila, and Ilkka Niemelä. Partitioning SAT instances for distributed solving. In *LPAR*, pages 372–386, 2010.

[Jovanovic and Janicic, 2005] Dejan Jovanovic and Predrag Janicic. Logical analysis of hash functions. In *FroCoS*, pages 200–215, 2005.

[Konev and Lisitsa, 2015] Boris Konev and Alexei Lisitsa. Computer-aided proof of Erdős discrepancy properties. *Artif. Intell.*, 224:103–118, 2015.

[Kullmann, 2009] Oliver Kullmann. Fundaments of branching heuristics. In *Handbook of Satisfiability*, pages 205–244. IOS Press, 2009.

[Lafitte *et al.*, 2014] Frédéric Lafitte, Jorge Nakahara Jr., and Dirk Van Heule. Applications of SAT solvers in cryptanalysis: Finding weak keys and preimages. *J. Satisf. Boolean Model. Comput.*, 9(1):1–25, 2014.

[Legendre *et al.*, 2012] Florian Legendre, Gilles Dequen, and Michaël Krajecki. Encoding hash functions as a SAT problem. In *ICTAI*, pages 916–921, 2012.

[Leurent, 2008] Gaëtan Leurent. MD4 is not one-way. In *FSE*, pages 412–428, 2008.

[Marques-Silva and Sakallah, 1999] João P. Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.

[Menezes *et al.*, 1996] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[Mironov and Zhang, 2006] Ilya Mironov and Lintao Zhang. Applications of SAT solvers to cryptanalysis of hash functions. In *SAT*, pages 102–115, 2006.

[Nejati *et al.*, 2017] Saeed Nejati, Jia Hui Liang, Catherine H. Gebotys, Krzysztof Czarnecki, and Vijay Ganesh. Adaptive restart and CEGAR-based solver for inverting cryptographic hash functions. In *VSTTE*, pages 120–131, 2017.

[Nossum, 2012] Vegard Nossum. SAT-based preimage attacks on SHA-1. Master's thesis, University of Oslo, Department of Informatics, 2012.

[Rivest, 1990] Ronald L. Rivest. The MD4 message digest algorithm. In *CRYPTO*, pages 303–311, 1990.

[Semenov *et al.*, 2020] Alexander A. Semenov, Ilya V. Otpuschennikov, Irina Gribanova, Oleg Zaikin, and Stepan Kochemazov. Translation of algorithmic descriptions of discrete functions to SAT with applications to cryptanalysis problems. *Log. Methods Comput. Sci.*, 16(1), 2020.

[Tseitin, 1970] Grigori S Tseitin. On the complexity of derivation in propositional calculus. *Studies in constructive mathematics and mathematical logic, part II, Seminars in mathematics*, pages 115–125, 1970.

[Wang *et al.*, 2005] Xiaoyun Wang, Xuejia Lai, Dengguo Feng, Hui Chen, and Xiuyuan Yu. Cryptanalysis of the hash functions MD4 and RIPEMD. In *EUROCRYPT*, pages 1–18, 2005.