EUROPEAN JOURNAL OF EMERGING ARTIFICIAL INTELLIGENCE

LEVERAGING CUBE-AND-CONQUER FOR CRYPTOGRAPHIC HASH FUNCTION PREIMAGE DISCOVERY: A SAT-BASED CRYPTANALYSIS PERSPECTIVE

Dr. Sophia Chen
School of Computing, National University of Singapore, Singapore

Dr. Marcus J. Rivera
Department of Engineering, Imperial College London, United Kingdom

V0LUME01 ISSUE01 (2024)
Published Date: 18 December 2024 // Page no.: - 54-70

ABSTRACT

Cryptographic hash functions are cornerstones of digital security, inherently designed to resist preimage attacks—the computational challenge of finding an input that generates a specific output hash. Despite this design principle, certain hash functions, particularly their reduced-round versions or older standards, can exhibit vulnerabilities that allow for practical preimage discovery. This comprehensive article delves into advanced methodologies for conducting preimage attacks on cryptographic hash functions, with a particular emphasis on the sophisticated integration of Satisfiability (SAT) solvers and the "Cube-and-Conquer" (CnC) paradigm. We meticulously explore the intricate process of encoding hash function inversion problems into Boolean formulas, elucidating the advantages conferred by parallel and distributed SAT solving environments. A core focus is placed on how the divide-and-conquer strategy, synergistically enhanced by look-ahead heuristics and strategic backdoor detection, can dramatically augment the efficiency and feasibility of such cryptanalytic endeavors. Empirical results pertaining to the inversion of various step-reduced MD4 and MD5 versions are critically examined, highlighting the practical implications of these findings for assessing the real-world security margins of cryptographic primitives.

Keywords: Cryptographic hash functions, Preimage attacks, Cube-and-Conquer, SAT solvers, MD4, MD5, Cryptanalysis, Boolean Satisfiability, Parallel SAT solving, Dobbertin's constraints.

INTRODUCTION

In the contemporary digital landscape, cryptographic hash functions serve as indispensable building blocks for a myriad of security applications, ranging from ensuring data integrity and securing passwords to facilitating digital signatures [61, 69, 62]. These functions are meticulously engineered to transform an arbitrary-length input message into a fixed-size output, known as a hash value or digest. A fundamental security attribute expected of any robust cryptographic hash function is its "one-way" property, formally termed preimage resistance [20, 73]. This property dictates that it should be computationally infeasible, given a hash output h, to determine any message M such that H(M)=h. While the theoretical robustness of modern, full-round hash functions against preimage attacks remains largely unchallenged, historical or reduced-round variants have frequently demonstrated susceptibilities that can be exploited in practice [3, 24, 22, 57, 84, 85].

The challenge of inverting cryptographic hash functions can be adeptly reformulated as an instance of the Boolean Satisfiability (SAT) problem [18, 47, 56, 75]. At its essence, the SAT problem involves determining whether there exists a truth assignment for the variables of a given propositional Boolean formula that renders the formula true. When applied to cryptanalysis, a hash function H and a target hash value h are translated into a Boolean formula ΦH(h). The satisfiability of ΦH(h) directly corresponds to the existence of a preimage M that produces the hash h. Consequently, uncovering a satisfying assignment to ΦH(h) provides a concrete preimage [18, 56, 22]. This systematic approach is often referred to as logical cryptanalysis, a specialized facet of algebraic cryptanalysis [7].

However, the sheer scale and complexity of the SAT instances generated from cryptographic problems often push the limits of computational feasibility. The SAT problem itself is classified as NP-complete, signifying that no known polynomial-time algorithm exists for its general solution [18, 29]. Nevertheless, over the past two and a half decades, remarkable advancements in SAT solver technology, particularly those employing the Conflict-Driven Clause Learning (CDCL) algorithm, have revolutionized their ability to tackle increasingly formidable problems [11, 59, 10, 40, 15]. To further extend their reach into highly complex cryptanalytic tasks, researchers have innovated with parallel and distributed

SAT solving methodologies [6, 12, 27, 37, 83], alongside highly specialized techniques such as "Cube-and-Conquer" (CnC) [42, 40].

This article presents a detailed examination of the "Cube-and-Conquer" (CnC) paradigm. CnC embodies a powerful divide-and-conquer strategy specifically tailored for SAT solving [42]. Its fundamental operation involves segmenting an intractable SAT problem into a multitude of smaller, more manageable subproblems, termed "cubes," which can subsequently be solved independently, often in parallel. This inherent parallelizability makes CnC exceptionally well-suited for problems possessing a discernible structure or those for which effective splitting variables can be identified. The application of CnC to the inversion of cryptographic hash functions, especially for the elusive preimage problem, opens a promising frontier for scrutinizing the practical limits of their one-way properties [88].

The subsequent sections of this article are structured as follows: Section 2 provides a comprehensive overview of the theoretical foundations, including detailed discussions on Boolean satisfiability, the intricacies of Cube-and-Conquer, the properties of cryptographic hash functions, and a deep dive into MD4, MD5, and Dobbertin's constraints. Section 3 outlines the proposed methodology, encompassing Dobbertin-like constraints and the iterative inversion algorithms employed. Section 4 elaborates on the experimental setup, including the chosen hashes, MD4 and MD5 problem definitions, SAT encoding techniques, and simplification strategies. Section 5 presents the empirical results and analysis for MD4 inversion. Section 6 details the results for unconstrained MD5 inversion. Section 7 places this work within the broader context of related research. Finally, Section 8 provides a conclusive summary of the findings and outlines promising avenues for future research.

2. Theoretical Foundations and Background

This section lays the groundwork by providing essential preliminaries on Boolean Satisfiability (SAT), the Cube-and-Conquer (CnC) approach, fundamental cryptographic hash function properties, and in-depth descriptions of MD4 and MD5, including the crucial Dobbertin's constraints.

2.1 Boolean Satisfiability Problem (SAT)

The Boolean Satisfiability Problem (SAT) is a foundational problem in theoretical computer science and a central challenge in artificial intelligence, logic, and automated reasoning [11]. At its core, SAT asks whether there exists a truth assignment for a set of Boolean variables that makes a given propositional Boolean formula evaluate to true [11, 48]. A formula is deemed satisfiable if such an assignment exists; otherwise, it is unsatisfiable. Historically, SAT was the first problem proven to be NP-complete, implying that finding a general, efficient (polynomial-time) algorithm for all instances is highly unlikely [18, 29].

A propositional Boolean formula is typically expressed in Conjunctive Normal Form (CNF). A CNF formula is a conjunction (AND) of clauses, where each clause is a disjunction (OR) of literals. A literal is either a Boolean variable (e.g., x) or its negation (e.g., ¬x). For example, $(x1 \vee \neg x2 \vee x3) \wedge (\neg x1 \vee x2)$ is a CNF formula with two clauses.

The Davis-Putnam-Logemann-Loveland (DPLL) Algorithm: The DPLL algorithm [21] is a complete, backtracking-based algorithm for solving SAT problems. It operates by building a decision tree, where internal nodes correspond to variable assignments (decisions), and edges represent the assigned values (true or false). The key components of DPLL include:

- Decision: Selecting an unassigned variable and assigning it a truth value.
- Unit Propagation (UP): If a clause contains only one unassigned literal and all other literals in that clause are false, then the unassigned literal must be assigned true to satisfy the clause. UP is applied iteratively, propagating assignments through the formula [26].
- Conflict Detection: If UP leads to an empty clause (a clause where all literals are false), a conflict is detected, indicating that the current set of assignments is inconsistent.
- Backtracking: Upon a conflict, the algorithm backtracks to a previous decision point, undoing assignments and exploring alternative paths in the search tree. Traditionally, DPLL uses chronological backtracking.

Look-ahead Algorithms: Look-ahead SAT solvers enhance the basic DPLL framework by incorporating sophisticated heuristics for variable selection [43]. Before making a decision, a look-ahead solver explores the immediate consequences of assigning true and false to each unassigned variable by performing limited Unit Propagation. A "reduction measure" is calculated for each variable, indicating how much the formula is simplified by its assignment. The variable that yields the highest reduction is chosen as the next decision variable. Key improvements include:

- Failed Literal Elimination: If assigning a literal (e.g., x) leads to a conflict during UP, then that literal is a "failed literal," and its negation (e.g., ¬x) must be true. If both literals of a variable are failed, the formula is unsatisfiable. This rule helps prune the search space.
- By proactively exploring the impact of decisions, look-ahead solvers can make more informed choices and simplify the formula effectively, particularly on structured instances [43].

Conflict-Driven Clause Learning (CDCL): Modern state-of-the-art SAT solvers are predominantly based on the CDCL algorithm, which significantly improves upon DPLL [59]. When a conflict occurs in CDCL, instead of simple chronological backtracking, the solver performs:

- Conflict Analysis: It analyzes the conflict to

understand its root causes, identifying a "reason" in the form of a set of assignments that led to the conflict.

- Clause Learning: A new clause, called a "conflict clause," is derived from the conflict analysis. This clause represents the learned reason for the conflict and is added to the formula. Conflict clauses prevent the solver from repeatedly exploring the same conflicting partial assignments.

- Non-chronological Backtracking: The solver backtracks not necessarily to the most recent decision, but to the highest decision level relevant to the learned conflict clause. This can cut off large portions of the search space.

- CDCL solvers are highly efficient on non-random instances and are the backbone of most successful SAT applications [11, 48]. Recent advancements include integrating local search techniques with CDCL, creating complete solvers that benefit from local search's exploratory power [15].

SAT-based Cryptanalysis: The application of SAT solvers to cryptanalytic problems is known as SAT-based cryptanalysis or logical cryptanalysis [19, 60, 55]. This is a specific type of algebraic cryptanalysis, where cryptographic functions are represented as systems of equations (in this case, Boolean equations) that are then solved by SAT solvers [7]. Over the past two decades, SAT-based cryptanalysis has achieved remarkable success in analyzing various cryptographic primitives, including stream ciphers, block ciphers, and cryptographic hash functions [7]. Problems from diverse fields like model checking, planning, and bioinformatics can also be efficiently reduced to SAT [11].

2.2 Cube-and-Conquer (CnC) in Detail

While CDCL solvers are powerful, certain extremely hard SAT instances can still overwhelm sequential solvers. In such scenarios, parallel SAT solving becomes essential [6, 5, 12]. There are two primary approaches to parallel SAT solving using complete algorithms:

- Portfolio Approach: Multiple distinct SAT solvers (or different configurations of the same solver) are run concurrently on the identical problem instance. The first solver to find a solution or prove unsatisfiability determines the overall result [37].

- Divide-and-Conquer Approach: The original problem is decomposed into a family of simpler, independent subproblems, which are then distributed among sequential solvers for parallel processing [12].

Cube-and-Conquer (CnC) is a sophisticated divide-and-conquer SAT solving methodology that synergistically combines the strengths of look-ahead and CDCL solvers [42, 40]. CnC partitions a given SAT formula into multiple independent subproblems, called "cubes," which can be solved in parallel.

Cubing Phase: In this initial phase, a modified look-ahead solver systematically splits the original SAT formula into numerous subproblems. The process involves identifying a set of "cube variables." For each possible truth assignment to these cube variables, a subproblem (a "cube") is generated by adding the assignments as unit clauses to the original formula and performing Unit Propagation (UP). The objective is to generate subproblems that are significantly simpler than the original, yet still capture its satisfiability. A crucial aspect of the cubing phase is deciding when to "cut off" a branch in the look-ahead search tree and declare it a cube. Two common cutoff heuristics are:

- Decision Depth: A branch is cut off after a predefined number of decisions are made along that path [46].

- Variable Threshold: A branch becomes a cube when the number of unassigned variables in the corresponding subproblem falls below a specified threshold [42]. The latter heuristic is generally preferred for hard instances as it better reflects the problem's complexity [40]. The choice of cube variables and cutoff thresholds is critical, as it directly impacts the number and hardness of the generated subproblems.

Conquer Phase: Once the cubing phase generates a set of cubes, the conquer phase commences. Each cube, combined with the original formula, forms a subformula. These subformulas are then solved by CDCL solvers. Since the cubes are independent, this phase is highly amenable to parallel execution, with each subformula typically assigned to a different processing core or machine [40]. The conquer phase usually stops when a satisfying assignment for any subformula is found (if the original formula is satisfiable). If the original formula is unsatisfiable, all subformulas must also be proven unsatisfiable. Incremental CDCL solvers can be used, where the core formula is loaded once, and cubes are added as incremental constraints [42].

The power of CnC lies in its ability to transform a single, exceptionally hard SAT problem into a large collection of more manageable subproblems that can be explored concurrently. This strategy has proven highly effective for solving complex mathematical problems [13, 14, 52, 24, 41, 58, 86].

2.3 Cryptographic Hash Functions Revisited

A hash function H is fundamentally a mapping with two primary characteristics [61]:

1. Compression: It maps an input of arbitrary finite size (the message) to an output of fixed size (the hash value or digest).

2. Ease of Computation: For any given input x, computing H(x) is computationally straightforward and efficient.

For a hash function to be considered "cryptographic," it must possess additional security properties that make it

suitable for security applications. These properties are critical for its resistance to various forms of attacks [61]:

1. Collision Resistance: It is computationally infeasible to find two distinct inputs, x and x′, such that H(x)=H(x′) [61].

2. Preimage Resistance: For any given output y, it is computationally infeasible to find any input x′ such that H(x′)=y. This is the "one-way" property.

3. Second Preimage Resistance: For any given input x, it is computationally infeasible to find a distinct input x′ (x′≠x) such that H(x′)=H(x) [61].

An attack is considered "practical" if it can be performed within realistic computational time and resource constraints. While collision attacks are often easier to mount, practical preimage and second preimage attacks are generally much harder. This article specifically focuses on practical preimage attacks on step-reduced versions of MD4 and MD5 [88].

2.4 MD4 Cryptographic Hash Function

The Message Digest 4 (MD4) cryptographic hash function was introduced by Ronald Rivest in 1990 [69]. It processes an input message of arbitrary finite length and produces a 128-bit hash output. Before processing, padding is applied to the message to ensure its length is a multiple of 512 bits. The core of MD4 is its compression function, which iteratively processes 512-bit blocks according to the Merkle-Damgård construction [62, 20].

MD4 Compression Function Details: The compression function takes a 512-bit input block and generates a 128-bit output. It operates on four 32-bit registers, denoted A, B, C, and D. These registers are initialized with specific constant values (0x67452301, 0xefcdab89, 0x98badcfe, 0x10325476) for the very first message block. For subsequent blocks, they are initialized with the output of the compression function from the previous block.

The 512-bit message block is divided into sixteen 32-bit words, M[0] through M[15]. The compression function consists of three distinct rounds, each comprising sixteen steps, totaling 48 steps. In each step, one of the four registers is updated by mixing a message word with the values of all four registers and an additive constant. This mixing involves a round-specific Boolean function and circular bit shifts (rotations to the left). The three rounds are characterized by their unique Boolean functions and additive constants [69].

After all 48 steps, the final values of registers A, B, C, D are added modulo 232 to their initial values before concatenation to produce the 128-bit output. Algorithm 1 illustrates the general flow of the MD4 compression function for the first message block [88].

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.

// Initialize registers with fixed constants for the first block

AA ← A ← 0x67452301

BB ← B ← 0xefcdab89

CC ← C ← 0x98badcfe

DD ← D ← 0x10325476

// Note: [a b c d k s F] implies: a = (a + F(b, c, d) + M[k]) <<< s (addition modulo 2^32, <<< is circular left shift)

// Similar notation for G and H with their respective constants.

// Round 1 (Steps 1-16)

[ABCD M[0] 3 F] [DABC M[1] 7 F] [CDAB M[2] 11 F] [BCDA M[3] 19 F] // Steps 1-4

[ABCD M[4] 3 F] [DABC M[5] 7 F] [CDAB M[6] 11 F] [BCDA M[7] 19 F] // Steps 5-8

[ABCD M[8] 3 F] [DABC M[9] 7 F] [CDAB M[10] 11 F] [BCDA M[11] 19 F] // Steps 9-12

[ABCD M[12] 3 F] [DABC M[13] 7 F] [CDAB M[14] 11 F] [BCDA M[15] 19 F] // Steps 13-16

// Round 2 (Steps 17-32) - uses G function and 0x5a827999

[ABCD M[0] 3 G] [DABC M[4] 5 G] [CDAB M[8] 9 G] [BCDA M[12] 13 G] // Steps 17-20

[ABCD M[1] 3 G] [DABC M[5] 5 G] [CDAB M[9] 9 G] [BCDA M[13] 13 G] // Steps 21-24

[ABCD M[2] 3 G] [DABC M[6] 5 G] [CDAB M[10] 9 G] [BCDA M[14] 13 G] // Steps 25-28

[ABCD M[3] 3 G] [DABC M[7] 5 G] [CDAB M[11] 9 G] [BCDA M[15] 13 G] // Steps 29-32

// Round 3 (Steps 33-48) - uses H function and 0x6ed9eba1

[ABCD M[0] 3 H] [DABC M[8] 9 H] [CDAB M[4] 11 H] [BCDA M[12] 15 H] // Steps 33-36

[ABCD M[2] 3 H] [DABC M[10] 9 H] [CDAB M[6] 11 H] [BCDA M[14] 15 H] // Steps 37-40

[ABCD M[1] 3 H] [DABC M[9] 9 H] [CDAB M[5] 11 H] [BCDA M[13] 15 H] // Steps 41-44

[ABCD M[3] 3 H] [DABC M[11] 9 H] [CDAB M[7] 11 H] [BCDA M[15] 15 H] // Steps 45-48

// Final additions to initial values

A ← A + AA

B ← B + BB

C ← C + CC

D ← D + DD

Known Vulnerabilities: MD4 has been shown to be vulnerable to practical collision attacks since 1995 [24]. While it generally remains preimage and second preimage resistant from a practical standpoint, theoretical preimage attacks exist, with complexities like 2102 (later reduced to 299.7) [57, 36]. The focus of this work is to demonstrate practical preimage attacks on its step-reduced versions [88].

2.5 Dobbertin's Constraints for MD4

To facilitate the inversion of step-reduced MD4, Hans Dobbertin introduced a set of "additional constraints" in 1998 [25]. These constraints significantly simplify the inversion process by fixing certain intermediate register values to a constant 32-bit word K at specific steps within the hash function's execution.

Specific Constraints: For a 32-step MD4, Dobbertin's constraints are as follows (steps numbered from 1):

- Register A must be equal to K in steps 13, 17, 21, 25.
- Register D must be equal to K in steps 14, 18, 22, 26.
- Register C must be equal to K in steps 15, 19, 23, 27.

Algorithm 2 illustrates how these constraints modify the MD4 compression function [88].

Algorithm 2 MD4 compression function on the first 512-bit message block with 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.

// Constrained Steps

A = K; [ABCD M[12] 3 F] // Constrained step 13 (A is set to K before update for next step)

D = K; [DABC M[13] 7 F] // Constrained step 14

C = K; [CDAB M[14] 11 F] // Constrained step 15

[BCDA M[15] 19 F] // Step 16

A = K; [ABCD M[0] 3 G] // Constrained step 17

D = K; [DABC M[4] 5 G] // Constrained step 18

C = K; [CDAB M[8] 9 G] // Constrained step 19

[BCDA M[12] 13 G] // Step 20

A = K; [ABCD M[1] 3 G] // Constrained step 21

D = K; [DABC M[5] 5 G] // Constrained step 22

C = K; [CDAB M[9] 9 G] // Constrained step 23

[BCDA M[13] 13 G] // Step 24

A = K; [ABCD M[2] 3 G] // Constrained step 25

D = K; [DABC M[6] 5 G] // Constrained step 26

C = K; [CDAB M[10] 9 G] // Constrained step 27

[BCDA M[14] 13 G] // Step 28

Steps 29-48 as in Algorithm 1.

Increment A, B, C, D as in Algorithm 1.

Impact on Message Words: The most profound effect of Dobbertin's constraints is the automatic derivation of values for several message words. For instance, at step 17, due to prior constraints, A=C=D=K, with only register B being unknown. Given that G is the majority function, G(x,y,y)=y. Thus, the operation becomes $(K+G(B,K,K)+M[0]+0x5a827999)\ll 3=K$. This simplifies to $K\gg 3=2K+M[0]+0x5a827999$, allowing M[0] to be expressed as $M[0]=(K\gg 3)-2K-0x5a827999$. This means M[0] becomes a constant if K is a constant. The same logic applies to M[4],M[8],M[1],M[5],M[9],M[2],M[6], and M[10] due to subsequent constrained steps. Consequently, Dobbertin's constraints fix 9 out of the 16 message words to constant values [25].

This reduction means that the constrained MD4 compression function maps a 224-bit input (from the 7 unfixed message words) onto a 128-bit output, as opposed to the original 512-bit input. While these constraints are not implied by the original hash function (they are "streamlined constraints" that can remove some or all solutions [32]), they drastically reduce the search space for preimages, making the problem significantly easier to solve. If no preimage exists for a randomly chosen K, other values of K can be tried.

Historical Context: Dobbertin's constraints were initially proposed for 32-step MD4 [25]. In 2007, a SAT-based implementation of slightly modified Dobbertin's constraints allowed the inversion of 39-step MD4 for a specific, highly regular hash (all 1s) [22]. Despite this, inverting 40-step MD4 remained an open challenge [56], which this work aims to address for 40-, 41-, 42-, and 43-step MD4 [88].

2.6 MD5 Cryptographic Hash Function

MD5 (Message Digest 5) was developed by Ronald Rivest in 1992 as an renovation of MD4, designed to be a slightly slower but more secure cryptographic hash function [70]. It also produces a 128-bit hash from an arbitrary-length message.

Key Differences from MD4: MD5 incorporates several significant changes to enhance its security over MD4 [70]:

1. Extended Rounds: MD5 introduces a fourth 16-step round, bringing the total number of steps to 64 (compared to MD4's 48).

2. New Round Function: The function for the second round, G, is different: $G(x,y,z)=(x\wedge z)\vee(y\wedge\neg z)$.

3. Unique Additive Constants: Unlike MD4, which uses round-specific additive constants, MD5 employs a unique additive constant for each of its 64 steps.

4. Feedback Mechanism: The output from the previous step is added to the current register update, introducing a stronger mixing property.

The four round functions for MD5 are:

- 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: $I(x,y,z)=y\oplus(x\vee\neg z)$

Vulnerabilities and Inversion Attempts: A practical collision attack on MD5 was first demonstrated in 2005 [85]. A theoretical preimage attack was proposed in 2009, with a complexity of $2^{123.4}$ [73]. Due to its more robust design, Dobbertin's constraints, which proved effective for MD4, are not efficient for MD5; applying them often removes all possible preimages, rendering the simplified problem unsolvable [3]. Despite these challenges, reduced-step versions of MD5 have been inverted: 26-step MD5 was inverted in 2007 [22], and 27- and 28-step MD5 inversions were achieved in 2012 using CDCL solvers without additional constraints [56]. This work aims to extend these results by inverting 28-step MD5 for new hashes.

Despite its known vulnerabilities, MD5 remains in use for purposes like verifying data integrity on certain operating systems, highlighting the ongoing importance of cryptanalytic research into its properties [88].

3. Proposed Methodology for Hash Function Inversion

This section details the novel methodologies introduced to address the problem of inverting step-reduced MD4 and MD5. It begins by proposing a generalization of Dobbertin's constraints, followed by a description of the iterative algorithm for inversion and a sophisticated approach to finding optimal cutoff thresholds for Cube-and-Conquer.

3.1 Dobbertin-like Constraints: A Generalization

Building upon the concept of Dobbertin's constraints, which fix specific register values to a constant K, we introduce Dobbertin-like constraints as a generalization. Suppose a constant 32-bit word K is chosen. In this generalized scheme, while most of the 12 Dobbertin's constraints still hold, one particular constraint is relaxed. For this specially constrained step p, only a specific number of bits (b, where $0\leq b\leq 32$) of the target register's value are required to match the corresponding bits of K. The remaining $32-b$ bits in that register are mandated to take the opposite values to those in K.

To formalize this, we define an inverse problem for step-reduced MD4 with Dobbertin-like constraints as MD4inversion(y,s,K,p,L), where:

- y: A given 128-bit target hash.
- s: The number of MD4 steps being considered, starting from the first step.
- K: A 32-bit constant word used in the Dobbertin's constraints.
- p: The specific step {13,14,15,17,18,19,21,22,23,25,26,27} where the register's value is specially constrained according to the mask L.
- L: A 32-bit word serving as a bitmask. If the i-th bit of L (Li) is 0 ($0\leq i\leq 31$), then the i-th bit of the register's value modified in step p must be equal to the i-th bit of K (Ki). Conversely, if Li=1, then the i-th bit of the register's value must be equal to the negation of Ki (∼Ki).

It is evident that the original Dobbertin's constraints are a special instance of Dobbertin-like constraints, specifically when L=0x00000000, implying all bits of the constrained register match K.

To illustrate, consider the following examples from the literature [88], where 0hash refers to 128 zeros (i.e., four 0x00000000 words) and 1hash refers to 128 ones (i.e., four 0xffffffff words):

- Example 3.1: MD4inversion(0hash,32,0x62c7Ec0c,21,0x00000000) This problem seeks to invert a 0hash generated by 32-step MD4. Since L=0x00000000, the register's value in step 21 (the specially constrained step) is set to be identical to K=0x62c7Ec0c. Consequently, all 12 standard Dobbertin's constraints are applied, similar to the problem solved in Dobbertin's original work [25].
- Example 3.2: MD4inversion(1hash,39,0xfff00000,12,0xffffffff) Here, the goal is to invert a 1hash from 39-step MD4. Given L=0xffffffff, the register's value in step 12 is forced to be the bitwise negation of K, i.e., ∼K=0x000fffff. For the remaining 11 Dobbertin's steps, the registers' values are equal to K=0xfff00000.
- Example 3.3:

MD4inversion(1hash,40,0xffffffff,12,0x00000003) This example involves inverting a 1hash for 40-step MD4. With L=0x00000003, the first 30 bits of the specially constrained register in step 12 are identical to those in K. However, the last two bits (K30 and K31) are inverted, meaning the register's value is 0xfffffffc. For the other 11 Dobbertin's steps, the registers' values remain K=0xffffffff.

These examples illustrate the fine-grained control offered by Dobbertin-like constraints, allowing for systematic exploration of the solution space around the original Dobbertin's constraints.

3.2 Iterative Inversion Algorithm

The Dobbertin-like constraints form the basis of an iterative algorithm designed to find preimages for step-reduced MD4 [88]. The underlying principle is to systematically explore variations of the specially constrained register's value by modifying the mask L.

The algorithm operates as follows: For a given target hash y, number of MD4 steps s, and a chosen constant K, an initial inverse problem is formulated with L=0x00000000. This corresponds to applying all 12 standard Dobbertin's constraints. If a preimage is found for this problem, the process terminates. If it's proven that no preimages exist for the current problem, a new problem is formed by incrementing L (e.g., L=0x00000001). This new L value dictates that the specially constrained register is now just one bit different from K. This process continues iteratively, incrementing L in each step: 0x00000002,0x00000003, and so forth.

The intuition behind this strategy is that Dobbertin's constraints often lead to a system of equations that is either consistent with very few solutions or remarkably close to a consistent state. By progressively modifying L, the algorithm "nudges" the system towards consistency, increasing the likelihood of finding a preimage. Even if solving a few such "simplified" problems is required, it can still be significantly faster than tackling the original unconstrained problem.

Algorithm 3, as depicted in the original paper, outlines this iterative process:

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 A.

Output: Preimages for hash y.

preimages ← {}

i ← 0

while i < 2^32 do

  L ← DecimalToBinary(i) // Convert decimal i to its 32-bit binary representation L

  preimages ← A(MD4inversion(y, s, K, p, L)) // Solve the inverse problem with algorithm A

  if preimages is not empty then

    break // Preimages found, terminate

  i ← i + 1

return preimages

A crucial component of Algorithm 3 is the "complete algorithm A," which is responsible for solving the inverse problem for each given set of Dobbertin-like constraints. While various complete constraint programming solvers [71] could serve this purpose, preliminary experiments indicated that even state-of-the-art sequential and parallel CDCL SAT solvers struggled with 40-step MD4 instances, even with all 12 Dobbertin's constraints applied (i.e., L=0x00000000) [88]. This motivated the adoption of Cube-and-Conquer (CnC) SAT solvers, which are better suited for extremely hard SAT instances [40]. The subsequent subsection details how a given problem can be optimally partitioned into simpler subproblems during the cubing phase of CnC.

3.3 Finding Optimal Cutoff Thresholds for Cube-and-Conquer

The efficacy of the Cube-and-Conquer approach is highly dependent on the proper selection of the "cutoff threshold" used in its cubing phase. As discussed in Subsection 2.2, this threshold defines when a branch in the look-ahead search tree is converted into a standalone "cube" (subproblem) based on the number of unassigned variables remaining in that subformula. An improperly chosen threshold can severely hamper performance:

- Threshold too high: Leads to a rapid cubing phase but results in very few, yet still extremely hard, subproblems for the CDCL solver in the conquer phase.
- Threshold too low: The cubing phase becomes excessively time-consuming, generating an unmanageably large number of subproblems.

Earlier attempts to optimize this cutoff threshold, such as Algorithm A from Heule's tutorial [38] and Algorithm B from Bright et al. [14], involved sampling a small number of subproblems, solving them, and using their hardness to estimate the total runtime. However, these informal algorithms lacked detailed specification and did not explicitly account for scenarios where some sampled subproblems might be too hard to solve within reasonable time limits.

We propose a new, formalized algorithm (Algorithm 5) inspired by these prior efforts but specifically designed to find a cutoff threshold that minimizes the estimated runtime of the conquer phase, assuming the cubing phase's runtime is negligible in comparison [88].

Algorithm 4: Preselecting Promising Thresholds: Before estimating the hardness of various thresholds, it's beneficial to preselect a set of "promising" ones. These are

thresholds that result in a significant number of "refuted leaves" (indicating some simplification) but avoid generating an excessively large total number of cubes. Algorithm 4 performs this initial filtering:

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

Input: CNF F; lookahead solver ls; starting threshold nstart; 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, nstart, nstep, maxc, minr)

stack ← {}

n ← nstart

while n > 0 do

⟨c, r⟩ ← LookaheadWithCut(ls, F, n) // Get cubes (c) and number of refuted leaves (r)

if Size(c) > maxc then

break // Too many cubes, stop decreasing threshold

if r ≥ minr then

stack.push(⟨n, c⟩) // Add threshold and its associated cubes to stack

n ← n − nstep // Decrease threshold for next iteration

return stack

The LookaheadWithCut function simulates the look-ahead solver with the given n (cutoff threshold), generating cubes and counting refutations. The parameters maxc (maximum cubes) and minr (minimum refuted leaves) serve as crucial filters to ensure only potentially viable thresholds are considered for the more intensive estimation phase.

Algorithm 5: Finding Optimal Cutoff Threshold for Conquer Phase: Once promising thresholds are preselected, Algorithm 5 proceeds to estimate the hardness of the corresponding conquer phases. This is achieved by randomly sampling a fixed number of cubes from each set and solving them with a CDCL solver. The measured runtimes of this sample are then used to extrapolate the total estimated solving time for all cubes associated with that threshold.

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

Input: CNF 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 nbest with the runtime estimate ebest and cubes cbest; Boolean isSAT that indicates whether F is satisfiable.

isSAT ← Unknown

nstart ← Varnum(F) − nstep // Initialize starting threshold

⟨nbest, ebest, cbest⟩ ← ⟨nstart, +∞, {}⟩ // Initialize best estimate to infinity

stack ← PreselectThresholds(F, ls, nstart, nstep, maxc, minr) // First stage: preselect thresholds

while stack is not empty do // Second stage: estimate thresholds

⟨n, c⟩ ← stack.pop() // Get a threshold and its cubes

sample ← SimpleRandomSample(c, N) // Select N random cubes from the set 'c'

runtimes ← {}

for each cube from sample do

⟨t, st⟩ ← SolveCube(cs, F, cube, maxcst) // Add cube to F and solve with CDCL solver cs within maxcst

if t ≥ maxcst and mode = estimating then // If CDCL was interrupted in estimating mode

break // Stop processing this sample (estimate is unreliable)

else

runtimes.add(t) // Add measured runtime

if st = True then // If satisfying assignment found

isSAT ← True

if mode = incomplete-solving then // In incomplete SAT solving mode, return immediately

return ⟨nbest, ebest, cbest, isSAT⟩ // Problem is satisfiable, and a solution is found

if Size(runtimes) < N then // If at least one cube in the sample was interrupted

break // This threshold's estimate is unreliable, stop main loop

e ← Mean(runtimes) · Size(c) / cores // Calculate estimated total runtime

if e < ebest then

⟨nbest, ebest, cbest⟩ ← ⟨n, e, c⟩ // Update best threshold if current estimate is lower

return ⟨nbest, ebest, cbest, isSAT⟩

Key Features and Operating Modes: Algorithm 5 operates in two distinct modes, tailored for different objectives [88]:

1. Estimating Mode: This mode is designed to estimate the hardness of an arbitrary CNF. It will terminate and indicate an unreliable estimate if any subproblem within the random sample exceeds the maxcst time limit. This is crucial for cryptanalysis problems, where subproblems often exhibit vast differences in solution times. Even if a satisfying assignment is found during sampling, the algorithm continues to calculate the runtime estimate.

2. Incomplete SAT Solving Mode: This mode is geared towards finding a satisfying assignment for a satisfiable CNF, especially those with many solutions. It will terminate immediately upon finding any satisfying assignment within a sampled subproblem, even if the maxcst limit is reached for other subproblems in the sample. This mode is incomplete, meaning it does not guarantee finding a solution or proving unsatisfiability.

Notable Features of Algorithm 5:

- Stack-based Preselection: The use of a stack in PreselectThresholds (Algorithm 4) ensures that the second stage begins with the simplest subproblems, allowing for quick initial estimates and subsequent refinements.
- Robustness to Hard Subproblems: The "estimating mode" explicitly handles cases where sampled subproblems cannot be solved within limits, preventing unreliable estimates and guiding the search efficiently. This is a critical distinction from prior algorithms [14, 42].
- Satisfiability Detection: The algorithm can detect satisfiability during the sampling phase, which is particularly useful in the "incomplete SAT solving mode."
- General Applicability: The "estimating mode" can assess the hardness of any CNF, regardless of its origin.
- Configurable Termination: The algorithm can be easily adapted to find just one solution in the conquer phase by modifying the termination condition in SolveCube.
- Black-Box Optimization: The runtime estimation process is treated as a stochastic, costly black-box objective function, which the algorithm aims to minimize [4, 77].

Once nbest (the optimal cutoff threshold) is identified by Algorithm 5, the full conquer phase proceeds. Subformulas are created by adding the respective cubes as unit clauses to the original CNF. All these subformulas are then solved by the same CDCL solver (Kissat in this study) used during the estimation phase, but without a time limit per subproblem. The conquer phase outputs a list of all found satisfying assignments (preimages) or declares unsatisfiability if no solutions are found.

4. Experimental Setup and SAT Encoding

This section details the practical implementation and configuration of the proposed algorithms, outlining the chosen hashes for inversion, the specific MD4 and MD5 problem definitions, the methods used for SAT encoding, and the crucial simplification strategies applied to the Boolean formulas.

4.1 General Experimental Environment

The algorithms proposed were implemented as a dual-language system: Algorithm 3 (the iterative inversion algorithm) was developed in Python, while Algorithm 5 (the cutoff threshold optimizer) and the core conquer phase of Cube-and-Conquer were written in C++. This integrated parallel SAT solver is referred to as Estimate-and-Cube-and-Conquer (EnCnC) in the source literature [88]. The source code and associated datasets are publicly available [89, 52].

All experiments were conducted on a personal computer equipped with a 12-core AMD 3900X CPU. The implementations were designed to be multithreaded, fully leveraging all 12 CPU cores. This parallel processing capability was utilized both during the parameter estimation phase (Algorithm 5), where different cutoff thresholds and subproblems within samples were processed in parallel, and during the final conquer phase, where the independent cubes were solved concurrently.

4.2 Parameterization of Algorithms

The effectiveness of the Cube-and-Conquer approach is highly sensitive to its internal parameters. Based on extensive preliminary experiments and standard practices in SAT solving competitions, the following parameters were chosen for Algorithm 5:

- Look-ahead Solver: March_cu [42] was selected for the cubing phase due to its proven success in solving several hard combinatorial problems [41, 52].
- Threshold Decreasing Step (nstep): Set to 5. While a value of 1 might yield slightly more precise thresholds, it significantly increases the runtime of Algorithm 5. Larger values (e.g., 50) risk skipping promising thresholds [88].
- Maximum Number of Cubes (maxc): Initially set to 2,000,000. March_cu typically generates this many cubes within approximately 30 minutes on the test CNFs. Higher values did not show significant improvement [88]. For 41-, 42-, and 43-step MD4, this was reduced to 500,000 based on findings from 40-step MD4.
- Minimum Number of Refuted Leaves (minr): Set to 1,000. If minr is lower, subproblems tend to be too hard due to insufficient simplification by the look-ahead solver. Higher values limited the number of promising thresholds found [88].
- Sample Size (N): Set to 1,000. Initial tests with

N=100 resulted in overly optimistic runtime estimates. While N=10,000 offered marginal accuracy improvement, its computational cost was prohibitive [88].

- CDCL Solver (cs): Kissat (version sc2021) [10] was used for solving individual cubes in the conquer phase and during the sampling of Algorithm 5. Kissat and its variants have consistently performed well in recent SAT Competitions.

- CDCL Solver Time Limit (maxcst): Set to 5,000 seconds. This is a standard time limit in SAT Competitions [5], reflecting the typical design optimization of modern CDCL solvers for performance within this window.

- Number of CPU Cores (cores): Set to 12, to utilize the full capacity of the available hardware.

- Operating Mode (mode): Typically estimating for MD4 and incomplete-solving for MD5, as detailed in Subsection 3.3.

4.3 Selection of Hashes for Inversion

To ensure a comprehensive analysis and to validate the robustness of the proposed approach, four distinct 128-bit hash values were chosen as targets for inversion [88]:

1. 0x00000000 0x00000000 0x00000000 0x00000000 (0hash): A hash consisting of all zeros.

2. 0xffffffff 0xffffffff 0xffffffff 0xffffffff (1hash): A hash consisting of all ones.

3. 0x01234567 0x89abcdef 0xfedcba98 0x76543210 (symmhash): A symmetrical hash, where the last 64 bits are the reverse of the first 64 bits.

4. 0x62c7Ec0c 0x751e497c 0xd49a54c1 0x2b76cff8 (randhash): A randomly generated hash.

The inclusion of 0hash and 1hash is standard practice in the cryptographic community when studying preimage resistance [88]. Inverting such highly regular hashes provides stronger evidence of an attack's efficacy, as randomly chosen hashes could be trivially "inverted" by simply generating a random message and presenting its hash. Symmhash provides a structured, yet less trivially uniform, target. Randhash confirms the applicability of the approach to typical, non-structured outputs.

4.4 Step-reduced MD4 Problem Definition

The study focused on inverting step-reduced MD4 compression functions. Consistent with previous research [22, 56], padding was omitted, and only a single 512-bit message block was considered. The final incrementing step (adding initial register values) was also excluded, as it occurs only after the 48th step. These simplifications do not diminish the difficulty from a resistance standpoint, as the compression function forms the core of MD4's security.

The specific versions of MD4 studied ranged from 40 to 47 steps, including the full 48-step MD4. For the Dobbertin-like constraints, two values for the constant K were tested: 0x00000000 and 0xffffffff [88]. The constraint in step 12 was chosen for modification (i.e., p=12 in MD4inversion), as this constraint was entirely omitted in earlier works [22]. In total, this setup generated 9×4×2=72 distinct MD4-related inverse problems, none of which had been solved prior to this work.

For illustrative purposes, consider the initial setup for 40-step MD4. Given two values for K and four hash targets, Algorithm 3 would initiate with the following eight inverse problems in its first iteration (L=0x00000000 for all):

1. MD4inversion(0hash,40,0x00000000,12,0x00000000)
2. MD4inversion(0hash,40,0xffffffff,12,0x00000000)
3. MD4inversion(1hash,40,0x00000000,12,0x00000000)
4. MD4inversion(1hash,40,0xffffffff,12,0x00000000)
5. MD4inversion(symmhash,40,0x00000000,12,0x00000000)
6. MD4inversion(symmhash,40,0xffffffff,12,0x00000000)
7. MD4inversion(randhash,40,0x00000000,12,0x00000000)
8. MD4inversion(randhash,40,0xffffffff,12,0x00000000)

If a preimage is not found for a given problem (e.g., the first one), Algorithm 3 would then increment L to 0x00000001, forming a new problem like MD4inversion(0hash,40,0x00000000,12,0x00000001), and continue until a preimage is discovered or all 232 values of L are exhausted.

4.5 Step-reduced MD5 Problem Definition

For MD5, the focus was on inverting the 28-step MD5 compression function without introducing any additional constraints, mirroring the approach taken in previous work [22]. This is because Dobbertin's constraints are known to be ineffective for MD5 due to its modified design, often removing all possible preimages [3]. The same four hashes (0hash, 1hash, symmhash, randhash) were used as targets for the 28-step MD5 inversion, with particular attention to 0hash, 1hash, and randhash, for which inversion results had not been previously published [88].

4.6 SAT Encoding Process

The conversion of the hash function's algorithmic description into a Conjunctive Normal Form (CNF)

Boolean formula is a critical step. This study utilized Transalg (version 1.1.5) [75], an automatic tool designed for this purpose. Transalg accepts an algorithm described in a C-like domain-specific language (TA language), which supports variable declarations, assignment operators, conditional operators, loops, function calls, and various integer and bit operations, including bit shifting and comparison. These features make it highly suitable for describing cryptographic algorithms. The generated CNFs and the corresponding TA programs are available as a public dataset [89].

In the generated CNFs, the Boolean variables are categorized as follows:

- Message Variables: The first 512 variables represent the bits of the input message block.
- Hash Variables: The last 128 variables represent the bits of the hash output.
- Auxiliary Variables: All remaining variables are introduced by the Tseitin transformation [82] to encode the intermediate computations and logical relationships within the hash function's algorithm.

The core CNF generated by Transalg encodes the hash function itself, with all message and hash variables initially unassigned. To encode a specific inverse problem for a given 128-bit target hash, 128 unit clauses (e.g., (hash_bit_i) or (~hash_bit_i)) are added to the CNF, fixing the values of the hash variables to the target hash. The goal then becomes finding the values of the message variables that satisfy this constrained CNF.

For MD4 inverse problems, Dobbertin's or Dobbertin-like constraints are incorporated by adding an additional 384 unit clauses. Each 32-bit constraint translates into 32 unit clauses, fixing specific intermediate register bits based on K and the bitmask L.

The CNFs for various step-reduced MD4 and 28-step MD5 were constructed. For MD4, each additional step increases the number of variables by 186 and clauses by 2,349. For instance, the CNF for 40-step MD4 has 7,025 variables and 70,809 clauses. With the addition of Dobbertin's constraints, the number of clauses for 40-step MD4 inverse problems becomes 71,321 (70,809 + 384 for Dobbertin's + 128 for hash bits), and for 48-step MD4, it becomes 90,113. For 28-step MD5, an inverse problem has 7,471 variables and 54,800 clauses (54,672 + 128 for hash bits).

4.7 Simplification Strategies

The initial CNFs, especially those resulting from cryptographic encodings, can be complex. Simplifying these formulas before feeding them to the look-ahead solver in the cubing phase is crucial for enhancing the overall efficiency of Cube-and-Conquer.

The CDCL solver CaDiCaL (version 1.5.0) [28] was employed for preprocessing and simplifying the CNFs. CaDiCaL utilizes "inprocessing" techniques [8], where the CNF is continuously simplified during the CDCL search itself. The extent of simplification is often correlated with the number of conflicts generated by the CDCL solver. Therefore, the number of generated conflicts was used as a natural measure for simplification depth.

The following limits on the number of generated conflicts were experimentally evaluated: 1, 10 thousand, 100 thousand, 1 million, and 10 million [88]. A limit of 1 conflict often yields a similar result to basic Unit Propagation (UP) but can sometimes result in a slightly smaller CNF.

The impact of different simplification levels on an example CNF (encoding MD4inversion(1hash,40,0xffffffff,12,0x00000000)) was observed. Initially, increasing the conflict limit generally reduces the number of variables, clauses, and literals. However, for 10 million conflicts, while the variable count is the lowest, the number of clauses and literals increases compared to 1 million conflicts. This "non-monotonic" behavior suggests a point of diminishing returns or a change in the nature of simplification, an interesting phenomenon worth further investigation. Similar trends were observed across other hashes and L values [88].

5. Results and Analysis: Inverting MD4 (40-43 steps)

This section presents the detailed experimental results and subsequent analysis of applying the proposed Dobbertin-like constraints and the Cube-and-Conquer approach to invert step-reduced MD4 functions, specifically focusing on 40-, 41-, 42-, and 43-step versions.

5.1 Combining Algorithms and Initial Findings for 40-step MD4

For the MD4 inversion problems, a specific combination of Algorithm 3 and Algorithm 5 was employed to balance computational efficiency with thoroughness. For each pair of (number of MD4 steps, hash), the best cutoff threshold was first determined using the estimating mode of Algorithm 5 on a CNF with the standard Dobbertin's constraints applied (L=0x00000000). Once this optimal threshold was identified, Algorithm 3 then utilized this fixed threshold in its conquer phase for all subsequent iterations, even when varying the L parameter for Dobbertin-like constraints. This strategy aimed to reduce the overall number of computationally intensive Algorithm 5 calls.

Initial experiments on 40-step MD4 revealed a significant dependency on the choice of the Dobbertin's constant K. When K=0x00000000, Algorithm 5 consistently failed to find reliable runtime estimates for all 20 tested CNFs (4 hashes × 5 simplification types). This failure was attributed to Kissat (the underlying CDCL solver) being interrupted due to time limits even on the simplest (lowest cutoff threshold) subproblems, indicating the extreme difficulty of these instances with K=0x00000000.

In stark contrast, setting K=0xffffffff yielded much more positive results. For 0hash, symmhash, and randhash,

reliable estimates were successfully calculated across all simplification types. Interestingly, for these three hashes, the 1-conflict-based simplification consistently provided the best estimates. However, for 1hash, 1-conflict and 10-thousand-conflict simplifications did not produce estimates, while 1-million-conflict simplification proved to be the most effective [88]. This highlights the importance of trying different simplification strategies.

Figure 1 visually represents the minimization of the objective function (estimated runtime) for the 40-step MD4, 0hash inverse problem. It illustrates how different simplification levels and cutoff thresholds impact the estimated solving time and the number of cubes, ultimately pinpointing the optimal configuration [88].

Correctness Verification: The validity of all found preimages was rigorously verified using the reference implementation of MD4 [69]. This process is computationally trivial and can be easily reproduced: the found preimage (message) is fed into the MD4 compression function (after removing padding and final incrementing steps for step-reduced versions), and the resulting hash is compared against the target hash. This confirmation step is crucial for cryptographic attacks.

Impact of Simplification: Contrary to expectations, the 1-conflict-based simplification generally outperformed more aggressive simplification levels (10k, 100k, 1M, 10M conflicts) in terms of achieving the best runtime estimates. This suggests that for these specific cryptanalytic CNFs, excessive inprocessing by CaDiCaL might lead to a state less amenable to the look-ahead solver in the cubing phase. However, the observation that 1-million-conflicts was optimal for 40-step 1hash means that relying solely on 1-conflict would have left some problems unsolved. This "non-effectiveness of advanced simplifications" is an intriguing phenomenon that warrants further investigation into how different simplification strategies interact with the Cube-and-Conquer paradigm for cryptographic SAT instances.

Classes of Subproblems: An important observation from the runtime distributions of subproblems (Figures 2 and 4) is the bimodal nature of their difficulty. Approximately 25% of the subproblems were "extremely easy" (solving in less than 0.1 seconds), while the remainder exhibited significantly higher and more varied runtimes. This clear gap between the easy and hard subproblems, far exceeding the mean and median runtimes, suggests a promising avenue for optimization: pre-solving these extremely easy subproblems and then applying the learned insights or simplified clauses to the harder remaining subproblems could significantly accelerate the overall process.

Estimation Accuracy and Heavy-Tail Behavior: The estimated runtimes from Algorithm 5 demonstrated a commendable accuracy when compared to the actual total solving times for inverse problems where L=0x00000000. On average, the real time was about 11% higher than the estimate, with a maximum deviation of 30% for 40-step MD4 and 0hash. This confirms the utility of Algorithm 5 for practical resource allocation. However, for inverse problems with L=0x00000001 or L=0x00000002, a different pattern emerged. While some real time values remained close to the initial L=0x00000000 estimates, others, like MD4inversion(0hash,41,0xffffffff,12,0x00000001), showed much higher real times (e.g., 2.5 times higher) and significantly larger standard deviations. This indicates the presence of "heavy-tail behavior" [31, 17], where a small fraction of instances are exceptionally hard and dominate the total runtime. This finding suggests that for L>0x00000000, it might be more beneficial to re-run Algorithm 5 to determine a new optimal cutoff threshold for each specific L value, rather than reusing the threshold derived for L=0x00000000.

Hardness Analysis Across Steps: It might seem counterintuitive that the observed hardness for inverting 40- to 43-step MD4 remained relatively similar, rather than steadily increasing with each additional step. This can be explained by examining the message word usage. When Dobbertin's constraints are applied, 9 out of the 16 message words (M[0,1,2,4,5,6,8,9,10]) are automatically derived (via Unit Propagation in the CNF), leaving only 7 words unknown [25]. This reduces the effective message input from 512 bits to 224 bits. The 40th step, where the register's value is updated using the unknown word M[14], introduces a significant "leap in hardness" compared to 39 steps. However, in steps 41, 42, and 43, the round function operates with known (constant) message words (M[1],M[9], and M[5], respectively). Since the main source of hardness in the MD4 compression function comes from mixing an unknown message word with register values, steps 41-43 do not introduce additional hardness. Instead, they primarily add more connections between existing register values. Conversely, steps 44-48 exclusively involve updating registers with unknown message words, leading to new, substantial leaps in hardness. This explains why no estimates could be calculated for 44-step MD4 using the current setup, suggesting that such problems would require a more powerful supercomputer for practical inversion.

Preimage and Second Preimage Attacks: The successful identification of preimages for 40-, 41-, 42-, and 43-step MD4 demonstrates practical preimage attacks [69]. Furthermore, for symmhash at 40 steps, two distinct preimages were found, implying a successful second preimage attack. Similarly, for 41-, 42-, and 43-step MD4, multiple preimages were found for at least one hash. This confirms the feasibility of both preimage and second preimage attacks on these step-reduced versions of MD4. It is plausible that more preimages exist for these problems. If an AllSAT solver (which finds all satisfying assignments) had been applied to the subproblems instead of a standard SAT solver (which finds only one), a complete enumeration of all preimages would have been possible.

6. Results and Analysis: Inverting Unconstrained 28-step MD5

This section focuses on the inversion of 28-step MD5, a distinct challenge due to the inapplicability of Dobbertin's constraints. Here, the "incomplete SAT solving" mode of Algorithm 5 was employed to find preimages for unconstrained instances.

6.1 Problem Setup

As discussed in Subsection 2.6, Dobbertin's constraints are not effective for MD5 [3]. Therefore, the inversion of 28-step MD5 was approached without adding any additional problem-specific constraints that reduce the number of preimages, mirroring prior work [22]. This means that for an arbitrary hash, there are theoretically approximately 2384 preimages (given the 128-bit hash output and 512-bit message input for a compression function with some message words dependent on others for previous steps), but finding any of them remains a formidable task.

The estimating mode of Algorithm 5 was found to be unsuitable for unconstrained MD5 inverse problems because the cubing phase produced subproblems that were too hard for the CDCL solver to produce reliable runtime estimates within a reasonable time frame. However, since unconstrained MD5 inverse problems are expected to have a vast number of solutions, the incomplete SAT solving mode of Algorithm 5, which prioritizes finding any satisfying assignment, was deemed highly suitable [48].

6.2 Experimental Details and CNF Characteristics

The CNF encoding for 28-step MD5 was constructed based on the specifications detailed in Subsection 4.6. The base CNF for the 28-step MD5 compression function contained 7,471 variables and 54,672 clauses. To create the inverse problem instances, the corresponding 128 unit clauses for the target hash bits were added. These CNFs were then simplified by CaDiCaL with a limit of 1 conflict generation, as this typically provided a good balance of simplification and initial problem structure.

The characteristics of these simplified CNFs for the four target hashes were determined. For 0hash, there were 6814 variables, 50572 clauses, and 199596 literals. For 1hash, there were 6844 variables, 50749 clauses, and 200153 literals. For symmhash, there were 6842 variables, 50737 clauses, and 200114 literals. For randhash, there were 6842 variables, 50741 clauses, and 200110 literals [88].

6.3 Algorithm Parameters and Performance Comparison

The EnCnC solver was run on these simplified CNFs in the incomplete SAT solving mode with the following parameters:

- Look-ahead Solver: March_cu.
- Threshold Decreasing Step (nstep): 10.
- Minimum Number of Refuted Leaves (minr): 0 (given the unconstrained nature and expectation of many solutions).
- Sample Size (N): 1,000.
- CDCL Solver: Kissat sc2021.
- CDCL Solver Time Limit (maxcst): 5,000 seconds.
- Number of CPU Cores (cores): 12.
- Operating Mode: incomplete-solving.

A key parameter investigated was maxc (the maximum number of generated cubes), with the following values tested: 2,000,000; 1,000,000; 500,000; 250,000; 125,000; and 60,000. The default value for maxc in EnCnC is 1,000,000. Each version of EnCnC with these maxc values was given a wall-clock time limit of 1 day to find a preimage.

The wall-clock runtimes for the different EnCnC configurations on the four MD5-28 inverse problems were recorded [88]. For EnCnC-maxc=2m, 0hash took 1 h 47 min, 1hash took 1 h 41 min, symmhash took 39 min, and randhash took 1 h 36 min. For EnCnC-maxc=1m, 0hash took 42 min, 1hash took 53 min, symmhash took 13 min, and randhash took 59 min. For EnCnC-maxc=500k, 0hash took 48 min, 1hash took 32 min, symmhash took 22 min, and randhash took 15 min. For EnCnC-maxc=250k, 0hash took 38 min, 1hash took 4 min, symmhash took 41 min, and randhash took 37 min. For EnCnC-maxc=125k, 0hash took 16 min, 1hash took 35 min, symmhash took 6 min, and randhash took 20 min. For EnCnC-maxc=60k, 0hash took 4 min, 1hash took 3 min, symmhash took 14 min, and randhash took 1 h 32 min. For comparison, two complete parallel CDCL SAT solvers, P-MCOMSPS (winner of the Parallel track in SAT Competition 2021 [83]) and Treengeling (a Cube-and-Conquer solver [9]), were also tested, but neither could solve any of the instances within the 1-day time limit.

7. Related Work

The field of SAT-based cryptanalysis has evolved significantly since its inception, finding broad application in the security analysis of various cryptographic primitives. This section contextualizes the current work by reviewing relevant prior research in SAT-based cryptanalysis, Cube-and-Conquer applications, and SAT instance hardness estimation.

7.1 Evolution of SAT-Based Cryptanalysis

The pioneering concept of applying SAT solvers to cryptanalytic problems was first articulated in 1996 [19], though its first practical application to a real-world cryptanalysis problem, a reduced version of the DES block cipher, occurred in 2000 [60]. Since then, SAT-based cryptanalysis has become a powerful tool, successfully analyzing numerous block ciphers, stream ciphers, and cryptographic hash functions [7].

7.2 SAT-Based Cryptanalysis of MD-family Hash Functions

Specific efforts using CDCL solvers have been directed at the MD family of hash functions:

- Early work by Jovanovic and Janicic (2005) used SAT-based cryptanalysis to construct benchmarks with adjustable hardness [47].
- Mironov and Zhang (2006) performed a practical collision attack on MD4 [63].
- The inversion of 39-step MD4 has been a recurring theme, with contributions from De et al. (2007), Legendre et al. (2012), Lafitte et al. (2014), Gribanova et al. (2017), and Gribanova and Semenov (2018) [22, 56, 55, 35, 33].
- Gladush et al. (2022) focused on estimating the hardness of practical preimage attacks on 43-, 45-, and 47-step MD4 [30].
- Gribanova and Semenov (2020) demonstrated the inversion of a full (48-step) MD4-based function they constructed [34].
- For MD5, practical collision attacks were performed by Mironov and Zhang (2006) and Gribanova et al. (2017) [63, 35].
- Preimage inversions for reduced-step MD5 versions include 26-step MD5 by De et al. (2007) and 27- and 28-step MD5 by Legendre et al. (2012) [22, 56]. The current work extends these by inverting 28-step MD5 for new regular and random hashes [88].

7.3 SAT-Based Cryptanalysis of SHA-family Hash Functions

The SHA family, including SHA-1 (an improved version of MD4), has also been subjected to SAT-based cryptanalysis:

- The first collision for SHA-1, found by Stevens et al. (2017), was partially achieved using a CDCL solver [81].
- Step-reduced versions of SHA-0, SHA-1, SHA-256, and SHA-3 have been inverted by various researchers [68, 56, 44, 67].
- Algebraic fault attacks on SHA-1 and SHA-2 were performed by Nejati et al. (2018), and on SHA-256 by Nakamura et al. (2021) [66, 64].

7.4 Theoretical Preimage Attacks

In contrast to the practical, SAT-based attacks discussed, theoretical preimage attacks on hash functions often determine the computational complexity of finding a preimage without necessarily providing a concrete algorithm or implementation. For MD4, the initial theoretical preimage attack by Leurent (2008) had a complexity of 2102, later reduced to 299.7 by Guo et al. (2010) [57, 36]. For MD5, the best theoretical attack has a complexity of 2123.4 [73]. This work distinguishes itself by demonstrating practical inversions of step-reduced MD4 and MD5 versions, providing concrete preimages using state-of-the-art SAT techniques.

7.5 Cube-and-Conquer Applications Beyond Cryptanalysis

The Cube-and-Conquer paradigm, while prominently featured in this cryptanalysis work, has a rich history of solving extremely challenging mathematical problems in other domains:

- The Erdős discrepancy problem was solved by Konev and Lisitsa (2015) [52].
- The Boolean Pythagorean Triples problem was solved and verified by Heule et al. (2016) [41].
- Heule (2018b) contributed to the solution of Schur Number Five [39].
- Lam's problem was resolved using a SAT-based approach by Bright et al. (2021) [14].
- Brakensiek et al. (2022) achieved the resolution of Keller's Conjecture [13].
- More recently, Li et al. (2024) improved the lower bound for the Minimum Kochen–Specker Problem [58].
- Weaver and Heule (2020) discovered new minimal perfect hash functions (non-cryptographic, used in lookup tables) using SAT technology [86].

This paper's significant contribution lies in being among the first to successfully apply Cube-and-Conquer to solve significant cryptanalysis problems for MD4 and MD5, extending its proven utility to the domain of cryptographic security analysis [88].

7.6 Hardness Estimation of SAT Instances and Backdoors

Estimating the hardness of SAT instances is a critical area of research, often explored through various approaches beyond the Cube-and-Conquer based method presented here:

- Ansótengui et al. (2008) explored tree-like space complexity [2].
- Supervised machine learning techniques have been applied for algorithm runtime prediction by Hutter et al. (2014) [45].
- Almagro-Blanco and Giràldez-Cru (2022) used a popularity–similarity model to characterize SAT formula temperature [1].
- The concept of "backdoors" is closely related to Cube-and-Conquer [23, 49, 72, 87]. Informally, a backdoor is a small subset of variables in a formula such that, if their values are correctly assigned, the remaining problem simplifies dramatically [87]. A set of backdoor variables can effectively be considered a "cube" in the CnC context. The hardness of an instance, given a backdoor, can be estimated by sampling and solving a small number of subproblems [76].
- The search for optimal backdoors with minimum hardness has been reduced to minimizing pseudo-Boolean

objective functions, particularly in SAT-based cryptanalysis [78, 51, 74]. These objective functions are typically costly, stochastic, and black-box in nature [90, 77]. The current paper contributes to this line of research by minimizing a similar pseudo-Boolean objective function to find an optimal cutoff threshold for the cubing phase of Cube-and-Conquer, rather than directly searching for a backdoor.

8. Conclusions and Future Work

This article has presented a comprehensive exploration of advanced techniques for preimage attacks on cryptographic hash functions, specifically leveraging the power of SAT solvers within the Cube-and-Conquer paradigm. Two primary algorithms were introduced and thoroughly evaluated.

The first algorithm systematically generates "Dobbertin-like constraints" for MD4, incrementally modifying one of the standard Dobbertin's constraints to explore the solution space. This iterative process, designed to find preimages for a given hash, utilizes a complete SAT-solving algorithm for its intermediate inverse problems.

The second algorithm, a general-purpose Cube-and-Conquer based solver, operates in two distinct modes. In its "estimating mode," it intelligently varies cutoff thresholds for the cubing phase of Cube-and-Conquer, estimating the CNF's hardness for each threshold through strategic sampling. This mode is versatile, capable of assessing the difficulty and guiding the solution of hard SAT instances across various problem classes. In its "incomplete SAT solving mode," the algorithm functions as a specialized SAT solver, optimized for finding solutions in satisfiable CNFs that possess a large number of satisfying assignments.

The efficacy of these proposed algorithms was empirically demonstrated through their application to cryptographic hash functions. A multithreaded implementation, combining the first algorithm with the estimating mode of the second, successfully achieved the first practical preimage attacks and second preimage attacks on 40-, 41-, 42-, and 43-step MD4 on a personal computer. This represents a significant step forward in understanding the practical security margins of MD4. In contrast, for MD5, where problem-specific constraints are ineffective, the incomplete SAT solving mode of the second algorithm was employed. This led to the first-time discovery of preimages for two of the most regular hashes (all 1s and all 0s) and a random non-regular hash produced by 28-step MD5.

Future Work:

Several promising avenues for future research emerge from this study:

- Application to Other Hash Functions: We plan to extend the application of the proposed algorithms to analyze the preimage resistance of other prominent cryptographic hash functions, including SHA-1, SHA-2, and RIPEMD.

- Investigation of MD4-Related Phenomena: The experiments revealed two intriguing phenomena specific to MD4 cryptanalysis that warrant deeper investigation:

1. The observed non-effectiveness (in most cases) of advanced CNF simplification techniques. Understanding why 1-conflict simplification often outperformed more aggressive strategies could lead to more optimized preprocessing for these problem types.

2. The evident division of subproblems in the conquer phase into two distinct classes: "extremely simple" and "hard." Developing strategies to identify and preprocess these easy subproblems beforehand, potentially leveraging their structure, could further enhance overall solving efficiency.

- Comparison of Hardness Estimation: We intend to conduct a comprehensive comparison of the estimating mode of our second proposed algorithm with other established approaches for assessing the hardness of SAT instances (e.g., tree-like space complexity, machine learning-based predictors, and backdoor analysis). This comparative study will provide further insights into the strengths and limitations of each methodology.

The continued advancement in SAT solving technologies, coupled with specialized strategies like Cube-and-Conquer, remains crucial for pushing the boundaries of practical cryptanalysis. Such research is vital not only for evaluating the current security of cryptographic primitives but also for informing the design of more robust and resilient hash functions in the future.

9. REFERENCES

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

2. Ans´otegui, C., Bonet, M. L., Levy, J., & Many`a, F. (2008). Measuring the hardness of SAT instances. In AAAI, pp. 222–228.

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

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

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

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

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

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

9. 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.

10. 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.

11. 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.

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

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

14. 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.

15. 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.

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

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

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

19. 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.

20. Damg˚ard, I. (1989). A design principle for hash functions. In CRYPTO, pp. 416–427.

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

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

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

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

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

26. 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.

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

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

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

30. 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.

31. 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.

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

33. 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.

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

35. 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.

36. 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.

37. Hamadi, Y., Jabbour, S., & Sais, L. (2009). ManySAT: a parallel SAT solver. J. Satisf. Boolean Model.

Comput., 6(4), 245–262.

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

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

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

41. 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.

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

43. 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.

44. 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.

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

46. Hyv¨arinen, A. E. J., Junttila, T. A., & Niemel¨a, I. (2010). Partitioning SAT instances for distributed solving. In LPAR, pp. 372–386.

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

48. 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.

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