Programmatic SAT for SHA-256 Collision Attack

By

Nahiyan Alamgir

A Thesis
Submitted to the Faculty of Graduate Studies
through the School of Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science
at the University of Windsor

Windsor, Ontario, Canada

2024

Programmatic SAT for SHA-256 Collision Attack

by

Nahiyan Alamgir

APPROVED BY:

H. Wu
Electrical and Computer Engineering

S. Samet
School of Computer Science

C. Bright, Supervisor
School of Computer Science

August 14, 2024

DECLARATION OF CO-AUTHORSHIP / PREVIOUS PUBLICATION

Co-Authorship

I hereby declare that this thesis incorporates material that is result of joint research, as follows:

In Chapters 4 and 5 of this thesis, the findings are the result of collaborative work with Dr. Curtis Bright and Dr. Saeed Nejati as co-authors. The computer algebraic techniques presented in Section 4.1 and Section 4.2.2 were proposed by Dr. Bright (based on the work of Mendel et al. [40]) and implemented by myself. Additionally, all aspects of the research were accomplished through joint efforts and collaboration with Dr. Bright and Dr. Nejati.

I am aware of the University of Windsor Senate Policy on Authorship and I certify that I have properly acknowledged the contribution of other researchers to my thesis, and have obtained written permission from each of the co-authors to include the above materials in my thesis. I certify that, with the above qualification, this thesis, and the research to which it refers, is the product of my own work.

Previous Publication

This thesis includes an original paper that has been previously published to a workshop for publication, as follows:

- **Portions of Chapter 3, 4, and 5**: Alamgir, N., Nejati, S., & Bright, C. (2024). SHA-256 Collision Attack with Programmatic SAT. In Kaufmann, D. and Brown, C., editors, Proceedings of the 9th International Workshop on Satisfiability Checking and Symbolic Computation, July 2, 2024, Nancy, France, Collocated with IJCAR 2024, volume 3717 of CEUR Workshop Proceedings, pages 91–110. CEUR-WS.org.

I certify that I have obtained a written permission from the copyright owners to include the above published materials in my thesis. I certify that the above material describes work completed during my registration as a graduate student at the University of Windsor.

General

ABSTRACT

Cryptographic hash functions play a crucial role in ensuring data security, generating fixed-length hashes from variable-length inputs. The hash function SHA-256 is trusted for data security due to its resilience after over twenty years of intense scrutiny. One of its critical properties is collision resistance, meaning that it is infeasible to find two different inputs with the same hash. Currently, the best SHA-256 collision attacks use differential cryptanalysis to find collisions in simplified versions of SHA-256 that are reduced to have fewer steps, making it feasible to find collisions. In this thesis, we use a satisfiability (SAT) solver as a tool to search for step-reduced SHA-256 collisions, and dynamically guide the solver with the aid of a computer algebra system (CAS) used to detect inconsistencies and deduce information that the solver would otherwise not detect on its own. Our hybrid SAT + CAS solver significantly outperformed a pure SAT approach, enabling us to find collisions in step-reduced SHA-256 with significantly more steps. Using SAT + CAS, we are able to find a 38-step slightly-modified SHA-256 collision first found with a highly sophisticated search tool by Mendel, Nad, and Schläffer. Conversely, a pure SAT approach could find collisions for no more than 28 steps. However, our work only uses the SAT solver CaDiCaL and its programmatic interface IPASIR-UP.

DEDICATION

I would like to dedicate this thesis to my family:

To my mother, for her altruism, kindness, and for being an important figure in my life.

To my father, for his financial support, sacrifices, and unwavering commitment to our family.

To my sister Nishat, who supported me in coming to Canada and pursuing my Master's degree, even while she faced the challenges of her Ph.D.

To my sister Shanta, for being my best friend and always being there for me, especially when I needed her the most.

ACKNOWLEDGEMENTS

My work would not have been possible without the support and guidance of many individuals. Their suggestions and inspiration have been invaluable, whether they are mentors, researchers, authors of influential projects, or even music artists. I would like to extend my heartfelt thanks to them, particularly:

- My supervisor Dr. Curtis Bright for his unwavering guidance throughout the past two years of this research. His exceptional flexibility, patience, and outstanding teaching and problem-solving skills have greatly contributed to my growth as a researcher and had a profound impact on me personally. Curtis's continual enthusiasm for mathematics and technology has made our discussions and collaborations both highly productive and immensely enjoyable.

- My mentor Dr. Saeed Nejati for providing directions to the research, sharing ideas and code, and his helpful suggestions and explanations along the way.

- Developers of the software I frequently used for my research, such as the operating system MANJARO and the entire ARCH LINUX community, the version control system GIT, the TeX and LaTeX typesetting systems, and the source-code editor VISUAL STUDIO CODE. I'd also like to thank Armin Biere and other authors of the SAT solver CADICAL, which played a critical role in this research.

- People behind the Digital Research Alliance of Canada, especially the team behind the clusters Narval, Béluga, and Graham for enabling me to run all the experiments.

- Dr. Oleg Zaikin for answering questions about his work related to 43-step MD4 inversion.

- Dr. Maria Eichlseder for her insights and answering questions related to her and her colleague's research related to SHA-256 collision attacks.

- My friends and colleagues under Dr. Bright—Yameen, Amadou, Aaron, Sadman, Aidan, Lamina, and Lily—for all the amusing and insightful discussions we've had about our work.

- All the members (current or past) of the rock bands Guns N' Roses and Led Zeppelin, especially Axl Rose, Robert Plant, and Jimmy Page, for their music. Their creative work uplifted me and brought me happiness, especially during tough times.

- My family members for their persistent support. Specifically, I'd like to thank my parents Alamgir and Lily, my sisters Shanta and Nishat, my nephews Arshan and Zabeer, my niece Amayra, and my brother-in-laws for being there beside me.

Nahiyan Alamgir
July, 2024

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

SAT	Boolean Satisfiability
CAS	Computer Algebra System
SHA	Secure Hash Algorithm
MD	Merkle–Damgård
SFS	Semi-free-start
CNF	Conjunctive Normal Form
DPLL	Davis–Putnam–Logemann–Loveland Procedure
CDCL	Conflict Driven Clause Learning
IPASIR-UP	Reentrant Incremental SAT API[1] (User Propagators)
SMT	Satisfiability Modulo Theories

[1]"IPASIR" is a reverse acronym (the reverse of RISAPI).

CHAPTER 1

Introduction

This thesis discusses several methodologies for finding collisions in cryptographic hash functions by augmenting a SAT solver. Our focus is on the cryptographic hash function SHA-256 (and weakened versions of SHA-256), and we investigate how augmenting a SAT solver can be programmed to search for collisions. Specifically, we explore integrating prominent differential cryptanalysis techniques into a SAT solver, which aids the search process and enables the discovery of collisions for less-weakened versions of SHA-256.

Cryptographic hash functions play a vital role in information security. They are widely relied on for data security and integrity. Due to their critical importance, cryptographic hash functions are frequent targets for cryptanalysis. Over time, many once-reliable hash functions have been compromised. Notable examples include MD5 and SHA-1, which were found to be vulnerable to collision attacks as demonstrated by Wang and Yu [58] and Stevens et al. [52], respectively. SHA-1, published by NIST in 1995 [43], showed vulnerabilities soon after the introduction of its successor, the SHA-2 family, in 2001. In 2011, due to identified weaknesses, NIST recommended migrating from SHA-1 to more secure hash functions like those in the SHA-2 family, which have since become widely used. For instance, SHA-256 is employed for transaction signatures and proof-of-work in the Bitcoin protocol [42].

Despite the arrival of SHA-3 [21], NIST still recommends both the SHA-2 and SHA-3 families. SHA-2 is attractive for its ease-of-computation while still being secure to all known attacks—no collision attack has ever been successful on the full version, despite a large number of attempts and partial results. One such attack by Mendel et al. [40] in 2013 utilized differential cryptanalysis for SHA-256. It was inspired by the 2005 work of Wang and Yu [58] that used a differential attack (involving modular integer differences) to find MD5 collisions. Mendel et al. found collisions for step-reduced versions of SHA-256 up to 28 steps and a "semi-free-start" collision (where the hash function is slightly modified to allow changing some predefined constants) of SHA-256 up to 38 steps. These records held for over ten years and were only broken in the last few months with the announcement of a 31-step SHA-256 collision [34] and a 39-step semi-free-start (SFS) SHA-256 collision [33].

Traditionally, the best collisions for step-reduced SHA-256 were found using highly sophisticated tools specifically designed to search for such collisions. Conversely, another line of research examined using satisfiability (SAT) solvers to search for collisions in step-reduced SHA-256, but the results of SAT solvers were not at all competitive with the best custom-written search tools. For example, in 2016, Prokop [46] successfully used a SAT solver to find a collision for 24 steps of SHA-256, but was not able to go higher. In 2019, Nejati and Ganesh [44] pushed this to 25 steps by using a SAT solver that was tuned to do programmatic propagation specifically for the collision-finding problem. However, this was still a long way from Mendel et al.'s 28 step collision or 38 step SFS collision from 2013 [40].

In this work, we develop a hybrid approach of using a programmatic SAT solver that uses a computer algebra system (CAS) to provide the SAT solver information it wouldn't be able to detect on its own—an approach that has been successful on many other problems recently [14]. We encode the collision-finding problem directly into SAT (see Section 5.1) and then programmatically encode several of the mathematical constraints exploited by Mendel et al. [40] that made their 2013 search so effective.

In particular, we are able to detect and block inconsistencies in the solver's state using programmatic inconsistency blocking (see Section 4.1) and are able to deduce nontrivial information about the solver's state using programmatic propagation (see Section 4.2). Our "SAT + CAS" solver was able to find several new 38-step SFS SHA-256 collisions, matching the same step count of the SFS collisions found by Mendel et al. [40], while a pure SAT approach was not able to go any further than 28 steps.

We note that our SAT-based tool is less efficient than the dedicated search tool of Mendel et al. [40] for finding 38-step SFS SHA-256 collisions. However, even though we did not reach the efficiency of a hand-crafted search tool, the novelty of our work is that we show that the efficiency of an off-the-shelf SAT solver can be dramatically improved by exploiting the IPASIR-UP interface. Moreover, the 38-step SFS SHA-256 collisions that we found are of independent interest as they have additional structure not present in Mendel et al. [40]'s 38-step SFS SHA-256 collisions—an additional two internal state variables have zero difference (see Table A.0.2) when compared with Mendel et al. [40]'s collision.

The rest of this chapter aims to introduce cryptographic hash functions (emphasizing SHA-256), differential cryptanalysis, and various types of collision attacks.

In Chapter 2, we discuss the use cases, Boolean formulae (and different other types of formulae), the Tseitin transformation, various types of SAT algorithms, and programmatic SAT.

Chapter 3 highlights progress of attacks on SHA-256 over the years.

Our methodology is discussed in Chapter 4—specifically our programmatic techniques of bitsliced and wordwise propagation.

Chapter 5 presents the results of our experiments, along with a discussion of the implementation and the SAT encoding used.

Finally, the conclusion and future work is discussed in Chapter 6.

1.1 Cryptographic Hash Functions

Cryptographic hash functions take an arbitrary-length input and produce a short fixed-length output that acts as a signature or fingerprint of the input. The fingerprint is called a hash value and the input is known as a message. Hash functions are extensively used for data integrity and security. They are particularly helpful in cases where storing the message would pose a security threat but a signature is still required for verification, such as a password in a database. The hashes of the passwords can be stored instead and each time the user enters their password, the hashes can be matched for verification.

Hashes can be used for data integrity as well. For example, when some data is stored or transferred, the integrity can be checked by comparing a known hash with the hash of the stored data. This integrity check ensures that the data was preserved without alteration.

Cryptographic hash functions are expected to have three primary characteristics:

- Preimage resistance: It's computationally infeasible to find an input, x, given a hash, y, such that y is the hash of x.
- 2nd preimage resistance: Given an input, x, and its hash, y, it's infeasible to find a different input, x', that produces the same hash y.
- Collision resistance: It's infeasible to find an input pair, x and x' ($x \neq x'$), that both produce the same hash.

If any of these characteristics break, the hash function can no longer be used depending on the purpose. For example, if a hash function is no longer preimage or 2nd preimage collision resistant, it cannot be reliably used for ensuring data integrity in applications such as password hashing or file checksums. This is because the hash is no longer unique for a specific input in practice.

An example of a weak hash function is MD4 [55] because it does not have collision resistance—an attacker can easily generate colliding message pairs.

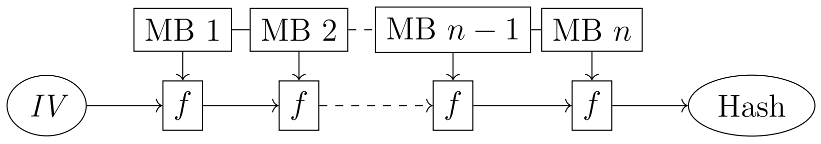

Fig. 1.2.1: SHA-256 processes the input (with padding if needed) into message blocks (abbreviated as “MB”), which are sequentially fed to the compression function, f. The output of each compression is used as the chaining value in the next compression. The compression of the last message block produces the final hash. The initial chaining value (IV) is fixed by the specification of SHA-256. The entire method is known as the Merkle–Damgård construction, which is popular for building collision-resistant hash functions.

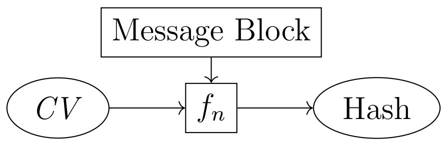

Fig. 1.2.2: A diagram depicting the simplified version of SHA-256 we consider in our work. f_n is the step-reduced compression function having n steps. The chaining value, CV, is arbitrary for semi-free-start (SFS) collisions and is not required to match the IV actually used by SHA-256—though a SFS colliding message pair is required to have matching CVs.

1.2 SHA-256

SHA-256 is a hash function that takes an arbitrary-length input and pads it as necessary to produce one or many 512-bit message blocks. Afterwards, the message blocks are processed iteratively to produce a 256-bit hash. Each message block is processed by a compression function that takes the message block and a 256-bit chaining value as inputs (see Figure 1.2.1).

In the compression function, 64 rounds (also called steps) of transformations are performed to produce a hash. The hash from processing a single message block is used as the chaining value for the next message block. This means that altering a message block will lead to cascading changes in the next message blocks in the sequence. The chaining value for the first message block is set by the specification [18] to a fixed value, known as the standard IV (initialization vector). The hash output is the chaining value produced after applying the compression function on the last message block.

In our work, we focus on a step-reduced version of SHA-256. This means that the number of rounds/steps in the compression function is reduced to make the problem easier. Moreover, we only consider messages with a single block of size 512 bits.

Because the hash output has 256 bits, there is certainly enough freedom in the input so that many collisions exist without needing to consider multiple blocks. A relaxed type of collision known as a semi-free-start (SFS) collision allows an arbitrary initial chaining value CV, so long as the *same* chaining value is used to initialize the hash function for both colliding messages in the SFS collision (see Figure 1.2.2).

In our work, we find SFS collisions for SHA-256 using up to 38 steps of the compression function. Note that the actual SHA-256 hash function has 64 steps, meaning we are still very far from finding a true SHA-256 collision (roughly speaking, as the number of steps increases the collision problem becomes exponentially more difficult). The best known collision attacks on SHA-256 are very far from the full 64 steps, so this provides evidence that SHA-256 is secure.

1.2.1 Message Expansion

SHA-256 performs operations on 32-bit words only. The input message block consists of 16 such words, M_i for $0 \leq i < 16$, but the compression function expands the M_i to more words (dependant on M_0 to M_{15}) to fill up for the rest of the 64 steps. Altogether there are 64 "extended" message words W_i for $0 \leq i < 64$ defined by

$$W_i = \begin{cases} M_i & \text{for } 0 \leq i < 16 \\ \sigma_1(W_{i-2}) \boxplus W_{i-7} \boxplus \sigma_0(W_{i-15}) \boxplus W_{i-16} & \text{for } 16 \leq i < 64 \end{cases} \tag{1}$$

where the functions $\sigma_0(x)$ and $\sigma_1(x)$ are defined as

$$\begin{aligned} \sigma_0(x) &= (x \ggg 7) \oplus (x \ggg 18) \oplus (x \gg 3), \text{ and} \\ \sigma_1(x) &= (x \ggg 17) \oplus (x \ggg 19) \oplus (x \gg 10). \end{aligned}$$

Here $\boxplus$ denotes addition modulo 2^{32}, $\oplus$ denotes bitwise XOR, $\gg$ denotes the right shift operator, and $\ggg$ denotes the right circular shift operator.

1.2.1.1 State Update Transformation

The compression function of SHA-256 takes as input a chaining value and message block and computes a new chaining value by applying 64 iterations of a state update procedure. We describe this state update procedure using equations similar to those presented by Mendel et al. [39]. The expanded message words W_i are used to compute internal state variables T_i, E_i, and A_i through the equations

$$\begin{aligned}
T_i &= E_{i-4} \boxplus \Sigma_1(E_{i-1}) \boxplus \mathrm{IF}(E_{i-1}, E_{i-2}, E_{i-3}) \boxplus K_i \boxplus W_i,\\
E_i &= A_{i-4} \boxplus T_i, \text{ and}\\
A_i &= T_i \boxplus \Sigma_0(A_{i-1}) \boxplus \mathrm{MAJ}(A_{i-1}, A_{i-2}, A_{i-3}).
\end{aligned}$$

Here the functions IF and MAJ are defined on words by applying bitwise the functions from $\mathbb{F}_2^3$ to $\mathbb{F}_2$

$$\mathrm{IF}(x, y, z) = xy + xz + z, \qquad \text{and} \qquad \mathrm{MAJ}(x, y, z) = xy + yz + xz,$$

and the linear functions Σ_0 and Σ_1 are defined by

$$\begin{aligned}
\Sigma_0(X) &= (X \ggg 2) \oplus (X \ggg 13) \oplus (X \ggg 22), \text{ and}\\
\Sigma_1(X) &= (X \ggg 6) \oplus (X \ggg 11) \oplus (X \ggg 25).
\end{aligned}$$

The chaining value is taken to be $[A_{-4}, \ldots, A_{-1}, E_{-4}, \ldots, E_{-1}]$. In other words, the chaining value sets the initial values of the state variables A and E. For example, A_{-4} will be initialized to the first 32-bit word of the chaining value while E_{-1} will be initialized to the last word of the chaining value. K_i is a constant given in SHA-256's specification and there is one unique constant for each step i. The auxiliary variable T_i is introduced to keep the modular additions from having more than 5 addends.

After the state update transformations, the last four A and E words are added with the chaining value to produce a new chaining value, which will be the output of the compression function (and following the final block will be the output of the hash function).

1.3 Types of Collision Attacks

Cryptographic hash functions are supposed to be collision resistant (see Section 1.1). This means that finding two different hash inputs leading to the same hash shouldn't be practically feasible.

While the basic idea of collisions is to find different hash inputs that yield the same hash, there are other constraints that can be enforced or relaxed. For example, the initial chaining value (see Figure 1.2.1) of SHA-256 can be constrained to a fixed value or fully relaxed to take an arbitrary value (see Figure 1.2.2).

The different types of collision attacks are categorized into 3 types:

- (Regular) Collision
- Semi-free-start (SFS) Collision
- Free-start (FS) Collision

Let $f(M, IV) = h$ be a hash function, such as SHA-256, following the MD-scheme (see Figure 1.2.1). The IV is a constant that is fixed by the specification of the hash function for a function like SHA-256, while M is a message of arbitrary size, producing the hash h.

In a (regular) collision, $f(M) = f(M')$, where $M \neq M'$. However, when the initialization vector can take an arbitrary value and is provided as an input to the hash function, finding collisions in this variant of the hash function is called a semi-free-start (SFS) collision. In other words, SFS collision solves for $f(CV, M) = f(CV, M')$ where $M \neq M'$.

To further relax the problem, CV can be allowed to be different in the two hash instances of the collision, as in $f(CV, M) = f(CV', M')$. The variables CV and CV' aren't constrained to be equal. This scenario, where the initialization values are relaxed and the relationship between the values provided to both hash instances is altered, is called a free-start collision.

Solutions for an attack with relaxed constraints (SFS and FS collision) are usually easier to find (as per Mendel et al. [39]), and may be extended to a collision through various techniques. The overall process may be more practically feasible than aiming straight for a collision.

Moreover, when finding collisions for a specific number of steps of a hash function gets practically infeasible, it may still be feasible to find the SFS and FS collisions. Benchmarking on the time and count of SFS and FS collisions may help analyze the magnitude of the difficulty over the reduced number of steps.

1.4 Differential Cryptanalysis

Differential cryptanalysis is a technique that analyzes how the input differences influence the output differences in, for example, a hash function. This technique is crucial in collision attacks of hash functions, since we're interested in studying the diffusion of the input differences to the output differences such that we get a zero output difference and a non-zero input difference.

In differential cryptanalysis of hash collisions, we have two hash inputs and we examine the differences in all the operations until the output for both inputs.

(x, x')	`0`	`u`	`n`	`1`	`x`	`-`	`?`
(0, 0)	+					+	+
(1, 0)		+			+		+
(0, 1)			+		+		+
(1, 1)				+		+	+

Table 1.4.1: This table shows the notation we use for differential conditions in our study. A '+' indicates whether a specific value pair is possible for (x, x'). For example, '`?`' indicates that the variables x and x' can take any value, '`x`' indicates the variables x and x' have distinct values, and '`-`' represents equal values. In the rest of the conditions, the exact values of the variables x and x' are known.

Usually differences between the values are calculated through XOR operations, such as $\Delta x = x \oplus x'$, where x is a single bit, x' is its counterpart in the second hash instance, and Δx is the difference of x and x'.

Hash functions such as SHA-256 operate on bitvectors (also called words) of 32 bits in length. If a Boolean variable representing a bit in one bitvector and its counterpart in the second hash instance form a pair, in general the pair may have up to 4 combinations, $\{(0,0),(0,1),(1,0),(1,1)\}$. The possibilities can be generalized as the differential conditions [20] presented in Table 1.4.1. For example, if a pair (x, x') has the possibilities $\{(0,0),(1,1)\}$, we can describe it as having the differential condition '`-`'.

For convenience, the differential conditions of a pair of words (A, A') can be collectively described in a vector $\nabla A = [c_n c_{n-1} \cdots c_1 c_0]$, where c_i is the differential condition of the ith bit pair (a_i, a'_i) with $A = [a_{n-1} \cdots a_0]$ and $A' = [a'_{n-1} \cdots a'_0]$.

The differential over a function $f(X) = Y$ where X and Y are bitvectors is denoted $\nabla X \rightarrow \nabla Y$. On a high level, we want f to be the hash function while ∇X and ∇Y are the input and output differences represented by differential conditions. In practice, analyzing this differential alone isn't helpful as it contains too little information. We want to study all the operations in between as well—chaining the operations in SHA-256 together as a series of steps starting from the input to the output. If we represent the differences in an operation's input and output values as a differential, we can represent the 2 hash function instances as a chain of differentials. This chain of differentials is called the differential path and analyzing this path shows how the differences propagate from the input differences all the way to the output differences, which is essential for finding collisions.

CHAPTER 2

Boolean Satisfiability (SAT)

Boolean satisfiability (SAT) solving involves searching for a solution of a Boolean formula (an assignment of the variables that makes the formula true). The tools designed for this purpose are called SAT solvers. SAT solving is NP-complete, which means that the solutions to a SAT problem can be verified in polynomial time, but currently there is no known way to solve the problem in polynomial time. Even though all known algorithms for SAT solving run in exponential time in the worst case, in practice many problems can be solved by modern SAT solvers in a reasonable amount of time. In fact, SAT solvers are so effective that in practice there are problems unrelated to logic that are most effectively solved by reducing them to SAT and calling a SAT solver [12].

The beauty of SAT solving lies in its generic nature, meaning that it can be applied to any domain as long as the problem can be encoded into a Boolean formula. This also allows solvers to be tuned for performance independent of any specific problem. Modern SAT solvers are adept at solving search problems, comprising sophisticated techniques like conflict analysis and clause learning, clever branching heuristics, and simplification techniques [8]. The combination allows them to be highly potent at solving search problems in general.

In this chapter, we look into the use cases of SAT, how a problem is encoded into CNF, the core concepts of modern SAT solvers, along with the concepts of programmatic SAT.

2.1 Use Cases

Due to its versatility in both searching for solutions under given constraints and proving the non-existence of solutions, SAT solvers have been proven useful in numerous fields. The following are some of the prominent applications of SAT.

Cryptanalysis. Cryptanalysis involves assessing the properties of crypographic algorithms to find weakenesses. For example, a cryptographic hash function can be analysed to find collisions [39, 40, 46, 44] or preimages [60]. Such problems usually involve searching through a huge space for which SAT solving has been found to be quite useful.

Conflict Detection. Conflict detection has been a popular problem ranging from package managers to compilers. Many package managers have been using SAT solvers to describe the dependencies and find contradictions (see [54, 30]). Many compilers have utilized SAT for finding bugs. Specifically, there are cases where the code can be converted to a Boolean formula that can be verified by the SAT solver (see [29]).

2.2 Boolean Formulae

SAT solving involves a Boolean formula composed of Boolean variables and constraints. The formula describes a problem and the SAT solver is a tool used to find solutions to the problem. If there exists no solution to the given problem, the solver can also prove that there is none. Modern SAT solvers conventionally work with formulas in Conjunctive Normal Form (CNF). Given a CNF formula, a SAT solver tries to find a set of variable assignments such that the formula is satisfied (yields true). CNF, in simple terms, is a conjunction (logical and) of clauses. For a set of n clauses $\{c_1, c_2, \ldots, c_{n-1}, c_n\}$, the CNF can be written as

$$c_1 \wedge c_2 \wedge \cdots \wedge c_{n-1} \wedge c_n = \bigwedge_{i=1}^{n} c_i.$$

Each clause is a disjunction (logical or) of literals, which defines a constraint. For a set of m literals $\{l_1, l_2, \ldots, l_{m-1}, l_m\}$, a clause can be expressed as

$$c_i = l_1 \vee l_2 \vee \cdots \vee l_{m-1} \vee l_m = \bigvee_{i=1}^{m} l_i.$$

Each literal is a Boolean variable or its negation. For example, a literal can be x or $\neg x$ for the Boolean variable x, and a clause can be $x \vee \neg y \vee z$ for the Boolean variables x, y, z.

CNF has been widely adopted by the SAT community because any Boolean expression can be efficiently converted to CNF, and most algorithms used by modern SAT solvers work efficiently with this form (see Section 2.3).

The DIMACS CNF file format. Many SAT solvers use a standardized file format called DIMACS CNF to accept CNF formulas, proposed in the DIMACS Challenge of 1993 [31]. This format provides a textual representation of the CNF formula and serves as an interface to the solver. DIMACS CNF takes a preamble comprising the number of variables (v) and clauses (c):

```
p cnf v c
```

This is followed by all the clauses each in their own line, with each clause terminated by a trailing `0`. Each variable is represented by a numerical ID (starting from 1) with a minus sign denoting a negation.

For example, to encode 2 clauses $(x \vee \neg y \vee z) \wedge (\neg x \vee y \vee z)$, we can write the following DIMACS code with the IDs 1, 2, and 3 representing the variables x, y, and z respectively:

```
 1  2  3  0
-1  2  3  0
```

Comments can also be placed anywhere in the file, even preceding the preamble. These optional lines start with the character "`c`", such as this:

```
c An example comment.
```

There are augmentations to the DIMACS format that are only to be handled by specific SAT solvers or algorithms. For example, CryptoMiniSat supports DIMACS CNF augmented with XOR clauses, meaning that the CNF file can have special clauses for XOR constraints. This is beneficial because it not only makes the problem's encoding easier to read in most cases but also allows the solver to detect and handle the XOR clauses directly, often resulting in an improved performance.

Another interesting augmentation to the DIMACS CNF format involves cardinality constraints, with the augmented format known as KNF [47]. Overall, this augmented format includes a generalization of clauses called "klauses" that define the constraint

$$\sum_{i=1}^{n} l_i \geq k$$

for a set of n literals $\{l_1, l_2, \ldots, l_{n-1}, l_n\}$ and a lower bound k. A KNF solver implementation, known as "Cardinality CDCL", includes a modification of the SAT solver CaDiCaL to provide native support for handling cardinality constraints (see their GitHub repository[1] for more details). The project also has support for extracting

[1] https://github.com/jreeves3/Cardinality-CDCL

klauses from DIMACS CNF encodings and also converting KNF encodings to DIMACS CNF.

2.3 Tseitin Transformation

To solve a problem using SAT solving, it must first be expressed in Conjunctive Normal Form (CNF), the format required by most modern SAT algorithms. A propositional logic formula can be transformed into CNF by applying the rules of Boolean Algebra [8]. However, this process often results in an exponential increase in the size of the resulting formula. While encoding a logical formula into CNF in a scalable manner may initially seem challenging, the Tseitin transformation [53] offers an efficient method to accomplish this. This process converts any problem into a formula in CNF that is satisfiable if and only if the original formula is.

Tseitin transformation, introduced in 1968 by Russian scientist Grigori Tseitin [53], is a technique for converting non-CNF formulas into CNF. It does this by breaking down the original formula into smaller subformulas and introducing auxiliary variables that are equivalent to these subformulas, meaning that they take the same truth values. For convenience, we use the “equivalence” operator ($\leftrightarrow$). The resulting CNF is a conjunction of these subformulas, each represented by an auxiliary variable. While this method increases the number of variables, it ensures that the encoding process scales linearly with the number of operations in the original expression.

Consider the logical expression that involves 4 operations:

$$(p \vee q) \wedge \neg(p \wedge q).$$

The expressions can be broken down into subexpressions, each equivalent to the newly introduced variables x, y, and z respectively:

$$\begin{gathered} x \leftrightarrow (p \vee q), \\ y \leftrightarrow (p \wedge q), \text{ and} \\ z \leftrightarrow \neg y. \end{gathered}$$

The subexpressions can then be expressed as the CNF clauses

$$\begin{gathered} (\neg p \vee x) \wedge (\neg q \vee x) \wedge (p \vee q \vee \neg x), \\ (p \vee z) \wedge (q \vee z) \wedge (\neg p \vee \neg q \vee \neg z), \text{ and} \\ (y \vee z) \wedge (\neg y \vee \neg z). \end{gathered}$$

Altogether, Tseitin transformation gives us the CNF formula

$$x \wedge z \wedge (\neg p \vee x) \wedge (\neg q \vee x) \wedge (p \vee q \vee \neg x) \wedge (p \vee z) \wedge (q \vee z) \wedge (\neg p \vee \neg q \vee \neg z) \wedge (y \vee z) \wedge (\neg y \vee \neg z),$$

which is equisatisfiable to the original formula (satisfiable if and only if the original formula is satisfiable).

Overall, the Tseitin transformation is a technique that converts a given logical formula into an equisatisfiable clausal form. This transformation operates with a time complexity of $O(n)$, where n represents the number of operations in the original formula. It achieves this at the cost of a linear increase in the number of variables, proportional to the number of operations in the original formula.

2.4 The DPLL Algorithm

Most modern SAT algorithms are tailored to handle CNF formulas. One of the most significant CNF-based SAT solving algorithms is the DPLL (Davis–Putnam–Logemann–Loveland) algorithm [19], widely recognized as the core of modern SAT solvers. It was introduced in 1962 as an improvement over the Davis–Putnam (DP) procedure. See [3] for more details on the implementation of the DPLL algorithm.

The DP algorithm. The Davis–Putnam procedure (predecessor of DPLL) is based on the resolution rule for variable elimination.

The resolution rule says that two clauses can be merged when one clause has a variable x while the other clause has the negation of that variable, $\neg x$. For example, if we have the following clauses containing x and $\neg x$ respectively,

$$p_1 \vee p_2 \vee \cdots \vee p_{n-1} \vee p_n \vee x$$
$$q_1 \vee q_2 \vee \cdots \vee q_{m-1} \vee q_m \vee \neg x$$

these clauses can be reduced to

$$p_1 \vee p_2 \vee \cdots \vee p_{n-1} \vee p_n \vee q_1 \vee q_2 \vee \cdots \vee q_{m-1} \vee q_m$$

as per the resolution rule. The resultant clause doesn't include x and $\neg x$.

The idea behind the DP algorithm is to remove trivially satisifiable clauses, deduce variables and remove clauses, and eliminate variables through resolution. This process continues until an empty clause is found (indicating that the problem is unsatisfiable) or if the entire clause set is empty (indicating a satisfiable solution) during any of the steps.

The DP algorithm works by sequentially performing the following steps:

- **Remove trivially satisfiable clauses**: Any clause that has a variable x and its negation $\neg x$ is trivially satisfiable and hence removed.
- **Deduce variables and remove clauses**: If a literal x appears in one or more clauses and $\neg x$ is absent from the entire clause set, the variable x can be set to true to satisfy those clauses. Similarly, if only $\neg x$ appears in the entire clause set, x should be set to false to satisfy the related clauses. Once the clauses are satisfied, they can be removed.
- **Variable elimination through resolution**: Pick a variable that appears positively in a set of clauses and negatively in another set of clauses (assuming that the sets are non-empty). Perform resolution between all the clauses in both the sets. The resulting clauses are added while the clauses in the two sets can be removed. At the end, the chosen variable is eliminated.

The problem with the DP algorithm is that the number of clauses can increase exponentially in the number of variables, which leads to high memory usage over time.

Boolean Constraint Propagation (BCP). To overcome the shortcoming of the DP algorithm, the DPLL algorithm has been introduced which utilizes a technique called Boolean Constraint Propagation (BCP) [59]. BCP ensures that variable elimination and clause removal can be efficiently done without increasing the clause set size.

BCP takes advantage of unit clauses, which are clauses with only one literal. The idea that unit clauses constrain a variable to a fixed value is useful in satisfying many clauses and removing them in the process.

For example, if BCP is performed on a clause set with a unit clause $\{u\}$ (let a set of literals represent a clause), all the literals $\neg u$ can be removed from all the clauses. On the other hand, all the clauses containing the literal u are satisfied and can be removed. This eliminates the variable u from the entire clause set. Another way to look at BCP is that the deduced information from the unit clauses propagates to the other clauses, satisfying many of them in the process.

Since more unit clauses could have been introduced through BCP, the process has to be repeatedly performed till there's no more unit clauses remaining in the clause set.

Overview of DPLL. Another key difference from the DP algorithm is the "branching" technique. Unlike DP which performs resolution till there's no variable left, BCP doesn't eliminate all the variables. As a result, DPLL has to ground a variable after BCP to progress, meaning that it has to choose a variable and decide to set it to true or false. It's done by adding a unit clause $\{x\}$ or $\{\neg x\}$ for a chosen variable x.

Overall, DPLL works in the following steps performed sequentially:

- **BCP**: Deduce variables from unit clauses and remove satisfied clauses.
- **Branch**: Pick a variable x and decide to set it to true or false by recursively calling the DPLL algorithm with a new clause $\{x\}$ or $\{\neg x\}$.

The entire process is repeated until the entire clause set is empty (SAT) or has an empty clause (UNSAT). Even if one of clauses is UNSAT, it doesn't mean that the SAT problem is UNSAT. The overall problem is deemed UNSAT only if it's UNSAT for both branches of every variable.

Despite BCP not increasing the size of the clause set, a major downside of DPLL is the depth of the recursion and the number of recursive calls made, which will be 2^n in the worst case.

2.5 The CDCL Algorithm

In 1996, Silva and Sakallah [50] introduced the concepts of Conflict-Driven Clause-Learning (CDCL) and non-chronological backtracking. These innovations, involving the concept of learning conflict clauses and backjumping in the search space greatly revolutionized SAT solving.

CDCL resolves a shortcoming of the DPLL algorithm, which is that the search is done in a rigid and inflexible manner for no good reason. However, CDCL retains key components of DPLL, such as Boolean Constraint Propagation (BCP) and branching.

The CDCL algorithm enhanced SAT solvers by providing increased freedom to explore the search space while also avoiding areas previously determined to be unsatisfiable. Whenever a conflicting state is reached, the algorithm identifies and learns the cause of the conflict as a reason clause, ensuring it avoids the same or similar conflicts in the future. This is because many areas of the search space lead to a conflict for the same reason.

The algorithm includes non-chronological backtracking, which allows undoing multiple conflict-related decisions simultaneously. This approach enables the solver to focus on more promising areas of the search space away from unsatisfiable regions.

Trail. In most CDCL implementations, a trail is maintained to track the decisions and propagatons. Each decision increments the "decision level," which decrements when a decision is undone through backjumping. The trail organizes decision and propagation literals by decision levels. Within each decision level, the decision literals along with the propagated literals implied by that decision are stored.

Implication graph. The implication graph is a directed acyclic graph that keeps track of the consequences of variable assignments. The root nodes represent decisions (nodes with no incoming edges) while the propagated literal nodes have at least one incoming edge. The parent nodes of a propagated literals node are the assignments that led to the propagation; in other words, these literals form the "antecedent" of the propagated literal. Any conflict caused by a set of assignments is also added to the graph as a node. Other information such as the decision level of each node is also stored.

Overall, the implication graph is crucial for generating better conflict reason clauses, and backjumping to a level where the conflict can be resolved.

Learning conflict clauses. A simple way to learn a conflict clause is to simply construct a reason out of all the decision literals that led to the conflict. For example, if the decision literals $x, \neg y$, and z lead to a conflict, the reason clause would be $\{\neg x, y, \neg z\}$, meaning that at least one of our decisions was incorrect and blocking the simultaneous assignment of the decisions.

The propagated literals aren't part of the conflict reason clause because these literals are implied by the decision literals. The exclusion of the propagated literals keeps the reason clause concise but still long if there were many decision levels existing during the conflict. A shorter and compact conflict reason clause can be constructed by analyzing and picking the decision literals directly responsible for the conflict, as not all the decision literals are necessarily responsible for the conflict.

In general, a shorter clause is easier for the SAT algorithm to process and is more powerful, as it describes the same constraint with fewer literals than an equivalent longer clause.

Backjumping. After a conflict clause is learned, the conflict is resolved by backjumping to the decision level where the conflict can be effectively resolved. Although backjumping by only one level is sufficient to address the conflict, it is not ideal in practice. To both resolve the conflict and explore more promising areas of the search space, the solver should backtrack to the farthest decision level where the conflict clause becomes a unit clause under the current partial assignment. In other words, the conflict clause must be asserting after the backjump and capable of performing unit propagation.

2.6 Programmatic SAT

Programmatic SAT involves injecting code into the solver to aid the solver and solve a problem more efficiently [27]. It is particularly useful when certain aspects of the problem are challenging to express in the Boolean formula. Moreover, solving parts of the problem in high-level code might be easier to implement and faster to solve programmatically (see [11, 14, 61]). For example, algorithms for solving a subproblem can be written in high-level languages like C/C++, which may be more efficient than using a SAT solver to handle the same subproblem expressed through a Boolean formula.

Programmatic SAT is usually domain-specific and combines the powerful techniques of search possessed by SAT solvers with the most efficient high-level algorithms and analysis tools for the problem. Some modern SAT solvers can be customized in a programmatic way through built-in interfaces like IPASIR-UP [25]. The solver may be aided through the following ways during search:

- **External propagation:** Assigning variables derived through high-level deduction.
- **External decisions:** Making decisions on picking important unassigned variables and guessing their values through high-level analysis.
- **External learning:** Injecting learned clauses during conflicts detected through the high-level conflict analysis.

Overall, the external routines function as an oracle for the SAT solver, providing information that either cannot be derived from the Boolean formula or can be obtained much faster than through conventional SAT solving techniques.

The SC^2 (Satisfiability Checking and Symbolic Computation) Project. SAT solving, ever since its inception, has been found to be useful for solving general search problems. On the other side, Computer Algebra Systems (CASs) include algorithms that are efficient at solving mathematical problems. However, many problems exist that can be solved using the methods of both satisfiability checking and symbolic computation. Bridging the two fields was proposed in 2015 in the work of Ábrahám [1] and Zulkoski et al. [62].

Shortly afterwards, the SC^2 project [2] was initiated to support the joint community. Since then, a wide variety of problems have been tackled using SC^2 techniques—from circuit verification [32], knot theory [37], graph theory [35], projective geometry [9], quantifier elimination and cylindrical algebraic decomposition [16], searching for combinatorial sequences [15] and matrices [13], and factoring integers with known bits [4]. See England's survey [24] for an overview of many other examples.

CHAPTER 3

Related Works

Cryptographic hash functions such as MD4, MD5, SHA-1, etc. have been extensively relied on for information security for many years. However, Wang et al. [55] devised an efficient method in 2005 for finding MD4 collisions with probability from 2^{-2} to 2^{-6} using at most 256 MD4 hash operations. Wang et al. also proposed an attack on MD5 [58] for finding collisions within 15–60 minutes of computational time in the same year. Also in 2005, Wang et al. [56] presented a collision attack on SHA-1 using at most 2^{69} SHA-1 hash computations.

3.1 Progress on SHA-256 Attacks

The SHA-2 family of hash functions, however, survived these remarkable attacks, likely due to their relatively complex design with message expansion. One of the earliest attacks on SHA-256 and its family members was in 2003 by Gilbert and Handschuh [28]. In FSE 2006, Mendel et al. [41] reported that the message expansion of the SHA-2 family of hash functions was one of the key points for their increased collision resilience over SHA-1. To tackle this, Mendel et al. applied a message modification technique and reached an 18-step collision for SHA-256. In INDOCRYPT 2008, Sanadhya and Sarkar [49] presented collisions up to 24 steps of SHA-256 and SHA-512, making improvements over the work of Nikolić and Biryukov [45] that presented collisions up to 21 steps of SHA-256 at FSE 2008.

In ASIACRYPT 2011, Mendel et al. [39] revealed a collision for 27-step SHA-256 and a semi-free-start (SFS) collision for 32 steps. They automated the search with a domain-specific tool that searches for differential characteristics for SHA-256. The tool utilizes propagation, analysis of the bit constraints, clever branching on the most constraints bits, and contradiction detection in the differential characteristics.

In EUROCRYPT 2013, Mendel et al. [40] came back with another breakthrough—a 28-step collision of SHA-256 along with a 38-step SFS collision. They further improved the automatic search tool that finds differential characteristics. The improvements included local collisions over a larger number of steps and improved decision/branching

Publication Year	Author	Collision	SFS Collision
2006	Mendel et al. [41]	18	-
2008	Sanadhya and Sarkar [49]	24	-
2011	Mendel et al. [39]	27	32
2013	Mendel et al. [40]	28	38
2024	Li et al. [34, 33]	31	39

Table 3.1.1: Progress of step-reduced SHA-256 collision attacks (including SFS collisions) from 2006 to 2024. The entries in the table indicate the number of steps for which the collisions (or SFS collisions) were found.

heuristics over [39].

Very recently, in the rump session of FSE 2024, Li et al. [34] announced a 31-step collision of SHA-256, and in a 2024 paper appearing in EUROCRYPT 2024 found a 39-step SFS collision [33]. These works also made an advancement in cryptanalysis with SAT solving, searching for characteristics by controlling the sparsity (number of variables with no difference).

The progress of the attacks on SHA-256 is presented in Table 3.1.1.

CHAPTER 4

Methodology

In this chapter, we present the methodology used in our SHA-256 collision attack. Our work is based on interacting with a SAT solver through a programmatic interface. Using differential crypanalysis techniques, we aid the solver with finding collisions. Specifically, we check for inconsistencies (see Section 4.1) and perform local (see Section 4.2.1) and global (see Section 4.2.2) propagation. The techniques allow us to find SFS collisions with a significantly higher number of steps than that with a plain SAT solver.

4.1 Programmatic Inconsistency Blocking

As discussed in Section 1.4, analyzing differential paths is essential in cryptanalysis. There are cases when a differential path has inconsistencies. In other words, parts of the differential path define a relation contradicting a relation defined by other parts of the differential path. For example, if we can derive the conditions $a = b$ and $a \neq b$ from differential conditions in the same path, then there certainly cannot exist any message pairs conforming to that path. During solving, it's crucial to analyze the current path for such inconsistencies and block them as early as possible to prevent the solver from exploring paths that are inconsistent.

The idea of looking for and blocking inconsistencies in the SHA-256 collision attack was utilized by Mendel et al. [39]. They described having linear equations relating two Boolean variables in SHA-256's state. Each of these equations can be derived from bitsliced differentials of bitwise functions and modular addition. Such relations can lead to conditions on the equality or inequality of two variables. Mendel et al. [39, 40] referred to these conditions as "two-bit conditions".

Two-bit conditions can be derived from bitsliced differentials of bitwise functions and addition operations. To deduce the two-bit conditions from a bitsliced differential, we enumerate all the possibilities and look for a pattern. As an example, consider the differential $\nabla[x_2 x_1 x_0] \to \nabla[y_0]$ of the XOR operation $x_0 \oplus x_1 \oplus x_2 = y_0$. If the differential is specifically `[-0-]` $\to$ `[0]`, it means that $(x_2, x_0) \in \{(0, 0), (1, 1)\}$ giving

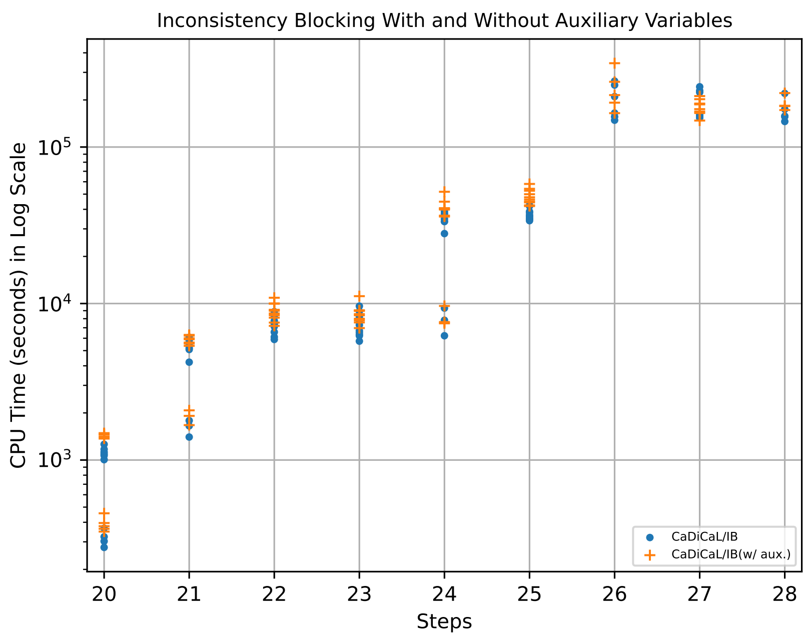

Fig. 4.1.1: The plot highlights the runtime of programmatic inconsistency blocking with and without auxiliary variables. Each instance is about finding SFS collisions in step-reduced SHA-256.

us the two-bit condition $x_2 = x_0$.

In practice, two-bit conditions are significantly more common in the bitsliced differentials of bitwise functions than that of addition operations. Thus, in our experiments, we only computed the two-bit conditions of these bitwise functions to reduce computational costs. Additionally, there are two-bit conditions involving Boolean variables other than that of A, E, and W. For example, two-bit conditions often involve the output variables of bitwise functions along with other auxiliary variables. However, we did not find it beneficial to address inconsistencies involving two-bit conditions of these auxiliary variables (see Figure 4.1.1). As a result, we focused solely on the two-bit conditions involving the primary variables to block inconsistencies.

The two-bit conditions are expressed in the form $x \oplus y = z$ where x and y are variables in the differential path and $z \in \{0, 1\}$. For example, the two-bit condition $x \neq y$ gives the equation $x \oplus y = 1$. The set of these linear equations often lead to inconsistencies that are non-trivial. For example, if $a = b$, $b = c$, and $c \neq a$, we have a contradiction involving the 3 two-bit conditions and can be visualized as a cycle $a = b = c \neq a$.

Such cycles of inconsistencies translate to cycles of inconsistent differentials, which in turn would are blocked to direct the search away from an invalid differential path.

To do this efficiently we employ a custom-written computer algebraic routine to detect cycles of inconsistent equations during solving. In particular, we use a graph for finding inconsistent cycles, where in the graph each vertex represents a variable and each edge represents a two-bit condition. Every time a new edge is added to the graph, we search for an inconsistent cycle involving that edge (and the shortest such cycle when one exists).

The graph algorithm we use for detecting inconsistent cycles involves a breadth-first search starting from vertex v_0 where (v_0, v_d) is a newly added edge. We look for all possible ways to reach v_d excluding the edge (v_0, v_d). Each edge (u, v) holds a Boolean variable $d(u, v) = u \oplus v$, called an edge value, which tells whether the Boolean variables u and v are equal or not.

For each path from v_0 to v_d found through the method described above, we get a cycle v_0, v_1, …, v_d, v_0 by adding the edge (v_0, v_d) to the path. We check if there's a contradiction in a cycle (connecting Boolean variables v_0 to v_n) by taking the $\mathbb{F}_2$ sum of all the edge values, $s = d(v_0, v_1) + d(v_1, v_2) + \cdots + d(v_{d-1}, v_d) + d(v_d, v_0)$. If the sum s is 1, there is an odd number of edges with inequal variables, indicating an inconsistent cycle. We iterate through the inconsistent cycles and take the shortest one for blocking.

When an inconsistency is detected it is blocked by adding a conflict clause constructed from the parts of the SAT solver's partial assignment (during the time of detection) implying the 2-bit conditions in the cycle. The IPASIR-UP interface [25] is used for feeding the new clause to the solver. The falsified clause causes the solver to backtrack right away, stepping out of the invalid differential path causing the solver to backtrack earlier than it otherwise would.

4.2 Programmatic Propagation

During the search for a collision, we work with a partial state that comprises known and unknown variables. Using the known variables, unknown variables may be derived. In other words, the information that we have can spread or propagate. This form of deduction is crucial in the search process. As mentioned in Section 5.1, many propagation rules such as `[xx-]` $\rightarrow$ `[-]` for the XOR function are encoded directly into the SAT encoding. However, it is not feasible to encode all possible propagation rules on 32-bit words because there are simply too many.

Ideally, one would use "perfect" propagation encoding the most stringent conditions possible given the current state. For example, we describe a simple example of perfect propagation given by Eichlseder [22, Ex. 3.4]. Suppose $X = [x_3x_2x_1x_0]$ is a 4-bit word, $\Sigma(X) = (X \ggg 1) \oplus (X \ggg 2) \oplus (X \ggg 3) = Y$, and we want to perform perfect propagation on the differential $\nabla X \to \nabla Y$. If the differential ∇X is known to be `[11--]`, then there are $2^2 = 4$ possibilities for (X, X') because $x_2 = x_2' \in \{0, 1\}$ and $x_3 = x_3' \in \{0, 1\}$ may be chosen independently. After trying all 4 possibilities one derives that $(\Sigma(X), \Sigma(X')) = (Y, Y')$ must be one of $(1100, 1100)$, $(0010, 0010)$, $(0001, 0001)$, or $(1111, 1111)$. In each case we have $Y = Y'$ meaning that we can derive that $\nabla Y =$ `[----]`.

Since all possibilities for "grounding" ∇X were explored, the maximum amount of possible information was propagated to ∇Y and this is said to be "perfect" propagation. Unfortunately, in general perfect propagation in infeasible because there are too many possibilities to explore.

4.2.1 Bitsliced Propagation

Perfect propagation is only feasible for small differentials with a small number of possibilities to explore. However, SHA-256 performs operations on 32-bit words, which means that every function operates on 32-bit words as input. If we want to propagate the output for a function, we'd have to deal with a relatively large number of bits.

To keep the process computationally feasible, we only perform perfect propagation on the output of a bitwise operation in each bit position independently. This reduces the number of bits involved in the propagation while still helping to deduce information. Each output condition is propagated by enumerating all possibilities conforming to the input conditions that the output condition is dependent on—the same as perfect propagation, but only perfect locally. This is a practical version of perfect propagation called "bitsliced" propagation.

For example, suppose we have $X \boxplus Y = Z$ and we want to propagate $\nabla X =$ `[x-x-]` and $\nabla Y =$ `[x---]` to ∇Z. We will focus on propagating information for the second-least significant bit; the conditions of this bitslice are enclosed in boxes in the depiction below:

$$\nabla Z = \begin{array}{rr} & \mathtt{?}\boxed{\mathtt{?-}} \\ & [\mathtt{x-}\boxed{\mathtt{x}}\mathtt{-}] \\ \boxplus & [\mathtt{x-}\boxed{\mathtt{-}}\mathtt{-}] \\ \hline & [\mathtt{??}\boxed{\mathtt{?}}\mathtt{-}] \end{array}$$

In the example above, the wordwise addition (modulo 2^4) involves 2 addends with the differential conditions `[x-x-]` and `[x---]`, and the first row denotes the differential conditions of the carries. In general the bitslices involve 3 input conditions and 2 output conditions (namely, a sum differential bit and a carry differential bit).

The conditions are derived through perfect propagation on each bitslice—in this case the slice having a width of 1 bit.

In this example, the highlighted bitsliced differential `[-x-??]` (with the last `?` denoting the carry) after propagation becomes `[-x-x?]` (i.e., the sum differential bit becomes an `x`). This process of bitwise propagation can be repeated for the rest of the bit positions, resulting in propagation over a wordwise operation with a low cost.

4.2.2 Wordwise Propagation

SHA-256's hash output is calculated by a series of Boolean operations on 32-bit words. Each step involves the state update equations of Section 1.2.1.1 that are used for transforming the state variables A and E. We also have the message expansion equation (1) defining W_i for all steps $i \geq 16$. All these equations involve modular additions and therefore to effectively search for collisions it is essential to have effective propagation for the modular additions. Bitsliced propagation is helpful in deriving information for modular additions, however, this technique doesn't capture all the relations between the bits as it is local to a bitslice and doesn't operate on the entirety of the 32-bit words.

To mitigate this shortcoming of bitwise propagation, we utilize a global "wordwise" propagation technique, that is significantly cheaper than perfect propagation on words pairs in practice but typically derives more information than bitwise propagation.

Wordwise propagation works by exploiting the constraints in the modular addition, such as the modular integer differences of words. Modular word differences were also used in the work of Wang and Yu [58] for performing the differential attacks.

When $A \boxplus B = C$ and $A' \boxplus B' = C'$, wordwise propagation may derive additional information on the differential conditions ∇A and ∇B if the modular difference of C and C' is known.

Denoting modular subtraction of two 32-bit words by $\boxminus$, the modular difference of C and C' is

$$\delta C \coloneqq C \boxminus C' = \sum_{i=0}^{31} (c_i - c_i')2^i \mod 2^{32} \tag{1}$$

where c_i and c_i' denote the ith least significant bits of C and C'.

In the previous example, the modular addition equations in both the hash instances can be combined via $(A \boxminus A') \boxplus (B \boxminus B') = C \boxminus C'$ which can be rewritten as $\delta A \boxplus \delta B = \delta C$.

In general, wordwise propagation is performed on equations like

$$\delta A_1 \boxplus \delta A_2 \boxplus \cdots \boxplus \delta A_n = C \tag{2}$$

where C can be determined in advance and we want to derive some additional information on at least one of the differential conditions ∇A_1 to ∇A_n. In practice, we propagate for equations of the form (2) with at most 2 word pairs with unknown modular differences. This limits the computational complexity while allowing us to deduce sufficient information.

For example, suppose we know $\delta A = \delta B$, $\nabla A =$ `[ux-]`, and $\nabla B =$ `[-n-]`. It follows that $\delta B = 0 \cdot 2^2 + (0 - 1) \cdot 2 + 0 = -2$ is known (modulo 8), but $\delta A = (1 - 0) \cdot 2^2 + (a_1 - a_1') \cdot 2 + 0 = 4 \pm 2$ is either 2 or 6 since $a_1 - a_1' = \pm 1$. From ∇A alone the value of δA cannot be determined exactly, but when the additional constraint $\delta A = \delta B$ is considered it is clear that the only solution is $\delta A = 6$ meaning that $a_1 - a_1' = 1$. Thus, wordwise propagation in this case would derive $\nabla A =$ `[uu-]`.

To avoid dealing with negative numbers, the differential conditions `x`, `n`, and `?` are normalized by adding an appropriate power of two. For example, in the above example 2^1 would be added making the equation

$$(1 - 0) \cdot 2^2 + w \cdot 2^1 + 0 = -2 + 2^1 = 0 \qquad \text{where } w \coloneqq a_1 - a_1' + 1 \in \{0, 2\}$$

becoming $(1 + v) \cdot 2^2 = 0$ where $v \coloneqq w/2 \in \{0, 1\}$. As a 3-bit bitvector equation (hence performed modulo 2^3), this is $[1 + v, 0, 0] = [0, 0, 0]$ which has just one solution $v = 1$.

Tracking the carries. After reducing equations of the form (2) to bitvector equations, we want to determine all solutions for the variables. However, for multi-word modular addition, we need to track the carries and include them in the bitvector equations.

For example, let's consider $\delta A \boxplus \delta B = C$ where

$$\begin{aligned} \nabla A &= \texttt{[---------x--]}, \\ \nabla B &= \texttt{[??????-??x--]}, \text{ and} \\ C &= 0. \end{aligned}$$

The problem is normalized by adding appropriate powers of 2 for the conditions `x` and `?` in ∇A and ∇B. Specifically, we add:

$$2 \cdot 2^2 + 2^3 + 2^4 + 2^6 + 2^7 + 2^8 + 2^9 + 2^{10} + 2^{11} = 4064$$

to ensure there are no negative difference for any of the conditions.

After normalization and adding the difference variables along with the carries, we obtain $\delta A \boxplus \delta B \boxplus 4064 = 4064$ generating the bitvector modular addition problem

$$\begin{array}{r}
[\ v_8,\ v_7,\ v_6,\ v_5,\ v_4,\ v_3, v_2,\ v_1,\ v_0,\ 0, 0, 0\ ] \\
[\ v_{15}, v_{14}, v_{13}, v_{12}, v_{11},\ 0,\ c_1, v_{10},\ v_9,\ 0, 0, 0\ ] \\
\boxplus\ [\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ 0,\ c_0,\ v_{16}, 0, 0, 0\ ] \\
\hline
1\ \ 1\ \ 1\ \ 1\ \ 1\ \ 1\ \ 1\ \ 0\ \ 0\ \ 0\ 0\ 0
\end{array}$$

where $v_0, \ldots, v_{16}, c_0, c_1 \in \{0, 1\}$.

The carry variable c_0 was added since $v_0 \oplus v_9 \oplus v_{16} = 0$ has the solutions $\{(0,0,0), (0,1,1), (1,1,0), (1,0,1)\}$ for (v_0, v_9, v_{16}), 3 of which can induce a carry. There are cases when a carry is ruled out, such as that for $v_2 \oplus c_1 = 1$, which has the solutions $\{(0,1), (1,0)\}$ for (v_2, c_1). If $v_2 \oplus c_1$ was 0 instead, a carry variable would have to be added to the next column.

Unit propagation. Once we've reduced the problem to bitvector equations and tracked the carries, the first step in solving for the variables is to perform unit propagation. In this step, columns in the bitvectors that contain only a single variable can be easily resolved. Since this process may result in more columns having a single variable, we repeat it until no such columns remain. In the above example, we deduce that $v_3 = 1$ in the first cycle of unit propagation.

Note that a `?` can have a difference of 0, 1, or 2 (if it's `n`, `-`, or `u` respectively) after normalization, so two Boolean variables are required to define its difference. During propagation, we can deduce the higher bit of the difference of `?` to be 0 if the lower bit is known to be 1, since its difference can't be 3. In this case, $\nabla B[6]$, whose difference is defined by v_3 and v_4, is propagated to `-`. The subsequent cycles of unit propagation deduce $\nabla B[7]$, …, $\nabla B[11]$ using the same logic.

Brute-force attack. After unit propagation, possibilities for the remaining unknown variables can be determined through a brute-force attack. Specifically, this involves enumerating all possible solutions for each column, given the constraints imposed by the chained columns. For example, if $c_0 = 0$ is known, then $(v_0, v_9, v_{16}) = (0, 0, 0)$ would be the only possible solution. However, in the above example, none of the remaining variables could be derived through a brute-force attack. Similar to unit propagation, we repeat the brute-force attack whenever a variable is deduced, as it may introduce new constraints for other columns. At most, we will have $2^6 = 64$ iterations for each column, as we restrict the number of addends in the modular addition equation to 2. This is because in a single column, we will have no more than 6 variables (4 difference variables + 2 carry variables). For example, in the most extreme case, a column can have 2 low-order difference variables from 2 '`?`', 2 high-order difference variables from 2 '`?`', and 2 carry variables.

Implementation details. Wordwise propagation is applied to all the modular addition equations, specifically the message expansion (1) and state update transformation equations, including the one for the auxiliary word T_i. However, the only variables that are propagated are the differential variables in ∇A_i, ∇E_i, and ∇W_i. This is because propagating the other (auxiliary) variables was found to hurt the solver's performance.

During the wordwise propagation routine, a heuristic that we used which we found dramatically improved the efficiency of the solver was to assume that any differential '`?`' in the auxiliary variables (including ∇T_i and the differential variables corresponding to the output of IF, MAJ, σ_0, Σ_0, etc.) was actually a '`-`' differential.

In practice, making this assumption allowed the modular difference of the auxiliary differential words to be calculable much more frequently, and increased the likelihood that variables in the word differentials ∇A_i, ∇E_i, and ∇W_i were derived. This heuristic is related to the 'decision' search strategy of Mendel et al. [39] which always first imposes a '`-`' for a '`?`' before imposing a '`x`'. However, we found that making assumptions on the primary word differentials ∇A_i, ∇E_i, and ∇W_i themselves significantly decreased the solver's performance, preventing us from finding SFS collisions for SHA-256 beyond 28 steps.

Updates since the SC^2 publication. Although the core concept of exploiting known modular differences remains unchanged, various improvements have been made to address some shortcomings. Previously, we divided the problem into subproblems and brute forced on the variables in the subproblem. This limited the brute-force attack to subproblems with a small set of variables. Our new form of brute-force attack no longer enumerates solutions for more than a single column while still deducing the same amount of variables for a given problem, if not more. Moreover, we do iterative unit propagation to deduce variables before any brute-force attack. In many cases, unit propagation, which is a cheaper technique, proved to be sufficient for deducing all the variables. There is also an update to the propagation of `?` conditions where we utilize a known lower-order difference bit to deduce a higher-order bit, and vice versa. Altogether the changes reduced the compuational complexity and also resulted in propagation of more conditions in differentials.

CHAPTER 5

Experiments and Results

Our programmatic SAT + CAS solver was implemented in CaDiCaL 1.8.0 [7] using the programmatic interface IPASIR-UP [25]. Our experiments were run in the Digital Research Alliance of Canada's [6] Narval cluster. Each SAT solver instance ran on a single core of an AMD Rome 7532 processor running at 2.4 GHz with 4 GiB of RAM. Our implementation is free software and is available online.[1]

5.1 The SAT Encoding

In our problem, we want the hashes in the two blocks to be the same while having a similar message pair. We use SAT solvers as a search tool for the collision attack. Our SAT formula comprises the encoding of two blocks of SHA-256 that we are aiming to find collisions for. For each block, the formula includes an n-step compression function that takes a 512-bit message block and a 256-bit chaining value, and then computes a 256-bit hash. The number of steps/rounds, n, is adjusted to generate a step-reduced version of the hash function.

Encoding the compression function includes bitwise Boolean functions such as IF and MAJ. The other functions, σ_0, σ_1, Σ_0, and Σ_1, boil down to 3-operand XOR functions after circular rotations and shifts. For each 3-bit XOR $a \oplus b \oplus c$, our encoding produces $x \leftrightarrow a \oplus b \oplus c$ (where x is an auxiliary variable) using $2^3 = 8$ clauses. An auxiliary variable is introduced for every gate in the circuit similar to how the Tseitin transformation [53] is performed (see Section 2.3).

The 32-bit modular addition is encoded as bitslices, where each bitslice involves at most 7 addends (including carries) and a 3-bit output (a high carry, a low carry, and a sum). The addition encoding is taken from the work of Nejati and Ganesh [44], which used the Espresso logic minimizer.

On top of the two hash function instances, we have the differential cryptanalysis layer. Each Boolean variable in one instance, say x, has a counterpart x' in the other instance. For the analysis of the differences as per differential cryptanalysis, we encode

[1] https://github.com/nahiyan/cadical-sha256

the bitwise differences as $\Delta x \leftrightarrow x \oplus x'$ (following Nejati and Ganesh [44]) where $\Delta x \in \{0, 1\}$ is a new auxiliary variable. Each triple $(x, x', \Delta x)$ defines a differential condition. For example, $(x, x', 1)$ defines an '`x`' while $(1, 0, \Delta x)$ defines a '`u`' (see Table 1.4.1 for the complete list of differential conditions).

The core ideas behind this SAT encoding are taken from the work of Prokop [46], which also uses bitwise XOR differences. The implementation of this SAT encoder utilizes a framework[2] for building SAT encoders.

A naive way to constrain collisions is to have zero differences in the hash pair while maintaining at least one difference in the message pair. However, we want to analyze all the differences between the 2 hash instances, especially the state update as well as the auxiliary variables, to capture as much information as possible. Thus, we follow the idea of a local collision as presented in the works of Mendel et al. [39, 40] and many others.

To induce a local collision, we constrain the differential conditions in the state update variables, A and E, along with the message words, W. This set of constraints on the differentials conditions is called the "starting point" (of the differential path), and are enforced through clauses added to the encoding. For example, if a differential condition on the variable x is constrained to be a '`-`', we add the unit clause $\neg\Delta x$ to set the difference to be zero (thus $x = x'$). The explicit starting points we used in our work are given in the appendix (Tables B.0.1–B.0.4).

Any solution found by the solver will be within the confinements set by the starting point. This reduction of the search space is found to very beneficial and the possibility and time required for finding collisions highly depend on a well-crafted starting point.

To make the base problem easier, we also add clauses for the propagation of common differentials, especially ones with `-` and `x`. For example, the encoding has a clause for propagation of `[xx-]` $\rightarrow$ `[-]` for the XOR function, encoding that when x is the auxiliary variable for $a \oplus b \oplus c$ we have $(\Delta a \wedge \Delta b \wedge \neg\Delta c) \rightarrow \neg\Delta x$. Such helpful clauses are present for all the operations XOR, MAJ, IF, and the modular addition of words from equation (1) and the state update equations of Section 1.2.1.1.

5.2 Implementation

Our framework comprises CaDiCaL 1.8 that includes the programmatic interface, IPASIR-UP. IPASIR-UP provides access to the current state of the solver for the relevant variables (i.e., those encoding the state of the hash function and the differential variables). IPASIR-UP also enables us to perform custom propagation and branching

[2]https://github.com/saeednj/SAT-encoding

as well as learning custom conflict clauses.

Only the Boolean variables necessary for the programmatic techniques are added to the list of observed/watched variables through the interface. This ensures that these variables are preserved after each simplification step performed by the solver. Furthermore, the solver will not send assignment notifications for any variable not included in the observed list.

Our implementation also maintains a high-level state of the differential cryptanalysis layer. In other words, for each variable pair (x, x'), the conditions $\{\mathtt{x}, \mathtt{-}, \mathtt{u}, \mathtt{n}, \mathtt{?}\}$ are updated in memory with respect to the variable assignments. This ensures that whenever the conditions are needed for the differential cryptanalysis techniques, they do not have to be re-computed.

For implementing inconsistency blocking, we used our own implementation of a graph library. This houses the graph algorithm described in Section 4.1 for detecting minimal inconsistent cycles in linear time. The implementations of bitsliced and wordwise propagation, as well as the two-bit condition detection engine used for inconsistency blocking, are based on the ideas described in the works of Mendel et al. [39] and Eichlseder [22, 23]. We used a least-recently-used cache data structure for caching propagation rules and rules for deriving two-bit conditions. The cache doesn't grow beyond the maximum available RAM, since the least frequently used entries are deleted on the fly.

In our experiments, most queries to the propagation and two-bit detection engines could be served from the cache, which is much faster than deriving the propagation rules or the two-bit conditions on the fly each time. Since the set of rules that are queried throughout the entire runtime is usually small (i.e., consumes a small portion of the total CPU time), it wasn't necessary to precompute any rules.

We perform bitsliced propagation for all operations in SHA-256 (including the modular additions) alongside the solver's built-in Boolean constraint propagation (BCP). As wordwise propagation is much more expensive in terms of computational cost, it's performed only when the SAT solver finishes with the other propagation methods. This ensures that wordwise propagation only deduces the conditions that bitsliced propagation and BCP could not.

IPASIR-UP asks for a "reason" clause of propagated literals when it becomes necessary for the solver to know why a literal was propagated. For bitsliced propagation, these reason clauses were relatively short, so in this case we provided reason clauses directly via IPASIR-UP's interface. On the other hand, wordwise propagation involves multiple word pairs and a single propagated literal may depend on a relatively large number of bits, leading to long reason clauses. To avoid overwhelming the solver with long reason clauses, we did not use IPASIR-UP's propagation interface for wordwise propagation and instead set the values of any literals deduced by wordwise propagation via branching.

Every time bitsliced propagation is performed, the reason clauses are stored in memory in case the solver requires them. However, the list of reason clauses grows continuously. To automatically prune obsolete reason clauses, the clauses are categorized by the solver's decision level, a process known as maintaining a trail. The decision level increments with each new decision made by the solver and decrements upon backtracking. Consequently, when the solver moves to a lower decision level, all reason clauses associated with higher levels are deleted. This form of garbage collection is also applied to two-bit conditions.

5.3 Results

In this section, we discuss finding SFS collisions of step-reduced SHA-256 for 20 to 39 steps and the attempts to find collisions for 40 steps. We also discuss the use of various programmatic techniques and also different encoders.

5.3.1 SFS Collisions: 20–38 steps

We performed the same experiment with three separate solvers: an unmodified version of CaDiCaL 1.8.0, a version of CaDiCaL with programmatic bitsliced and wordwise propagation, and a version of CaDiCaL with both programmatic propagation and inconsistency blocking. We also tried using CryptoMiniSat 5.11.21 [51], given that it supports XOR constraints natively and has been tuned to work on cryptographic problems. However, we did not pursue this extensively as CryptoMiniSat currently does not have a programmatic interface and did not perform as well as CaDiCaL.

With each solver we searched for semi-free-start collisions for step-reduced SHA-256 with 20 to 38 steps. In order to reduce the randomness inherent in the search, each instance was solved ten times independently using 10 different random seeds, though these 10 different seeds were consistently used across all our experiments. Each instance was run for a time limit of 500,000 seconds (roughly 5.8 days). The number of instances successfully solved in each case is given in Table 5.3.1, and all the running times for finding a collision are plotted in Figure 5.3.1 and Figure 5.3.2. The starting points used for the 21-step, 25-step, 28-step, and 38-step instances are given in the appendix. The starting points for all other step counts were formed from one of these starting points by dropping a number of rows at the bottom, e.g., the 26-step

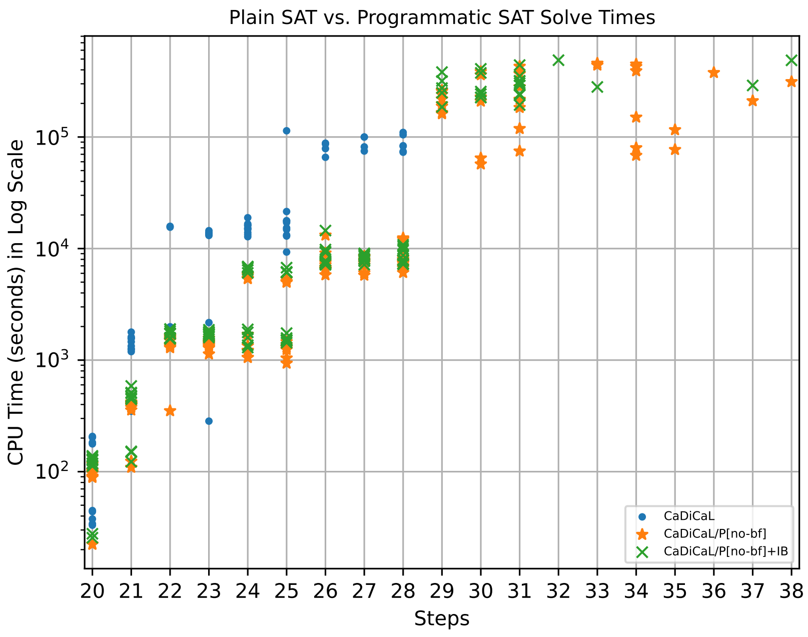

Fig. 5.3.1: Running times for finding a SFS collision for step-reduced SHA-256 for a varying number of steps. The plot compares a plain SAT solver with two programmatic SAT+CAS solvers. CaDiCaL/P[no-bf] represents bitsliced and wordwise propagation (without any brute-force attack). CaDiCaL/P[no-bf]+IB represents bitsliced and wordwise propagation (without any brute-force attack) along with inconsistency blocking. The lack of a data point indicates no collisions were found within 500,000 seconds.

instance matches the 28-step starting point with two rows removed. This means that the instances in the step ranges 20–21, 22–25, 26–28, and 29–38 can be expected to roughly have similar difficulty as they were created from the same starting point.

The results show that programmatic propagation was clearly effective at helping the solver find SFS collisions. The plain SAT solver could only find SFS collisions up to 28 steps, while CaDiCaL with programmatic propagation, both with and without inconsistency blocking, successfully found SFS collisions for every step count from 20 to 38, with the exception of 35—no 35-step instances were solved when both programmatic propagation and inconsistency blocking were enabled (see Table 5.3.1). We could get significantly lower solve times for 30, 31, 34, and 35 when wordwise propagation doesn't include any brute-force attack. In general, we found that the wordwise propagation brute-force method and the inconsistency blocking method tended to decrease the efficiency of the solver.

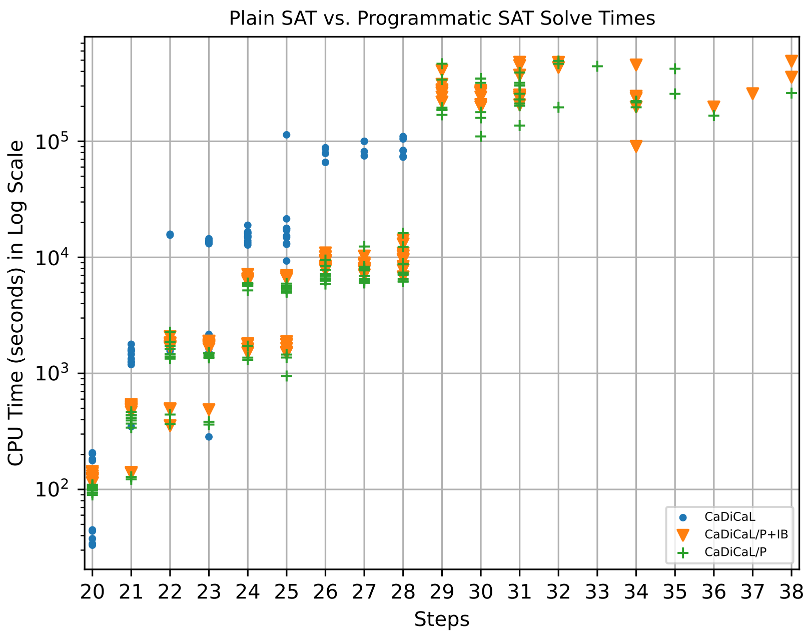

Fig. 5.3.2: Running times for finding a SFS collision for step-reduced SHA-256 for a varying number of steps. The plot compares a plain SAT solver with two programmatic SAT+CAS solvers. CaDiCaL/P represents bitsliced and wordwise propagation. CaDiCaL/P+IB represents bitsliced and wordwise propagation along with inconsistency blocking. Note that unlike Figure 5.3.1, the wordwise propagation here involves a brute-force attack, as detailed in Section 4.2.2. The lack of a data point indicates no collisions were found within 500,000 seconds.

Steps	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38
CaDiCaL	10	10	10	10	10	10	4	4	6	0	0	0	0	0	0	0	0	0	0
CaDiCaL/P	10	10	10	10	10	10	10	10	10	6	6	9	3	1	3	2	1	0	1
CaDiCaL/P[no-bf]	10	10	10	10	10	10	10	10	10	8	6	7	0	2	6	2	1	1	1
CaDiCaL/P+IB	10	10	10	10	10	10	10	10	10	7	7	7	2	0	5	0	1	1	2
CaDiCaL/P[no-bf]+IB	10	10	10	10	10	10	10	10	10	5	6	9	1	1	0	0	0	1	1

Table 5.3.1: Number of step-reduced SFS collisions found in each instance for the 5 methods: plain CaDiCaL, CaDiCaL with Propagation (P), and CaDiCaL with Propagation and Inconsistency Blocking (P+IB), and also 2 variants of P and P+IB where brute-force attack is turned off (P[no-bf] and P[no-bf]+IB). For each number of steps and solver, we solved the same instance using 10 different SAT solver seeds.

5.3.2 SFS Collisions: 39–40 steps

Although our experiments showed that programmatic propagation could improve the performance of the solver and find SFS collisions up to 38 steps, we could not find SFS collisions beyond 38 steps.

The work of Li et al. [33] involves an SMT (Satisfiability Modulo Theories) encoding that could find SFS collisions up to 39 steps. SMT is the problem of determining the satisfiability of mathematical formulas. It's similar to SAT, but SMT supports more expressive formulas involving complex entities like real numbers, integers, bitvectors, strings, and cardinality constraints. Li et al. [33]'s work involves cardinality constraints to find a differential characteristic with minimum differences in the ∇W_i, ∇A_i, and ∇E_i variables.

The quantity of differences is the sum of the Hamming weights of the differences. For example, the Hamming weight for the expanded message variable differences is

$$H_W = \sum_{i=0}^{n-1} \sum_{j=0}^{31} W_i[j] \oplus W_i'[j]$$

where n is the number of steps. Similarly, it's H_A and H_E for the sums of the ∇A and ∇E differences.

The minimization for H_W, H_A, and H_E is done in the following steps in order:

- Minimize H_W: Find the minimum H_W. Let this minimum H_W be t_w.
- Minimize H_A: Find the minimum H_A with $H_W = t_w$ constrained. Let this minimum H_A be t_a.
- Minimize H_E: Find the minimum H_E with $H_W = t_w$ and $H_A = t_a$ constrained.

Let this minimum be t_e.

The minimization process involves proving that a valid characteristic doesn't exist below a certain point. For example, if $t_w = 25$, it means that there exists no valid characteristic for $H_W < 25$.

Minimizing the Hamming weight leads to a sparse differential characteristic, thereby increasing the chances of finding a message pair that conforms to an SFS collision. For example, the minimum Hamming weights for a 39-step SFS collision are $t_w = 25$, $t_a = 10$, and $t_e = 90$ with the starting point B.0.5.

The relation between the sparsity and the likelihood of finding SFS collisions is also utilized in the work of Mendel et al. [40]. In our experiments, we've observed that the likelihood of finding SFS collisions is very high within the constraints of sparse characteristics found with Li et al. [33]'s encoding.

Improving Li et al. [33]'s encoder. The original encoder[3] of Li et al. [33] generates high-level code describing a problem, which is then processed by an SMT (Satisfiability Modulo Theories) solver called STP (Simple Theorem Prover) [26] to find a solution.

SMT solvers determine the satisfiability of logical formulas. Unlike SAT solvers, SMT solvers accept high-level problem descriptions, typically supporting arithmetic operations, bit-vectors, and arrays. However, SMT solvers often translate the problem into a SAT problem and then use a SAT solver to solve it.

In their implementation, STP is configured to use CryptoMiniSat as the SAT solver in portfolio mode. However, we modified the implementation to output DIMACS CNF instead, allowing it to be solved with CaDiCaL and most other SAT solvers.

Moreover, the original implementation's cardinality constraints, which were implemented using modular addition for Hamming weights, have been replaced by cardinality constraints based on the totalizer algorithm [5]. Overall, these changes significantly reduced the solve time and allowed us to retrieve the variable mapping, which was not possible with STP. Having the variable mapping and gaining the ability to solve with CaDiCaL opened the pathway to trying out our programmatic techniques on this different type of SAT encoding.

Note that STP has an option to output DIMACS CNF, but it doesn't provide the variable mapping and also doesn't let us select the algorithm used for encoding the cardinality constraints.

Interestingly, we didn't benefit from the programmatic techniques that provided a performance boost for finding 20–38 steps SFS collisions (as specified in Section 5.3.1). See Figure 5.3.3 for the results using our modification of Li et al. [33]'s encoder.

[3] `https://github.com/Peace9911/sha_2_attack`

The minimum Hamming weights for each number of steps are provided in Table 5.3.2. Note that the times for finding the characteristics (see Figure 5.3.3) are much higher for 26–28 steps compared to that for ≥ 29 steps. This likely was a result of cardinality constraints that were relatively tighter in the instances with 26–28 steps.

To find message pairs conforming to an SFS collision out of the found characteristics, we utilize our SAT encoder (see Section 5.1). We encode an SFS collision problem using the found characteristic as the starting point. Solving this problem gives us a colliding message pair that conforms to the characteristic. The running times for finding the SFS collisions from the found characteristics are shown in Figure 5.3.4. Note that there exists no collision conforming to the characteristics involving 26–28 steps, meaning that the solver yields UNSAT for all instances of these steps. The tight cardinality constraints might explain why no collisions exist for the characteristics found for 26–28 steps.

Steps	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
H_A	1	4	1	1	4	4	12	12	12	6	6	7	7	7	7	7	7	7	7	10
H_E	68	35	73	73	33	33	58	58	58	151	151	140	141	141	141	141	141	141	141	90
H_W	2	11	5	5	22	25	37	37	37	19	19	23	23	23	23	23	23	23	23	25

Table 5.3.2: Minimum Hamming weights for each number of steps found with Li et al. [33]'s encoding.

Cardinality Clauses We have benchmarked the performance with cardinality clauses [47] and experimented with our implementation of the totalizer encoding [5]. The comparison is highlighted in Figure 5.3.5. The results show that Cardinality CDCL is significantly slower than CaDiCaL 1.5.2 for problems based on Li et al. [33]'s SAT encoder. We chose CaDiCaL 1.5.2 for comparison because Cardinality CDCL is based this version of CaDiCaL.

Searching for 40-step SFS collisions. As shown in Figure 5.3.3, Li et al. [33]'s form of encoding could be used to find SFS characteristics up to 39 steps with low solve times. This brings up a question: can it find valid SFS characteristics for 40 steps? To investigate, we used a starting point for 40 steps (see Table B.0.6).

The initial stage in the search for a 40-step SFS collision with this type of encoding involves sequentially minimizing the Hamming weights H_W, H_A, and H_E. This step is crucial, as finding a message pair that conforms to a differential characteristic is highly unlikely without minimizing the Hamming weights.

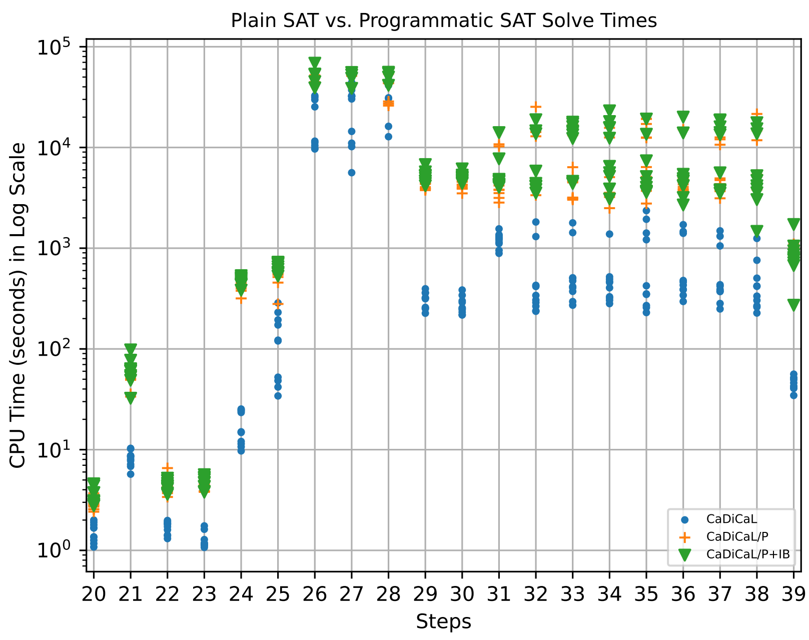

Fig. 5.3.3: Running times for finding a SFS characteristic using Li et al. [33]'s technique for step-reduced SHA-256 for a varying number of steps. The plot compares a plain SAT solver with two programmatic SAT solvers. CaDiCaL/P represents bitsliced and wordwise propagation. CaDiCaL/P+IB represents bitsliced and wordwise propagation along with inconsistency blocking. The lack of a data point indicates no collisions were found within 86,400 seconds.

The characteristics with the lowest Hamming weights H_W, H_A, and H_E that we found were 77, 13, and 140 respectively. The lowest Hamming weight H_W that we proved to have no valid characteristics is 55, meaning that there might exist characteristics with a lower H_W for $55 < H_W < 77$. However, we couldn't find a characteristic for $H_W < 77$, and none of the found characteristics could be extended to an SFS collision, meaning that no message pair confirming to an SFS collision exists for these characteristics and the solver returned UNSAT.

We've also tried constraining the Hamming weight for the last active message word ∇W_{30} to be 1 on top of constraining H_W. This idea is based on the fact that most of the solutions for SFS collisions with 38 and 39 steps also had a Hamming weight of 1 for the last active message word. With this approach, the lowest Hamming weights that we found were $H_W = 83$, $H_A = 20$, and $H_E = 155$. On the other side, the highest H_W that we could be proved to have no valid characteristics is 74, leaving a range for the possible valid characteristics between $74 < H_W < 83$. However, none of the found characteristics could be extended to an SFS collision, just like in the previous case without the cardinality constraint on ∇W_{30}.

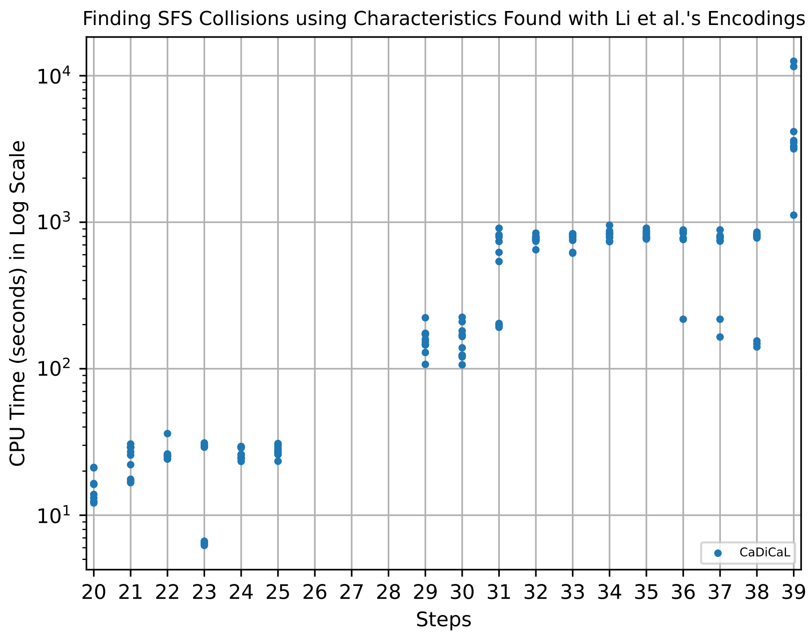

Fig. 5.3.4: Running times for finding a SFS collision using the characteristic found with Li et al. [33]'s encoding for step-reduced SHA-256 for a varying number of steps. The lack of a data point indicates no collision exists for the characteristic.

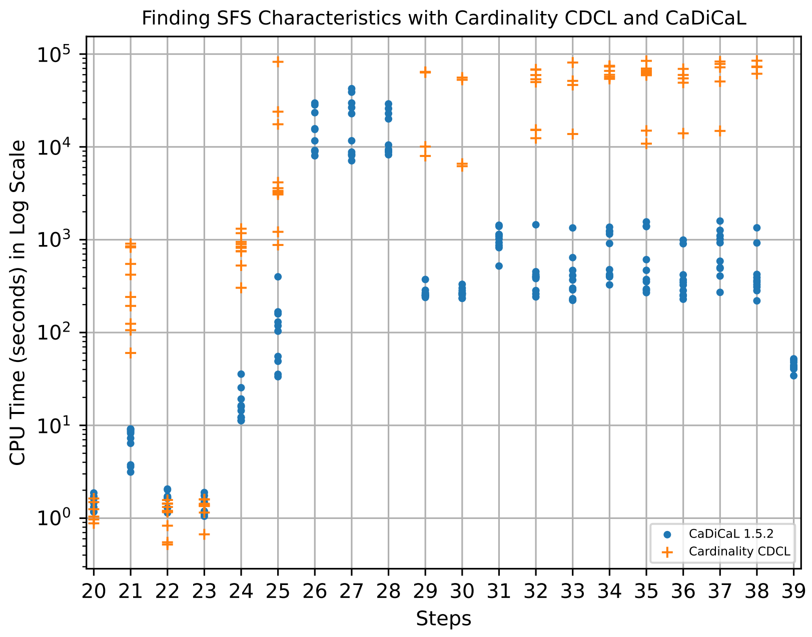

Fig. 5.3.5: Running times for finding a SFS characteristic using Li et al. [33]'s technique for step-reduced SHA-256 for a varying number of steps. The plot compares solving with and without native cardinality constraints support. The lack of a data point indicates no collisions were found within 86,400 seconds.

CHAPTER 6

Conclusion

In this work we combine the programmatic SAT+CAS paradigm with the differential cryptanalysis techniques used in previous collision attacks on SHA-256. In the process, we demonstrate that these computer algebraic techniques can dramatically improve the performance of the SAT solver, enabling the SAT+CAS solver to find a semi-free-start collision of SHA-256 with 38 steps, while a plain SAT solver could go no further than 28 steps. Moreover, previous 38-step SFS collisions [40] were found with a highly sophisticated search tool specifically written to find SHA-256 collisions, while our work used the general purpose SAT solver CaDiCaL coupled with the IPASIR-UP interface [25] for custom propagation, branching, and learning. Thus, we were able to exploit the power of modern SAT solvers without needing to write a search tool from scratch.

A few months before this thesis was written, the current best SFS collision for SHA-256, involving up to 39 steps, was found by Li et al. [33]. This approach, based on an SMT/SAT methodology, employs a significantly different encoding. Our SAT+CAS approach wasn't found to be useful with this alternate encoding for reasons that are currently unclear. Determining other programmatic techniques or modifying our current techniques to improve the solver's performance with this new encoding is potential future work.

It's possible that the cardinality constraints in Li et al.'s encodings may be limiting the performance of our SAT+CAS approach. If this is indeed the case, we could adapt our programmatic techniques to work better with these constraints. Additionally, instead of strictly enforcing minimum Hamming weights (see Section 5.3.2), we might consider allowing slightly higher Hamming weights to provide greater flexibility for our programmatic techniques, as long as we can still obtain a valid characteristic for which there exists a message pair conforming to an SFS collision. We've already seen that minimum Hamming weights do not always lead to optimal solve times, nor do they consistently result in characteristics that can be extended to an SFS collision.

REFERENCES

[1] Ábrahám, E. (2015). Building bridges between symbolic computation and satisfiability checking. In *Proceedings of the 2015 ACM on International Symposium on Symbolic and Algebraic Computation*, pages 1–6.

[2] Ábrahám, E., Abbott, J., Becker, B., Bigatti, A. M., Brain, M., Buchberger, B., Cimatti, A., Davenport, J. H., England, M., Fontaine, P., et al. (2017). Satisfiability checking and symbolic computation. *ACM Communications in Computer Algebra*, 50(4):145–147.

[3] Ahmed, T. (2009). *An implementation of the DPLL algorithm.* PhD thesis, Concordia University.

[4] Ajani, Y. and Bright, C. (2024). SAT and lattice reduction for integer factorization. In *Proceedings of the 2024 ACM on International Symposium on Symbolic and Algebraic Computation*, pages 391–399.

[5] Bailleux, O. and Boufkhad, Y. (2003). Efficient CNF encoding of Boolean cardinality constraints. In *International conference on principles and practice of constraint programming*, pages 108–122. Springer.

[6] Baldwin, S. (2012). Compute Canada: advancing computational research. In *Journal of Physics: Conference Series*, volume 341. IOP Publishing. Article 012001.

[7] Biere, A., Fleury, M., and Pollitt, F. (2023). CaDiCaL_vivinst, IsaSAT, Gimsatul, Kissat, and TabularaSAT entering the SAT competition 2023. *SAT COMPETITION 2023*, page 14.

[8] Biere, A., Heule, M., van Maaren, H., and Walsh, T., editors (2021). *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications.* IOS Press.

[9] Bright, C., Cheung, K. K., Stevens, B., Kotsireas, I., and Ganesh, V. (2021a). A SAT-based resolution of Lam's problem. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 3669–3676.

[10] Bright, C. and Davenport, J. H., editors (2022). *SC-Square 2021: Satisfiability Checking and Symbolic Computation*, volume 3273 of *CEUR Workshop Proceedings*. CEUR.

[11] Bright, C., Ganesh, V., Heinle, A., Kotsireas, I., Nejati, S., and Czarnecki, K. (2016). MathCheck2: A SAT+CAS verifier for combinatorial conjectures. In *Computer Algebra in Scientific Computing*, pages 117–133. Springer International Publishing.

[12] Bright, C., Gerhard, J., Kotsireas, I., and Ganesh, V. (2020a). *Effective Problem Solving Using SAT Solvers*, page 205–219. Springer International Publishing.

[13] Bright, C., Kotsireas, I., and Ganesh, V. (2020b). Applying computer algebra systems with SAT solvers to the Williamson conjecture. *Journal of Symbolic Computation*, 100:187–209.

[14] Bright, C., Kotsireas, I., and Ganesh, V. (2022). When satisfiability solving meets symbolic computation. *Communications of the ACM*, 65(7):64–72.

[15] Bright, C., Kotsireas, I., Heinle, A., and Ganesh, V. (2021b). Complex Golay pairs up to length 28: A search via computer algebra and programmatic SAT. *Journal of Symbolic Computation*, 102:153–172.

[16] Brown, C. W. (2017). Projection and quantifier elimination using non-uniform cylindrical algebraic decomposition. In *Proceedings of the 2017 ACM on International Symposium on Symbolic and Algebraic Computation*, ISSAC'17. ACM.

[17] Brown, C. W. and Vale-Enriquez, F. (2020). From simplification to a partial theory solver for non-linear real polynomial constraints. *Journal of Symbolic Computation*, 100:72–101. Symbolic Computation and Satisfiability Checking.

[18] Dang, Q. H. (2015). *Secure Hash Standard*.

[19] Davis, M., Logemann, G., and Loveland, D. (1962). A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397.

[20] De Canniere, C. and Rechberger, C. (2006). Finding SHA-1 characteristics: General results and applications. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 1–20. Springer.

[21] Dworkin, M. J. (2015). SHA-3 standard: Permutation-based hash and extendable-output functions. *Federal Information Processing Standards.*

[22] Eichlseder, M. (2013). Linear propagation of information in differential collision attacks. Master's thesis, Graz University of Technology.

[23] Eichlseder, M. (2018). *Differential Cryptanalysis of Symmetric Primitives.* PhD thesis, Graz University of Technology.

[24] England, M. (2022). SC-Square: Overview to 2021. In [10], pages 1–6.

[25] Fazekas, K., Niemetz, A., Preiner, M., Kirchweger, M., Szeider, S., and Biere, A. (2023). IPASIR-UP: User propagators for CDCL. In *26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023)*, pages 8:1–8:13. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[26] Ganesh, V. and Dill, D. L. (2007). A decision procedure for bit-vectors and arrays. In *Computer Aided Verification: 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007. Proceedings 19*, pages 519–531. Springer.

[27] Ganesh, V., O'Donnell, C. W., Soos, M., Devadas, S., Rinard, M. C., and Solar-Lezama, A. (2012). *Lynx: A Programmatic SAT Solver for the RNA-Folding Problem*, page 143–156. Springer Berlin Heidelberg.

[28] Gilbert, H. and Handschuh, H. (2003). Security analysis of SHA-256 and sisters. In *International workshop on selected areas in cryptography*, pages 175–193. Springer.

[29] Gopinath, D., Malik, M. Z., and Khurshid, S. (2011). Specification-based program repair using SAT. In *Tools and Algorithms for the Construction and Analysis of Systems: 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011. Proceedings 17*, pages 173–188. Springer.

[30] Ignatiev, A., Janota, M., and Marques-Silva, J. (2014). Towards efficient optimization in package management systems. In *Proceedings of the 36th International Conference on Software Engineering*, pages 745–755.

[31] Johnson, D. S. and Trick, M. A. (1996). *Cliques, coloring, and Satisfiability: Second DIMACS implementation challenge, October 11-13, 1993*, volume 26. American Mathematical Soc.

[32] Kaufmann, D., Biere, A., and Kauers, M. (2019). Verifying large multipliers by combining SAT and computer algebra. In *2019 Formal Methods in Computer Aided Design (FMCAD)*. IEEE.

[33] Li, Y., Liu, F., and Wang, G. (2024a). *New Records in Collision Attacks on SHA-2*, pages 158–186. Springer Nature Switzerland.

[34] Li, Y., Liu, F., Wang, G., Dong, X., and Sun, S. (2024b). A practical colliding message pair for 31-step SHA-256. FSE 2024 Rump Session.

[35] Li, Z., Bright, C., and Ganesh, V. (2024c). A SAT solver + computer algebra attack on the minimum Kochen–Specker problem. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*, IJCAI-2024. International Joint Conferences on Artificial Intelligence Organization.

[36] Liang, J. H., Ganesh, V., Poupart, P., and Czarnecki, K. (2016). Learning rate based branching heuristic for SAT solvers. In *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5–8, 2016, Proceedings*, pages 123–140.

[37] Lisitsa, A. and Vernitski, A. (2017). *Automated Reasoning for Knot Semigroups and π-orbifold Groups of Knots*, page 3–18. Springer International Publishing.

[38] Liu, F., Wang, G., Sarkar, S., Anand, R., Meier, W., Li, Y., and Isobe, T. (2023). *Analysis of RIPEMD-160: New Collision Attacks and Finding Characteristics with MILP*, page 189–219. Springer Nature Switzerland.

[39] Mendel, F., Nad, T., and Schläffer, M. (2011). Finding SHA-2 characteristics: searching through a minefield of contradictions. In *Advances in Cryptology–ASIACRYPT 2011: 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings 17*, pages 288–307. Springer.

[40] Mendel, F., Nad, T., and Schläffer, M. (2013). Improving local collisions: new attacks on reduced SHA-256. In *Advances in Cryptology–EUROCRYPT 2013: 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings 32*, pages 262–278. Springer.

[41] Mendel, F., Pramstaller, N., Rechberger, C., and Rijmen, V. (2006). Analysis of step-reduced SHA-256. In *Fast Software Encryption: 13th International Workshop,*

FSE 2006, Graz, Austria, March 15-17, 2006, Revised Selected Papers 13, pages 126–143. Springer.

[42] Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf.

[43] National Institute of Standards and Technology (1995). *Federal Information Processing Standards Publication: Secure Hash Standard.*

[44] Nejati, S. and Ganesh, V. (2019). CDCL(Crypto) SAT solvers for cryptanalysis. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, CASCON '19, page 311–316, USA. IBM Corp.

[45] Nikolić, I. and Biryukov, A. (2008). Collisions for step-reduced SHA-256. In *Fast Software Encryption: 15th International Workshop, FSE 2008, Lausanne, Switzerland, February 10-13, 2008, Revised Selected Papers 15*, pages 1–15. Springer.

[46] Prokop, L. (2016). *Differential cryptanalysis with SAT solvers.* PhD thesis, Ph. D. Dissertation. University of Technology, Graz.

[47] Reeves, J. E., Heule, M., and Bryant, R. E. (2024). From clauses to klauses. In *International Conference on Computer Aided Verification (CAV-24)*. Springer.

[48] Rivest, R. (1992). RFC1321: The MD5 message-digest algorithm.

[49] Sanadhya, S. K. and Sarkar, P. (2008). New collision attacks against up to 24-step SHA-2. In *Progress in Cryptology-INDOCRYPT 2008: 9th International Conference on Cryptology in India, Kharagpur, India, December 14-17, 2008. Proceedings 9*, pages 91–103. Springer.

[50] Silva, J. M. and Sakallah, K. A. (1996). GRASP-a new search algorithm for satisfiability. In *Proceedings of International Conference on Computer Aided Design*, pages 220–227. IEEE.

[51] Soos, M., Nohl, K., and Castelluccia, C. (2009). *Extending SAT Solvers to Cryptographic Problems*, pages 244–257. Springer Berlin Heidelberg.

[52] Stevens, M., Bursztein, E., Karpman, P., Albertini, A., and Markov, Y. (2017). The first collision for full SHA-1. In *Advances in Cryptology–CRYPTO 2017: 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20–24, 2017 , Proceedings, Part I 37*, pages 570–596. Springer.

[53] Tseitin, G. S. (1983). *On the Complexity of Derivation in Propositional Calculus*, pages 466–483. Springer Berlin Heidelberg.

[54] Wang, C., Wu, R., Song, H., Shu, J., and Li, G. (2022). smartPip: A smart approach to resolving Python dependency conflict issues. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12.

[55] Wang, X., Lai, X., Feng, D., Chen, H., and Yu, X. (2005a). Cryptanalysis of the Hash Functions MD4 and RIPEMD. In *Advances in Cryptology–EUROCRYPT 2005: 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005. Proceedings 24*, pages 1–18. Springer.

[56] Wang, X., Yin, Y. L., and Yu, H. (2005b). Finding collisions in the full SHA-1. In *Advances in Cryptology–CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005. Proceedings 25*, pages 17–36. Springer.

[57] Wang, X., Yin, Y. L., and Yu, H. (2005c). *Finding Collisions in the Full SHA-1*, page 17–36. Springer Berlin Heidelberg.

[58] Wang, X. and Yu, H. (2005). How to break MD5 and other hash functions. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 19–35. Springer.

[59] Zabih, R. and McAllester, D. A. (1988). A rearrangement search strategy for determining propositional satisfiability. In *AAAI*, volume 88, pages 155–160.

[60] Zaikin, O. (2022). Inverting cryptographic hash functions via cube-and-conquer. *arXiv preprint arXiv:2212.02405.*

[61] Zulkoski, E., Bright, C., Heinle, A., Kotsireas, I., Czarnecki, K., and Ganesh, V. (2017). Combining SAT solvers with computer algebra systems to verify combinatorial conjectures. *Journal of Automated Reasoning*, 58(3):313–339.

[62] Zulkoski, E., Ganesh, V., and Czarnecki, K. (2015). *MathCheck: A Math Assistant via a Combination of Computer Algebra Systems and SAT Solvers*, page 607–622. Springer International Publishing.

Appendices

APPENDIX A

SFS Collisions

In this appendix we provide an example of a 38-step semi-free-start SHA-256 collision that we found (Table A.0.1).

Table A.0.1: SFS collision for 38 steps found with programmatic propagation and inconsistency blocking. h_0 is the chaining value, (M, M') is the colliding message pair, and h_1 is the hash of M and M'. Word pairs in M and M' that have differences are enclosed in a box.

h_0	afea2566 1e0a73e2 da747de7 34381a7f 06f4c0d9 8897dd98 c592ba6a d2aa5e80
M	5b5058d2 901f87fb 254bcfa2 5f8d7dc1 fb1053be 0622e1f8 da8801c2 $\boxed{\texttt{a951cfbb}}$ $\boxed{\texttt{5db42ffd}}$ 683b4391 $\boxed{\texttt{f87eabbd}}$ e928b976 3675cc55 6ebe78be e3031536 $\boxed{\texttt{c2de906f}}$
M'	5b5058d2 901f87fb 254bcfa2 5f8d7dc1 fb1053be 0622e1f8 da8801c2 $\boxed{\texttt{9737d17b}}$ $\boxed{\texttt{5db43001}}$ 683b4391 $\boxed{\texttt{f8812bbd}}$ e928b976 3675cc55 6ebe78be e3031536 $\boxed{\texttt{c2de906b}}$
h_1	d0e019f7 408269d3 24296a7b 30df8e7f 95d2bff8 34e2bca6 6c50a294 ddb4254a

Table A.0.2: The differential characteristic for the 38-step semi-free-start collision presented in Table A.0.1. The words with a nonzero difference (i.e., including a '`u`' or '`n`' differential) are enclosed in a box. Interestingly, compared to the 38-step semi-free-start collision presented by Mendel et al. [40], an additional two words (∇A_8 and ∇E_{10}) have a zero difference.

i	∇A_i	∇E_i	∇W_i
-4	001101000011100000001101001111111	11010010101010100101111010000000	
-3	11011010011101000111110111100111	11000101100100101011101001101010	
-2	00011110000010100111001111100010	10001000100101111101110110011000	
-1	10101111111010100010010101100110	00000110111101001100000011011001	
0	11111110001111000010010101011001	01101011101011110101110100111011	01011011010100000101100011010010
1	01001011000010110110000101101010	11000001100000000001101010100000	10010000000111111000011111111011
2	01100101111011000011000000100010	11100000101101110101010101101101	00100101010010111100111110100010
3	01110101001010010111110000101000	00001101001111101101010100100110	01011111100011010111110111000001
4	11000011100011110110101000111111	00000010101010001100011111011001	11111011000100000101001110111110
5	11001111011111111011000001010111	00011000001010000101111110011101	00000110001000101110000111111000
6	00111101011110100100110110011011	01001100100101100011110010101001	11011010100010000000000111000010
7	nnn0nnnunnnnu01uuuuuuuu0un000010	010u1u0nnnn0nunnuuuuuuu00u001010	10ununn10un10nn1110nuuu1un111011
8	11001111111110101001001011100001	001nu010011001010000000010010010	0101110110110100001nuuuuuuuuuu01
9	01u001n0n0unnnnn00n0nuunuu10nu11	00010111011010010111110001101n00	01101000001110110100001110010001
10	000000001110000001100000100000111	100010110000001111011101010010110	11111000nuuuuuunu010101110111101
11	110110001010000000101000100000110	0unnun011u0n0u1110100n0un1nuuu01	11101001001010001011100101110110
12	101100000001000101011101010001100	10000000001000100100001001000101	00110110011101011100110001010101
13	01100100110001111111110111111000	1un0nu0uuunnnnnn00n1nuu0nu100n11	01101110101111100111100010111110
14	00001110111111011001101011100111	00110110001100111100010010010000	11100011000000110001010100110110
15	1100n0101011001u0100111101111101	1111n110000111un1011101111111100	11000010110111101001000001101u11
16	11011101100011000101110010100u10	unnnnnn0001111000110011111u0nu10	10110000011011100011010101111110
17	11101100000110000000111011010101	0001111nuuu111nu0unn1110101010nu	11010101101101000100100110000001
18	00110001010011100101110110001111	00000110000111110000111110111100	11000000111100000000000101101111
19	01111001001101010100100010010001	0010n101unnnnnnn0001100110110011	10111010010001001001101110101000
20	00000011110100011000001001011001	1001100001011100101101unnnnnnnn00	10110011001011110000100101010110
21	11000111011100100100001111011010	01101101111111111111000000000001	01100110010010100111100100010011
22	00100011101011111111110111000001	10101011100001110101011111111110	00010100101010100100011010011101
23	10000010100110001101110100111111	11011110110001000111101101110110	1100u000011001nu0000110101111000
24	01000000000101011100011011111110	01010101111111011000110111111001	10011110001100111100000011001n10
25	11001011100110110010110011011100	10011111100000110000101010100110	00011110010111001111111110111100
26	111001011001000000010111000100010	000011011111100100111101000100101	00101001110010011101001110101010
27	11001011111100101100101000101100	00000100000101110110010111001110	01011100001111001111111000110101
28	11100111000010101101101110111100	101100101101110000001011001011100	01011111100011111001000101011010
29	00101001101011001111100101111010	0011111100110100111011111100000101	00011001000111111010011001001101
30	00101111100001101001111000010001	00110111000011000101010101011000	010010011011111110100101000011010
31	11011100111011101011001110011111	110110000100001000000001110000101	11101111001011100111000001011100
32	01111100111101110111110011000001	011001010100100001101000100000001	001001101010010011001100010000010
33	00001111011000110010100101101000	01010111011001000010011011011001	10111101011000000100111100101100
34	11111100101001110111010000000000	00001011000010011100011011001010	01111101101001000011001100101100
35	01001001101101001110110010010100	10100110101111011110100000101010	11000001100001101101001000000100
36	00100010011101111111010111110001	10101100010010101101111100001110	001000000011111001000101001001111
37	001000000111101011111010010010001	10001110110111011111111100011111	0000110010010111101101010111101000

APPENDIX B

Starting Points

This appendix presents the explicit starting points that we used in our search. The 21-step starting point (Table B.0.1) is taken from the work of Prokop [46]. The starting point for 25 steps (Table B.0.2) is an extended version of the 24-step starting point provided by Prokop [46]. The 28-step starting point (Table B.0.3) is a slightly modified version of the starting point used by Mendel et al. [40]. The 39-step starting point (Table B.0.5) is taken from the work of Li et al. [33]. Li et al. [33]'s starting point is a modified version of the 39-step starting point for SHA-512 in Eichlseder [23]. The 40-step starting point (Table B.0.6) has been provided to us by Maria Eichlseder.

The 38-step starting point (Table B.0.4) is constructed based on the 38-step differential characteristic provided by Mendel et al. [40]—in particular the differential words ∇W_{15}, ∇W_{23}, ∇W_{24}, ∇A_{15}, and ∇A_{16}. The '`x`'s in these words are placed under the heuristic assumption that these words have a low (but nonzero) Hamming weight. The differential word ∇W_{24} (with a Hamming weight of 1 and an '`x`' in position 2) was taken from their starting point. This propagates to the '`x`'s in ∇W_{15} and ∇A_{16} (and both those words were assumed to have a Hamming weight of 1 as well) as well as the '`x`' in position 16 of ∇W_{23}. Then setting position 16 of ∇A_{15} to '`x`' causes it to propagate to $\nabla E_{19}[16]$ and cancel out with $\nabla W_{23}[16]$ in the state update transformation equation of T_{23} (and similarly for position 27 of ∇A_{15}). $\nabla W_{23}[27]$ is set to '`x`' to cancel out with $\sigma_0(W_{15})$ in step 30 of the message expansion equation.

More details related to the collisions are available in the data repository in the GitHub repository.[1]

[1] https://github.com/nahiyan/sha256-data

Table B.0.1: Starting point for a 21-step semi-free-start collision.

i	∇A_i	∇E_i	∇W_i
-4	--------------------------------	--------------------------------	
-3	--------------------------------	--------------------------------	
-2	--------------------------------	--------------------------------	
-1	--------------------------------	--------------------------------	
0	--------------------------------	--------------------------------	--------------------------------
1	--------------------------------	--------------------------------	--------------------------------
2	--------------------------------	--------------------------------	--------------------------------
3	--------------------------------	--------------------------------	--------------------------------
4	--------------------------------	--------------------------------	--------------------------------
5	x???????????????????????????????	????????????????????????????????	????????????????????????????????
6	--------------------------------	????????????????????????????????	????????????????????????????????
7	--------------------------------	????????????????????????????????	????????????????????????????????
8	--------------------------------	????????????????????????????????	????????????????????????????????
9	--------------------------------	????????????????????????????????	--------------------------------
10	--------------------------------	--------------------------------	--------------------------------
11	--------------------------------	--------------------------------	--------------------------------
12	--------------------------------	--------------------------------	--------------------------------
13	--------------------------------	--------------------------------	????????????????????????????????
14	--------------------------------	--------------------------------	--------------------------------
15	--------------------------------	--------------------------------	--------------------------------
16	--------------------------------	--------------------------------	--------------------------------
17	--------------------------------	--------------------------------	--------------------------------
18	--------------------------------	--------------------------------	--------------------------------
19	--------------------------------	--------------------------------	--------------------------------
20	--------------------------------	--------------------------------	--------------------------------

Table B.0.2: Starting point for a 25-step semi-free-start collision.

i	∇A_i	∇E_i	∇W_i
-4	--------------------------------	--------------------------------	
-3	--------------------------------	--------------------------------	
-2	--------------------------------	--------------------------------	
-1	--------------------------------	--------------------------------	
0	--------------------------------	--------------------------------	--------------------------------
1	--------------------------------	--------------------------------	--------------------------------
2	--------------------------------	--------------------------------	--------------------------------
3	--------------------------------	--------------------------------	--------------------------------
4	--------------------------------	--------------------------------	--------------------------------
5	--------------------------------	--------------------------------	--------------------------------
6	--------------------------------	--------------------------------	--------------------------------
7	--------------------------------	--------------------------------	--------------------------------
8	x???????????????????????????????	????????????????????????????????	????????????????????????????????
9	--------------------------------	????????????????????????????????	????????????????????????????????
10	--------------------------------	????????????????????????????????	--------------------------------
11	--------------------------------	????????????????????????????????	????????????????????????????????
12	--------------------------------	????????????????????????????????	--------------------------------
13	--------------------------------	--------------------------------	--------------------------------
14	--------------------------------	--------------------------------	--------------------------------
15	--------------------------------	--------------------------------	--------------------------------
16	--------------------------------	--------------------------------	????????????????????????????????
17	--------------------------------	--------------------------------	--------------------------------
18	--------------------------------	--------------------------------	--------------------------------
19	--------------------------------	--------------------------------	--------------------------------
20	--------------------------------	--------------------------------	--------------------------------
21	--------------------------------	--------------------------------	--------------------------------
22	--------------------------------	--------------------------------	--------------------------------
23	--------------------------------	--------------------------------	--------------------------------
24	--------------------------------	--------------------------------	--------------------------------

Table B.0.3: Starting point for a 28-step semi-free-start collision.

i	∇A_i	∇E_i	∇W_i
-4	--------------------------------	--------------------------------	
-3	--------------------------------	--------------------------------	
-2	--------------------------------	--------------------------------	
-1	--------------------------------	--------------------------------	
0	--------------------------------	--------------------------------	--------------------------------
1	--------------------------------	--------------------------------	--------------------------------
2	--------------------------------	--------------------------------	--------------------------------
3	--------------------------------	--------------------------------	--------------------------------
4	--------------------------------	--------------------------------	--------------------------------
5	--------------------------------	--------------------------------	--------------------------------
6	--------------------------------	--------------------------------	--------------------------------
7	--------------------------------	--------------------------------	--------------------------------
8	????????????????????????????????	????????????????????????????????	x???????????????????????????????
9	????????????????????????????????	????????????????????????????????	????????????????????????????????
10	????????????????????????????????	????????????????????????????????	--------------------------------
11	--------------------------------	????????????????????????????????	--------------------------------
12	--------------------------------	????????????????????????????????	--------------------------------
13	--------------------------------	????????????????????????????????	????????????????????????????????
14	--------------------------------	????????????????????????????????	--------------------------------
15	--------------------------------	--------------------------------	--------------------------------
16	--------------------------------	--------------------------------	????????????????????????????????
17	--------------------------------	--------------------------------	--------------------------------
18	--------------------------------	--------------------------------	????????????????????????????????
19	--------------------------------	--------------------------------	--------------------------------
20	--------------------------------	--------------------------------	--------------------------------
21	--------------------------------	--------------------------------	--------------------------------
22	--------------------------------	--------------------------------	--------------------------------
23	--------------------------------	--------------------------------	--------------------------------
24	--------------------------------	--------------------------------	--------------------------------
25	--------------------------------	--------------------------------	--------------------------------
26	--------------------------------	--------------------------------	--------------------------------
27	--------------------------------	--------------------------------	--------------------------------

Table B.0.4: Starting point for a 38-step semi-free-start collision.

i	∇A_i	∇E_i	∇W_i
-4	--------------------------------	--------------------------------	
-3	--------------------------------	--------------------------------	
-2	--------------------------------	--------------------------------	
-1	--------------------------------	--------------------------------	
0	--------------------------------	--------------------------------	--------------------------------
1	--------------------------------	--------------------------------	--------------------------------
2	--------------------------------	--------------------------------	--------------------------------
3	--------------------------------	--------------------------------	--------------------------------
4	--------------------------------	--------------------------------	--------------------------------
5	--------------------------------	--------------------------------	--------------------------------
6	--------------------------------	--------------------------------	--------------------------------
7	????????????????????????????????	????????????????????????????????	????????????????????????????????
8	????????????????????????????????	????????????????????????????????	????????????????????????????????
9	????????????????????????????????	????????????????????????????????	--------------------------------
10	--------------------------------	????????????????????????????????	????????????????????????????????
11	--------------------------------	????????????????????????????????	--------------------------------
12	--------------------------------	????????????????????????????????	--------------------------------
13	--------------------------------	????????????????????????????????	--------------------------------
14	--------------------------------	????????????????????????????????	--------------------------------
15	----x----------x----------------	????????????????????????????????	-----------------------------x--
16	-----------------------------x--	????????????????????????????????	--------------------------------
17	--------------------------------	????????????????????????????????	--------------------------------
18	--------------------------------	--------------------------------	--------------------------------
19	--------------------------------	????????????????????????????????	--------------------------------
20	--------------------------------	????????????????????????????????	--------------------------------
21	--------------------------------	--------------------------------	--------------------------------
22	--------------------------------	--------------------------------	--------------------------------
23	--------------------------------	--------------------------------	----x---------xx----------------
24	--------------------------------	--------------------------------	-----------------------------x--
25	--------------------------------	--------------------------------	--------------------------------
26	--------------------------------	--------------------------------	--------------------------------
27	--------------------------------	--------------------------------	--------------------------------
28	--------------------------------	--------------------------------	--------------------------------
29	--------------------------------	--------------------------------	--------------------------------
30	--------------------------------	--------------------------------	--------------------------------
31	--------------------------------	--------------------------------	--------------------------------
32	--------------------------------	--------------------------------	--------------------------------
33	--------------------------------	--------------------------------	--------------------------------
34	--------------------------------	--------------------------------	--------------------------------
35	--------------------------------	--------------------------------	--------------------------------
36	--------------------------------	--------------------------------	--------------------------------
37	--------------------------------	--------------------------------	--------------------------------

Table B.0.5: Starting point for a 39-step semi-free-start collision. Note that there is no difference constraint unlike the other starting points. Instead, we rely on the cardinality constraints to impose a difference.

i	∇A_i	∇E_i	∇W_i
-4	--------------------------------	--------------------------------	
-3	--------------------------------	--------------------------------	
-2	--------------------------------	--------------------------------	
-1	--------------------------------	--------------------------------	
0	--------------------------------	--------------------------------	--------------------------------
1	--------------------------------	--------------------------------	--------------------------------
2	--------------------------------	--------------------------------	--------------------------------
3	--------------------------------	--------------------------------	--------------------------------
4	--------------------------------	--------------------------------	--------------------------------
5	--------------------------------	--------------------------------	--------------------------------
6	--------------------------------	--------------------------------	--------------------------------
7	--------------------------------	--------------------------------	--------------------------------
8	????????????????????????????????	????????????????????????????????	????????????????????????????????
9	????????????????????????????????	????????????????????????????????	????????????????????????????????
10	--------------------------------	????????????????????????????????	????????????????????????????????
11	--------------------------------	????????????????????????????????	????????????????????????????????
12	--------------------------------	????????????????????????????????	????????????????????????????????
13	--------------------------------	????????????????????????????????	--------------------------------
14	--------------------------------	????????????????????????????????	--------------------------------
15	????????????????????????????????	????????????????????????????????	--------------------------------
16	????????????????????????????????	????????????????????????????????	????????????????????????????????
17	--------------------------------	????????????????????????????????	????????????????????????????????
18	????????????????????????????????	????????????????????????????????	--------------------------------
19	--------------------------------	????????????????????????????????	--------------------------------
20	--------------------------------	????????????????????????????????	--------------------------------
21	--------------------------------	????????????????????????????????	--------------------------------
22	--------------------------------	????????????????????????????????	--------------------------------
23	--------------------------------	--------------------------------	--------------------------------
24	--------------------------------	--------------------------------	????????????????????????????????
25	--------------------------------	--------------------------------	--------------------------------
26	--------------------------------	--------------------------------	????????????????????????????????
27	--------------------------------	--------------------------------	--------------------------------
28	--------------------------------	--------------------------------	--------------------------------
29	--------------------------------	--------------------------------	--------------------------------
30	--------------------------------	--------------------------------	--------------------------------
31	--------------------------------	--------------------------------	--------------------------------
32	--------------------------------	--------------------------------	--------------------------------
33	--------------------------------	--------------------------------	--------------------------------
34	--------------------------------	--------------------------------	--------------------------------
35	--------------------------------	--------------------------------	--------------------------------
36	--------------------------------	--------------------------------	--------------------------------
37	--------------------------------	--------------------------------	--------------------------------
38	--------------------------------	--------------------------------	--------------------------------

Table B.0.6: Starting point for a 40-step semi-free-start collision. Note that there is no difference constraint unlike the other starting points. Instead, we rely on the cardinality constraints to impose a difference.

i	∇A_i	∇E_i	∇W_i
-4	--------------------------------	--------------------------------	
-3	--------------------------------	--------------------------------	
-2	--------------------------------	--------------------------------	
-1	--------------------------------	--------------------------------	
0	--------------------------------	--------------------------------	--------------------------------
1	--------------------------------	--------------------------------	--------------------------------
2	--------------------------------	--------------------------------	--------------------------------
3	--------------------------------	--------------------------------	--------------------------------
4	--------------------------------	--------------------------------	--------------------------------
5	--------------------------------	--------------------------------	--------------------------------
6	--------------------------------	--------------------------------	--------------------------------
7	--------------------------------	--------------------------------	--------------------------------
8	--------------------------------	--------------------------------	--------------------------------
9	--------------------------------	--------------------------------	--------------------------------
10	--------------------------------	--------------------------------	--------------------------------
11	????????????????????????????????	????????????????????????????????	????????????????????????????????
12	????????????????????????????????	????????????????????????????????	????????????????????????????????
13	????????????????????????????????	????????????????????????????????	????????????????????????????????
14	????????????????????????????????	????????????????????????????????	????????????????????????????????
15	????????????????????????????????	????????????????????????????????	--------------------------------
16	????????????????????????????????	????????????????????????????????	????????????????????????????????
17	????????????????????????????????	????????????????????????????????	--------------------------------
18	????????????????????????????????	????????????????????????????????	????????????????????????????????
19	????????????????????????????????	????????????????????????????????	????????????????????????????????
20	????????????????????????????????	????????????????????????????????	--------------------------------
21	????????????????????????????????	????????????????????????????????	????????????????????????????????
22	????????????????????????????????	????????????????????????????????	--------------------------------
23	--------------------------------	????????????????????????????????	--------------------------------
24	--------------------------------	????????????????????????????????	--------------------------------
25	--------------------------------	????????????????????????????????	????????????????????????????????
26	--------------------------------	????????????????????????????????	????????????????????????????????
27	--------------------------------	--------------------------------	--------------------------------
28	--------------------------------	--------------------------------	????????????????????????????????
29	--------------------------------	--------------------------------	????????????????????????????????
30	--------------------------------	--------------------------------	????????????????????????????????
31	--------------------------------	--------------------------------	--------------------------------
32	--------------------------------	--------------------------------	--------------------------------
33	--------------------------------	--------------------------------	--------------------------------
34	--------------------------------	--------------------------------	--------------------------------
35	--------------------------------	--------------------------------	--------------------------------
36	--------------------------------	--------------------------------	--------------------------------
37	--------------------------------	--------------------------------	--------------------------------
38	--------------------------------	--------------------------------	--------------------------------
39	--------------------------------	--------------------------------	--------------------------------

VITA AUCTORIS

NAME:	Nahiyan Alamgir
PLACE OF BIRTH:	Dhaka, Bangladesh
YEAR OF BIRTH:	1998
EDUCATION:	University of Windsor, M.Sc. in Computer Science, Windsor, Ontario, 2024
	North South University, B.Sc. in Computer Science & Engineering, Dhaka, Bangladesh, 2021