SHA-256 Collision Attack with Programmatic SAT

Nahiyan Alamgir[1], Saeed Nejati[2,†] and Curtis Bright[1]

[1]*University of Windsor, Canada*
[2]*Amazon, USA*

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 paper, 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 find a 38-step collision of SHA-256 with a modified initialization vector—something first found by a highly sophisticated search tool of 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.

Keywords

Hash Functions, Differential Cryptanalysis, SAT Solving, Inconsistency Blocking, Computer Algebra System

1. Introduction

Cryptographic hash functions play a vital role in information security. They are widely relied on for data security and integrity. Due to the high reliance on cryptographic hash functions for security, they have been constantly targeted for cryptanalysis. Through these cryptanalytic attacks, some hash functions that were heavily relied on for security purposes have met their end of life. Some prominent examples include MD5 and SHA-1—hash functions that were compromised in terms of collision resistance in the works of Wang and Yu [1] and Stevens et al. [2] respectively. SHA-1 was published by NIST in 1995 [3]. Six years later, NIST has also published a new family of hash functions called SHA-2. Soon after, weaknesses were found in SHA-1 [4], and in 2011 NIST formally recommended anyone relying on SHA-1 for security migrate to other hash functions like SHA-2. Consequently, hash functions in the SHA-2 family have become very widely used. For example, the function SHA-256 is used for transaction signatures and for proof-of-work in the Bitcoin protocol [5].

Despite the arrival of SHA-3 [6], 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. [7] in 2013 utilized differential cryptanalysis for SHA-256. It was inspired by the 2005 work of Wang and Yu [1] 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

9th International Workshop on Satisfiability Checking and Symbolic Computation, July 2, 2024, Nancy, France, Collocated with IJCAR 2024

† The contribution to this paper does not relate to this author’s position at Amazon.

cbright@uwindsor.ca (C. Bright)
https://www.curtisbright.com/ (C. Bright)
0000-0002-0462-625X (C. Bright)

only broken in the last few months with the announcement of a 31-step SHA-256 collision [8] and a 39-step semi-free-start (SFS) SHA-256 collision [9]. For more details and background, see Section 2.

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 [10] 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 [11] 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 [7].

In our 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 [12]. We encode the collision-finding problem directly into SAT (see Section 3) and then programmatically encode several of the mathematical constraints exploited by Mendel et al. [7] 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) and are able to deduce nontrivial information about the solver's state using programmatic propagation (see Section 5). 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. [7], while a pure SAT approach was not able to go any further than 28 steps—see Section 6 for a summary of our results.

We note that our SAT-based tool is significantly slower than the dedicated search tool of Mendel et al. [7] for finding 38-step SFS SHA-256 collisions. However, the novelty of our work is that we show that the performance 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. [7]'s 38-step SFS SHA-256 collisions—an additional two internal state variables have zero difference (see Table 5) when compared with Mendel et al. [7]'s collision.

2. Background

In this section we provide background on cryptographic hash functions, especially SHA-256, and then discuss differential cryptanalysis and the notation we use in our work (see Section 2.3). We also briefly discuss SAT solving and summarize previous collision attacks on SHA-256.

2.1. 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.

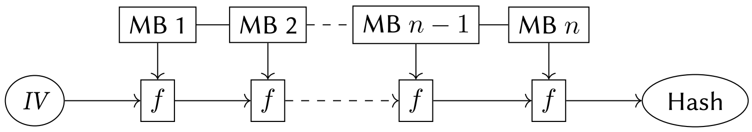

Figure 1: SHA-256 processes the input (with padding if needed) into message blocks (abbreviated as "MB"), which are sequentially fed to the compression function, f. The output of each compression is used as the chaining value in the next compression. The compression of the last message block produces 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.

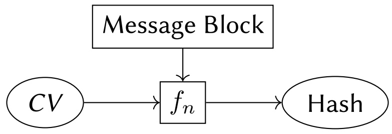

Figure 2: A diagram depicting the simplified version of SHA-256 we consider in our work. f_n is the step-reduced compression function having n steps. The chaining value, *CV*, is arbitrary for semi-free-start (SFS) collisions and is not required to match the *IV* actually used in SHA-256—though a SFS colliding message pair is required to have matching *CV*s.

- Collision resistance: It's infeasible to find an input pair, x and x' ($x \neq x'$), that both produce the same hash.

An example of a weak hash function is MD4 [13] because it does not have collision resistance—an attacker can easily generate colliding message pairs.

2.2. 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).

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 [14] 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 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.

2.2.1. 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 "expanded" 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 σ_0 and σ_1 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.

2.2.2. 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. [15]. 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).

2.3. 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. Usually differences between the values are

Table 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.

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

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'. In general, a pair (x, x') representing a bit in one bitvector and its counterpart in the second hash instance may have up to 4 combinations, $\{(0,0),(0,1),(1,0),(1,1)\}$. In some cases, some of the four possibilities may be ruled out, though. The possibilities for (x, x') can be generalized as the differential conditions [16] presented in Table 1. For example, if a pair (x, x') has the possibilities $\{(0,0),(1,1)\}$, we describe it as having the differential condition '-', whereas the differential condition 'x' describes the possibilities $\{(0,1),(1,0)\}$.

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 $\nabla X \to \nabla Y$. 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 is not 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.

2.4. 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. 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.

The beauty of SAT solving lies in its generic nature, which means 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 independently of a specific problem. Modern SAT solvers can be surprisingly effective at solving SAT problems by incorporating sophisticated techniques like conflict analysis and clause learning, clever branching heuristics, and simplification [17]. This combination allows them to be highly potent at general-purpose search.

The SC^2 Project. SAT solving, ever since its inception, has been found to be useful for satisfiability checking—determining whether a logical formula is satisfiable. On the other side, Computer Algebra Systems (CASs) include algorithms that are efficient at solving mathematical problems. However, many problems exist that involves both satisfiability checking and symbolic computation, and bridging the two fields was proposed in 2015 in the work of Ábrahám [18] and Zulkoski et al. [19]. Shortly afterwards, the SC^2 project [20] was initiated to support the joint community. Since then, a wide variety of problems have been tackled using SC^2 techniques, from circuit verification [21] to knot theory [22] to quantifier elimination and cylindrical algebraic decomposition [23], and factoring integers with known bits [24]. See England's survey [25] for an overview of many other examples.

Programmatic SAT. Programmatic SAT involves injecting code into the solver to aid the solver and solve a problem more efficiently than it otherwise could [26]. It's useful especially when aspects of the problem are difficult to express in conjunctive normal form, the typical input of SAT solvers [27].

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 [28]. 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.

2.5. Previous Work

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. [13] devised an efficient method in 2005 for finding MD4 collisions with probability from 2^{-6} to 2^{-2} using at most 256 MD4 hash operations. Wang et al. also proposed an attack on MD5 [1] for finding collisions within 15–60 minutes of computational time in the same year. Also in 2005, Wang et al. [29] presented a collision attack on SHA-1 using at most 2^{69} SHA-1 hash computations, resulting in a SHA-1 collision found in 2017 [30].

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 [31]. In FSE 2006, Mendel et al. [32] 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 [33] presented collisions up to 24 steps of SHA-256 and SHA-512, making improvements over the work of Nikolić and Biryukov [34] that presented collisions up to 21 steps of SHA-256 at FSE 2008.

In ASIACRYPT 2011, Mendel et al. [15] 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. [7] 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 heuristics over [15].

Very recently, in the rump session of FSE 2024, Li et al. [8] announced a 31-step collision of SHA-256, and in a EUROCRYPT 2024 paper found a 39-step SFS collision [9]. 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 2.

Table 2
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.

Publication Year	Author	Collision	SFS Collision
2006	Mendel et al. [32]	18	-
2008	Sanadhya and Sarkar [33]	24	-
2011	Mendel et al. [15]	27	32
2013	Mendel et al. [7]	28	38
2024	Li et al. [8, 9]	31	39

3. The SAT Encoding

Our problem is to find two different messages whose hashes match, and to do this we use SAT solvers as a search tool for the collision attack. Our SAT formula contains variables encoding two messages (each containing one 512-bit message block) that we want to collide after applying SHA-256. For each block, the formula includes an n-step compression function taking a 512-bit message block and a 256-bit chaining value, and from them computes a 256-bit hash. The number of steps/rounds, n, is adjusted to generate a step-reduced version of SHA-256.

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 a new auxiliary variable) using $2^3 = 8$ clauses. A new variable is introduced for every gate in the circuit similar to how the Tseitin transformation is performed [35].

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 [11], which used the Espresso logic minimizer [36].

On top of the two hash function instances, we have the differential cryptanalysis layer. Each Boolean variable in one instance, say x, has its counterpart x' in the other instance. For the analysis of the differences as per differential cryptanalysis, we encode the bitwise differences as $\Delta x \leftrightarrow x \oplus x'$ (following Nejati and Ganesh [11]) where $\Delta x \in \{0, 1\}$ is a new auxiliary variable. Each triple (x, x', Δx) defines a differential condition. For example, $(x, x', 1)$ defines an x while $(1, 0, \Delta x)$ defines a u (see Table 1 for the complete list of differential conditions).

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 two 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. [15, 7] 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. The encoding includes clauses for constraining of the conditions as such, and is called the **starting point** of the differential path. 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 (and thus $x = x'$). The explicit starting points we used in our work are given in the appendix (Tables 6–9).

Any solution found by the solver will be within the confinements set by the starting point. This reduction of search space is found to very beneficial and the possibility and time required for finding collisions highly depends 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 2.2.2.

4. Programmatic Inconsistency Blocking

As discussed in Section 2.3, 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. [15]. 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. [15, 7] refers 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 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. 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 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, $\ldots$, 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 [28] 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.

5. 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 3, 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.

There is a large body of work studying propagation in SAT encodings and metrics by which propagation can be studied, including propagation completeness [37], propagation strength [38], and unit-refutation completeness [39]. In general, the basic unit propagation mechanism used in SAT solvers will *not* propagate all logically implied information, though IPASIR-UP supports the inclusion of more advanced custom propagation routines. 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 [40, 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 $\nabla X \rightarrow \nabla Y$. 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 is infeasible because there are too many possibilities to explore.

5.1. 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; this bitslice is highlighted in bold in the depiction below:

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

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.

5.2. 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 2.2.2 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 differences of words were also used in the work of Wang and Yu [1] for a different purpose.

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

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

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

$$\delta A_1 \boxplus \delta A_2 \boxplus \cdots \boxplus \delta A_n = C \tag{3}$$

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.

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 $v = 1$.

Dividing into subproblems. After reducing equations of the form (3) to bitvector equations, we want to determine all solutions for the variables. To do so, we search for possible values through brute force. Since this has an exponential time complexity, we divide the problem into smaller components by analyzing the cascading effects of the carries. In practice, this reduces the computational cost significantly as the subproblems can usually be solved quickly, and any subproblem that is too expensive to solve can be skipped without affecting the other subproblems. In our work, we consider any subproblem with more than 10 variables as expensive, limiting the number of (brute force) iterations to $2^{10} = 1024$ for a subproblem.

As an example, suppose $\delta A \boxplus \delta B = \delta C$ where $\nabla A =$ `[u1xxx]`, $\nabla B =$ `[xx-nx]`, and $\nabla C =$ `[---u-]`. Then $\delta C = 2$, and after normalizing A and B we derive

$$(\delta A + 2^2 + 2 + 1) \boxplus (\delta B + 2^4 + 2^3 + 2 + 1) = \delta A \boxplus \delta B \boxplus 2 = 4,$$

becoming the bitvector modular addition problem

$$\delta A \boxplus \delta B \boxplus 2 = \begin{array}{r} [\ 1,\ v_3,\ v_2,\ v_0,\ 0\ ] \\ \boxplus\ [\ v_4,\ 0,\ \ 0,\ v_1,\ 0\ ] \\ \hline 0 \quad 0 \quad 1 \quad 0 \quad 0 \end{array}$$

where $v_0, \ldots, v_4 \in \{0, 1\}$.

Now, a linear scan is performed starting from the least significant digit (the rightmost column). Since there's no variable in the first column, we skip it. Next, we have a subproblem candidate $v_0 + v_1 \equiv 0 \pmod 2$. Since this addition can overflow, the next column must be included in the subproblem. Including the next column results in the subproblem $v_0 + v_1 + 2v_2 \equiv 2 \pmod 4$ which cannot overflow since $v_0 = v_1 = v_2 = 1$ is not a solution. Thus, the problem starting in the next column is independent.

The next problem is then $v_3 \equiv 0 \pmod 2$, which only has one solution $v_3 = 0$ and thus also cannot overflow and is independent of the final column. Similarly, the final column results in the subproblem $1 + v_4 \equiv 0 \pmod 2$ which only has the single solution $v_4 = 1$.

In this example wordwise propagation thus determines that $v_4 = 1$ and $v_3 = 0$. Since v_4 arose from the second 'x' in ∇B which encodes the difference $b_3 \oplus b_3'$, i.e., $v_4 = (b_3 - b_3' + 1)/2$, we derive that $(b_3, b_3') = (1, 0)$. Similarly, v_3 arose from the 'x' in ∇A encoding $a_2 \oplus a_2'$, i.e., $v_3 = (a_2 - a_2' + 1)/2$, and we derive that $(a_2, a_2') = (0, 1)$. Thus, in this example wordwise propagation deduces the updated differential conditions $\nabla A =$ `[u1nxx]` and $\nabla B =$ `[xu-nx]`.

Implementation Details. Wordwise propagation was 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 were propagated were the differential variables in ∇A_i, ∇E_i, and ∇W_i.

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. [15] 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.

6. Implementation and Results

Our programmatic SAT solver was implemented in CaDiCaL 1.8.0 [41] using the programmatic interface IPASIR-UP [28]. Our experiments were run in the Digital Research Alliance of Canada's [42] 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]

6.1. Implementation

The IPASIR-UP programmatic interface 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 as well as learning custom conflict clauses.

For implementing inconsistency blocking, we used our own implementation of a graph library. This houses the graph algorithm described in Section 4 for detecting minimal inconsistent cycles in linear time. Our 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. [15] and Eichlseder [40, 43]. 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 way, wordwise propagation only deduces the conditions that bitsliced propagation couldn't.

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.

6.2. Results

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 [44], 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 3, and the minimum time for finding a collision is plotted in Figure 3. 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 instance matches the 28-step starting point with two rows removed. This means that the instances in the step

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

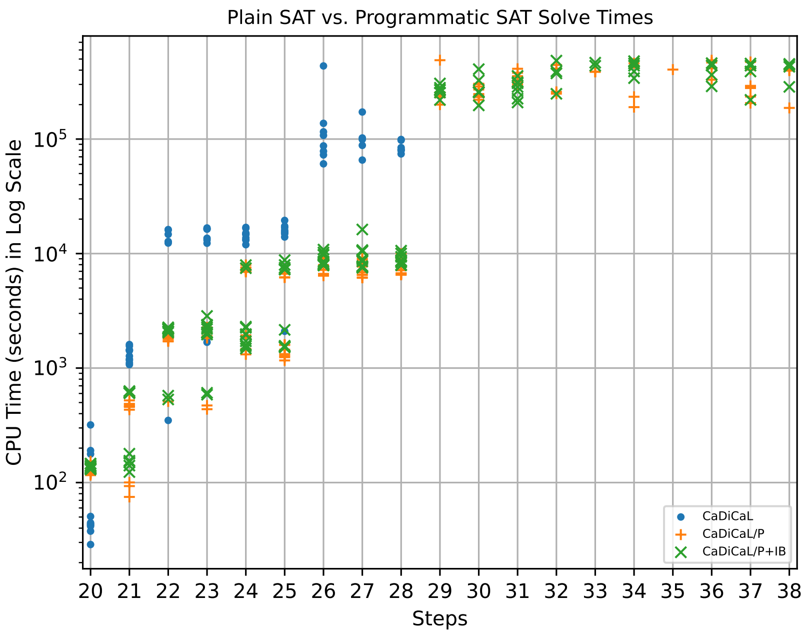

Figure 3: Running times for finding a SFS collision for step-reduced SHA-256 for a varying number of steps. The plot compares a plain SAT solver with two programmatic SAT solvers. The lack of a data point indicates no collisions were found within 500,000 seconds.

Table 3
Number of step-reduced SFS collisions found in each instance for the 3 methods: plain CaDiCaL, CaDiCaL with Propagation (P), and CaDiCaL with Propagation and Inconsistency Blocking (P+IB). For each number of steps and solver, we solved the same instance using 10 different SAT solver seeds.

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	8	5	7	0	0	0	0	0	0	0	0	0	0
CaDiCaL/P	10	10	10	10	10	10	10	10	10	6	9	8	4	2	3	1	4	7	3
CaDiCaL/P+IB	10	10	10	10	10	10	10	10	10	6	7	7	4	2	5	0	4	4	4

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 3). In general, we found inconsistency blocking tended to decrease the efficiency of the solver, although using inconsistency blocking did result in the fastest solve times for the instances with 30, 31, 32, and 36 steps.

7. 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 [7] 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 [28] 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.

At the time the work in this paper was performed, the best SFS collision ever found for SHA-256 contained 38 steps [7]. Just prior to submitting this work, the authors became aware of the work of Li et al. [9] appearing at EUROCRYPT 2024 that finds for the first time a 39-step SHA-256 SFS collision. Li et al. [9] also use a SAT-based approach, but with a significantly different encoding. Determining if the SAT+CAS approach can also be useful with this alternate encoding will be the subject of future work.

Acknowledgements

We thank the chairs of the SC-Square 2024 workshop, Daniela Kaufmann and Chris Brown, for their flexibility during the publication process of this paper, and the anonymous reviewers for their detailed comments. We also thank Maria Eichlseder for her insight and answering a number of questions concerning state-of-the-art hash function collision search tools, as well as Oleg Zaikin for answering questions about his work on inverting 43-step MD4 [45].

References

[1] X. Wang, H. Yu, How to break MD5 and other hash functions, in: Annual international conference on the theory and applications of cryptographic techniques, Springer, 2005, pp. 19–35. doi:10.1007/11426639_2.

[2] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, Y. Markov, 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, Springer, 2017, pp. 570–596. doi:10.1007/978-3-319-63688-7_19.

[3] National Institute of Standards and Technology, Federal Information Processing Standards Publication: Secure Hash Standard, 1995. doi:10.6028/nist.fips.180-1.

[4] X. Wang, Y. L. Yin, H. Yu, Finding Collisions in the Full SHA-1, Springer Berlin Heidelberg, 2005, p. 17–36. doi:10.1007/11535218_2.

[5] S. Nakamoto, Bitcoin: A peer-to-peer electronic cash system (2008). URL: https://bitcoin.org/bitcoin.pdf. doi:10.2139/ssrn.3440802.

[6] M. J. Dworkin, SHA-3 standard: Permutation-based hash and extendable-output functions, Federal Information Processing Standards (2015). doi:10.6028/NIST.FIPS.202.

[7] F. Mendel, T. Nad, M. Schläffer, 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, Springer, 2013, pp. 262–278. doi:10.1007/978-3-642-38348-9_16.

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

[9] Y. Li, F. Liu, G. Wang, New Records in Collision Attacks on SHA-2, Springer Nature Switzerland, 2024, pp. 158–186. doi:10.1007/978-3-031-58716-0_6.

[10] L. Prokop, Differential cryptanalysis with SAT solvers, Master's thesis, University of Technology, Graz, 2016. doi:10.3217/qxkra-gh483.

[11] S. Nejati, V. Ganesh, CDCL(Crypto) SAT solvers for cryptanalysis, in: Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering, CASCON '19, IBM Corp., USA, 2019, pp. 311–316. doi:10.48550/arXiv.2005.13415.

[12] C. Bright, I. Kotsireas, V. Ganesh, When satisfiability solving meets symbolic computation, Communications of the ACM 65 (2022) 64–72. doi:10.1145/3500921.

[13] X. Wang, X. Lai, D. Feng, H. Chen, X. Yu, 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, Springer, 2005, pp. 1–18. doi:10.1007/11426639_1.
[14] Q. H. Dang, Secure Hash Standard, 2015. doi:10.6028/nist.fips.180-4.
[15] F. Mendel, T. Nad, M. Schläffer, 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, Springer, 2011, pp. 288–307. doi:10.1007/978-3-642-25385-0_16.
[16] C. De Canniere, C. Rechberger, Finding SHA-1 characteristics: General results and applications, in: International Conference on the Theory and Application of Cryptology and Information Security, Springer, 2006, pp. 1–20. doi:10.1007/11935230_1.
[17] A. Biere, M. Heule, H. van Maaren, Handbook of Satisfiability, volume 185, IOS press, 2009.
[18] E. Ábrahám, Building bridges between symbolic computation and satisfiability checking, in: Proceedings of the 2015 ACM on International Symposium on Symbolic and Algebraic Computation, 2015, pp. 1–6. doi:10.1145/2755996.2756636.
[19] E. Zulkoski, V. Ganesh, K. Czarnecki, MathCheck: A Math Assistant via a Combination of Computer Algebra Systems and SAT Solvers, Springer International Publishing, 2015, p. 607–622. doi:10.1007/978-3-319-21401-6_41.
[20] E. Ábrahám, J. Abbott, B. Becker, A. M. Bigatti, M. Brain, B. Buchberger, A. Cimatti, J. H. Davenport, M. England, P. Fontaine, et al., Satisfiability checking and symbolic computation, ACM Communications in Computer Algebra 50 (2017) 145–147. doi:10.1145/3055282.3055285.
[21] D. Kaufmann, A. Biere, M. Kauers, Verifying large multipliers by combining SAT and computer algebra, in: 2019 Formal Methods in Computer Aided Design (FMCAD), IEEE, 2019, pp. 28–36. doi:10.23919/FMCAD.2019.8894250.
[22] A. Lisitsa, A. Vernitski, Automated Reasoning for Knot Semigroups and π-orbifold Groups of Knots, Springer International Publishing, 2017, pp. 3–18. doi:10.1007/978-3-319-72453-9_1.
[23] C. W. Brown, 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, 2017, pp. 53–60. doi:10.1145/3087604.3087651.
[24] Y. Ajani, C. Bright, SAT and lattice reduction for integer factorization, in: Proceedings of the 2024 ACM on International Symposium on Symbolic and Algebraic Computation, 2024. doi:10.1145/3666000.3669712, to appear.
[25] M. England, SC-Square: Overview to 2021, in: C. Bright, J. H. Davenport (Eds.), SC-Square 2021: Satisfiability Checking and Symbolic Computation, volume 3273 of *CEUR Workshop Proceedings*, CEUR, 2022, pp. 1–6. URL: https://ceur-ws.org/Vol-3273/invited1.pdf. doi:10.48550/arXiv.2209.04359.
[26] V. Ganesh, C. W. O'Donnell, M. Soos, S. Devadas, M. C. Rinard, A. Solar-Lezama, Lynx: A Programmatic SAT Solver for the RNA-Folding Problem, Springer Berlin Heidelberg, 2012, p. 143–156. doi:10.1007/978-3-642-31612-8_12.
[27] C. Bright, V. Ganesh, A. Heinle, I. Kotsireas, S. Nejati, K. Czarnecki, MathCheck2: A SAT+CAS verifier for combinatorial conjectures, in: Computer Algebra in Scientific Computing, Springer International Publishing, 2016, pp. 117–133. doi:10.1007/978-3-319-45641-6_9.
[28] K. Fazekas, A. Niemetz, M. Preiner, M. Kirchweger, S. Szeider, A. Biere, IPASIR-UP: User propagators for CDCL, in: 26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023), Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023, pp. 8:1–8:13. doi:10.4230/LIPIcs.SAT.2023.8.
[29] X. Wang, Y. L. Yin, H. Yu, 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, Springer, 2005, pp. 17–36. doi:10.1007/11535218_2.
[30] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, Y. Markov, The First Collision for Full SHA-1, Springer International Publishing, 2017, pp. 570–596. doi:10.1007/978-3-319-63688-7_19.
[31] H. Gilbert, H. Handschuh, Security analysis of SHA-256 and sisters, in: International workshop on selected areas in cryptography, Springer, 2003, pp. 175–193. doi:10.1007/978-3-540-24654-1_

13.
[32] F. Mendel, N. Pramstaller, C. Rechberger, V. Rijmen, 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, Springer, 2006, pp. 126–143. doi:10.1007/11799313_9.
[33] S. K. Sanadhya, P. Sarkar, 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, Springer, 2008, pp. 91–103. doi:10.1007/978-3-540-89754-5_8.
[34] I. Nikolić, A. Biryukov, 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, Springer, 2008, pp. 1–15. doi:10.1007/978-3-540-71039-4_1.
[35] G. S. Tseitin, On the Complexity of Derivation in Propositional Calculus, Springer Berlin Heidelberg, 1983, pp. 466–483. doi:10.1007/978-3-642-81955-1_28.
[36] R. Rudell, A. Sangiovanni-Vincentelli, Multiple-valued minimization for PLA optimization, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 6 (1987) 727–750. doi:10.1109/tcad.1987.1270318.
[37] L. Bordeaux, J. Marques-Silva, Knowledge Compilation with Empowerment, Springer Berlin Heidelberg, 2012, pp. 612–624. doi:10.1007/978-3-642-27660-6_50.
[38] M. Brain, L. Hadarean, D. Kroening, R. Martins, Automatic Generation of Propagation Complete SAT Encodings, Springer Berlin Heidelberg, 2015, pp. 536–556. doi:10.1007/978-3-662-49122-5_26.
[39] L. Bordeaux, M. Janota, J. M. Silva, P. Marquis, On unit-refutation complete formulae with existentially quantified variables, in: G. Brewka, T. Eiter, S. A. McIlraith (Eds.), Principles of Knowledge Representation and Reasoning: Proceedings of the Thirteenth International Conference, KR 2012, Rome, Italy, June 10–14, 2012, AAAI Press, 2012, pp. 75–84. URL: http://www.aaai.org/ocs/index.php/KR/KR12/paper/view/4534.
[40] M. Eichlseder, Linear Propagation of Information in Differential Collision Attacks, Master's thesis, Graz University of Technology, 2013. doi:10.3217/p0rxz-6rb72.
[41] A. Biere, M. Fleury, F. Pollitt, CaDiCaL_vivinst, IsaSAT, Gimsatul, Kissat, and TabularaSAT entering the SAT competition 2023, Proceedings of SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions (2023) 14–15.
[42] S. Baldwin, Compute Canada: Advancing computational research, in: Journal of Physics: Conference Series, volume 341, IOP Publishing, 2012, p. 012001. doi:10.1088/1742-6596/341/1/012001.
[43] M. Eichlseder, Differential Cryptanalysis of Symmetric Primitives, Ph.D. thesis, Graz University of Technology, 2018. doi:10.3217/yqfs0-a5c98.
[44] M. Soos, K. Nohl, C. Castelluccia, Extending SAT Solvers to Cryptographic Problems, Springer Berlin Heidelberg, 2009, pp. 244–257. doi:10.1007/978-3-642-02777-2_24.
[45] O. Zaikin, Inverting 43-step MD4 via cube-and-conquer, in: Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-2022, International Joint Conferences on Artificial Intelligence Organization, 2022, pp. 1894–1900. doi:10.24963/ijcai.2022/263.

Appendix

In the appendix, we provide an example of a 38-step semi-free-start SHA-256 collision that we found (Table 4), a table showing its differential characteristic (Table 5), and the starting points that we used in our search.

The 21-step starting point (Table 6) is taken from the work of Prokop [10]. The starting point for 25 steps (Table 7) is an extended version of the 24-step starting point provided by Prokop [10]. The 28-step starting point (Table 8) is a slightly modified version of the starting point used by Mendel et al. [7].

The 38-step starting point (Table 9) is constructed based on the 38-step differential characteristic provided by Mendel et al. [7]—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.

Table 4

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	6f4c0d98	897dd98c	592ba6ad	2aa5e80
M	5b5058d2	901f87fb	254bcfa2	5f8d7dc1	fb1053be	0622e1f8	da8801c2	a951cfbb
	5db42ffd	683b4391	f87eabbd	e928b976	3675cc55	6ebe78be	e3031536	c2de906f
M'	5b5058d2	901f87fb	254bcfa2	5f8d7dc1	fb1053be	0622e1f8	da8801c2	9737d17b
	5db43001	683b4391	f8812bbd	e928b976	3675cc55	6ebe78be	e3031536	c2de906b
h_1	d0e019f7	408269d3	24296a7b	30df8e7f	95d2bff8	34e2bca6	6c50a294	ddb4254a

Table 5
The differential characteristic for the 38-step semi-free-start collision presented in Table 4. 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. [7], an additional two words (∇A_8 and ∇E_{10}) have a zero difference.

i	∇A_i	∇E_i	∇W_i
-4	00110100001110000001101001111111	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	000101110110100101111100011010n00	01101000001110110100001110010001
10	00000000111000001100000100000111	10001011000001111011101010010110	11111000nuuuuuunu010101110111101
11	11011000101000000101000100000110	0unnun011u0n0u1110100n0un1nuuu01	11101001001010001011100101110110
12	10110000000100010101110101000110 0	10000000001000100100001001000101	00110110011101011100110001010101
13	01100100110001111111110111111000	1un0nu0uuunnnnnn00n1nuu0nu100n11	01101110101111100111100010111110
14	00001110111110110011010111100111	00110110001100111100010010010000	11100011000000110001010100110110
15	1100n0101011001u0100111101111101	1111n110000111un1011101111111100	110000101101111010010000011 01u11
16	11011101100011000101110010100u10	unnnnnn00011110001100111 11u0nu10	10110000011011100011010101111110
17	11101100000110000000111011010101	0001111nuuu111nu0unn1110101010nu	11010101101101000100100110000001
18	00110001010011100101110110001111	00000110000111110000111110111100	11000000111100000000000101101111
19	01111001001101010100100010010001	0010n101unnnnnnn0001100110110011	10111010010001001001101110101000
20	00000011110100011000001001011001	1001100001011100101 10unnnnnnnn00	10110011001011110000100101010110
21	11000111011100100100001111011010	01101101111111111111000000000001	01100110010010100111100100010011
22	00100011101011111111110111000001	10101011100001110101011111111110	00010100101010100100011010011101
23	10000010100110001101110100111111	11011110110001000111101101110110	1100u000011001nu0000110101111000
24	01000000000101011100011011111110	01010101111111011000110111111001	100111100011001111000000011001n10
25	11001011100110110010110011011100	10011111100000110000101010100110	00011110010111001111111110111100
26	11100101100100000010111000100010	00001101111100100111101000100101	00101001110010011101001110101010
27	11001011111100101100101000101100	00000100001011101100101111001110	01011100001111001111111000110101
28	11100111000010101101101110111100	10110010110111000000101100101110 0	01011111100011111001000101011010
29	00101001101011001111100101111010	00111110011010011101111100000101	00011001000111111010011001001101
30	00101111100001101001111000010001	00110111000011000101010101011000	01001001101111110100101000011010
31	11011100111011101011001110011111	11011000010000100000000111000010 1	11101111001011100111000001011100
32	01111100111101110111110011000001	01100101010010000110100010000001	00100110101001001100110001000010
33	00001111011000110010100101101000	01010111011001000010011011011001	10111101011000000100111100101100
34	11111100101001110111010000000000	00001011000010011100011011001010	01111101101001000011001100101100
35	01001001101101001110110010010100	10100110101111011110100000101010	11000001100001101101001000000100
36	00100010011101111111010111110001	10101100010010101101111100001110	00100000001111100100010100100111 1
37	00100000111101011111010010010001	10001110110111011111111100011111	00001100100101111011010111101000

Table 6
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 7
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 8
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 9
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	--------------------------------	--------------------------------	--------------------------------