SAT Solver Attacks on CubeHash

Benjamin W. Bloom
Department of Computer Science
Rochester Institute of Technology
Rochester, NY, USA

Committee Members

Chair: Alan Kaminsky

Reader: Warren Carithers

Observer: Reynold Bailey

April 27, 2010
Abstract

The use of a SAT solver to attack a hash function is a fairly unexplored field in Computer Science. This paper examines the use of a state of the art SAT solver to attempt to break a strong hash function. The way that the boolean expression is set up to encompass the function is covered, as well as various results that show specific cases to be unbreakable; other cases are breakable, and example collisions are given. This paper is based on a project which generates the boolean expression for specific function parameters, and is able to interpret the SAT solver output to retrieve and show the actual collision generated.

Contents

1 Introduction 4
  1.1 Hash Functions ...................................................... 4
    1.1.1 Cryptographic Hash Functions ................................. 4
    1.1.2 Applications of a Cryptographic Hash Function ................. 4
  1.2 Boolean Satisfiability ............................................ 5
    1.2.1 Conjunctive Normal Form .................................... 5
    1.2.2 SAT Solvers ................................................ 5

2 CubeHash 6

3 The Attack 7

4 Design 8
  4.1 Building the CNF .................................................. 8
    4.1.1 XOR ..................................................... 9
    4.1.2 AND .................................................... 9
    4.1.3 OR ...................................................... 9
    4.1.4 Addition ................................................ 9
  4.2 Variable Structure ................................................ 10
    4.2.1 Linking the Variables ...................................... 12
  4.3 Variable Storage .................................................. 13
  4.4 Assembling the Expressions ...................................... 13
    4.4.1 Number of Variables and Clauses ........................... 14
    4.4.2 CNF Format ............................................. 16
    4.4.3 Byte Order ............................................. 16
  4.5 Retrieving the Solution .......................................... 16
    4.5.1 Colliding the Messages .................................... 17

5 The Programs 17

6 Results 18
  6.1 Testing Setup ..................................................... 18
  6.2 Brute Force ...................................................... 18
  6.3 Solve Time ....................................................... 18
6.4 Comparing to Brute Force .................................................. 22

7 Future Work ..................................................................... 24

8 Conclusions .................................................................... 25

A Appendix ........................................................................... 26
  A.1 Collision in CubeHash 1/49-512 ........................................ 26
  A.2 Collision in CubeHash 2/96-512 ........................................ 26

B Appendix ........................................................................... 27
1 Introduction

This project covers a largely unexplored topic in Computer Science and Computer Security: Using a SAT solver to attack a hash function. There has been some work done in this area with SAT solvers having been used to attack MD5 and SHA1 [8]. Inspiration for this project came from examination of how to use mathematical inference to attack a hash function, as well as using fixed numbers of message blocks to obtain a collision [4, 5].

1.1 Hash Functions

A hash function takes a message of arbitrary length, and produces a fixed length output from that message. Mathematically, this can be expressed as:

\[ h(M) = H \]

For function \( h \) with variable-length message \( M \), producing fixed-length output \( H \).

They are used for a variety of applications, depending on the strength of the function. Definitions of strength can be found in the section below, on Cryptographic Hash Functions. The size of a hash function’s output is typically expressed with the number of bits that is used to produce the output.

1.1.1 Cryptographic Hash Functions

Cryptographic hash functions have several additional constraints on top of that broad definition. They must be collision resistant, as well as first and second preimage resistant. Collision resistance means that it should be computationally infeasible to find two random messages such that they both hash to the same fixed length output. Preimage resistance means that the algorithm must not be readily invertible. In other words, it should be infeasible to find a message that produces a given hash, when provided with only the hash. Second-preimage resistance means that given a message and an output, it should be infeasible to find a second message which hashes to the same output as the first.

A weak hash function is one that does not meet one or more of the criteria for a Cryptographic hash function. Such algorithms are often used in applications where this kind of security is not required, such as computing keys for hash tables. A strong hash function is one that meets all three criteria for a Cryptographic hash function. A strong hash function is the focus of this project.

The first and second preimage attacks expect to take \( 2^n \) hash function evaluations to successfully find a preimage, where \( n \) is the number of bits to search. A collision attack operates on the principle of the birthday problem, which states that in a group of twenty-three random people, the chance that two of them share the same birthday is fifty percent. The birthday attack to find a collision in a hash function states that it should take \( 2^{n/2} \) evaluations of the hash function in order to expect to find two messages that hash to the same value.

1.1.2 Applications of a Cryptographic Hash Function

Cryptographic hash functions are widely used for applications varying from password exchange protocols to verifying data integrity to digital signatures. If a hash function that is used in one of
these applications can be shown to be weak, then it is no longer fit for use, and the security of the 
application is compromised.

Consider a digital signature. This is generally a form of public key cryptographic security. A 
message is hashed and the resulting hash is encrypted with a Public or Private key (depending 
on which party is sending the message). When the recipient gets the message, they can hash the 
message using the same hash algorithm, and decrypt the encrypted hash using the Public or Private 
key (whichever one was not used for encrypting). If the locally computed hash and the decrypted 
value are the same, then the message is accepted as authentic. If the hash function used for this 
purpose is not cryptographically secure, then an attacker can substitute a different message which 
hashes to the same thing as the original message, and keep the signature, and the recipient will 
accept the attacker’s message as authentic.

These types of applications are the motivation for trying to break a hash function. The suc-
cessful break of the function will rule it out as a good candidate for a secure hash algorithm. 
The inability to break the algorithm does not mean it cannot be broken, but does instill a level of 
confidence in the security of the algorithm that was not there before examination.

1.2 Boolean Satisfiability

The Boolean Satisfiability (SAT) problem is a well-known NP-Complete problem in Computer 
Science theory. It was the first problem shown to be NP-Complete [7]. An expression made 
entirely of boolean variables and operators gets created, and a computer program tries to find an 
assignment for all variables such that the expression as a whole evaluates to true. It may be the case 
that such an assignment cannot be found, in which case the expression is said to be unsatisfiable.

1.2.1 Conjunctive Normal Form

Boolean expressions can be put into many forms; the form that SAT solvers use is called Conjunc-
tive Normal Form (CNF). This form is a conjunction (AND) of disjunctions (ORs). If one wishes 
to express “(A or B or C) and (not A or B or not C)” in CNF, the most common way of doing it is 
to show it like this:
\[(A \lor B \lor C)(\bar{A} \lor B \lor \bar{C})\]

The AND is not shown with an explicit symbol, but it is shown implicitly by parenthetical 
groupings. These groupings are called clauses, and can have arbitrarily many variables in them. 
To satisfy the expression, some combination of truth values for A, B, and C must be made to 
satisfy both clauses. This is a very simple CNF, and a human (or computer) can quickly determine 
that setting \(B = 1\) will satisfy the whole expression, no matter what values are assigned to \(A\) or \(C\).

1.2.2 SAT Solvers

A SAT solver is a program that accepts a CNF as input, and either determines that the expression is 
unsatisfiable, or returns an assignment to all variables that satisfies the expression. State of the art 
SAT solvers have many complicated heuristics that they employ to efficiently find a solution if it 
exists or to determine unsatisfiability if no solution exists. The SAT solver used for this project is 
called glucose. It was built by Audemard and Simon, and was submitted to the SAT competition in 
2009 [1]. This solver was chosen after running some tests with other SAT solvers, and determining
that they were not performing as well as desired. The SAT competition 2009 website showed that the glucose SAT solver performed extremely well in most of the competitions, which is why it was chosen for this project.

2 CubeHash

This is a hash algorithm developed by Daniel J. Bernstein and is a candidate in the National Institute of Standards and Technology (NIST) competition to select a new Secure Hash Algorithm (SHA-3). This algorithm is a good candidate to attack with an SAT solver attack, as it features a very simple round function compared to other hash algorithms, and the only mathematical operations are addition and exclusive-or (XOR). It also uses rotation and swap operators, which add mathematical complexity to the attack, but are not directly expressed with clauses in the CNF for the function. CubeHash is currently one of fourteen candidate algorithms that has made it through the first two rounds of the NIST competition. There are many published attacks on weakened versions of CubeHash, but none on the full-strength versions.

The function features three tuneable parameters, which can be adjusted to change the strength of the function.

- \( r \) - The number of rounds per message block, defined for \( \{1, 2, 3, \ldots\} \)
- \( b \) - The message block size, in bytes, defined for \( \{1, 2, 3, \ldots 128\} \)
- \( h \) - The size of the hash output, defined for \( \{8, 16, 24, 32, \ldots 512\} \)

CubeHash variants are written in the form: “CubeHash \( r/b - h \)” which denotes the three parameters used to tune the function. The algorithm that CubeHash encompasses is very easy to understand, making it nice to examine. The algorithm computes an Initialization Vector (IV) based on the \( r \), \( b \), and \( h \) values. The first \( b \) bytes of the message input are given to the algorithm, it XORs them into the state of the function, which is the IV at the first stage of the algorithm, and then it runs through \( r \) rounds of computation. This same thing is done for each set of \( b \) bytes of input until the entire message has been used; namely, the message block is XORed into the state and then the state runs through \( r \) rounds. The function refers to “words,” which are four byte blocks that encompass a 32-bit integer. When a word of state is referred to in this paper, that means a 32-bit integer, or a 4-byte integer. After all input has been mixed into the state, it is finalized by XORing a 1 into the final state bit, and running through \( 10 \times r \) more rounds. The final hash output is contained in the first \( b \) bytes of the state, interpreted in Big Endian format. The overall structure of this algorithm is shown in Figure 1a.

The message blocks are XORed into the state in Little Endian format. This becomes significant later, during the attack itself. The byte order must be very carefully watched, to make sure that the input blocks are flagged correctly.

Each round consists of ten operations, two sets of Addition, and two sets of Exclusive-Or, four sets of word swaps, and two sets of rotations. After each Addition, but before each XOR, the first 16 words of the state are shifted left by a certain amount. After every Add or XOR operation, words swap positions, to increase the complexity of the round function. The round function is fully invertible, meaning that if you know the full state at the end of a round, you can go backwards.
through the function to retrieve the input state. The rotations and swaps are set up so that if you
know only a partial piece of the state at the end of the function, it is very difficult to backtrack and
find the state at the start of the round.

The CubeHash round function’s structure is shown in Figure 1b. This diagram was taken from
Brier and Peyrin’s paper on Cryptanalysis of CubeHash[6]. A very detailed layout of how this
works is in Section 4.2, where variable structure is discussed.

3 The Attack

Due to the nature of the complete algorithm, the approach chosen for the attack focuses on only
the first section of the algorithm, that is to say the input-mixing stage, as opposed to going from
a final hash output and trying to work back to an input that produces that output. This means that
this will be a Collision attack, which is briefly discussed above, as an attempt to find two random
messages that produce the same hash output. Recall that a strong hash function must be resistant to
a collision attack, in that it must be computationally infeasible to find two messages that produce
the same hash output.

The general structure of the attack will be to find two messages that collide in the last $(128 - b)$
bytes of state, accounting for byte order. However, the collision in those bytes only needs to happen after a certain number of message blocks has been made. The two messages as a whole must also be unequal; this is shown in Figure 2. In this figure, M1_B1 represents Message 1, Block 1, and M2_S2 represents Message 2, State 2, which is the state after one round of mixing has been completed. This shows that the message blocks as a whole must be unequal to the other message’s blocks as a whole. Only one bit needs to be different across the two messages. This diagram shows three message blocks, but the idea carries from 1 message block to \( n \) message blocks, the setup is the same. Notice that only the last \((128 - b)\) bytes of the state of each message after however many blocks needs to be equal. The reason for this is that if those bytes are equal, then we can select a message block which wipes out the differences in the first \( b \) bytes of the state. At that point, the state of the algorithm will be the same for two distinct messages, and the messages will then hash to the exact same hash output.

4 Design

4.1 Building the CNF

To build a CNF expression to encompass this attack, we need to know what operations to use. There are two operations that matter for actually turning into CNF, those are Addition and Exclusive-Or. The rotation and swap operations are not expressed in CNF, except in the way that the connections between variables are set up. While extremely important, it does not add significance to what the
generic expressions say. The way that Addition will be set up also requires that the operations OR and AND be expressed in CNF as well.

The XOR, AND, and OR gates in CNF are pretty straightforward. They consist of three variables, $A$, $B$, and $C$, and are shown below. $\overline{X}$ means the negation of variable $X$. In all three of these gates, they are relating variables $A$, $B$, and $C$ such that $A < op > B = C$. These expressions come from representing the truth table for each operation, and then producing a CNF for that truth table.

4.1.1 XOR

An exclusive or gate that expresses $A \oplus B = C$ in CNF looks like this:
$$(\overline{A} \lor \overline{B} \lor \overline{C})(A \lor B \lor C)(A \lor \overline{B} \lor C)(\overline{A} \lor B \lor C)$$

4.1.2 AND

An AND gate that expresses $A \& B = C$ looks like this:
$$(\overline{A} \lor \overline{B} \lor C)(A \lor \overline{B} \lor C)(A \lor B \lor \overline{C})(A \lor B \lor C)$$

4.1.3 OR

An OR gate that expresses $A \mid B = C$ looks like this:
$$(\overline{A} \lor \overline{B} \lor C)(A \lor B \lor C)(A \lor \overline{B} \lor C)(\overline{A} \lor B \lor C)$$

4.1.4 Addition

The addition operation is more complicated. There are three cases for addition. One in which the carry-in is known to be zero (the first adder in each word), one in which the carry-out does not matter (the last adder in each word), and the more general case where both carry-in and carry-out matter. A slight optimization that we can give to the SAT solver is that we know the glucose
SAT solver will want to change all clauses to 3-SAT before doing computation. This means that it wants all clauses to have at most three variables in them. An addition gate usually consists of five variables: Two inputs, the carry in, the sum output, and the carry out. However, what we can do is assign temporary variables, such that the CNF of the ADD gate comes out to have only three variables in each clause. These temporary variables are shown in Figure 3.

Due to the way the add gate works, it turns out that it can be shown as the concatenation of other gates. That is what is shown below. It is easiest to show the Addition gates as macro-like wrappers around operations/expressions formed above. A good way to interpret the macro expressions is to think of them as: operator(input1, input2, output).

**Carry-in equals zero**  The case where the carry-in is known to be zero only needs to compute the Sum output and the Carry out using A and B. The Sum output is simply $A \oplus B$, and the carry out is $A \land B$, so the CNF for it will be these expressions, expanded out:

$$(\text{xor}(A, B, S))(\text{and}(A, B, Co))$$

**Carry-out doesn’t matter**  The case where the carry-out does not matter only needs to compute the Sum output. However, it needs to compute the temporary variable $M = A \oplus B$, and then compute $S = M \oplus Ci$, which ends up encompassing the expression $S = A \oplus B \oplus Ci$.

$$(\text{xor}(A, BM))(\text{xor}(M, Ci, S))$$

**General case of Addition**  The general case which is used in the vast majority of cases requires calculating both the carry out and the sum output, using the three inputs. This is shown below:

$$(\text{xor}(A, B, M))(\text{xor}(M, Ci, S))(\text{and}(A, B, P))(\text{and}(M, Ci, Q))(\text{or}(P, Q, Co))$$

### 4.2 Variable Structure

Since we know the general structure of the expression, we need to know how to set up the variables themselves. That is to say, we need to know where to place them in memory, and in the CNF, so that they can be set and retrieved in a meaningful way. The way to approach this is to first look more deeply into the round function, and see how many variables we need to describe a round. The best way to show this is with a diagram, the one shown in Figure 4 does this job. The “⊗” symbols in the diagram are Addition gates, and the “⊕” symbols are XOR gates. Bit rotations are not shown, but word-swapping is shown. The initial state of the round is contained in words 0-31. The output from the round is contained in words 64-95, but in the jumbled order shown. What this means is that if the function were dealing with a 2-round variant of CubeHash, then input word 0 to the second round of computation would actually be output word 80 from the first round. The way that this variable structure was set up is starting from the base state of the round (words 0-31), and doing the first addition produces 16 new words (32-47), the first XOR produces words 48-63, the second addition produces words 64-79, and the second XOR produces words 80-95. When the round function’s structure is examined more closely, one can see that if all but one byte of word 0 are locked in place, then nearly all of the output state is already resolved, leaving very little for the SAT solver to try and guess. In fact, if any bytes in the first 16 words of state are locked in place, and assuming that the upper 16 words are also locked, then some of the round’s output bytes will be known as well.
This kind of structure is unique to the first round of input, even in a multi-round setup (2 or more rounds per mixed input block). However, the overall pattern continues for multi-round variable structuring, and only the first Add and XOR of the first round is different from the first Add and XOR of the subsequent rounds. In all of these word assignments, what it’s actually showing is the overall structure of what is getting assigned. Within each word are the 32 variables (bits) that are getting linked together with the other variables in the words that are adding or XORing together. Each word is treated as a Big Endian word; however, they could also be treated as Little Endian. There is a lot of Endianness checking and switching that occurs in this algorithm, especially when flagging equality and inequality of various message blocks, as well as retrieving the final solution. The reason that Big Endian is preferred is that while the CubeHash specification says that the state words are interpreted as little endian, the algorithm itself does the initialization in Big Endian, and the message blocks are put into the state in little endian format, so it seems to instead leave the state as big endian, and interpret the message input as little endian. Everything works correctly when you assume that the message is being input in little endian format into the state, but it does mean that one must be very careful of byte order when flagging variables for equality and inequality.
4.2.1 Linking the Variables

Following along with the diagram will help with this section. All of the variables need to be linked together, starting with the first Add gate in the first round, which adds together words 0 and 16, and puts the result into word 32. This can be shown algorithmically, like so:

\[
\text{for } i < 16 \text{ do} \\
\quad \text{state}_{i+32} \leftarrow \text{state}_i + \text{state}_{i+16} \\
\text{end for}
\]

However, when the number of rounds is greater than 1, the above algorithm is still how to assign the first adder for the first round, but the first adder for rounds 2 and up must be done slightly differently, in that it is split up into two separate loops. This is because the sets of two words that must be added together are no longer 16 apart. Word 80 must add to word 65, and word 81 must add to word 64. This pattern does continue for all subsequent rounds in the multi-round version of the algorithm. This addition structure looks like the following:

\[
\text{for } i = 0, i < 16, i+ = 2 \text{ do} \\
\quad \text{state}_{i+33} \leftarrow \text{state}_i + \text{state}_{i+17} \\
\text{end for} \\
\text{for } i = 1, i < 16, i+ = 2 \text{ do} \\
\quad \text{state}_{i+31} \leftarrow \text{state}_i + \text{state}_{i+15} \\
\text{end for}
\]

In the actual algorithm, there are also offsets defined to tell which round we’re at in the computation, to be able to tell which variables we are setting. This is because round 2’s input starts at word 64, round 3’s starts at 128, and round n’s starts at \(64 \times (n - 1)\).

The first XOR gate in the first round XORs together words 0 and 40 and puts the result into word 56, however, words 0-15 are swapped around, as seen in the figure, so the algorithm must account for this. This also means that words 8 and 32 are XORED together, and the result placed into word 48. This is shown in the figure, but we can show it algorithmically as well. Those first 16 words are also rotated left by 7 bits (not shown in the figure), but shown below with the “\(\ll\)” symbol.

\[
\text{for } i < 8 \text{ do} \\
\quad \text{state}_{i+56} \leftarrow (\text{state}_i \ll 7) \oplus \text{state}_{i+40} \\
\quad \text{state}_{i+48} \leftarrow (\text{state}_{i+8} \ll 7) \oplus \text{state}_{i+32} \\
\text{end for}
\]

Similar to the Addition gate for the first round, the first round for the multi-round version for rounds 2 and above is slightly different from the version shown above, which is only for the first round per mixed message block.

\[
\text{for } i < 8 \text{ do} \\
\quad \text{state}_{i+56} \leftarrow (\text{state}_{i+16} \ll 7) \oplus \text{state}_{i+40} \\
\quad \text{state}_{i+48} \leftarrow (\text{state}_{i+24} \ll 7) \oplus \text{state}_{i+32} \\
\text{end for}
\]

At this point, we can show the second addition and second XOR functions. These are the same for the 1-round and the multi-round versions of the algorithm. We’ll start off with the second addition. Notice that in the diagram, the second addition operates on the third row of 32 words, and produces the second half of the fourth row. There is a pattern to the way things are added together,
but the input words jump by two. An easy way to take care of this is to just have a static array that you iterate across and use to show the offsets that we’re adding together. So, keeping in mind how the diagram shows this second add happening, this is what we have:

```plaintext
for i ∈ [34, 35, 38, 39, 42, 43, 46, 47] do
  state_{i+30} ← state_i + state_{i+14}
end for
for i ∈ [32, 33, 36, 37, 40, 41, 44, 45] do
  state_{i+34} ← state_i + state_{i+18}
end for
```

Then, the second XOR does a similar style computation, but uses four loops instead. As with the above code, follow along with the diagram to see why the loops are set up like they are. The second XOR also has a rotate left by 11 bits, so that must be accounted for as well.

```plaintext
for i < 4 do
  state_{i+80} ← (state_{i+52} ≪ 11) ⊕ state_{i+64}
  state_{i+84} ← (state_{i+48} ≪ 11) ⊕ state_{i+68}
  state_{i+88} ← (state_{i+60} ≪ 11) ⊕ state_{i+72}
  state_{i+92} ← (state_{i+56} ≪ 11) ⊕ state_{i+76}
end for
```

### 4.3 Variable Storage

This shows the overall structure of the round variables, but we have not yet covered all of the temporary variables that must be assigned for the Add gates. Each Add gate creates 64 words of temporary storage for its temporary variables. These names are shown in Figure 3. Each Add gate has 16 words for its carry variables, 16 words for $M = A \oplus B$, 16 words for $P = A \& B$, and 16 words for $Q = M \& Ci$. These temporary variables are placed after the end of the round block’s main variables. This is shown in Figure 5.

This shows the storage allocation for two blocks of two rounds each. The “128 – $b$ bytes for equality” is used to set the last 128 – $b$ bytes of the previous block’s output (second Add and second XOR operations’ variables) equal to the last 128 – $b$ bytes of the “current” block’s input variables. This is done so that the current block’s first $b$ bytes may be free to take on any values that they wish. If this were not done, then for intermediate blocks of input, the only values they could take on would be all zeroes, as they would have to exactly equal the previous round’s output in the big chain of rounds. Initially, we had tried adding on more rounds for adding on more message blocks, but found this zeroing behavior, and had to switch to the current method, which allows those blocks to range as necessary. The temporary bytes used for equality ensure that the XOR operation that would be done in the normal CubeHash algorithm is upheld, since it would not be able to freely change the last 128 – $b$ bytes of input either.

### 4.4 Assembling the Expressions

Now that we have a way to describe the operations that we need to express, and the variable structure is exposed, the work of crafting the main CNF can begin. The best way to break this
down is piece by piece. There are two sets of message inputs. Set 1 will be for message 1, and set 2 will be for message 2. Within each set are a number of round blocks, one block represents a single message block of input, and the rounds and variables for those rounds that are needed to mix that input block into the state. Then, within each block, there are a number of rounds, and within each round, there are a bunch of operations (additions and XORs).

4.4.1 Number of Variables and Clauses

Each round consists of thirty-two 32-bit adders, and thirty-two 32-bit XORs. Each adder also has 32 bits for carry variables, and three more sets of 32 bits for the temporary variables. That is 5 \times 32 variables per 32-bit adder. This becomes 32 words for the initial state, the XORs also create 32 words, and the Adds create 160 words. This is a total of \((32 + 32 + 160) \times 32 = 7168\) variables per
round function.

However, we can have multiple rounds per message block, and the multi-round version does not need double the number of variables, since the output of one round and the input to the next are the same words. Each additional round adds on 32 words for the XORs, and 160 words for the Adders. What this comes out to is \((224 + 192 \times (r - 1)) \times 32\) total variables for one round block. There also needs to be space reserved for the equality from this message block to the next, so that adds on \((128 - b) \times 8\) more variables.

Additionally, there can be any number of round blocks (one round block equals one message block of input). This comes out to:

- \(variables\_per\_block = (224 + 192 \times (r - 1)) \times 32 + 8 \times (128 - b)\)
- \(variables\_per\_set = variables\_per\_block \times num\_blocks\)

Each operation must be expressed in CNF as many clauses consisting of ORs, ANDed together. Using the definitions made in Section 4.1.4, there are \(8 + 20 \times 30 + 8 = 616\) clauses per Add gate, since the two edge cases use 8 clauses, and the 30 general cases use 20 clauses. In the XOR section above the Addition section, it is shown that an XOR takes 4 clauses. There are thirty-two adds and thirty-two XORs, making \((8 + 20 \times 30 + 8) \times 32 + (4 \times 32) \times 32 = 23808\) clauses per round.

There are two sets of clauses denoting the two messages, each set has a number of \(r\)-round blocks. The two messages need to be non-equal. The SAT solver must know about this constraint, and it cannot simply do an equivalence check, so the inequality check must be built into the CNF expression. The best way to achieve this check is to XOR each bit of message 1 with its corresponding bit in message 2, put the output of that into some variables, and then OR all of those output variables together. If any one of the pairs of message bits are not equal, then the XOR of the input bits will come out to 1, and the large OR clause will evaluate to true, meaning that the messages are not equal as a whole. This means that there are \(8 \times b \times num\_blocks\) variables that need to be added to the CNF to accommodate this extra clause for inequality, and 4 clauses for each of those variables, since they are XORs.

The output side also has \(128 - b\) bytes that need to be equal (refer to Figure 2 if clarification is required). This inequality will be achieved by XORing those \(8 \times (128 - b)\) variables at the end of each set together, and putting the results into a bunch of variables, and then ANDing together the negation of all of those variables. What this achieves is \(8 \times (128 - b)\) unary clauses and \(4 \times 8 \times (128 - b)\) XOR clauses that say that all of those unary clause variables must be false (meaning that the XOR of the variables at the end of the sets is zero, making the variables equal).

There must also be XOR clauses for the equality variables from one round block to the next, so that the intermediate blocks of the CNF are able to be non-zero. So there are \(8 \times (128 - b) \times (num\_blocks - 1) \times 5\) clauses for that, since the last block does not need the extra variables or clauses.

Finally, there must also be an IV specification for both message blocks, so this requires \(2 \times 8 \times (128 - b)\) more clauses (but no more variables).

In total, this comes out to:

- \(Number\_of\_Clauses = (23808 \times r \times num\_blocks) \times 2 + 1 + 4 \times (8 \times b \times num\_blocks) + 7 \times 8 \times (128 - b) + 8 \times (128 - b) \times (num\_blocks - 1) \times 5\)
\begin{itemize}
  \item \textit{Number of Variables} = \((224 + 192 \times (r - 1)) \times 32 + 8 \times (128 - b)) \times \text{num\_blocks} \times 2 + 8 \times b \times \text{num\_blocks} + 8 \times (128 - b)\)
\end{itemize}

### 4.4.2 CNF Format

The SAT solver accepts an input file in DIMACS CNF format. This format asks for a specific number of variables and clauses that the CNF contains. The variables in the CNF start at 1 and go up from there. The variables do not have to be continuous numbers, and the negation of variable \(x\) is expressed \(-x\). All variables are integers. The variables do not start at 0, because each clause is separated by a “0” term. That is to say, that a zero should be at the end of every disjunction. Disjunctions can span multiple lines, it is the zero term that matters for separating clauses from each other. Multiple clauses can even appear on the same line. The output from my program is in this format, with one disjunction per line, this is to make the number of lines of the CNF be equal to the number of clauses that is declared at the top of the file.

### 4.4.3 Byte Order

It is mentioned above that byte order is very important for setting equality and inequality of message blocks and state variables. When showing the general outline of the attack, and referencing Figure 2, the message blocks as a whole must be unequal to each other. This inequality is set in Little Endian format, since the message blocks are put into the state in Little Endian format, so if a state word is of the form \([b_3, b_2, b_1, b_0]\), and a message block is interpreted as a stream of bytes \([\text{word}_{1b3}, \text{word}_{1b2}, \text{word}_{1b1}, \text{word}_{1b0}]\), then the first XOR operation that must take place is \(b_0 = b_0 \oplus \text{word}_{1b3}\) when inputting message bytes. So \(b_0\) must be the first byte of that state word flagged as “not equal” to the corresponding byte in the corresponding word of the other set for the big clause for the inequality of the two input messages.

Then, at the same time, the last \((128 - b)\) bytes of every message block input must be set equal to the last \((128 - b)\) bytes of the block output from the previous input block for the given set (but this does not carry between sets). This equality must be set in Big Endian format, since the inequality of the first \(b\) bytes of that very same state block is being flagged in Little Endian format. This equality and inequality will cover the whole of the 128 bytes of state for that message block, so it is very important to get the byte order and variable flagging order correct, since this matters very greatly when grabbing the solution out of the SAT solver output.

### 4.5 Retrieving the Solution

The SAT solver output lists all of the variables from the CNF and shows the values that satisfy the CNF. These are still in DIMACS CNF format, so a variable without a minus sign means that variable should be set to true and a variable with a minus sign means that variable should be set to false. We calculated above how many variables there are per message block, so what we know is that the first \(b\) bytes of each block are the bytes that we need to grab for the messages. These bytes must be grabbed in Little Endian format. My program retrieves the variables in Little Endian format and reassembles them in Big Endian format (since the main algorithm that will re-hash both messages to make sure the outputs are equivalent needs them in Big Endian so that it can transform
them to little endian while using them as input). Byte order is very important, and careful attention must be paid when getting the solution out of the file, since the endianness switches.

### 4.5.1 Colliding the Messages

We have calculated how many variables are in each set of message blocks, so it is easy to obtain both messages from each set. Then, the hashing algorithm can run up through the last message block that has been resolved. In Figure 2, this is \( M_{1 \cdot S3} \) and \( M_{2 \cdot S3} \) for the two messages. They are put through \( r \) rounds, and produce \( M_{1 \cdot S4} \) and \( M_{2 \cdot S4} \). At that point, we know that the last \((128 - b)\) bytes of those two blocks are equal. A message block needs to be crafted such that all 128 bytes of both blocks become equal after the message is XORed in. The simplest way to do this is to assign \( \text{block} = M_{1 \cdot S4} \oplus M_{2 \cdot S4} \) and use that as the message block, recalling that byte order must be preserved, so the XOR must be done and made into the message block by switching the endianness of the resulting block. One thing to note is that any message at all can be put in for message 1’s block portion, and it just needs to be XORed with message 2’s block in the same fashion. The big thing is that the differences need to be wiped out, so that after inputting the message block, the two states are identical. Represented in its final form, \( \text{block}_1 = M_{1 \cdot S4} \oplus \text{msg} \), and then \( \text{block} = \text{block}_1 \oplus M_{2 \cdot S4} \), to have an arbitrary crafted \( \text{msg} \) for input 1.

The messages have now been successfully collided and can then be hashed. The final hash output is identical. More data can be appended to the end of both messages, as long as the appended data is the same for both messages.

### 5 The Programs

The CNF producer and solution reader are written in java. The RoundCNFProducer creates a CNF given the inputs \( r, b, h, \) blocks, and an output file. These represent CubeHash’s \( r, b, h \) values, as well as a number of message blocks, and the file to which the CNF should be written. The CNFSolutionToHex program will take inputs \( r, b, h, \) blocks, and an input file, and retrieve the solution from the input file based on those parameters. As the name of that program suggests, it will retrieve the solution, and show the hexadecimal formatted pair of messages that collide. It also shows whether or not the hashes of the two messages do indeed collide, since it’s possible that the solution that is being retrieved had specified the wrong parameters. It shows the hash of each message as well.

The programs might be invoked as follows:

```
java RoundCNFProducer 1 49 512 1 cnf.txt
./glucose_static cnf.txt solution.txt 2> progress.txt
java CNFSolutionToHex 1 49 512 1 solution.txt
```

The `glucose` program must be invoked in a linux environment, the other two programs do not have to be. The reason that the standard error stream is redirected for the SAT solver is that it prints a lot of progress messages, including the final timing data of how long it took to run. The format of the solution that is output has two lines. The first line says if the solution were Satisfiable or Unsatisfiable. The second line contains all of the variables and their assignments that make the
whole expression evaluate to true (if a solution exists). If no solution exists, then the second line is blank. The format of the variables is the same as the DIMACS input format, in that if a variable \( x \) is set to 1, then it is just \( x \), but if it is set to 0, then it is \( -x \). For example, with variables 1, 2, 3, 4, and 5, if 1 and 5 are set to zero, then the solution will be “\(-1 2 3 4 -5\)” when read from the file.

6 Results

6.1 Testing Setup

All tests were done on an AMD 2600+ (2.1 GHz) processor chip, in Ubuntu 8.1, with 1 GB of RAM. The tests were all executed with the glucose SAT solver, after other SAT solvers that had been tried proved to be too inefficient.

6.2 Brute Force

A brute force attack will expect to take a certain amount of time, based on the block size, number of blocks, and number of rounds \([3, 2]\). Determined based on how long it takes to hash a single message, the solve time for a variant of the function given the number of blocks used for finding a message block can be observed. This is done by hashing a message with each variant hundreds of thousands of times, and calculating the average hash time based on the aggregate time used divided by the total number of hashes performed. Only the first part of the hash actually needs to be done, the finalization stage does not need to be completed, since the attack never touches the finalization stage. Thus, only the specified number of round blocks are actually hashed, using an initialization vector that was precomputed for that function variant. This method assumes that the attack will try to use the least number of bits when attempting to break the function. This implies that the largest search space will be 512 bits, since if \( b \leq 64 \), then the input side will be tested, and if \( b > 64 \), the output side will be tested. This is because the outputs need to be equal with the inputs being unequal; the last \((128 - b)\) bytes of output would be set and the inputs would be searched. Alternatively, the inputs would be searched, and then there would be a search for a collision in the last \((128 - b)\) bytes of output. The search problem should become more and more difficult the closer to \( b = 64 \) that the attack reaches. The brute force attack’s solve time will be the time that it takes to hash one message, multiplied by the expected number of messages. With \( b = 1 \), the number of expected messages is \( 2^{8b/2} = 2^{4b} = 2^4 = 16 \). The time taken to hash a single message is \( \tau \), so the total time to find a collision is \( \tau \times 2^{4b} \) for any given message where \( b \leq 64 \) and \( \tau \times 2^{4(128-b)} \) for a given message where \( b > 64 \).

6.3 Solve Time

The ideal results to show with this attack would be a break of a strong version of the function, namely CubeHash 8/1 or CubeHash 16/32. However, it quickly became apparent that anything dealing with more than two rounds was very difficult to get a solution (or unsatisfiability) for the CNF encompassing that algorithm variant. The baseline from which other tests will be measured, is therefore CubeHash \( 1/b - 512 \), with a single message block to collide. This means that only one message block will be searched, and there will only be one round to go through. This is the
simplest possible case to produce a collision in the full algorithm. This is also the most promising and probably the most interesting case, due to the solve time (or the time taken to resolve unsatisfiability). Due to the way a brute force collision attack works, we would expect it to take $2^{n/2}$ messages to find a collision in two messages, where $n$ is the number of bits to search. Keeping this in mind, let’s look at Figure 6 and observe the time it took to solve for every available value of $b$.

![Figure 6: Solve time for CubeHash 1/b-512, with 1 message block](image)

The highest Unsatisfiable $b$ value was 48, and the lowest Satisfiable $b$ value was 49. All of $b$ values from 1 through 48 were Unsatisfiable, and all of $b$ values from 49 through 128 were Satisfiable. The longest solve time for CubeHash $1/b-512$ with one message block was at $b = 55$, with 35 seconds to solve. The collision generated for CubeHash $1/49-512$ with 1 message block may be found in the Appendix. It shows that there are two message blocks; however, this is because only one block is used to generate the collision in the last $(128 - b) = 79$ bytes of the state after that one block, the second block is used to wipe out the differences in the lower 49 bytes of the state. This is very clearly doing something better than a brute force attack would, since it found a solution across 49 bytes (392 bits) in about 20 seconds. With a standard brute force collision attack we would expect it to take $2^{196}$ messages to resolve a collision with 392 bits to search. The reason the solve time creates that triangular shape is because with $b = 1$, there are 127 bytes of IV that are locked in place, and cannot change, meaning that there are only 8 bits that are free to range and change the end hash value. Recall from the way that variables are structured, that since most of the input state is known, then most of the output state is also known, leaving very little for the SAT solver to try and guess.

An interesting note is that if the solver returns saying that the variant is unsatisfiable, then there is no possible solution for that function variant with that specified number of message blocks. This is an interesting point, because Brier and Peyrin showed that CubeHash $1/36$ can be broken with
two message blocks[6], and my results show that CubeHash 1/36 cannot be broken with only one message block.

As the value for $b$ climbs up, the solve time looks like it’s scaling exponentially with the number of bits to search; however, it does not look like it is scaling with $2^{8b}$. This scaling also switches and looks to be exponentially decaying after $b = 55$. This is likely due to having less and less of the final $(128 - b)$ bytes of output locked in place, and it becomes much easier to find two messages which collide in those last bytes of output after a round of the hash function. This is because while the search space for the input is increasing, the search space for the output is decreasing. We would probably expect the midpoint ($b = 64$) to be the cross point.

This is showing something known as the “phase change phenomenon” in random k-SAT problems[]. This states that the closer that the ratio $\alpha = \text{Clauses}/\text{Variables}$ is to $\alpha = 4.26$, the harder it becomes to determine a solution. To search and see if this is what is happening, the number of variables and clauses that the SAT solver has to search must be found. This is done by letting the solver do its pre-processing phase, where it assigns all unary clauses, and propagates variables until it cannot make any further assignments without guessing. At that point, the number of variables and clauses are recorded, and the ratio $\alpha$ is obtained. This is shown in Figure 7. It is extremely interesting to note that the ratio never comes close to 4.26. It becomes most difficult to solve for CubeHash 1/b-512 when $\alpha = 1.79$, and CubeHash 2/b-512 around the same $\alpha$ value. The reason for this is likely due to the fact that the $\alpha = 4.26$ value is for randomly structured k-SAT problems, whereas this is a highly structured problem.

![Figure 7: Clause/Variable ratio for CubeHash 1/b-512, with 1-2 message blocks](image)

This gives us a clue to how the solve time will scale with $b$ changing but everything else held constant, so the next case examined is how solve time scales with the number of message blocks used. It was found to be more informative to look from the largest $b$ values going down to smaller values, and seeing how the time scaled with that. With that in mind, looking at values of $b$ from 64
through 128, we come up with Figure 8.

**Figure 8: Solve time for CubeHash 1/b-512, with 1-4 message blocks**

With two message blocks, $b = 48$ comes back satisfiable, but $b$ values lower than that took too long to resolve an answer one way or the other. This would be a great area for future work to look. We know from Brier and Peyrin’s paper[6] that there is a collision with two message

**Figure 9: Solve time for CubeHash r/1-512, with 1 message block**
blocks in CubeHash $1/b-512$. This test was attempted, but the solver ran for over 6 days (525,000 seconds) without finding a solution. A large part of the issue here is that the SAT solver treats all variables equally. It does not know that it only has to look at a fairly small subset of the total variables in order to find a solution. Recall that from the variable storage section, the majority of the variables in the CNF are temporary variables associated with the Addition gates. This shortcoming becomes obvious when examining CubeHash $r/1-512$, and seeing how the solve time scales up when there is only one message block, one byte block size, and a changing $r$ value, seen in Figure 9.

Due to the way that the round function is set up, there were some notable jumps in solve time between $b = 64$ and $b = 63$, as well as between 4-byte boundaries below $b = 64$. Looking at Table 1, these jumps stand out, between $b = 59$ and $b = 60$ as well.

Delving further into looking at 1-4 blocks, for all $b$ values, we come up with Figure 10. With 2 message blocks, it looks like the peak will occur somewhere around $b = 36$. With 3 message blocks, it looks like a similar $b$ value will have that solve-time peak. Mapping these peaks is a good area for future work, as well as figuring out why exactly the peaks are different for different numbers of message blocks. The values in between the two sides of the data, where the peaks occur, are unknown. These were values for which the solve time was large, in terms of real time.

### 6.4 Comparing to Brute Force

Comparing the solve time to brute force is important for determining the viability of this attack, when viewed in relation to the time a brute force collision search would take when treating the round as a black box. The first comparison is for the best-performing case of CubeHash 1/b-512, with 1 message block. Figure 11 shows the analysis of solve time on a logarithmic plot. This is calculated by figuring out how long a message takes to hash with the given variant of CubeHash, and then multiplying that time by the expected number of messages ($2^{4b}$), and then taking the base 2 logarithm of that result. Using the notation in section 6.2 on Brute Force, this formula is $\log_2(\tau \times 2^{4b})$. This gives the expected time that a brute force collision attack would take. Also shown is the logarithm of the time taken by the SAT solver to come up with either a solution, or deem the problem unsatisfiable. Figure 12 shows this same style of plot for CubeHash 2/b-512,
Figure 10: Solve time for CubeHash 1/b-512, with 1-4 message blocks

with 1 message block. In both cases, the SAT solver time does appear to be doing an upward slope as it gets towards $b = 64$, however, this slope is a lot less steep than the brute force solve time. Based on these plots, it seems very clear that the SAT solver is performing much, much better than a brute force collision attack would perform. This is especially obvious when looking at the absolute solve time for $b = 55$, which was 35 seconds, and comparing that to $\tau \times 2^{220}$ seconds, there basically is no meaningful comparison, other than to say that the SAT solver completely dwarfs the brute force attack.
7 Future Work

There is great potential for future work in this area, even with the CNF generator from this project. Some things that could be done to make it more flexible are to allow for differing numbers of
message blocks between the two sets, for a collision (i.e. colliding a message with 3 blocks with a message that has 1 block). Providing exact input for one of the messages, thus executing a second-preimage attack, instead of a collision attack. Starting with a hash output, and working backwards through the finalization stage, then through an arbitrary number of message blocks to an input that matches the IV in the last \((128 - b)\) bytes of state. That would be a first preimage attack. Being able to tell the SAT solver which variables are important (the blocks representing the two input messages, and the output bytes that are checked for equality denoting a collision). Doing some of the SAT solver’s work for it, by doing some inference and figuring out a lot of the variables that are already in the system and do not need to be guessed.

8 Conclusions

SAT solver technology will need to be improved to account for the fact that not all variables need to be treated equally in the CNF expression. However, the glucose SAT solver performed incredibly well on CubeHash \(1/b - 512\) with one message block. Based on this project, CubeHash may have the potential to be broken by a SAT solver that is geared towards Cryptographic Hash Functions. The results presented here do not approach breaking the recommended-strength versions of the function (CubeHash 8/1 or CubeHash 16/32). However, the results do show that the lower round versions of the function can be subjected to a SAT solver attack that results in either collisions, or a statement that no collision can possibly be found, in less time than a brute force attack. A state of the art SAT solver performs well for specific variants of the function, but did not do well on more complicated versions.

References


A Appendix

A.1 Collision in CubeHash 1/49-512

The collision message blocks are specified in Big Endian format.

| Message 1 Block 1 | 70eefd5d 4d9b2e72 3dd8e222 66514ab4 8df74891 41b80b46 9ab4d3c7 b26ae02b bd00cd12 9c46ee6e ac86c42c 926c6bd1 12 |
| Message 1 Block 2 | a42b7f07 045411c9 698082cc 74995b49 afb9ca3f f86198e1 77086416 f26662c1 e18b3e41 06f37fed 5d3bfc13 453b1a00 fb |
| Message 2 Block 1 | 702ec25d 4d9b2e72 3d18ed22 66514ab4 8df74891 41b80b46 9ab4d3c7 b26ae02b bd00cd12 9c46ee6e ac86e42c 926c6bd1 12 |
| Message 2 Block 2 | a42b7f07 045411c9 698082cc 74995b49 afb9cc21 f86198e1 88086494 f26662c1 e18b3e41 06f37fed 5d3bfc13 453b1a00 04 |

Hash

345c303bc5b8391b4eb2ade91dd4957f2eda7cbcaa973f7baa8f79e556c2ef52 dbd818f1cca433c35a1eb0d58e1dc0b2f2a85c93d3cf842b3da9439d2846aa7f

A.2 Collision in CubeHash 2/96-512

The collision message blocks are specified in Big Endian format.

| Message 1 Block 1 | 5376fa60 3a78b2be 49bfe958 8487eab8 569ca09f dd7e7d22 da64a353 6f2886b7 dad94oa1 a6b7caaa5 e8063b1e 35992d48 23378f37 373dd8d5 f45852a2 9950c3a5 2db4d9cb 61aee861 77209db7 5b2c61b5 d7914542 e33566ba 4154b1d2c 04353790 |
| Message 1 Block 2 | 79b1a8e1 1d60740b 02554941 e30f8df6 5223670b 71e5b962 9f9a9422 6e563116 5ff8f775b d03a3b32 531e23fb d4ef58a farc36c6 6b8215b 632c6d6b 6e2d2096 da695ab7 abea0d8 d049bee8 9a7a597d c663a80 2038e40a 06ac46ac c1fa80c6 |
| Message 1 Block 3 | 104eeae3 2cd3e73a 64623e65 a7181c68 47ade04d 1eed07d4 b0237c69 978aa446 6206cf58 bcf613e5 28901d26 291bcdd 5e9cfb00 7d5afdf6 c85ffede 0c1bd404 1974b63 0cc3375 5e129a9a 9d880ea9 d2b97d70 c9f162de 470c4ef6 58ef016 |
| Message 2 Block 1 | 6c957dec 975ac562 91e7da84 654e6809 f6ed05ef d5e9f9ed e5871ae9 74003488 e04b9bc7 d103ab50 7de4c8b 2d613bb1 2dad23c6 b7ca7577 2ee1b80e 8f3f07f 086b2baa d6a6c4ee fa854ab0 fadfd8f7 3e6820f4 0a7a7c0c 5c27a27c |
| Message 2 Block 2 | cdcc1c9a 08d487ef e2e01866 53cb163c d9b2a56 81a56c73 2ac8800f 534fdd99 bc19f3e2 f49e6a0 30e49f69 1290c5ba 49516051 1cd3f34b 86cd5980 741ef5ca 50927e0f 594703b3 55aab64a f39f9e9e c7f6a0b6 9540b57 6ae123b 9c72a572 |
| Message 2 Block 3 | a83211eb eea7cb5f 71ac6ae 0954b2ef 30723adc 717d9f00 965b14a 130342cc c433e652 395c7e7b cb0bcd6b 20ee644 30fc501e 03790b4f b75358c1 baae8f8f d418f924 063ad3ce 6bf70944 fd68d1d8 89626bec 24385257 939c879 af4a648c |
B Appendix

In order to compile the programs, all that needs to be done is to run javac on them. The *glucose* binary that is included is precompiled, and will run on linux. I ran it on Ubuntu 8.1, so it should work out-of-the-box with that.