SAT Based Attacks on SipHash

by

Santhosh Kantharaju Siddappa

A Project Report Submitted
in
Partial Fulfillment of the
Requirements for the Degree of
Master of Science
in
Computer Science

Supervised by

Prof. Alan Kaminsky

Department of Computer Science

B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York

January 2014

The project “SAT Based Attacks on SipHash” by Santhosh Kantharaju Siddappa has been examined and approved by the following Examination Committee:

Prof. Alan Kaminsky
Professor
Project Committee Chair

Prof. Stanisław P. Radziszowski
Professor

Prof. Hans-Peter Bischof
Professor

Abstract

SAT Based Attacks on SipHash

Santhosh Kantharaju Siddappa

Supervising Professor: Prof. Alan Kaminsky

SipHash is a proposed pseudorandom function (PRF) that is optimized for small message inputs. It is intended to be used as a message-authentication code (MAC). It uses a 128-bit secret key to compute the tag of a message. This project uses SAT based attacks on the primitive to perform partial key recovery and compares the effectiveness of these attacks against standard brute force approach that involves trying all possible combinations for the key bits.

The primitive is converted into CNF and fed to an off-the-shelf SAT solver. The solver uses clause learning and if satisfiable, returns a set of values for the missing key bits. It also reports the number of conflicts that occurred before a solution was found. This is repeated several times for varying number of missing key bits and different versions of SipHash. It is then compared to the number of attempts to retrieve the missing key bits using brute force and the results are analyzed to check the effectiveness of SAT based attacks.

Contents

Abstract . . . . **iii**

1 Introduction **1**
1.1 Message Authentication Code 1
1.1.1 SipHash 2
1.2 Boolean Satisfiability Problem 4
1.3 SAT solver 4

2 Design **6**
2.1 Building the CNF 6
2.1.1 CNF for AND 7
2.1.2 CNF for OR 7
2.1.3 CNF for XOR 8
2.1.4 CNF for ADD 9
2.1.5 Rotation 10
2.2 Converting SipHash-c,d to CNF 11
2.2.1 Variables 11
2.2.2 Intermediate variables 12
2.2.3 Initialization 12
2.2.4 SipRound 12
2.2.5 Variable parameters 13
2.3 Data Collected 15

3 Analysis **16**
3.1 SipHash-c,d 16
3.1.1 SipHash-1,x 17
3.1.2 SipHash-x,1 19
3.1.3 SipHash-1,0 With More Missing Key Bits 19

4 Related work **21**
4.1 Cryptanalysis of Data Encryption Standard 21
4.1.1 Logical cryptanalysis as a SAT problem 22
4.1.2 SAT attacks for cryptographic key search 22
4.1.3 Algebraic cryptanalysis of DES 23
4.2 SAT solver attacks on CubeHash 23
4.3 SAT attacks on Bivium 25

5 Future Work **27**

6 Conclusions **28**

Bibliography **29**

A Variable Parameters **31**

B Sample output **32**

C CNF **35**

D Data tables of results **36**

List of Figures

1.1 SipHash-2,4 processing a 15-byte message[1] 3
1.2 SipRound[1] 3

2.1 ADD gate 9
2.2 Adding two 64-bit numbers using a full adder 10
2.3 Vector before rotation 11
2.4 Vector after rotation 11

3.1 SipHash-1,0 17
3.2 SipHash-1,1 17
3.3 SipHash-1,2 18
3.4 SipHash-1,3 18
3.5 SipHash-2,1 19
3.6 SipHash-1,0 19

4.1 DES algorithm[8] 22
4.2 Solve time for CubeHash 1/b - 512[2] 24
4.3 Solve time for CubeHash 2/b - 512[2] 25

Chapter 1

Introduction

This project explores the effectiveness of SAT based attacks on the cryptographic primitive SipHash and compares it to the effectiveness of brute force attacks on the same. SipHash employs a hash function that uses a secret key to generate the tag of a given message. This project aims at a partial key recovery and is designed as follows. The SipHash primitive is converted to a Conjunctive Normal Form (CNF) and fed to a SAT solver. The partially known key bits are encoded into the CNF along with the original message block and the expected tag output. If solvable, the solver returns the values for the missing key bits and the number of conflicts it took to find a solution. The number of conflicts is used as a parameter to test the effectiveness of SAT based attacks versus brute force attacks.

The rest of this section gives a brief introduction to SipHash, boolean satisfiability problem and SAT solvers. The next section contains the design of the project and explains how the primitive is converted into a CNF. The results section contains the analysis of the data collected.

1.1 Message Authentication Code

A MAC takes a long message of arbitrary length as an input and produces a shorter fixed length tag of the message as output. MACs are used to verify data integrity or to authenticate packets sent over the Internet. A MAC should be pre-image resistant [6]. A MAC is said to be pre-image resistant if, given only the tag of a message, it is computationally

infeasible to compute the original message.

1.1.1 SipHash

Many Internet based services have a server that processes requests for several clients. Requests sent to such a server can be intercepted, modified and retransmitted to the server. This can be prevented by using MACs. MACs use a secret key to find the tag of the request which is then sent from the client to the server along with the request. The server recomputes the tag of the message and if it matches, processes the request. To process several such requests, the MAC needs to be fast and secure.

SipHash is a pseudorandom function that is used to hash contents of packets sent over the Internet. It achieves this by using a secret key to find the tag of the message sent over the Internet which is then used to authenticate the sender. It can be shown as

$$f\,(k, m) = t,$$

$f \longrightarrow$ MAC algorithm

$k \longrightarrow$ secret key

$m \longrightarrow$ message of arbitrary length

$t \longrightarrow$ tag of the message.

SipHash uses a 128-bit key to hash varying bytes of data. It uses a series of XORs, additions and rotation functions to produce the 64-bit final tag.

A version of SipHash-c-d contains c compression rounds and d finalization rounds. A general flow of SipHash is shown in Figure 1.1.

It uses 4 initialization vectors v_0 through v_3, each 64 bits long. They are created by XOR-ing the lower and higher order 64 bits of the key with pre-chosen constants. Vectors v_0 and v_2 are XOR-ed with the lower 64 bits of the key. Vectors v_1 and v_3 are XOR-ed with the higher 64 bits of the key. It then breaks down the given input into blocks of 8 bytes, padding extra bytes in case the input is not a multiple of 8. The last byte in the last block

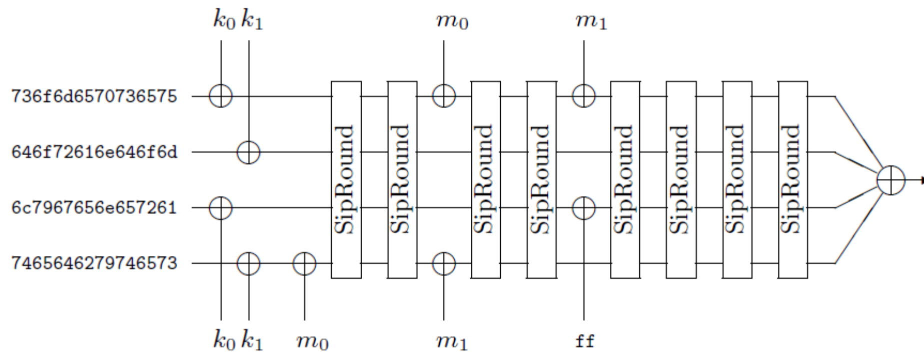

Figure 1.1: SipHash-2,4 processing a 15-byte message[1]

contains the value b mod 256, where b is the length of the message. The blocks are then XOR-ed with the initialization vector v_3 and c SipRounds are performed.

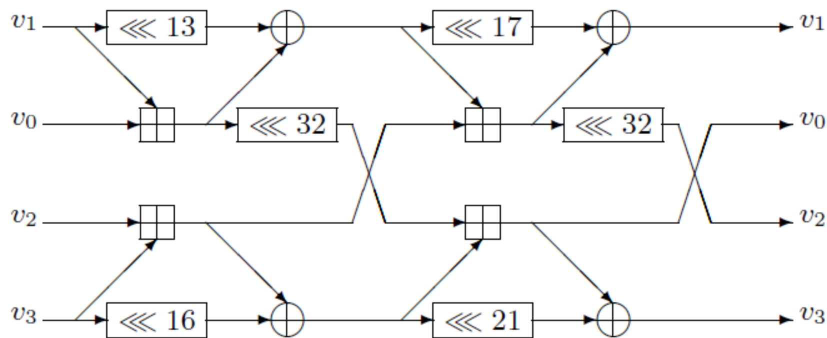

Figure 1.2: SipRound[1]

Each SipRound consists of 4 additions, 4 XOR operations and 6 rotations as shown in Figure 1.2. Vector v_1 is added to v_0 and v_3 to v_2. Vectors v_1 and v_3 are rotated to the left by 13 and 16 bits respectively. Vector v_1 is XOR-ed with v_0 and v_3 with v_2. v_0 is then rotated to the left by 32 bits. v_1 is added to v_2 and v_3 to v_0. v_1 and v_3 are then rotated to the left by 17 and 21 bits. v_1 is XOR-ed with v_2 and v_3 with v_0. Finally v_2 is rotated to the left by 32 bits to complete a single SipRound.

After this, v_0 is XOR-ed with the message block and finalized by XOR-ing v_2 with 0xff followed by d iterations of SipRound. Finally the 4 vectors are XOR-ed together to return the 64-bit tag value.

1.2 Boolean Satisfiability Problem

Given a boolean formula involving several variables, the satisfiability (SAT) problem is to find values for all the variables such that substituting the values will evaluate the formula as true. It is an NP-complete problem [3]. A problem is said to be NP if there is no known polynomial time algorithm to solve it but the solution to the problem can be verified in polynomial time. NP-complete problems are a set of NP problems for which if a polynomial time solution is found for any problem, then that algorithm can be used to solve all the NP-complete problems in polynomial time.

1.3 SAT solver

A SAT solver is a program that takes as input a boolean expression and if it exists, provides a set of values for the literals for which the expression is true. Otherwise the boolean expressions is said to be unsatisfiable. A boolean expression can be expressed in many forms and one such form is the CNF. A CNF is a conjunction of clauses where each clause is a disjunction of literals. The variable or the negation of the variable in a boolean expression is called a literal. An example of a CNF involving the literals A, B and C is (A $\vee\neg$B $\vee$ C) ($\neg$A$\vee\neg$B $\vee$ C). The OR operation is represented as $\vee$ and the NOT operator is represented using $\neg$. The AND operator is not shown but is implicitly present between all clauses. For a CNF to evaluate to be true, each clause must evaluate to true. In this example, we can assign either B as 0 or C as 1 and the expression would evaluate to be true regardless of what value A is.

A conflict-driven clause learning SAT solver uses information from each clause to find

possible values for each of the literals [7]. It uses backtracking and depth first search to find the solution. It reads in the boolean expression and assigns a variable as true or false. It branches on the variable being true or false. Once the variable is assigned the value, it simplifies the entire boolean expression. This phase is called propagation [11] where the program tries to find if assigning the variable affects the value of other variables. If it does, the affected variables are also assigned the thus inferred values and the process continues recursively until no other variables are affected by such assignments. During this phase, it is possible to discover clauses that evaluate to false because all the literals have been assigned as false. This is known as a conflict. The assignment of values to variables that led to the conflict is stored and this clause is known as a learnt clause [11]. The program then backtracks to the point where the assignments were made and reverses the assignment and the program continues to run until a solution is found or the tree is fully traversed.

The SAT solver uses clause learning and assigns values to the variables. It then evaluates the boolean expression using this set of values to see if it is true or false. This is analogous to trying different set of values using a brute force approach. Hence the number of conflicts is used as a parameter for comparison between brute force approach and SAT based approach.

In this project, we will use CryptoMiniSAT version 2.9.5 as the SAT solver. The workings of this is explained in [11].

Chapter 2

Design

To solve for the missing key bits, SipHash needs to be converted to a CNF which is then fed to a SAT solver. This section explains how the CNF is built. SipHash consists of basic AND, OR, XOR and ADD operations. Variables involved with each operation are converted to CNF and appended to the CNF for entire SipHash. The key bits and message blocks are randomly generated for each iteration for different versions of SipHash-c,d. They are then fed to a SipHash implementation to find the tag. The tag and the message block are added to the CNF and random key bits are left out before adding it to the CNF. This is then exported to a CNF file which is then fed to the SAT solver. The output from it is parsed and the number of conflicts is recorded. Each of these steps is explained in detail in the following subsections.

2.1 Building the CNF

The CNF clauses represent the operations performed involving all the variables. The operations are expressed as conjuntion of disjunctions involving the variables. The SAT solver learns from these clauses and finds the correct value. In this project, four basic operations are used namely AND, OR, XOR, ADD and rotate functions. This section will explain how to transform these operations into CNF.

2.1.1 CNF for AND

Consider the operation $A \wedge B = C$, where A and B are variables that have to be ANDed together and the result has to be stored in C.

A	B	C	$A \wedge B = C$
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Table 2.1: Truth table for $A \wedge B = C$

Table 2.1 is used to build the CNF. The first three columns contain all possible combinations for the three variables A, B and C. The final column in the table checks the boolean condition whether $A \wedge B = C$. In the first row, we see that $0 \wedge 0 = 0$. Hence the fourth column is 1. In the second row, we have $0 \wedge 0 = 1$ which is not true. Hence the fourth column is a 0. Similarly, the entire table is built. Since the clauses impose the possible values that the variables can have, the combination of values that evaluated to 0 from the previous table are chosen and they are negated and imposed as clauses in the CNF. The second row from table 2.1 becomes $(A \vee B \vee \neg C)$. Combining all the clauses from the table, the CNF for $A \wedge B = C$ is

$$(A \vee B \vee \neg C)\ (A \vee \neg B \vee \neg C)\ (\neg A \vee B \vee \neg C)\ (\neg A \vee \neg B \vee C)$$

2.1.2 CNF for OR

Table 2.2 represents the truth table for the operation $A \vee B = C$. Just like the AND function, we choose the rows whose last column has a 0 and invert the values.

A	B	C	$A \vee B = C$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Table 2.2: Truth table for $A \vee B = C$

From the truth table, the CNF for $A \vee B = C$ is

$$(\neg A \vee B \vee C)\ (A \vee \neg B \vee C)\ (A \vee B \vee \neg C)\ (\neg A \vee \neg B \vee C)$$

2.1.3 CNF for XOR

Table 2.3 represents the truth table for the operation $A \oplus B = C$. Similar to the last two operations, we select the rows for which the last column is a zero.

A	B	C	$A \oplus B = C$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Table 2.3: Truth table for $A \oplus B = C$

From the table, the CNF for $A \oplus B = C$ is

$$(\neg A \vee B \vee C)\ (A \vee \neg B \vee C)\ (A \vee B \vee \neg C)\ (\neg A \vee \neg B \vee \neg C)$$

2.1.4 CNF for ADD

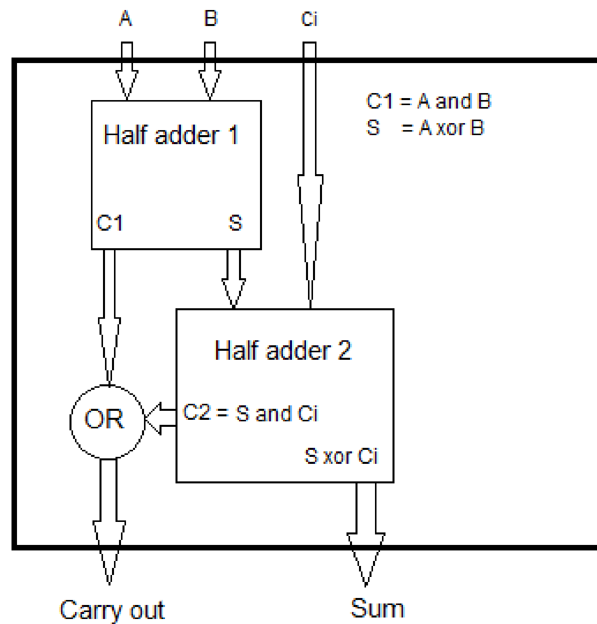

Figure 2.1: ADD gate

A full adder adds two bits along with a carry-in bit and gives as output the sum and the carry-out. This is implemented by using two half adders as shows in Figure 2.1. Half adder 1 takes as input two values A and B. The carry C1 from this adder is given by evaluating (A $\vee$ B) and the sum S is determined by evaluating (A $\oplus$ B). S is then fed as the input to Half adder 2 along with the carry-in Ci. The carry out C2 from Half adder 2 is determined as (S $\vee$ Ci). The final sum of the full adder is evaluated as (S $\oplus$ Ci) and the carry out is given by (C1 $\wedge$ C2).

Two numbers, each several bits long can be added by chaining many such ADD gates. The corresponding bits from each number are fed as input and the final sum from the full adders is stored. The carry-out from each ADD operation is fed as the carry-in for the next

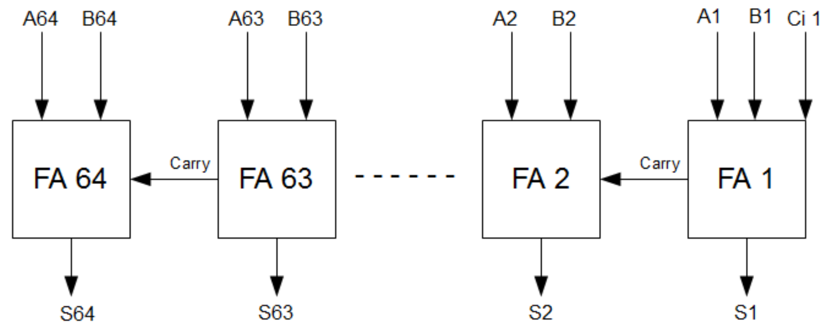

Figure 2.2: Adding two 64-bit numbers using a full adder

ADD operation. Two special cases arise, namely the carry-in for the first ADD gate which is always a zero and the carry-out from the last ADD gate which is ignored.

Based on this, the CNF for the ADD function can be written as a combination of AND, XOR and OR operations between the two input variables and the intermediate sum and carry values. The CNF is

$$(A \oplus B = S) \;\; (A \wedge B = C1) \;\; (Ci \oplus S = Sum) \;\; (S \wedge Ci = C2) \;\; (C1 \vee C2 = Carryout)$$

for the first 63 bits. For the addition of the last bit, we ignore the carry out. Hence the CNF becomes

$$(A \oplus B = S) \;\; (A \wedge B = C1) \;\; (Ci \oplus S = Sum) \;\; (S \wedge Ci = C2)$$

2.1.5 Rotation

Rotate operations are performed on the initialization vectors and the state has to be maintained across compression rounds. For this, the vectors are stored as a sequence of numbers where each number represents the variable name for each bit in the vector.

```
64 63 62 61 60 59 58 57 56 55 54 53 52 51 50 49
48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33
32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17
16 15 14 13 12 11 10 9  8  7  6  5  4  3  2  1
```

Figure 2.3: Vector before rotation

```
50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35
34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19
18 17 16 15 14 13 12 11 10 9  8  7  6  5  4  3
2  1  64 63 62 61 60 59 58 57 56 55 54 53 52 51
```

Figure 2.4: Vector after rotation

Figure 2.3 represents the vector before a rotate operation is performed. The vector is stored starting with the most significant bit(MSB), namely 64 and ends with the least significant bit(LSB), 1. To perform a left rotate by 14 bits, the last 14 bits starting from the MSB are placed at the fourteenth position from the LSB in that order while moving the rest of the vector to the left by 14 bits. The final state of the vector after rotation is shown in Figure 2.4.

2.2 Converting SipHash-c,d to CNF

As described in the previous section, SipHash consists of simple AND, XOR, ADD and rotate functions. The state is stored in the four initialization vectors and keeping track of these will help us find the final tag.

2.2.1 Variables

Variables in the CNF are tracked using numbers. Each of the vectors are assigned initial numbers. When any operation is performed on any of the vectors, the variable numbers are incremented to the next unassigned variable. The CNF clauses linking the new and old variable counts are added onto the final CNF.

There are a few essential values that persist through different versions of SipHash, namely the key, message block, finalization constant, initialization values and the final tag. The variable count starts from 1 and certain numbers are reserved for these. 1 to 128 are reserved for the key bits. 129 to 192 are reserved for the message block. 193 to 256 are reserved for the finalization constant and 257 to 320 for the final tag. The four vectors are assigned values from 321 to 576. Each vector is stored as a list of numbers starting from

321 until the next 63 values, and so on. This makes it easy to handle rotation, especially when there are multiple rounds of compressions and finalization. For each of these values, the least significant bit is assigned the lowest variable count.

2.2.2 Intermediate variables

A counter is used to keep track of the intermediate variable count. Intermediate variables are created on the go and are assigned values beginning at the lowest unused variable count. After reserving key bits for all the known attributes, the first intermediate variable starts at 577. When any operation is performed, the counter is incremented. For an operation involving a vector, the intermediate variables representing the new state of the vector are stored from 577 through 640 and the next intermediate variables are assigned from 641.

2.2.3 Initialization

Initially, vectors v_0 and v_2 are XOR-ed with the lower half of the 128 bit key. Vectors v_1 and v_3 are XOR-ed with the higher 64 bits of the key. Each vector is assigned the next 64 unused variables. The message block is limited to 8 bytes of data. Increasing the size of the message block will not affect the complexity of the solution since all the message bits are known. Hence we resort to the simplest case of having only 8 bytes of data.

2.2.4 SipRound

Each SipRound contains a few XOR, ADD and rotations. The XOR might be between two vectors or a vector and a constant. The variable count of the vector in which the final value is to be stored is incremented. The XOR of two vectors is handled similarly. Rotation of a vector is handled by simply rearranging the range values. For a left rotation by k bits, the last k bits are rotated and placed at the start of the vector.

Adding two vectors is a bit more tedious as there are a lot more intermediate variables. As mentioned in the previous section, a full adder is implemented as a combination of two

half adders. Variable space is reserved for the sum and carry of each half adder. The first carry-in for the second half adder is asserted as 0 and is added as a clause in the CNF.

2.2.5 Variable parameters

Given a version of SipHash-c,d, the basic CNF generated will remain the same. The basic CNF comprises the initialization, c compressions rounds and d finalization rounds and calculation of the final tag. This is generated only once for a given version and all the other varying parameters are asserted in the simulation program. The key bits are generated randomly for each simulation run. The number of key bits are varied in order to check its effect on the total number of conflicts. The number of missing key bits vary from 1 to 28 for lower versions and increasing this further increases the computing time by a substantial amount. Higher versions of SipHash have fewer missing key bits, as less as 14. Each time the position of the missing key bits is also randomly generated. The size of the message block is fixed as 8 bytes. For each simulation round, the message blocks are also randomly generated.

The random parameters are generated using a random number generator. Since this project is implemented in Java, the Math.random() function is used to generate the random numbers. For generating the 128-bit key, the random number generator is invoked to get a random number between 0 and 128. This in converted into a byte which is then appended to a 16 byte array to create the key. Similarly, the 8 byte data is also generated. The key and the data arrays are fed to an implementation of SipHash to get the tag which is pushed onto the CNF. This is done by selecting the value of each bit and checking if it is a 1 or a 0. If the bit is 1, the positive value of the tag variable is pushed onto the CNF as a single clause else the negative value is pushed. Similarly the data bits are also pushed onto the CNF as single clauses. While pushing the key onto the CNF, random key bits have to remain unknown. This is done by creating a list of numbers from 1 to 128. The random number generator is used to generate a number between 1 and 128. The value of the corresponding

bit is retrieved using bit manipulation and similar to pushing the tag, a corresponding single clause is pushed onto the CNF. For n missing key bits, the process is repeated 128-n times and the key bits are pushed onto the CNF.

The clauses are exported into an external CNF file in the DIMACS format. The format contains all the clauses in single or several lines. Each clause is separated by a 0. True values for variables are represented as the positive integer value and negations are represented as the negative value of that variable number. A sample line in the CNF file looks like -137 20 153 0. This represents a disjunction clause containing the negation of 137 and variables 20 and 153. A DIMACS file looks as follows

```
-321 -1 -577 0        321 -1 577 0
321 1 -577 0          -321 1 577 0
```

Given a version of SipHash-c,d, once all parameters are randomly generated, the key bits and the message block are fed into an implementation of SipHash and the final tag is retrieved. The message block and the initialization values are asserted onto the CNF along with the final tag. Random key bits are asserted onto the CNF too. This iteraton is run 100 times for each version of SipHash. The simulation is further repeated by varying the number of compression and finalization rounds. The compressions rounds are varied between 1 and 2. The finalization rounds vary from 0 to 4.

Sample input and output are shown in Appendix A and B. Appendix A contains the input message, the key and the tag. The 16 missing key bit positions are also shown. Appendix B shows the output of CryptoMiniSAT. The program outputs the values of all unknown variables including the intermediate variables.

The following pseudo code summarizes the flow of the program.

$for\ c \longrightarrow 1\ to\ 2$

$for\ d \longrightarrow 1\ to\ 4$

$List < Clauses > clauses = GenerateCNF(c, d);$

$for\ k \longrightarrow 1\ to\ 28$

$avgConflicts = 0;$

$for\ i \longrightarrow 1\ to\ 100$

$List < Clauses > temp = new\ List < Clauses > (clauses);$

$message = new\ Message();$

$key = new\ Key();$

$tag = SipHash(message, key);$

$AppendValue(temp, message);$

$AppendValue(temp, tag);$

$AppendRandomKey(temp, key, 128 - k);$

$avgConflicts += SATSolver(temp);$

$plotConflicts(c, d, k, avgConflicts/100);$

2.3 Data Collected

Once the SAT solver completes execution, it writes the result to the output stream. This data is then parsed to find the total number of conflicts. As explained previously, a conflict arises when a chosen set of values for the key bits does not satisfy a clause. This is measured and recorded, and the average number of conflicts for a given number of missing key bits for a given version of SipHash-c,d is recorded over 100 iterations.

Chapter 3

Analysis

The average number of conflicts reported for varying number of missing key bits is recorded. Higher versions of SipHash-c-d are tested for fewer missing key bits as the computation time was very long. The number of conflicts reported is compared with the estimated average number of brute force trials needed to find the key bits. For n missing key bits, a brute force approach would need an average of 2^{n-1} trials to find the key bits. The following graphs display the results for varying versions of SipHash. The x-axis represents the number of missing key bits and the y-axis represents the logarithm to the base 2 of the number of conflicts. The values are an average number of conflicts from running the iteration 100 times.

The program was executed on an Intel Xeon X5560 @ 2.80GHz running Ubuntu 12.04.3 LTS and it approximately took about 4 days to generate all the results. In most cases, increasing the number of bits beyond 16 greatly increased the computational time. Hence, the results are analyzed for up to 16 unknown key bits only.

3.1 SipHash-c,d

In the Figures 3.1 and 3.2, the points along the red line indicate the logarithm to the base 2 of the number of attempts to retrieve a key using brute force approach. The dots represent the logarithm to the base 2 of the average number of conflicts as reported by the SAT solver.

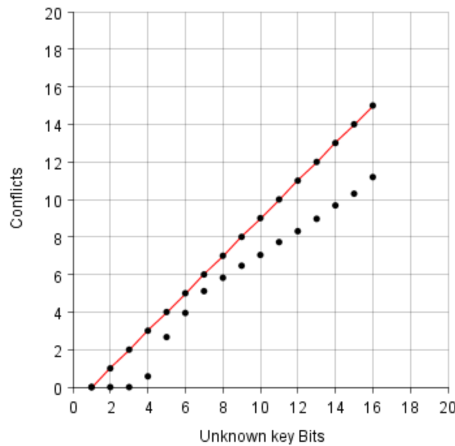

Figure 3.1: SipHash-1,0

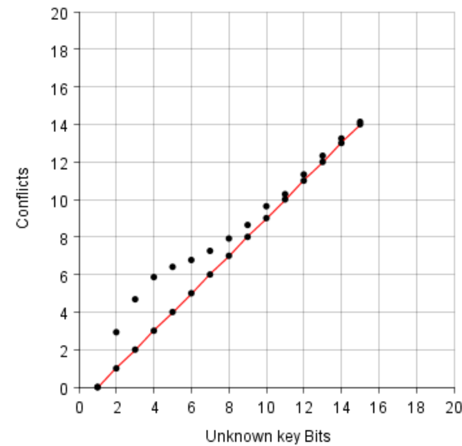

Figure 3.2: SipHash-1,1

For a single round of compression, the SAT solver is able to retrieve the missing key bits more efficiently as compared to a brute force approach. But as the number of SipRounds increases, the solver takes more conflicts before finding the right key bits. In Figure 3.2 we notice that the SAT solver reports more conflicts for fewer missing key bits when compared to the brute force approach. This behavior is not seen in Figure 3.1. As the number of key bits is small, brute force finds the right value more efficiently. The SAT solver tries different values based on clause learning. It then evaluates the CNF with the given set of values. If the set of values fails to satisfy the formula, it will report a conflict. Based on the different operations performed, there will be many intermediate variables created. The SAT solver tries to assign values to these variables. This in turn will create many more conflicts which is explained by the graph. The number of conflicts by brute force approach is smaller than that by the SAT solver. But as the number of unknown key bits increases, the number of unknowns increases the complexity for a brute force approach. Hence the performance of the SAT solver becomes more comparable with the brute force approach.

3.1.1 SipHash-1,x

As the number of SipRounds increases, more intermediate variables are created and it reaches a point where brute force approach is more efficient. In Figures 3.3 and 3.4, we

see that the number of conflicts reported by the SAT solver is more than the brute force approach. As the complexity increases, it approaches the number of conflicts reported by a brute force approach.

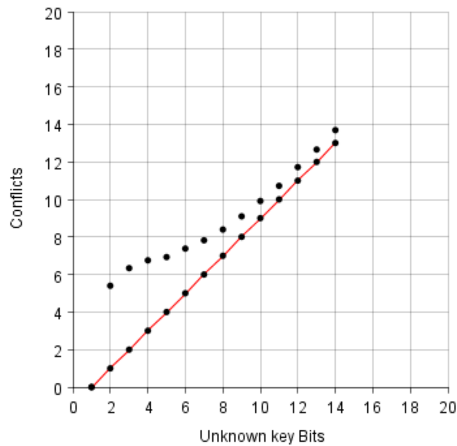

Figure 3.3: SipHash-1,2

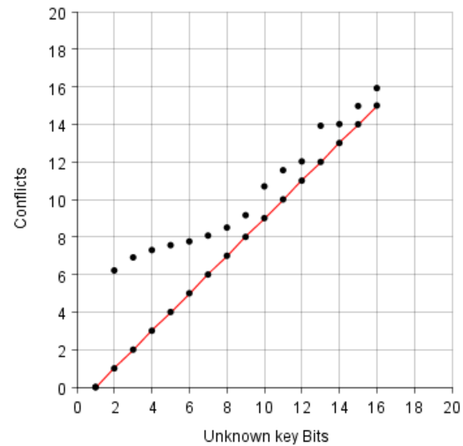

Figure 3.4: SipHash-1,3

Increasing the finalization rounds yields similar results. The number of conflicts reported is greater than the number of attempts using a brute force approach. However, as the number of SipRounds increases, the curve moves further away and takes longer to normalize and converge to the conflicts using the brute force approach.

3.1.2 SipHash-x,1

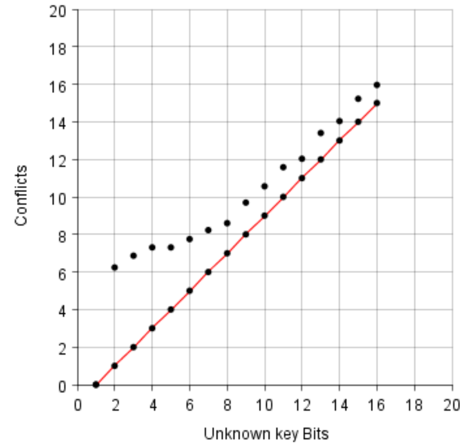

Figure 3.5: SipHash-2,1

From Figures 3.1 and 3.5 we can see that increasing the number of compression rounds has a similar effect as increasing finalization rounds. This is expected as they both perform SipRounds and hence create almost the same number of intermediate variables.

3.1.3 SipHash-1,0 With More Missing Key Bits

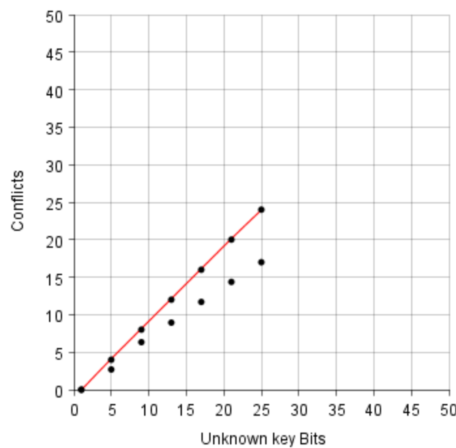

Figure 3.6: SipHash-1,0

Missing key bits	Brute force	SAT solver
1	0	0.00
5	4	2.70
9	8	6.33
13	12	8.94
17	16	11.70
21	20	14.37
25	24	17.00

Table 3.1: Conflicts for SipHash-1,0

From the results, the SAT solver performed better than brute force for only SipHash-1,0. Increasing SipRounds caused the performance to decline further than brute force approach. Further investigating this version for more number of missing key bits gives us the graph shown in Figure 3.6. The SAT solver tends to get more efficient as the number of missing key bits increases. Increasing the number of missing key bits drastically increased the computation time and hence the simulation was stopped at 25 missing key bits. But based on the curvature of the graph, we can predict that the SAT solver will perform better for larger missing key bits.

Data tables of all results are listed in Appendix C.

Chapter 4

Related work

Based on the results, we can see that the SAT solver works efficiently on lower versions of SipHash as compared to brute force attacks. But as the complexity increases, it becomes inefficient to use SAT solvers. The performance is not sufficient enough to break the recommended version of SipHash-4,8. Similar conclusions are reached in other projects where a SAT solver is used to attack a cryptographic primitive [2, 10]. SAT based attacks were deployed for CubeHash to find collisions and it works efficiently for lower versions of CubeHash [2]. But as the complexity was increased, the SAT solver returned unsatisfiable.

4.1 Cryptanalysis of Data Encryption Standard

Data Encryption Standard (DES) is a block cipher that encodes a 64-bit blocks of plaintext into 64-bit blocks of ciphertext using a 56-bit secret key [8]. Figure 4.1 shows the basic workings of DES. The algorithm takes the plaintext as input and performs some initial preprocessing. It then splits the plaintext into two halves. It combines one half of the plaintext with the key using a special function and then XORs this with the other half of the key. It then swaps the two resulting halves. These operations comprise a basic round. DES has 16 such rounds.

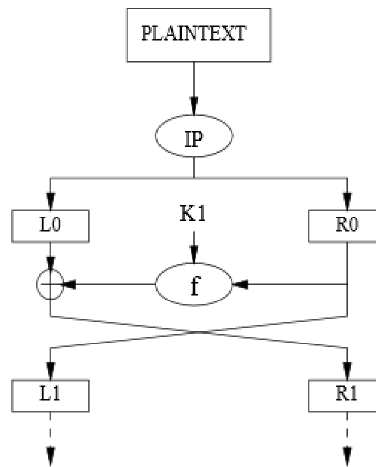

Figure 4.1: DES algorithm[8]

4.1.1 Logical cryptanalysis as a SAT problem

The authors of [9] present the idea of transforming the cryptanalysis as a SAT problem. The authors use the DES algorithm to perform the attack. The attack is designed assuming that the plaintext and the ciphertext is known. The authors propose to track the state after each operation and represent it using propositional logic which is ultimately converted into a boolean formula. The details are explained in [9].

Given the ciphertext and plaintext pairs, the authors were able to break DES with three rounds. However, they were unable to apply the attack to a full strength DES with sixteen rounds. The authors conclude that viewing the cryptographic primitive as a boolean formula and trying to solve it is a feasible approach.

4.1.2 SAT attacks for cryptographic key search

The authors in [8] build upon the proposed idea in [9] and applies SAT based attacks to perform key search on DES. They compare the effectiveness of two solver algorithms, namely Walk-SAT and Rel-SAT.

Walk-SAT algorithm uses a local minimum algorithm to narrow the search space in order to find the key bits. It randomly assigns value to a variable. It then continues to randomly assign values to other variables and finds the number of satisfied clauses that result from the assignments. When the number of satisfied clauses reaches a local minimum, it restarts with another random assignment. Walk-SAT was able to find the key bits for DES with two rounds.

Rel-SAT algorithm works similar to a clause learning algorithm as described in section 1.3. Rel-SAT performed better compared to Walk-SAT and was able to break the DES with three rounds. The results from [8] emphasize the significance of clause learning to the success of SAT solvers.

4.1.3 Algebraic cryptanalysis of DES

The authors of [4] propose a new attack that combines SAT solver attacks with algebraic attacks. The DES algorithm is represented as a series of multivariate equations and the authors propose that solving these set of equations will break the DES algorithm. The authors propose converting the equations to a CNF and then feeding the CNF to a SAT solver. The attack was able to break a DES with six rounds with only one known plaintext. The authors conclude that combining SAT attacks with algebraic attacks will allow us to attack a wider array of algorithms that show weakness to either types of attacks.

4.2 SAT solver attacks on CubeHash

CubeHash is a hashing algorithm that was listed as a Secure Hash Algorithm (SHA-3) candidate [2]. A variant of CubeHash can be represented as CubeHash r/b - h where r represents the number of rounds per message block, b represents the size of the message block in bytes and h represents the size of the hash output. The hash algorithm feeds a b block size message and performs r rounds for each message block and outputs a hash of size h.

More details can be found in [2].

SAT solver attacks were employed to find collisions in CubeHash. The attack must find two messages m_1 and m_2 such that $m_1 \neq m_2$ and $h(m_1) = h(m_2)$, where $h(m)$ represents the hash of the message m. The findings are as follows.

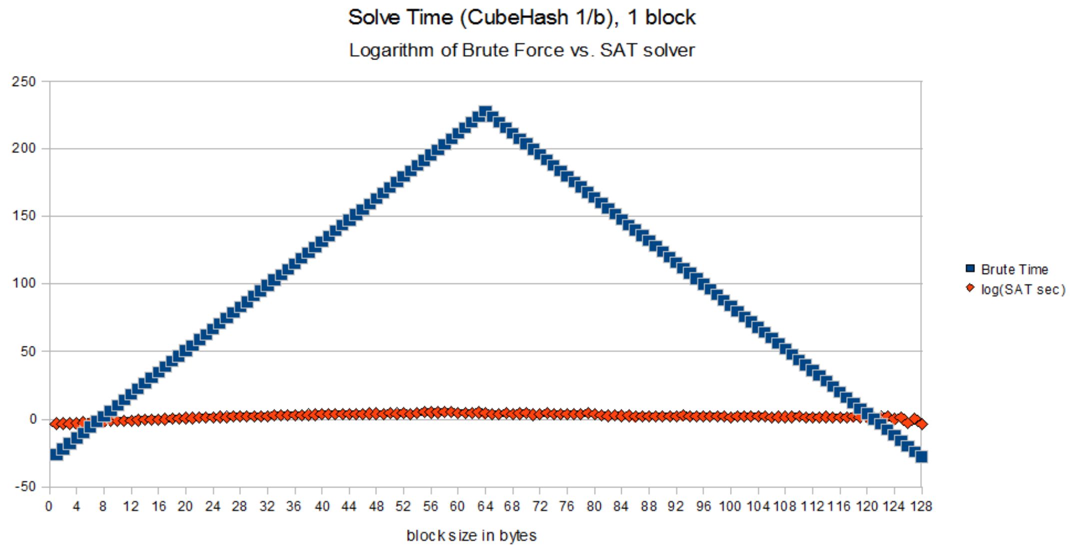

Figure 4.2: Solve time for CubeHash 1/b - 512[2]

The design of the attack implied that it was hardest to find a collision when b was 64. Figure 4.1 shows a plot of the logarithm of the time taken using brute force attack versus the logarithm of the time taken by the SAT solver. From the figure we can see that for one round per message block, the brute force approach took the longest when b = 64. The SAT solver however, consistently was way more efficient irrespective of the message block size.

Figure 4.2 shows the results when the number of rounds per message block is increased to 2. The SAT solver is only able to solve instances when b $\leq$ 28 and b $\geq$ 89. For all other instances, the SAT solver took too long to find a solution and the solver timed out.

Based on the findings from [2], we can conclude that SAT based attacks are only successful

on reduced versions of CubeHash and further improvements to SAT solver algorithms will have to be made to attack a full strength CubeHash 16/32.

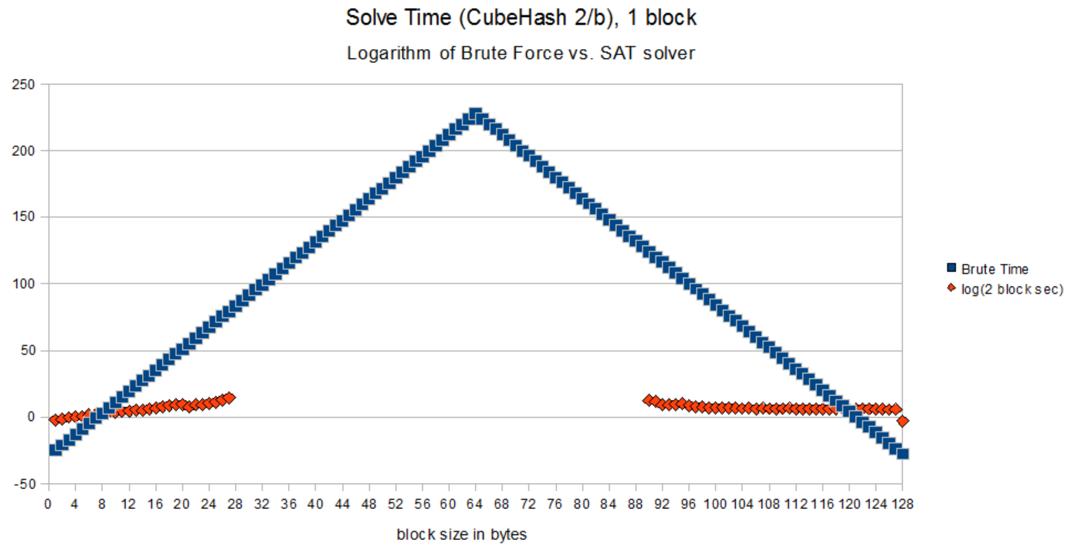

Figure 4.3: Solve time for CubeHash 2/b - 512[2]

4.3 SAT attacks on Bivium

Bivium is a stream cipher where two parties communicate by sharing a secret key [5]. The internal state of the cipher is stored in an initialization vector (IV) which is 177 bits long and the secret key which is 80 bits long. It is stored as two registers each 93 and 84 bits long, respectively. In [5], SAT solver attacks are employed to attack the primitive and its performance is compared with other attacks such as Binary Decision Diagrams and Gröbner bases. The authors design various attacks and select the fastest attack on the primitive. They then apply various methods to perform the attack and compare the performance of these methods using the solve time as the comparison parameter. The authors compare different strategies for the attacks and deduce that finding the last 48 bits of the second register leads to the fastest solving time to crack the cipher [5]. This is termed as ”Ending2” strategy.

The authors conduct various tests using BDD, SAT solver and Gröbner bases. They conclude by saying that SAT attacks are by far the best amongst these attacks on Bivium [5].

Chapter 5

Future Work

SAT based attacks offer a different approach to break cryptographic primitives. However, the SAT solver was not used to the full extent in this project. Further work can be conducted to better understand the SAT solver and its applications to cryptanalysis. Following is a list of a few new approaches that can be tried with SAT solvers and SipHash.

- The project explored up to 25 missing key bits for a partial key recovery. The number of missing key bits can be further increased to get a deeper understanding of the performance comparison between SAT based attacks and brute force approach to partial key recovery.

- The project explored the SAT solver performance on SipHash up to two compression rounds and three finalization rounds. The number of compression and finalization rounds can be increased further to examine more results.

- The project used CryptoMiniSAT as the SAT solver. Several other SAT solvers can be employed and their relative performance can be analyzed.

- The project can be extended to incorporate the application and analysis of parallel SAT solvers to partial key recovery.

- The project assumed that certain key bits were known in advance and aimed at a partial key recovery. SAT based attacks can be combined with other cryptanalysis techniques such as linear or differential cryptanalysis and other attacks can be forged.

Chapter 6

Conclusions

This project aimed at a partial key recovery in a newly proposed MAC, SipHash using SAT based attacks. The given cryptographic primitive was converted into a CNF and fed into an off the shelf SAT solver, namely CryptoMiniSAT which then evaluated whether the given boolean expression was satisfiable or not. If it was satisfiable, it returned the value for the missing key bits. SipHash uses a 128-bit private key in varying versions of SipHash-c,d. The simulation was conducted for different versions of SipHash and for varying number of missing key bits. It was then compared with a brute force approach to break the primitive and we found that for lower versions of SipHash and very few missing key bits, the SAT based approach outperformed brute force approach. But as the complexity of SipHash increased, brute force attacks were more efficient as the conversion to CNF was creating too many intermediate variables. In conclusion, SAT based attacks perform relatively well compared to brute force approach for lower versions of SipHash but more research is needed to improve the performance to make it viable to crack more complex versions.

Bibliography

[1] Jean-Philippe Aumasson and Daniel J. Bernstein. SipHash: A fast short-input PRF. In Steven D. Galbraith and Mridul Nandi, editors, *INDOCRYPT*, volume 7668 of *Lecture Notes in Computer Science*, pages 489–508. Springer, 2012.

[2] Benjamin Bloom. SAT solver attacks on CubeHash @ONLINE, April 2010.

[3] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[4] Nicolas T. Courtois and Gregory V. Bard. Algebraic cryptanalysis of the Data Encryption Standard. Cryptology ePrint Archive, Report 2006/402, 2006. `http://eprint.iacr.org/`.

[5] Tobias Eibach, Enrico Pilz, and Gunnar Völkel. Attacking Bivium using SAT solvers. In *Proceedings of the 11th international conference on Theory and applications of satisfiability testing*, SAT'08, pages 63–76, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] Dejan Jovanović and Predrag Janičić. Logical analysis of hash functions. In *Proceedings of the 5th international conference on Frontiers of Combining Systems*, FroCoS'05, pages 200–215, Berlin, Heidelberg, 2005. Springer-Verlag.

[7] Tero Laitinen, Tommi Junttila, and Ilkka Niemelä. Conflict-driven xor-clause learning. In *Proceedings of the 15th international conference on Theory and Applications of Satisfiability Testing*, SAT'12, pages 383–396, Berlin, Heidelberg, 2012. Springer-Verlag.

[8] Fabio Massacci. Using Walk-SAT and Rel-Sat for cryptographic key search. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, IJCAI '99, pages 290–295, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[9] Fabio Massacci and Laura Marraro. Logical cryptanalysis as a SAT problem: the encoding of the Data Encryption Standard. *In Journal of Automated Reasoning*, 24:165–203, 1999.

[10] Ilya Mironov and Lintao Zhang. Applications of SAT solvers to cryptanalysis of hash functions. In *Proceedings of the 9th international conference on Theory and Applications of Satisfiability Testing*, SAT'06, pages 102–115, Berlin, Heidelberg, 2006. Springer-Verlag.

[11] Mate Soos. Enhanced Gaussian elimination in DPLL-based SAT solvers. In *Pragmatics of SAT*, pages 1–1, Edinburgh, Scotland, GB, July 2010.

Appendix A

Variable Parameters

Key	0x46 0x31 0x30 0x05 0x2c 0x03 0x37 0x5b 0x2e 0x47 0x79 0x07 0x5d 0x75 0x12 0x58
Message	0x0d 0x51 0x38 0x0c 0x59 0x0a 0x22 0x28
Missing positions	1 5 7 10 13 35 43 44 66 70 71 83 101 115 125 126
Tag	0xab939c262b0322ba

Appendix B

Sample output

****
SipHash-1,0
****

****
Unknown key bits : 16
****

****
Trial 1
****

c Outputting solution to console
c This is CryptoMiniSat 2.9.5
c compiled with gcc version 4.6.3
c WARNING: for repeatability, setting FPU to use double precision
c Reading file 'CNF'
c – clauses added: 0 learnts, 15960 normals, 0 xors
c – vars added 4416
c Parsing time: 0.04 s
c N st 0 0 4416 15264 0 32 0 45856 0 no data no data –
c Flit: 5 Blit: 566 bXBeca: 0 bXProp: 935 Bins: 0 BRemL: 0 BRemN: 0 P: 0.2M T: 0.09
c Cleaning up after failed var search: 0.01 s
c vivif2 – cl tried 7216 cl rem 56 cl shrink 18 lits rem 44 time: 0.01 Vivification of clauses
c vivif2 – cl tried 0 cl rem 0 cl shrink 0 lits rem 0 time: 0.00 Vivification of clauses
c asymm cl-useful: 274/7172/7172 lits-rem:822 time: 0.07 Asymmetric branching
c bin-w-bin subsume rem 0 bins time: 0.00 s Binary clause self-subsume
c Removed useless bin: 0 fixed: 0 props: 0.02M time: 0.00 s
c lits-rem: 1385 cl-subs: 2156 v-elim: 523 v-fix: 0 time: 0.08 s
c Finding binary XORs T: 0.00 s found: 0 Binary XOR clauses
c Finding non-binary XORs: 0.02 s (found: 962, avg size: 3.0) Non-binary XOR clauses
c x-sub: 0 x-cut: 0 vfix: 0 v-elim: 124 locsubst: 40 time: 0.00
c calculated reachability. Time: 0.00

```
c Calc default polars - time: 0.00 s pos: 33 undec: 3338 neg: 1045
c ====================================================================================================
c types(t): F = full restart, N = normal restart
c types(t): S = simplification begin/end, E = solution found
c restart types(rt): st = static, dy = dynamic
c t rt Rest Confl Vars NormCls XorCls BinCls Learnts ClLits LtLits LGlueHist SGlueHist
c B st 0 0 1077 1532 834 101 0 7432 0 no data no data –
c Decided on static restart strategy
c E st 8 4098 0 1532 834 139 3865 7432 56787 no data no data –
c Verified 2366 clauses.
c Solution needs extension. Extending.
c Verified 2366 clauses.
c num threads : 1
c restarts : 8
c dynamic restarts : 0
c static restarts : 8
c full restarts : 0
c total simplify time : 0.00
c learnts DL2 : 0
c learnts size 2 : 3540                                  Binary learnt clauses
c learnts size 1 : 1752 (39.67 % of vars)                Single learnt clauses
c filedLit time : 0.10 (5.45 % time)                     Time exploring failed literals
c v-elim SatELite : 523 (11.84 % vars)                   Variables eliminated using SatELite
c SatELite time : 0.07 (4.09 % time)
c v-elim xor : 124 (2.81 % vars)                         Variables eliminated using XOR clauses
c xor elim time : 0.00 (0.23 % time)
c num binary xor trees : 362
c binxor trees' crown : 952 (2.63 leafs/tree)
c bin xor find time : 0.00
c OTF clause improved : 200 (0.05 clauses/conflict)
c OTF impr. size diff : 208 (1.04 lits/clause)
c OTF cl watch-shrink : 617 (0.15 clauses/conflict)
c OTF cl watch-sh-lit : 694 (1.12 lits/clause)
c tried to recurMin cls : 1597 (38.97 % of conflicts)
c updated cache : 0 (0.00 lits/tried recurMin)
c clauses over max glue : 0 (0.00 % of all clauses)
c conflicts : 4098 (2328.26 / sec)                       Total conflicts
c decisions : 5153 (0.12 % random)
c bogo-props : 25038696 (14225642.72 / sec)
c conflict literals : 58309 (47.23 % deleted)
c Memory used : 15.71 MB
```

c CPU time : 1.76 s
s SATISFIABLE
v -1 2 3 -4 -5 -6 7 -8 9 -10 -11 -12 13 14 -15 -16 -17 -18 -19 -20 21 22 -23 -24 25 -26 27 -28 -29 -30 -31 -32 -33 -34 35 36 -37 38 -39 -40 41 42 -43 -44 -45 -46 -47 -48 49 50 51 -52 53 54 -55 -56 57 58 -59 60 61 -62 63 -64 -65 66 67 68 -69 70 -71 -72 73 74 75 -76 -77 -78 79 -80 81 -82 -83 84 85 86 87 -88 89 90 91 -92 -93 -94 -95 -96 97 -98 99 100 101 -102 103 -104 105 -106 107 -108 109 110 111 -112 -113 114 -115 -116 117 -118 -119 -120 -121 -122 -123 124 125 -126 127 -128 129 -130 131 132 -133 -134 -135 -136 137 -138 -139 -140 141 -142 143 -144 -145 -146 -147 148 149 150 -151 -152 -153 -154 155 156 -157 -158 -159 -160 161 -162 -163 164 165 -166 167 -168 -169 170 -171 172 -173 -174 -175 -176 -177 178 -179 -180 -181 182 -183 -184 -185 -186 -187 188 -189 190 -191 -192 193 194 195 196 197 198 199 200 -201 -202 -203 -204 -205 -206 -207 -208 -209 -210 -211 -212 -213 -214 -215 -216 -217 -218 -219 -220 -221 -222 -223 -224 -225 -226 -227 -228 -229 -230 -231 -232 -233 -234 -235 -236 -237 -238 -239 -240 -241 -242 -243 -244 -245 -246 -247 -248 -249 -250 -251 -252 -253 -254 -255 -256 -257 258 -259 260 261 262 -263 264 -265 266 -267 -268 -269 270 -271 -272 273 274 -275 -276 -277 -278 -279 -280 281 282 -283 284 -285 286 -287 -288 -289 290 291 -292 -293 294 -295 -296 -297 -298 299 300 301 -302 -303 304 305 306 -307 -308

. . . . 4007 variables omitted . . . .

-4315 -4316 4317 -4318 4319 -4320 -4321 -4322 4323 4324 4325 4326 4327 -4328 -4329 -4330 -4331 4332 -4333 4334 4335 4336 -4337 4338 4339 -4340 -4341 4342 -4343 -4344 4345 -4346 4347 -4348 -4349 -4350 -4351 4352 -4353 4354 4355 4356 -4357 -4358 4359 4360 4361 4362 4363 -4364 -4365 4366 -4367 4368 4369 -4370 4371 -4372 4373 4374 4375 4376 -4377 -4378 -4379 -4380 -4381 -4382 -4383 -4384 -4385 -4386 4387 -4388 4389 4390 -4391 4392 -4393 -4394 -4395 -4396 -4397 -4398 4399 4400 -4401 -4402 -4403 -4404 4405 4406 -4407 4408 -4409 -4410 -4411 4412 4413 4414 4415 -4416 0

Appendix C

CNF

Following is a part of the CNF file presented in five columns where each line of each column represents a clause.

-321 -1 -577 0	321 1 -577 0	321 -1 577 0	-321 1 577 0	-322 -2 -578 0
322 2 -578 0	322 -2 578 0	-322 2 578 0	-323 -3 -579 0	323 3 -579 0
323 -3 579 0	-323 3 579 0	-324 -4 -580 0	324 4 -580 0	324 -4 580 0
-324 4 580 0	-325 -5 -581 0	325 5 -581 0	325 -5 581 0	-325 5 581 0
-326 -6 -582 0	326 6 -582 0	326 -6 582 0	-326 6 582 0	-327 -7 -583 0
327 7 -583 0	327 -7 583 0	-327 7 583 0	-328 -8 -584 0	328 8 -584 0
328 -8 584 0	-328 8 584 0	-329 -9 -585 0	329 9 -585 0	329 -9 585 0
-329 9 585 0	-330 -10 -586 0	330 10 -586 0	330 -10 586 0	-330 10 586 0
-331 -11 -587 0	331 11 -587 0	331 -11 587 0	-331 11 587 0	-332 -12 -588 0
332 12 -588 0	332 -12 588 0	-332 12 588 0	-333 -13 -589 0	333 13 -589 0
—	—	—	—	—
	46516	lines	omitted	
—	—	—	—	—
-128 0	-71 0	3 0	-64 0	-86 0
10 0	52 0	-122 0	22 0	117 0
84 0	57 0	-9 0	-114 0	-113 0
103 0	-127 0	107 0	46 0	-91 0
-79 0	-15 0	-77 0	-116 0	-42 0
-23 0	-115 0	-99 0	-50 0	-17 0
-49 0	-101 0	109 0	76 0	69 0
89 0	-66 0	11 0	-8 0	-33 0
59 0	125 0	-111 0	-29 0	102 0
-72 0	41 0	-112 0	-93 0	92 0
63 0	37 0	53 0	-62 0	-48 0
-32 0	-1 0	-104 0	-25 0	108 0

Appendix D

Data tables of results

The column labeled brute force represents the average number of attempts required by brute force to find a solution for the given number of missing key bits. The column labeled SAT solver represents the average number of conflicts reported by the SAT solver to find the missing key bits.

Missing key bits	brute force	SAT solver
1	0	0.0
2	1	2.93
3	2	4.73
4	3	5.90
5	4	6.32
6	5	6.77
7	6	7.24
8	7	7.85
9	8	8.71
10	9	9.44
11	10	10.39
12	11	11.36
13	12	12.28
14	13	13.27
15	14	14.14
16	15	15.37

Table D.1: SipHash-1,1

Missing key bits	brute force	SAT solver
1	0	0.0
2	1	5.43
3	2	6.13
4	3	6.71
5	4	7.23
6	5	7.33
7	6	7.87
8	7	8.24
9	8	9.01
10	9	9.79
11	10	10.78
12	11	11.75
13	12	12.61
14	13	13.77
15	14	14.72
16	15	15.80

Table D.2: SipHash-1,2

Missing key bits	brute force	SAT solver
1	0	0.0
2	1	6.21
3	2	6.90
4	3	7.29
5	4	7.56
6	5	7.75
7	6	8.07
8	7	8.49
9	8	9.15
10	9	10.69
11	10	11.55
12	11	12.01
13	12	13.92
14	13	14.01
15	14	14.97
16	15	15.92

Table D.3: SipHash-1,3

Missing key bits	brute force	SAT solver
1	0	0.0
2	1	6.24
3	2	6.86
4	3	7.30
5	4	7.30
6	5	7.74
7	6	8.22
8	7	8.59
9	8	9.69
10	9	10.56
11	10	11.58
12	11	12.03
13	12	13.40
14	13	14.03
15	14	15.22
16	15	15.96

Table D.4: SipHash-2,1