M.S. Project Proposal

SAT Based Attacks on SipHash

Santhosh Kantharaju Siddappa
Department of Computer Science
Rochester Institute of Technology

Chair

Prof. Alan Kaminsky

Reader

Prof. Stanislaw P. Radziszowski

Observer

Prof. Hans-Peter Bischof

24 April 2013
1. Introduction

A hash function takes a long message of arbitrary length as an input and produces a shorter fixed length hash of the message as output. Cryptographic hash functions are used to verify data integrity or to authenticate packets sent over the Internet using digital signatures. A hash function should be preimage resistant and collision resistant [1]. A hash function is said to be preimage resistant if given only the hash of a message, it is computationally infeasible to compute the original message. A hash function is said to be collision resistant if it is computationally infeasible to find two distinct messages that produce the same hash. Attackers often use collision attacks to break applications that use hash functions [1].

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 but when the boolean formula is expressed in a standard form such as the Conjunctive Normal Form (CNF), it is possible to find a solution if it exists. A CNF is a conjunction of many clauses where each clause is a disjunction of many literals. An example of a CNF involving the literals A, B and C is \((A \lor \neg B \lor C) \land (\neg A \lor \neg B \lor C)\). A SAT solver takes as input a CNF and if it exists, provides a set of values for the literals for which the CNF will evaluate to be true. For a CNF to evaluate to be true, each clause must evaluate to true. A conflict-driven clause learning SAT solver uses information from each clause to find possible values for each of the literals [2]. If a set of values do not satisfy a clause, a conflict arises and the set of values is discarded. Eventually the SAT solver generates the set of values that satisfy the CNF.

SipHash is a pseudorandom function that is used to hash contents of packets sent over the Internet. It is primarily used to authenticate network traffic and prevent denial of service attacks. It achieves this by using a private key to hash data packets which are then transmitted over the Internet. Packets are authenticated at the destination by recomputing the hash of the packets and comparing it with the attached hash value. Assuming that we know the input to the hash function and the final hash value, this project aims to compute the key used to create the hash value. SAT based attacks are going to be employed to deduce the key value used for reduced versions of SipHash. Its performance will be measured by comparing the time taken to deduce the key using SAT based attacks against the time taken using brute force attacks. This project also intends to find the point where brute force attacks are more effective than SAT based attacks.

2. Overview

SipHash uses a 128-bit key to hash varying bytes of data producing a 8 byte hash that is appended to the data and sent over the Internet. It uses a series of XORs, ANDs and rotation functions only which makes it a really fast algorithm for short inputs and it has a very small overhead with only 8 bytes of data making it ideal for short input messages.

Figure 1: Flow of SipHash [3]
A version of SipHash-c-d contains c number of compression rounds and d number of finalization rounds. A general flow of SipHash is shown in Figure 1. It uses 4 initialization vectors v₀ through v₃, 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 v₀ and v₂ are XOR-ed with the lower 64 bits of the key. Vectors v₁ and v₃ 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. It makes it simpler to pad the last few bytes rather than having to deal with lesser blocks as the overhead of the padded bytes is negligible. The blocks are then XOR-ed with the initialization vector v₃ and c number of SipRounds are performed. Each SipRound consists of 4 AND, XOR operation and 6 rotations as shown in Figure 2. Vector v₀ is AND-ed with v₁ and v₂ with v₃. Vectors v₁ and v₃ are rotated to the left by 13 and 16 bits respectively. Vector v₁ is XOR-ed with v₀ and v₃ with v₂. v₀ is then rotated to the left by 32 bits. v₂ is AND-ed with v₁ and v₀ with v₃. v₁ and v₃ are then rotated to the left by 17 and 21 bits. v₁ is Xor-ed with v₂ and v₃ with v₀. Finally v₂ is rotated to the left by 32 bits to complete a single SipRound. After this, the blocks are again XOR-ed with v₁ and finalized by XOR-ing v₂ with 0xff followed by d number of iterations of SipRounds. Finally the 4 vectors are XOR-ed together to return the 64-bit hash value.

![Figure 2: SipRound [3]](image)

3. **Hypothesis**

This project will explore the claim that inference based SAT attacks on reduced versions of SipHash are more effective than brute force attacks to retrieve the key used to create the hash. It will also try to find the complexity of SipHash at which SAT based attacks take as much time as brute force attacks. On an average, brute force attacks would take an average of $2^{127}$ evaluations to find the correct key. Here we assume that we know n bits of the key in which case a brute force attack would take $2^{127} - n$ attempts. This project will try to reinforce that SAT based attacks take lesser number of trials to deduce the key.

4. **Related work**

As mentioned in [5], SAT based attacks have been deployed on several hash functions. SAT based attacks have also been employed on CubeHash as outlined in [4]. SipHash is a relatively new hashing function. The authors have tested various types of attacks as outlined in [3] to ensure SipHash is a secure algorithm. This project will be building off of the design outlined in [4] and will look to explore the effectiveness of SAT based attacks on various versions of SipHash. This project will use an off the shelf SAT solver like CryptoMiniSAT2 that reads a SAT formula from a text file as input and solves the formula if a solution is possible.
5. **Architecture**

The project will be able to create SAT formulae for different number of compression and finalization rounds for a given version of SipHash-c-d with n known bits in the key. This will be done by a module that will take as input the number of compression and finalization rounds and the n known bits. Since we know the values of the plaintext and the final output, the rest of the values will be substituted into the formula which will provide us with a satisfiability formula involving the unknown 128-n bits of the key. The final output of the hash function is based on the state of the function that is stored in the four initialization vectors \( v_0 \) through \( v_3 \). The value of each bit of the vectors is determined by the initial key value, the initialization constants, the input text and the operations that are performed during each SipRound.

Based on the number of SipRounds, each bit is represented as a boolean formula of AND and XOR operations involving the unknown key bits. This will then be converted into the CNF form by a separate module before it is fed into a SAT solver like CryptoMiniSAT. Each boolean formula representing a bit of the vector will be converted into several clauses. As described in [4], an XOR operation equivalent to \( A \oplus B = C \) in CNF would be \((\neg A \lor \neg B \lor \neg C)(A \lor B \lor \neg C)(A \lor \neg B \lor C)(\neg A \lor B \lor C)\). Similarly the AND operation \( A \land B = C \) in CNF would be \((\neg A \lor \neg B \lor \neg C)(\neg A \lor B \lor \neg C)(A \lor \neg B \lor \neg C)(A \lor B \lor \neg C)\). The value of the variable C from the above mentioned expressions is determined by retrieving the expected hash value for the particular bit. Substituting the value into the CNF will simplify each of the clauses which are then fed to the SAT solver. The SAT solver will then process the formula and will decide if the formula is satisfiable or not. Based on the result, the number of trials to reach a solution is noted accordingly and then compared to an estimate of a brute force attack to compute the same. This process will be repeated several times and an average will be derived for each version of SipHash. The project will start testing on the simplest version of SipHash-1-0 and progress towards more complex SipHash versions until the SAT attacks take longer than brute force attacks. This project will only use only one message block to test the performance of the system. Having more blocks of the message will not make the satisfiability formula more complex as they are only XOR-ed with the initialization vector. Hence the performance of the system will not be affected by having more blocks of the message.

6. **Evaluation**

The performance of the system will be measured by comparing the number of conflicts reported by the SAT solver against the estimated number of tries using a brute force attack to find the bits of the key. A naive brute force attack will try all combinations of the key bits until we get the desired hash as the output. This will take an average of \( 2^{127} \) evaluations to find the correct key. A SAT solver uses clause learning to eliminate invalid entries and measuring the number of conflicts before the correct set of values is reached will give us a way to measure the performance of the SAT solver. A plot of the logarithm of the number of trials to find the solution versus the number of known bits in the key will give us an estimate of the difference in performance between using SAT attacks and brute force attacks. Number of tries to find the right solution will be calculated for varying complexities of SipHash-c-d for varying number of known bits of the key. If the number of tries using SAT solver do not exceed that using a brute force attack, sample values will be chosen and new points will be interpolated to find the complexity of SipHash-c-d at which SAT solver becomes less effective than brute force attacks.

The results of the project will be compared with other applications of SAT solver to cryptanalysis [5] to analyze the effectiveness of SAT-based attacks on cryptographic primitives.
7. **Deliverables**

- Implemented code
- Design documents
- Analysis of the results
- Report containing the details of the project
- Sample input files

8. **References**


9. **Schedule**

26 April 2013: Proposal draft

03 May 2013: Proposal finalized and start on design

17 May 2013: Implement conversion of AND and XOR operation into CNF

31 May 2013: Implement mapping of bits

14 June 2013: Implement simplification of equation before converting into CNF

28 June 2013: Improvements to existing code

12 July 2013: Prepare draft of report and have results ready to be examined

26 July 2013: Review results and make necessary changes

02 August 2013: Prepare for defense

10. **Current Status**

Currently, I have done some preliminary research and gathered references. I have modified an existing SipHash-2-4 code to generate the hash for any version of SipHash-c-d.