Appears in A. Balint, A. Belov, M.J.H. Heule, and M. Järvisalo (eds.), *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of Department of Computer Science Series of Publications B, University of Helsinki 2013. ISBN 978-952-10-8991-6.

Instance generator for encoding preimage, second-preimage, and collision attacks on SHA-1

Vegard Nossum
Department of Informatics, University of Oslo
Oslo, Norway

***Abstract*—The instance generator described in this document encodes three attacks on the cryptographic hash function SHA-1. Unlike most instance generators for cryptographic hash functions, our encoding is not based purely on the Tseitin transformation. In particular, we encode modular addition using column sums represented as pseudo-boolean constraints and minimised in clausal form using the heuristic logic minimiser ESPRESSO.**

I. Introduction

SHA-1 is a cryptographic hash function that was published by the NIST in 1995 [1]. Cryptographic hash functions are functions which are "hard to invert"; in particular, this means that a given function f should satisfy the following three properties:

1) **Preimage resistance.** Given a hash H, it is infeasible to find a message M such that $f(M) = H$.
2) **Second-preimage resistance.** Given a message M, it is infeasible to find another message M' such that $f(M) = f(M')$.
3) **Collision resistance.** It is infeasible to find distinct messages M and M' such that $f(M) = f(M')$.

In this document, we describe an instance generator that encodes the corresponding attacks against the compression function of SHA-1 as SAT problems. The attack is successful if the SAT solver is able to find a solution to the instance. Given that SHA-1 was designed with these three properties in mind, we expect the resulting instances to be among the most difficult combinatorial problems for a SAT solver.

II. Parameters

A. General parameters

To seed the generator's random number generator, use `--seed` i, where i is an integer.

The type of attack to encode can be specified using one of `--attack preimage`, `--attack second-preimage`, or `--attack collision`.

B. Difficulty parameters

For preimage attacks, there are three difficulty parameters: `--rounds` t, where $16 \leq t \leq 80$, `--hash-bits` m, where $0 \leq m \leq 160$, and `--message-bits` n, where $0 \leq n \leq 512$. See section III for more information about the impact these parameters have on the expected hardness of the resulting instances.

C. Format options

The generator supports output in both CNF and OPB formats, using `--cnf` and `--opb`; exactly one of these must be given. Since the rules of the SAT Competition 2013 mandate that no comment follows the "p" line, we provide an option `--sat2011` that outputs CNF files that strictly follow the rules of the SAT Competition 2011.

III. Expected hardness

Given that SHA-1 was designed to be hard to crack, it is highly unlikely that any solver will be able to solve an instance in reasonable time; however, we have measured the mean running time for MiniSat on a series of *reduced-difficulty* instances encoding preimage attacks.

The three difficulty parameters for preimage attacks are *number of rounds*, *number of fixed hash bits*, and *number of fixed message bits*.

The full SHA-1 algorithm has 80 rounds, of which (only) the first 16 take input directly from the message to be hashed. Therefore, the possible number of rounds are between 16 and 80, where 16 is a very easy instance and 80 is a very hard instance. To see the effect of the number of rounds, we lowered the number of fixed hash bits. We observed three distinct phases; between 16 and 21 rounds, the instances are trivial to solve. Between 22 and 26 rounds, the difficulty increases extremely rapidly (an instance with 26 rounds takes approximately 2^{11} times longer to solve than an instance with 22 rounds), and from 27 rounds onwards, the difficulty increases very slowly (an instance with 80 rounds takes approximately only twice as long to solve as an instance with 27 rounds).

The number of fixed hash bits varies between 0 and 160 and effectively allows us to adjust the number of bits in the hash; with a value of 0, any message will be a solution (thus, an extremely easy instance), and with a value of 160, we require the message to hash exactly to the given hash value. The difficulty of the instance is roughly (but not quite) exponential in the number of fixed hash bits.

In the full SHA-1 algorithm, the input to the compression function is 512 bits of the message. Thus, 2^{512} is an upper limit on the size of the search space of a brute force search for a preimage. By adjusting the number of fixed message bits, we effectively give the solver parts of one known solution. By increasing this number, we effectively lower the search space of a brute force search. However, our observations indicate that by fixing a small number of bits (i.e. less than 32), the problem

becomes drastically more difficulty to solve. Only by fixing a very large number of bits (i.e. more than approximately $512-24$) does the problem become easier to solve.

See [2] for more detailed information about the hardness as a function of these parameters.

For the SAT Competition 2013, for instances that on average roughly take around the time limit of 5000 s to solve using MINISAT, we suggest the following combinations of parameters:

- 22 rounds, 128–160 hash bits, and 0 fixed message bits;
- 23 rounds, 64–96 hash bits, and 0 fixed message bits;
- 80 rounds, 8–12 hash bits, and 0 fixed message bits.

IV. Encoding of 5-ary 32-bit modular addition

Each round of SHA-1 includes exactly one 5-ary 32-bit adder. Expressing a constraint over 160 boolean variables (one of the inputs is an integer constant and is therefore disregarded), this actually constitutes a large part of the instance in terms of the number of clauses needed to encode it.

One very simple and frequently used way to encode addition is to use the Tseitin transformation on a standard ripple-carry adder circuit. This typically means introducing a lot of extra variables: one for each gate in the circuit. We take a different (and, we believe, novel) approach based on column sums expressed as pseudo-boolean constraints and further encoded in clausal form using the ESPRESSO heuristic logic minimiser.

Consider the following grade school addition schema for three binary numbers, x, y, and z, and their sum, w:

$$\begin{array}{cccccc} & c_3 & & & & \\ & c_2 & c_1 & & & \\ & & & c_0 & & \\ \hline & x_3 & x_2 & x_1 & x_0 \\ & y_3 & y_2 & y_1 & y_0 \\ + & z_3 & z_2 & z_1 & z_0 \\ \hline = & w_3 & w_2 & w_1 & w_0 \end{array}$$

The sum of three bits is either 0, 1, 2, or 3, and thus can be represented by a two-bit number. For the first (rightmost) column, we let $c_0 w_0$ be the sum of x_0, y_0, and z_0, and express it with the following pseudo-boolean constraint:

$$x_0 + y_0 + z_0 = 2c_0 + w_0$$

The *carry bit* c_0 is added to the next column, which is summed in a similar way. However, the sum of four bits is at most 4 and must be represented with three bits, in this case $c_2 c_1 w_1$. Thus, we obtain the constraint for the second column sum:

$$c_0 + x_1 + y_1 + z_1 = 4c_2 + 2c_1 + w_1$$

We continue in the same way for the remaining columns, with one small exception: since we are encoding *modular* addition, we will get some extraneous carry bits towards the end that should simply be discarded. These carry bits are termed *dummy* bits (as they are only ever used as placeholders for any value) and denoted with the letter d. The last two columns of this particular example are therefore encoded as follows:

$$c_1 + x_2 + y_2 + z_2 = 4d_0 + 2c_3 + w_2$$
$$c_3 + c_2 + x_3 + y_3 + z_3 = 4d_2 + 2d_1 + w_3$$

Having obtained a set of k pseudo-boolean constraints (for a k-bit adder), we now encode these constraints in CNF using ESPRESSO. Since the number of variables in each constraint is fairly small ($n + \lfloor 1 + \log_2 n \rfloor$ for an n-ary adder; at most 10 variables for 5-ary 32-bit modular addition), enumerating their truth tables (of at most 2^{10} entries) is completely feasible. The final number of clauses for each column sum depends on the constraint, but is in any case bounded by the size of its truth table.

V. Comparison with other generators

We make a brief comparison with other encodings of SHA-1 preimage attacks found in the literature:

Encoding	Variables	Clauses	Ratio
Our encoding	13,408	478,476	35.69
CRYPTLOGVER [3]	44,812	248,220	5.54
Plain Tseitin [4]	$\approx$ 55,000	$\approx$ 235,000	$\approx$ 4.27

In short, our encoding has fewer variables, more clauses, and is easier to solve than the variants of the Tseitin encoding.

VI. Verifier

In addition to the instance generator, we also provide a *verifier* for instances encoding preimage attacks. The verifier takes the instance and a solution (as found by a SAT solver) and verifies that the solution is indeed a valid preimage for the (possibly partial) hash value encoded in the instance.

The verifier does not simply check that the solution satisfies the clauses in the instance; rather, it calculates the SHA-1 hash of the message part of the solution and checks that it matches the hash part of the solution. This ensures not only that the solver is correct, but that the encoding itself is correct. (Of course, we can only ensure that a particular solution to and encoding of a particular instance is correct, but this is good enough in practice.)

VII. Availability

The program and scripts are available as Free Software (under the GNU General Public License version 3) from https://github.com/vegard/sha1-sat/. The program depends on the logic minimiser ESPRESSO in order to run.

References

[1] "Secure Hash Standard," ser. FIPS, vol. 180-1. National Institute of Standards and Technology, 1995.

[2] V. Nossum, "SAT-based preimage attacks on SHA-1," Master's thesis, University of Oslo, 2012.

[3] P. Morawiecki and M. Srebrny, "A SAT-based preimage analysis of reduced KECCAK hash functions," Cryptology ePrint Archive, Report 2010/285, 2010. [Online]. Available: http://eprint.iacr.org/2010/285.pdf

[4] M. Srebrny, M. Srebrny, and L. Stepień, "SAT as a programming environment for linear algebra and cryptanalysis," in *ISAIM*, 2007.