Security Margin Evaluation of SHA-3 Contest Finalists through SAT-Based Attacks

Ekawat Homsirikamol[2], Paweł Morawiecki[1],
Marcin Rogawski[2], and Marian Srebrny[1,3]

[1] Section of Informatics, University of Commerce, Kielce, Poland
[2] Cryptographic Engineering Research Group, George Mason University, USA
[3] Institute of Computer Science, Polish Academy of Sciences, Poland

Abstract. In 2007, the U.S. National Institute of Standards and Technology (NIST) announced a public contest aiming at the selection of a new standard for a cryptographic hash function. In this paper, the security margin of five SHA-3 finalists is evaluated with an assumption that attacks launched on finalists should be practically verified. A method of attacks is called logical cryptanalysis where the original task is expressed as a SATisfiability problem. To simplify the most arduous stages of this type of cryptanalysis and helps to mount the attacks in a uniform way a new toolkit is used. In the context of SAT-based attacks, it has been shown that all the finalists have substantially bigger security margin than the current standards SHA-256 and SHA-1.

Keywords: Cryptographic hash algorithm, SHA-3 competition, algebraic cryptanalysis, logical cryptanalysis, SATisfiability solvers.

1 Introduction

In 2007, the U.S. National Institute of Standards and Technology (NIST) announced a public contest aiming at the selection of a new standard for a cryptographic hash function. The main motivation behind starting the contest has been the security flaws identified in SHA-1 standard in 2005. Similarities between SHA-1 and the most recent standard SHA-2 are worrisome and NIST decided that a new, stronger hash function is needed. 51 functions were accepted to the first round of the contest and in July 2009 among those functions 14 were selected to the second round. At the end of 2010 five finalists were announced: BLAKE [15], Groestl [21], JH [28], Keccak [13], and Skein [2]. The winning algorithm will be named 'SHA-3' and most likely will be selected in the second half of 2012.

Security, performance in software, performance in hardware and flexibility are the four primary criteria normally used in evaluation of candidates. Out of these four criteria, security is the most important criterion, yet it is also the most difficult to evaluate and quantify. There are two primary ways of estimating the security margin of a given cryptosystem. The first one is to compare the complexities of the best attack on the full cryptosystem. The problem with this

A. Cortesi et al. (Eds.): CISIM 2012, LNCS 7564, pp. 56–67, 2012.

approach is that for many modern designs there is no known successful attack on the full cryptosystem. Security margin would be the same for all algorithms where there is nothing better than the exhaustive search if this approach is used. This is not different for SHA-3 contest where no known attacks on the full functions, except JH, have been reported. For JH there is a preimage attack [19] but its time complexity is nearly equal to the exhaustive search and memory accesses are over the exhaustive search bound. Therefore estimating the security margin using this approach tells us very little or nothing about differences between the candidates in terms of their security level. The second approach of how to measure the security margin is to compare the number of broken rounds to the total number of rounds in a given cryptosystem. As a vast majority of modern ciphers and hash functions (in particular, the SHA-3 contest finalists) has an iterative design, this approach can be applied naturally. However, there is also a problem with comparing security levels calculated this way. For example, there is an attack on 7-round Keccak-512 with complexity 2^{507} [3] and there is an attack on 3-round Groestl-512 with complexity 2^{192} [23]. The first attack reaches relatively more rounds (29%) but with higher complexity whereas the second attack has lower complexity but breaks fewer rounds (21%). Both attacks are completely non-practical. It is very unclear how such results help to judge which function is more secure.

In this paper we follow the second approach of measuring the security margin but with an additional restriction. We assume that the attacks must have practical complexities, i.e., can be practically verified. It is very similar to the line of research recently presented in [5,6]. This restriction puts the attacks in more 'real life' scenarios which is especially important for SHA-3 standard. So far a large amount of cryptanalysis has been conducted on the finalists, however the majority of papers focuses on maximizing the number of broken rounds which leads to extremely high data and time complexity. These theoretical attacks have great importance but the lack of practical approach is evident. We hope that our work helps to fill this gap to some extent.

The method of our analysis is a SAT-based attack. SAT was the first known NP-complete problem, as proved by Stephen Cook in 1971 [7]. A SAT solver decides whether a given propositional (boolean) formula has a satisfying valuation. Finding a satisfying valuation is infeasible in general, but many SAT instances can be solved surprisingly efficiently. There are many competing algorithms for it and many implementations, most of them have been developed over the last two decades as highly optimized versions of the DPLL procedure [10] and [11].

SAT solvers can be used to solve instances typically described in the Conjunctive Normal Form (CNF) into which any decision problem can be translated. Modern SAT solvers use highly tuned algorithms and data structures to find a solution to a given problem coded in this very simple form. To solve your problem: (1) translate the problem to SAT (in such a way that a satisfying valuation represents a solution to the problem); (2) run your favorite SAT solver to find a solution. The first connection between SAT and crypto dates back to [8], where a suggestion appeared to use cryptoformulae as hard benchmarks for propositional satisfiability checkers. The first application of SAT solvers in cryptanalysis was due to Massacci et al. [18] called logical cryptanalysis. They ran a SAT solver on DES key search, and then also for faking an RSA signature for a given message by finding the e-th roots of a (digitalized) message m modulo n, in [12]. Courtois and Pieprzyk [9] presented an approach to code in SAT their algebraic cryptanalysis with some gigantic systems of low degree equations designed as potential procedures for breaking some ciphers. Soos et al. [26] proposed enhancing a SAT solver with some special-purpose algebraic procedures, such as Gaussian elimination. Mironov and Zhang [20] showed an application of a SAT solver supporting a non-automatic part of the attack [27] on SHA-1.

In this work we use SAT-based attacks to evaluate security margin of the 256-bit variant SHA-3 contest finalists and also compare them to the current standards, in particular SHA-256. We show that all five finalists have a big security margin against these kind of attacks and are substantially more secure than SHA-1 and SHA-256. We also report some interesting results on particular functions or its building blocks. Preimage and collision attacks were successfully mounted against 2-round Keccak. At the time of publication, this is the best known practical preimage attack on reduced Keccak. A pseudo-collisions on 6-round Skein-512-256 was also found. For the comparison, the Skein's authors reached 8 rounds but they found only pseudo-near-collision [2]. In the attacks we use our toolkit which is a combination of the existing tools and some newly developed parts. The toolkit helps in mounting the attacks in a uniform way and it can be easily used for cryptanalysis not only of hash functions but also of other cryptographic primitives such as block or stream ciphers.

2 Methodology of our SAT-Based Attacks

2.1 A Toolkit for CNF Formula Generation

One of the key steps in attacking cryptographic primitives with SAT solvers is CNF (conjunctive normal form) formula generation. A CNF is a conjunction of clauses, i.e., of disjunctions of literals, where a literal is a boolean valued variable or its negation. Thus, a formula is presented to a SAT solver as one big 'AND' of 'ORs'. A cryptographic primitive (or a segment of it) which is the target of a SAT based attack has to be completely described by such a formula. Generating it is a non-trivial task and usually very laborious. There are many ways to obtain a final CNF and the output results differ in the number of clauses, the average size of clauses and the number of literals. Recently we have developed a new toolkit which greatly simplifies the generation of CNF.

Usually a cryptanalist needs to put a considerable effort into creating a final CNF. It involves writing a separate program dedicated only to the cryptographic primitive under consideration. To make it efficient, some minimizing algorithms (Karnaugh maps, Quine-McCluskey algorithm or Espresso algorithm) have to be used [17]. These have to be implemented in the program, or the intermediate results are sent to an external tool (e.g., Espresso minimizer) and then the minimized form is sent back to the main program. Implementing all of these procedures requires a good deal of programming skills, some knowledge of logic synthesis algorithms and careful insight into the details of the primitive's operation. As a result, obtaining CNF might become the most tedious and error-prone part of any attack. It could be especially discouraging for researchers who start their work from scratch and do not want to spend too much time on writing thousands lines of code.

To avoid those disadvantages we have recently proposed a new toolkit consisting basically of two applications. The first of them is Quartus II — a software tool released by Altera for analysis and synthesis of HDL (Hardware Description Language) designs, which enables the developers to compile their designs and configure the target devices (usually FPGAs). We use a free-of-charge version Quartus II Web Edition which provides all the features that we need. The second application, written by us, converts boolean equations (generated by Quartus) to CNF encoded in DIMACS format (standard format for today's SAT solvers). The complete process of CNF generation includes the following steps:

1. Code the target cryptographic primitive in HDL;
2. Compile and synthesize the code in Quartus;
3. Generate boolean equations using Quartus inbuilt tool;
4. Convert generated equations to CNF by our converter.

Steps 2, 3, and 4 are done automatically. Using this method the only effort a researcher has to put is to write a code in HDL. Normally programming and 'thinking' in HDL is a bit different from typical high-level languages like Java or C. However it is not the case here. For our needs, programming in HDL looks exactly the same as it would be done in high-level languages. There is no need to care about typical HDL specific issues like proper expressing of concurrency or clocking. It is because we are not going to implement anything in a FPGA device. All we need is to obtain a system of boolean equations which completely describes the primitive we wish to attack. Once the boolean equations are generated by the Quartus inbuilt tool, the equations are converted into CNF by the separate application. The conversion implemented in our application is based on the boolean laws (commutativity, associativity, distributivity, identity, De Morgan's laws) and there are no complex algorithms involved.

It must be noted that Quartus programming environment gives us two important features which may help to create a possibly compact CNF. It minimizes the functions up to 6 variables using Karnaugh maps. Additionally, all final equations have at most 5 variables (4 inputs, 1 output). It is because Quartus is dedicated to FPGA devices which are built out of 'logic cells', each with 4 inputs/1 output. (There are also FPGAs with different parameters; e.g., 5/2. But we chose 4/1 architecture in all the experiments.) This feature is helpful when dealing with linear ANF equations with many variables (also referred as 'long XOR equations'). A simple conversion of such an equation to CNF gives an exponential number of clauses; an equation in n-variables corresponds to 2^{n-1} clauses in CNF. A common way of dealing with this problem is to introduce new variables and cut the original equation into a few shorter ones.

Example 1. Let us consider an equation with 5 variables:

$$a + b + c + d + e = 0$$

A CNF corresponding to this equation consists of 2^{5-1} clauses with 5 literals in each clause. However, introducing two new variables, we can rewrite it as a system of three equations:

$$\begin{aligned} a + b + x &= 0 \\ c + d + y &= 0 \\ e + x + y &= 0 \end{aligned}$$

A CNF corresponding to this system of equations would consist of $2^2 + 2^2 + 2^2 = 12$ clauses.

Quartus automatically introduces new variables and cuts long equations to satisfy the requirements for FPGA architecture. Consequently a researcher needs not be worried that the CNF would be much affected by very long XOR equations (which may be a part of the original cryptographic primitive's description).

To the best of our knowledge, there are only two other tools which provide similar functionality to our toolkit — automate the CNF generation and help to mount the uniform SAT-based attacks. First is the solution proposed in [16] where the main idea is to change the behaviour of all the arithmetic and logical operators that the algorithm uses, in such a way that each operator produces a propositional formula corresponding to the operation performed. It is obtained by using C++ implementation and a feature of the C++ language called operator overloading. Authors tested their method on MD4 and MD5 functions. The proposed method can be applied to other crypto primitives but it is not clear how it would deal with more complex operations, e.g. an S-box described as a look-up table. The second tool is called Grain of Salt [25] and it incorporates some algorithms to optimize a generated CNF. However it can be only used with a family of stream ciphers.

In comparison to these two tools, our proposal is the most flexible. It can be used with many different cryptographic primitives (hash functions, block and stream ciphers) and it does not limit an input description to only simple boolean operations. The toolkit handles XOR equations efficiently and also takes an advantage of logic synthesis algorithms which help to provide more compact CNF.

2.2 Our SAT-Based Attack

All the attacks reported in the paper have a very similar form and consist of the following steps.

1. Generate the CNF formula by our toolkit;
2. Fix the hash and padding bits in the formula;
3. Run a SAT solver on the generated CNF.

The above scheme is used to mount a preimage attack, i.e., for a given hash value h, we try to find a message m such that $h = f(m)$. CryptoMiniSAT2, gold medalist from recent SAT competitions [24], is selected as our SAT solver. In the preliminary experiments, we also tried other state-of-art SAT solvers (Lingeling [4], Glucose [1]) but overall CryptoMiniSAT2 solves our formulas faster.

We attack functions with 256-bit hash. When constructing a CNF coding a hash function, one has to decide the size of the message (how many message blocks are taken as an input to the function). It is easier for a SAT solver to tackle with a single message block because coding each next message block would make a formula twice as big. However, each of the five finalists has a different way of padding the message. If only one message block is allowed, BLAKE-256 can take maximally 446 bits of message which are padded to get a 512-bit block. On the other hand, Keccak-256 can take as many as 1086 bits of message in a single block. To avoid the situation where one formula has much more message bits to search for by a SAT solver than the other formula, message is fixed to 446 bits (maximum value for BLAKE-256 with one message block processed, other finalists allow more).

To find a second preimage or a collision, only a small adjustment to the aforementioned attack is required. Once the preimage is found, we run SAT solver on exactly the same formula but with one message bit fixed to the opposite value of that from the preimage (rest of the message bits are left unknown). It turns out that in a very similar time the SAT solver is able to solve such slightly modified formula, providing a second preimage and a collision. The second preimages/collisions are expected because with a size of the message fixed to 446 bits we have 446 to 256 bits mapping.

3 Results

We have conducted the preimage attack described in Section 2.2 on the five finalists and also on the two standards SHA-256 and SHA-1. As a SAT solver we used CryptoMiniSat2, 2.9.0 version, with the parameters *gaussuntil=0* and *restart=static*. These settings were suggested by the author of CryptoMiniSat2. The experiments were carried out on Intel Core i7 2.67 GHz with 8 Gb RAM. Starting with 1-round variants of the functions, the SAT solver was run to solve the given formula and gave us the preimage. The time limit for each experiment was set to 30 hours. If the solution was found, we added one more round, encoded in CNF and gave it to the solver. The attack stopped when the time limit was exceeded or memory ran out. Table 1 shows the results. The second column contains the number of broken rounds in our preimage attack and the third column shows the security margin calculated as a quotient of the number of broken rounds and the total number of rounds. For clarity, we are reporting our preimage attack but, as explained above, it can be easily modified to get a second preimage or a collision. Therefore the numbers from Table 1 remain valid for all three types of attacks.

All the SHA-3 contest finalists have substantially bigger security margin than SHA-256 and SHA-1 standards. On the other hand, the finalists differ slightly (maximally 7%) and all have the security margin over 90%. For Groestl we were not able to attack even a 1-round variant, nor a simplified Groestl with the output transformation replaced by a simple truncation. The only successful attack on Groestl (or rather part of it) is the attack on the output transformation in the 1-round variant of Groestl. The output transformation is not a complete round but giving 100% of security margin would not be fair neither. Therefore we try to estimate 'a weight' of the output transformation. Essentially all the operations (equations) in Groestl compression function come from two very similar permutations (P and Q). The output transformation is built on the P permutation only so it can be treated as a half-operation of the compression function. Hence the attack on the output transformation in a 1-round variant of Groestl is shown in Table 1 as half the round.

All the reported attacks on the finalists took just a few seconds. Only for 16-round SHA-256 the attack lasted longer — one hour. Despite the fact a conservative time limit (30 hours) was set for this type of experiments, it did not help to extend the attack to reach one more round. It seems that the time of the attack grows superexponentially in the number of rounds. The same behaviour was observed by Rivest et al. when they tested MD6 function with their SAT-based analysis [22]. For MD6 with 256-bit hash size, they reached 10 rounds which gives 90% of security margin. For a reader interested in estimating the asymptotic complexity of our attacks, we report that it would be very difficult mainly because Altera does not reveal details of algorithms used in Quartus.

Table 1. Security margin comparison calculated from the results of our preimage attacks on round-reduced hash functions

Function	No. of rounds	Security margin
SHA-1	21	74% (21/80)
SHA-256	16	75% (16/64)
Keccak-256	2	92% (2/24)
BLAKE-256	1	93% (1/14)
Groestl-256	0.5*	95% (0,5/10)
JH-256	2	96% (2/42)
Skein-512-256	1	99% (1/72)

* Only output transformation broken. It is estimated as an equivalent to one half-operation of the Groestl compression function.

It is interesting to see if the parameters of CNF formula, that is the number of variables and clauses, could be a good metric for measuring the hardness of the formula and consequently the security margin. Table 2 shows the numbers of variables and clauses for full hash functions. The values are rounded to the nearest thousand. For SHA-1, SHA-256, BLAKE, and Skein, we have generated the complete formula with our toolkit. For the other functions, we have extrapolated the numbers from round-reduced variants as the toolkit had some memory problems with those huge instances (over 1 million of clauses). As every round in the given function is basically the same (consists of the same type of equations), the linear extrapolation is straightforward. For the examined functions, the CNF formula parameters could be a good metric for measuring the hardness of the formula but only to some extent. Indeed, the smallest formulas (SHA-1 and SHA-256) have the lowest security margin but, for example, BLAKE and Keccak have nearly the same security level while Keccak formula is more than twice as big.

Table 2. The parameters of our CNF formulas coding hash functions

Function	Variables	Clauses
SHA-1	29 000	200 000
SHA-256	61 000	400 000
BLAKE-256	57 000	422 000
Keccak-256	88 000	1 075 000
Skein-512-256	148 000	1 041 000
JH-256	169 600	1 998 000
Groestl-256	279 000	3 568 000

Besides the attacks on (round-reduced) hash functions, we have also mounted the attacks on the compression functions — main building blocks of hash functions. First we tried the preimage attack on a given compression function and if it did not succeed we attacked the function in a scenario where an adversary can choose IV (initial value) and get a pseudo-preimage. Table 3 summarizes these attacks. Similarly as for the results from Table 1, the numbers remain valid for all three types of attacks (a preimage, a second preimage and a collision attack). Among the finalists our best attack was on 6-round Skein-512-256 compression function for which we found pseudo-collisions. For comparison, the Skein's authors reached 8 rounds but they found only a pseudo-near-collision. For Groestl compression function we were not able to mount any successful attack. Keccak has a completely different design from MD hash function family — there is no a typical compression function taking IV and a block of a message. Therefore in Table 3 we do not report any result for these two functions.

It is very difficult to give a good and clear answer which designs or features of a hash function are harder for SAT solvers. One observation we have made is that those designs which use S-boxes (JH and Groestl) have the biggest CNF formula and are among the hardest for SAT solvers. Equations describing an S-box are more complex than equations describing addition or boolean AND. Consequently, the corresponding CNF formula for the S-box is also more complex with greater number of variables and clauses than in the case of other typical operations. Before we give an example, let us first take a closer look at the addition operation. This operation is used in SHA-1, SHA-256, BLAKE, and Skein. In our toolkit the addition of two words is described by the following equations (a full adder equations):

Table 3. Attacks on the compression functions

Function	Type of attack	No. of rounds	Security margin
SHA-1	preimage	21	74% (21/80)
SHA-256	preimage	16	75% (16/64)
BLAKE-256	preimage	1	93% (1/14)
JH-256	preimage	2	96% (2/42)
Skein-512-256	pseudo-preimage	6	92% (6/72)

$$S_i = A_i \oplus B_i \oplus C_{i-1}$$

$$C_i = (A_i \cdot B_i) \oplus (C_{i-1} \cdot (A_i \oplus B_i))$$

S_i is the i-th bit of the sum of two i-bit words A and B. C_i is the i-th carry output.

Now let us compare the CNF sizes of the addition operation and AES S-box used in Groestl. A CNF of 32-bit addition has 411 clauses and 124 variables while AES S-box given to our toolkit as a look-up table gives a CNF with 4800 clauses and 900 variables. We also experimented with an alternative description of AES S-box expressed as boolean logic equations, instead of a look-up table [14]. This description reduces the CNF size approximately by half but still it is a degree of order greater than the CNF from the 32-bit addition operation.

We have also observed that there is no clear limit in size of CNF formulas beyond which a SAT solver fails. For example the CNF of 2-round JH with 59 thousand clauses is solved within seconds whereas the CNF of 2-round Skein with 27 thousand clauses was not solved having 30 hours of time limit. What exactly causes the difference between hardness of formulas is a good point for further research.

4 Conclusion

The security margin of the five finalists of the SHA-3 contest using our SAT-based cryptanalysis has been evaluated in this paper. A new toolkit which greatly simplifies the most tedious stages of this type of analysis and helps to mount the attacks in a uniform way has been proposed and developed. Our toolkit is more flexible than the existing tools and can be applied to various cryptographic primitives. Based on our methodology, we have shown that all the finalists have substantially bigger security margin than the current standards SHA-256. We stress that 'bigger security margin' we refer only to the context of our SAT-based analysis. Using other techniques (e.g., linear cryptanalysis) could lead to a different conclusion.

As a side effect of our security margin evaluation, we have also carried out some attacks on compression functions and reported some new state-of-the-art results. For example, we have found pseudo-collisions for 6-round Skein-512-256 compression function.

References

1. Audemard, G., Simon, L.: Glucose SAT Solver, http://www.lri.fr/~simon/?page=glucose
2. Schneier, B., et al.: The Skein Hash Function Family, http://www.skein-hash.info/sites/default/files/skein1.1.pdf
3. Bernstein, D.J.: Second preimages for 6 (7? (8??)) rounds of Keccak? NIST mailing list (2010), http://ehash.iaik.tugraz.at/uploads/6/65/NIST-mailing-list_Bernstein-Daemen.txt
4. Biere, A.: Lingeling, http://fmv.jku.at/lingeling
5. Biryukov, A., Dunkelman, O., Keller, N., Khovratovich, D., Shamir, A.: Key Recovery Attacks of Practical Complexity on AES Variants With Up To 10 Rounds. Cryptology ePrint Archive, Report 2009/374 (2009), http://eprint.iacr.org/2009/374
6. Bouillaguet, C., Derbez, P., Dunkelman, O., Keller, N., Rijmen, V., Fouque, P.A.: Low Data Complexity Attacks on AES. Cryptology ePrint Archive, Report 2010/633 (2010), http://eprint.iacr.org/2010/633
7. Cook, S.A.: The complexity of theorem-proving procedures. In: Proceedings of the Third Annual ACM Symposium on Theory of Computing, STOC 1971, pp. 151–158. ACM, New York (1971)
8. Cook, S.A., Mitchell, D.G.: Finding hard instances of the satisfiability problem: A survey, pp. 1–17. American Mathematical Society (1997)
9. Courtois, N.T., Pieprzyk, J.: Cryptanalysis of Block Ciphers with Overdefined Systems of Equations. In: Zheng, Y. (ed.) ASIACRYPT 2002. LNCS, vol. 2501, pp. 267–287. Springer, Heidelberg (2002)
10. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Communications of the ACM 7(5), 394–397 (1962)
11. Davis, M., Putnam, H.: A computing procedure for quantification theory. Journal of the ACM 7, 201–215 (1960)
12. Fiorini, C., Martinelli, E., Massacci, F.: How to fake an RSA signature by encoding modular root finding as a SAT problem. Discrete Applied Mathematics 130, 101–127 (2003)
13. Bertoni, G., et al.: Keccak sponge function family main document, http://keccak.noekeon.org/Keccak-main-2.1.pdf
14. Gaj, K., Chodowiec, P.: FPGA and ASIC Implementatios of AES. In: Koc, C.K. (ed.) Cryptographic Engineering, ch. 10, pp. 235–294. Springer (2009)
15. Aumasson, J.P., et al.: SHA-3 proposal BLAKE, http://www.131002.net/blake/
16. Jovanović, D., Janičić, P.: Logical Analysis of Hash Functions. In: Gramlich, B. (ed.) FroCos 2005. LNCS (LNAI), vol. 3717, pp. 200–215. Springer, Heidelberg (2005)
17. Lala, P.K.: Principles of modern digital design. Wiley-Interscience (2007)
18. Massacci, F.: Using Walk-SAT and Rel-SAT for cryptographic key search. In: Proceedings of the International Joint Conference on Artificial Intelligence, pp. 290–295 (1999)
19. Mendel, F., Thomsen, S.: An Observation on JH-512 (2008), http://ehash.iaik.tugraz.at/uploads/d/da/Jh_preimage.pdf
20. Mironov, I., Zhang, L.: Applications of SAT Solvers to Cryptanalysis of Hash Functions. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 102–115. Springer, Heidelberg (2006)
21. Gauravaram, P., et al.: Grøstl — a SHA-3 candidate, http://www.groestl.info

22. Rivest, R., et al.: The MD6 hash function, http://groups.csail.mit.edu/cis/md6/
23. Schlaffer, M.: Updated Differential Analysis of Grøestl. Grøstl website (January 2011), http://groestl.info/groestl-analysis.pdf
24. Soos, M.: CryptoMiniSat 2.5.0. In: SAT Race Competitive Event Booklet (July 2010), http://www.msoos.org/cryptominisat2
25. Soos, M.: Grain of Salt — An Automated Way to Test Stream Ciphers through SAT Solvers. In: Workshop on Tools for Cryptanalysis (2010)
26. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT Solvers to Cryptographic Problems. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 244–257. Springer, Heidelberg (2009)
27. Wang, X., Yin, Y.L., Yu, H.: Finding Collisions in the Full SHA-1. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 17–36. Springer, Heidelberg (2005)
28. Wu, H.: Hash Function JH, http://www3.ntu.edu.sg/home/wuhj/research/jh/

Appendix

For the reader's convenience, we provide an example SystemVerilog code for SHA-1 used in the experiments with our toolkit. In many cases a code strongly resembles a pseudocode defining a given cryptographic algorithm. A reader familiar with C or Java should have no trouble adjusting the code to our toolkit's needs.

```
module sha1(IN, OUT);
   input [511:0] IN; // input here means 512-bit message block
   output [159:0] OUT;  // output here means 160-bit hash
   reg [159:0] OUT;
   reg [31:0] W_words [95:0]; // registers for W words
   reg [31:0] h0 ,h1, h2, h3, h4;
   reg [31:0] a, b, c, d, e, f, k, temp, temp2;
   integer i;

  always @ (IN, OUT)
  begin

h0 = 32'h67452301; h1 = 32'hEFCDAB89;
h2 = 32'h98BADCFE; h3 = 32'h10325476;
h4 = 32'hC3D2E1F0;

a = h0; b = h1; c = h2; d = h3; e = h4;

 W_words[15] = IN[31:0];  W_words[14] = IN[63:32];
 W_words[13] = IN[95:64]; W_words[12] = IN[127:96];
 W_words[11] = IN[159:128]; W_words[10] = IN[191:160];
 W_words[9] = IN[223:192];  W_words[8] = IN[255:224];
 W_words[7] = IN[287:256];  W_words[6] = IN[319:288];
 W_words[5] = IN[351:320];  W_words[4] = IN[383:352];
 W_words[3] = IN[415:384];  W_words[2] = IN[447:416];
 W_words[1] = IN[479:448];  W_words[0] = IN[511:480];
```

```
for (i=16; i<=79; i=i+1)
 begin
 W_words[i] = W_words[i-3] ^ W_words[i-8] ^ W_words[i-14] ^ W_words[i-16];
 W_words[i] = {W_words[i][30:0], W_words[i][31]}; // leftrotate 1
 end

 for (i=0; i<=79; i=i+1) // main loop
  begin
  if ((i>=0) && (i<=19))
     begin
     f = (b & c) | ((~b) & d); k = 32'h5A827999;
     end
  if ((i>=20) && (i<=39))
     begin
     f = b ^ c ^ d;  k = 32'h6ED9EBA1;
     end
  if ((i>=40) && (i<=59))
     begin
     f = (b & c) | (b & d) | (c & d);  k = 32'h8F1BBCDC;
     end
  if ((i>=60) && (i<=79))
     begin
     f = b ^ c ^ d;  k = 32'hCA62C1D6;
     end

temp2 = {a[26:0], a[31:27]}; // a leftrotate 5
temp = temp2 + f + e + k + W_words[i];
e = d;
d = c;
c = {b[1:0], b[31:2]}; // b leftrotate 30
b = a;
a = temp;
end // end of main loop

h0 = h0 + a; h1 = h1 + b;
h2 = h2 + c; h3 = h3 + d; h4 = h4 + e;
OUT = {h0, h1, h2, h3, h4}; //HASH
end
endmodule
```