A SAT-based preimage analysis of reduced KECCAK hash functions

Paweł Morawiecki[1] and Marian Srebrny[2]

[1] University of Commerce, Kielce, Poland, pawelm@wsh-kielce.edu.pl

[2] Institute of Computer Science,
Polish Academy of Sciences, Warsaw, Poland;
and University of Commerce, Kielce, Poland; marians@ipipan.waw.pl

Abstract. In this paper, we present a preimage attack on reduced versions of KECCAK hash functions. We use our recently developed toolkit CryptLogVer for generating CNF (conjunctive normal form) which is passed to the SAT solver PrecoSAT [14]. We found preimages for some reduced versions of the function and showed that full KECCAK function is secure against the presented attack.

Key words: preimage attack, KECCAK, satisfiability, algebraic cryptanalysis, logical cryptanalysis, SAT solvers

1 Introduction

KECCAK is a family of cryptographic hash functions and was submitted as a SHA-3 candidate. The security of a publicly known cryptographic algorithm is accepted if there is no known successful attack on it. Often some partial trust is additionally based on some good statistical properties and reported failure of breaking attempts with some known methods, like differential or linear cryptanalysis. The recent new hash function MD-6 [16] has been also tested, among other methods, with logical (SAT-based) analysis.

SAT Solvers solve problems described in Conjunctional Normal Form (CNF) into which any decision problem can be translated. Modern SAT solvers use highly tuned algorithms and data structures to quickly find a solution to the problem described 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 the currently best SAT solver to find a solution. The propositional encoding formula can be thought of as a declarative program. One can treat the propositional calculus and the SAT solvers as a powerful programming environment that makes it possible to create and to run the propositional declarative programs for solving the encoded tasks. The hope you can get a solution relatively fast is based on the fact that the SAT solving algorithm is one of the best optimized.

A SAT testing algorithm decides whether a given propositional (boolean) formula has a satisfying valuation. SAT was the first known NP-complete problem, as proved by Stephen Cook in 1971. 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 of [5] and [6].

In this paper, we present a preimage attack on reduced versions of Keccak hash functions. We use our recently developed toolkit CryptLogVer for generating CNF (conjunctive normal form) which is passed to the SAT solver PrecoSAT [14].

The paper is organized as follows. In section 2 we present a short description of Keccak family functions. Then a CNF generation method is given. In section 3 the detailed attack scenario is described followed by our experimental results. Comparison to related work is indicated in section 7. The last section consists of some conclusion and future research.

2 Keccak — a brief description

In this section we present only a brief description of Keccak which can be helpful for understanding the attack described in the paper. For a complete information we refer the interested reader to [10].

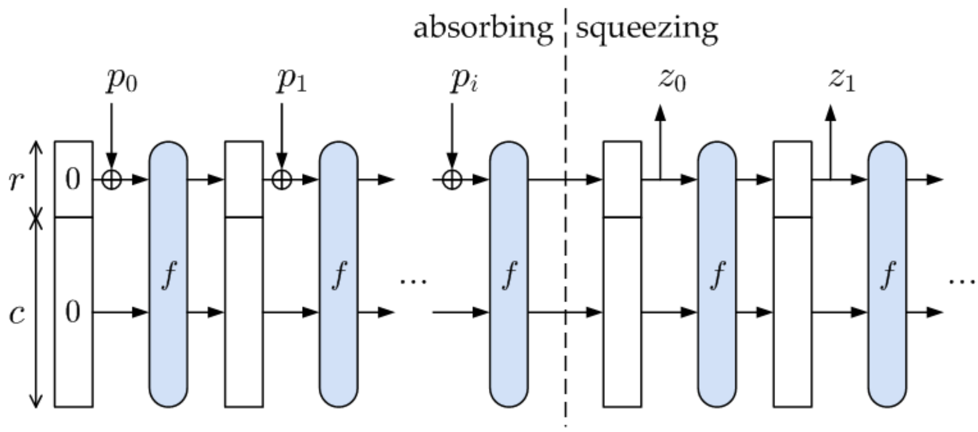

Fig. 1. Sponge Construction [18]

Keccak is a family of hash functions which makes the use of the sponge construction. Figure 1 shows the construction. It has two main parameters r and c which are called bitrate and capacity respectively. A sum of those two gives the state size which Keccak operates on. For a SHA-3 proposals, the state size is 1600 bits. A different values for bitrate and capacity give trade-off between speed and security. The higher bitrate gives the faster function but less secure. Keccak proceeds in two phases. In the first phase (absorbing) the r-bit input message blocks are xored into the first r bits of the state, interleaved with applications of the function f (called Keccak-f in the specification). This phase is finished when all message blocks are processed. In the second phase (squeezing) the first r bits of the state are returned as hash bits, interleaved with applications of the function f. The phase is finished when the desired length of hash is produced.

The default values for Keccak are $r = 1024$, $c = 576$ which gives 1600-bit state. However, Keccak can also operate on smaller states (25, 50, 100, 200, 400 and 800-bit state). The state size determines the number of rounds in Keccak-f function. For a default 1600-bit state there are 24 rounds. In the experiments we often used reduced versions with smaller number of rounds.

3 CNF formula generation

One of the key steps in attacking cryptographic primitves with SAT solvers is CNF formula generation. Such a formula completely describes the primitive (or a segment of the primitive) which is the target of the attack. Generating it is a non-trivial task and usually is very laborious. There are many ways to obtain the 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 called CryptLogVer which greatly simplifies the creation of CNF. Here we describe only the main concepts. The detailed description of CryptLogVer will be published in a separate paper [13].

Usually the cryptanalist needs to put a considerable effort into creating the final CNF. It involves writing a separate program dedicated only to a cryptographic primitive under consideration. To make it efficient, some minimizing algorithms (Karnaugh maps, Quine-McCluskey algorithm or Espresso algorithm) have to be used. These are 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 time-consuming 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 hundreds lines of code.

To avoid those disadvantages we have recently proposed a new toolkit consisting basically of two applications. First of them is Quartus II - a software tool produced 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 synthesise the code in Quartus
3. Generate boolean equations using Quartus inbuilt tool
4. Convert generated equations to CNF by a separate application.

Steps 2, 3, 4 are done automatically. 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 issues like proper expressing of concurrency or clocking. It is because we are not going to implement anything in FPGA device. All we need is to obtain a system of boolean equations which completely describes the primitive we wish to attack.

4 Description of the attack

We have carried out the preimage attack, i.e., for a given hash value h, we tried to find a message m such that $h = f(m)$. Our attack was applied on reduced versions of Keccak with smaller state and smaller number of rounds comparing to Keccak default settings. We experimented with the message lengths between 24 and 40 bits. Our attack scheme can be divided into three steps:

1. Generate CNF by CryptLogVer toolkit
2. Set output bits (hash) and part of input bits (padding bits)
3. Run PrecoSAT on created CNF

When a message searched for is supposed to be short (the number of message bits and required padding bits is less than or equal to bitrate r) it fits into one block P_i (see Figure 1). Consequently, there is only one invocation of Keccak-f function and the CNF used in the attacks can encode only Keccak-f. In general function f is crucial for the security of the sponge construction and its strength in many cases comes down to CICO problem (problem defined by Keccak designers, section 4.2.4 in Keccak main document [10]). The preimage attacks presented in this paper can be also treated as an attempt of solving CICO problem.

5 The experimental results

We attacked reduced versions of Keccak-f with different number of rounds, state sizes and message lengths. Table 1 summarizes our results. The experiments were carried out on an Intel Core2Duo 2GHz machine and for harder instances on a 4-core Intel Xeon 2.5 GHz which was a part of Grid'5000 system [7]. However only one core was used as PrecoSat is not a parallel solver. The function name specifies the state size, e.g., KECCAK-f[200] means that the function operates on 200-bit state. The exhaustive search was done with C++ implementation provided by KECCAK designers. The exhaustive search time is the time needed for checking all the combinations of the unknown message bits.

Input parameters				Attack times [seconds]	
Function	Number of rounds	Message size [bit]	Hash size [bit]	SAT-Solver attack	Exhaustive search
KECCAK-f[1600]	3	24	1024	1	86
KECCAK-f[1600]	3	32	1024	10	21805
KECCAK-f[1600]	3	40	1024	1852	5582135
KECCAK-f[200]	3	24	80	6	93
KECCAK-f[200]	3	32	80	53	15748
KECCAK-f[200]	3	40	80	56536	4031542
KECCAK-f[50]	4	24	24	4439	112
KECCAK-f[50]	5	24	24	7357	140

Table 1. Preimage attacks: SAT–based attacks vs. exhaustive search.

We also experimented with 4-round versions of KECCAK-f[200] and KECCAK-f[1600], but PrecoSAT was not able to find the solution. The time limit was 48 hours and it was tested for 32- and 40-bit messages. For toy versions such as KECCAK–f[50] the exhaustive search turned out to be better.

6 A note on two other SHA-3 candidates

We have managed to estimate the complexity of a boolean formula in its conjunctive normal form coding the hash function Grøstl (Austrian-Dutch SHA-3 candidate of Knudsen, Matusiewicz, et al.). Grøstl uses the AES S-boxes. [8] announces that the simplest version of Grøstl (with the 256-bit hash values) contains 1280 such S-boxes. A single such S-box has been coded by our CryptLogVer toolkit as a formula with around 4800 clauses and 900 variables. A straigthforward calculation gives at least 1280 * 4800 = 6 mln 144 thousand clauses in total. Hence, no SAT-based attack can be feasible (with no extra financial effort), even for reduced versions. From this perspective Grøstl seems to be very strong. For comparison, the AES standard has "only" 200 S-boxes.

We also have looked closer at Bernstein's CubeHash function [4], encouraged by its simplicity. We have obtained the following estimate of the CNF formula size possibly coding it. For the version originally submitted to the SHA-3 contest its CNF would have around 1 mln 760 thousands clauses and 270 thousand variables.

7 Related work

The designers of KECCAK made some experiments to solve the CICO problem for KECCAK-f[25]. They used SAGE computer algebra software and were able to solve the problem only for two rounds of KECCAK-f[25]. For more rounds the program ran out of memory. The work by Courtois and Bard [2] showed that SAT solvers can be a better option solving cryptographic problems (often comprising of large systems of equations) than computer algebra systems (such as SAGE, MAGMA or Singular) due to their much lower memory requirement.

In other work [1] the triangulation algorithm was used to solve the CICO problem. They reached 3 rounds for KECCAK-f[1600]. They fixed only a few bits which was enough to show non-randomness of the function but did not lead to any real attack. For smaller state sizes of KECCAK-f they did not pass 3 rounds.

We also carried out our SAT–based preimage attack on reduced versions of SHA1. For full SHA1, the CNF formula encoding the function has 181 thousand clauses and 31 thousand variables, while full KECCAK-f[1600] has 775 thousand clauses and 181 thousand variables. It could be already the sign that KECCAK is much stronger function, as the CNF formula is over 4 times bigger. We found a short preimage for 27-round SHA1 (out of its full 80 rounds), but only for 3 rounds out of 24 for KECCAK-f[1600].

8 Conclusion

We have carried out the SAT–based preimage attack on reduced KECCAK hash functions. The results proved the strength of the function against this kind of attack — we found preimages only for much reduced versions of KECCAK. We have found a preimage for the 3-round KECCAK-f[1600] with 40 unknown message bits. For future research it might be interesting to try extrapolating results for a full function. Such a technique was used in [17].

Acknowledgment

The authors gratefully acknowledge Mate Soos for his cooperation in carrying out some of the experiments on Grid'5000 [7].

References

1. J.-P. Aumasson and D. Khovratovich First Analysis of Keccak. `http://131002.net/data/papers/AK09.pdf`.
2. Courtois, N. T. and Bard, G. V. 2006. *Algebraic cryptanalysis of the Data Encryption Standard*. Cryptology ePrint Archive, Report 2006/402. `http://eprint.iacr.org/`.
3. Courtois, N. and Pieprzyk, J. 2002. Cryptanalysis of Block Ciphers with Overdefined Systems of Equations. In *ASIACRYPT 2002*, 267-287, .
4. *CubeHash*. `http://cubehash.cr.yp.to`.

5. Davis, M. and Putnam, H. 1960. A Computing Procedure for Quantification Theory. *Journal of the ACM* 7(1):201-215.
6. Davis, M.; Logemann, G. and Loveland, D.W. 1962. A Machine Program for Theorem Proving. *Communications of the ACM* 5(7):394-397.
7. *Grid'5000*. www.grid5000.fr.
8. *Grøstl* . http://www.groestl.info.
9. Jovanovic, D. and Janicic, P., Logical Analysis of Hash Functions. In: B. Gramlich (Ed.). *FroCoS 2005*. LNAI 3717. Springer-Verlag, 2005, 200-215.
10. *Keccak hash function*. http://keccak.noekeon.org/.
11. Massacci, F. 1999. Using walk-SAT and rel-sat for cryptographic key search. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI99)*, T. Dean, Ed. Morgan Kaufmann.
12. Mironov, I., and Zhang, L. 2006. Applications of SAT Solvers to Cryptanalysis of Hash Functions. In *Theory and Applications of Satisfiability Testing - SAT 2006*. Lecture Notes in Computer Science, volume 4121. Springer-Verlag, 2006, 102-115.
13. Morawiecki, P., Srebrny, M. and Srebrny, M. 2010. Towards CryptLogVer toolkit for logical cryptanalysis and verification - to appear.
14. Biere, A. 2009 P{re,i}coSAT@SC09. SAT 2009 competitive events booklet http://fmv.jku.at/precosat/.
15. *Quartus II web edition software*. https://www.altera.com/support/software/download/altera_design/quartus_we/dnl-quartus_we.jsp.
16. Rivest, R. L., et al. 2008. *The MD6 hash function – A proposal to NIST for SHA-3*. http://groups.csail.mit.edu/cis/md6/submitted-2008-10-27/Supporting_Documentation/md6_report.pdf. Submission to NIST.
17. Soos, M., Nohl, K. and Castelluccia, C. 2009. Extending SAT Solvers to Cryptographic Problems. In: *SAT 2009*, Kullmann, O., ed., Lecture Notes in Computer Science, volume 5584, Springer-Verlag, pp. 244-257.
18. *Sponge construction*. http://sponge.noekeon.org.

Appendix

For the reader's convenience, we provide Verilog code used in the experiments with our toolkit CryptLogVer. In most cases such a code strongly resembles a pseudocode defining a given cryptographic function.

```
module keccak(MESSAGE, HASH);

   parameter laneSize = 64;
   parameter numberOfRounds = 24;
   parameter stateSize = 5*5*laneSize;
   parameter hashSize = 1024;
   input [stateSize-1:0] MESSAGE; // input message
   output [hashSize-1:0] HASH; // hash value
   reg [hashSize-1:0] HASH;

   reg [laneSize-1:0] A [0:4][0:4];
   reg [stateSize-1:0] AVector;
   reg [63:0] RoundConstants [0:23];
```

```
integer i,j,k,r;

function [stateSize-1:0] Round;
input [63:0] RC;
input [stateSize-1:0] AVector;
reg [laneSize-1:0] A [0:4][0:4];
reg [laneSize-1:0] B [0:4][0:4];
reg [laneSize-1:0] C [0:4];
reg [laneSize-1:0] D [0:4];
reg [laneSize-1:0] temp [0:4];
reg [laneSize-1:0] tempLane;
reg [laneSize-1:0] tempLaneRotated;
integer i,j,k,r,offset;
integer RotationOffsets [0:4][0:4];


begin

RotationOffsets[0][0]=0;
RotationOffsets[0][1]=36;
RotationOffsets[0][2]=3;
RotationOffsets[0][3]=41;
RotationOffsets[0][4]=18;
RotationOffsets[1][0]=1;
RotationOffsets[1][1]=44;
RotationOffsets[1][2]=10;
RotationOffsets[1][3]=45;
RotationOffsets[1][4]=2;
RotationOffsets[2][0]=62;
RotationOffsets[2][1]=6;
RotationOffsets[2][2]=43;
RotationOffsets[2][3]=15;
RotationOffsets[2][4]=61;
RotationOffsets[3][0]=28;
RotationOffsets[3][1]=55;
RotationOffsets[3][2]=25;
RotationOffsets[3][3]=21;
RotationOffsets[3][4]=56;
RotationOffsets[4][0]=27;
RotationOffsets[4][1]=20;
RotationOffsets[4][2]=39;
RotationOffsets[4][3]=8;
RotationOffsets[4][4]=14;

//change AVector to two-dimensional array
for (i=0; i<=4; i=i+1)
    begin
    for (j=0; j<=4; j=j+1)
        begin
        for (k=0; k<laneSize; k=k+1)
```

```
                begin
                A[i][j][k] = AVector[5*i*laneSize+j*laneSize+k];
                end
            end
        end

    // main part of Round function
    for (i=0; i<=4; i=i+1)
        begin
        C[i] = A[i][0] ^ A[i][1] ^ A[i][2] ^ A[i][3] ^ A[i][4];
        end

    for (i=0; i<=4; i=i+1)
        begin
        temp[i] = {C[(i+1)%5][laneSize-2:0], C[(i+1)%5][laneSize-1]}; // C cyclic leftrotate 1
        D[i] = C[(i-1+5)%5] ^ temp[i];
        end

    for (i=0; i<=4; i=i+1)
        begin
        for (j=0; j<=4; j=j+1)
            begin
            A[i][j] = A[i][j] ^ D[i];
            end
        end

    for (i=0; i<=4; i=i+1)
        begin
        for (j=0; j<=4; j=j+1)
            begin
            offset = RotationOffsets[i][j]%laneSize;
            tempLane = A[i][j];
            for (k=0; k<laneSize; k=k+1)
                 begin
                 tempLaneRotated[k]=tempLane[(k-offset+laneSize)%laneSize]; // cyclic left shift
                 end
            B[j][(2*i+3*j)%5] = tempLaneRotated;
            end
        end


    for (i=0; i<=4; i=i+1)
        begin
        for (j=0; j<=4; j=j+1)
             begin
             A[i][j] = B[i][j] ^ (~(B[(i+1)%5][j]) & B[(i+2)%5][j]);
             end
        end

    A[0][0] = A[0][0] ^ RC;
```

```
   // make A array back to vector format (because function can not return arrrays)
   for (i=0; i<=4; i=i+1)
       begin
       for (j=0; j<=4; j=j+1)
           begin
           for (k=0; k<laneSize; k=k+1)
                begin
                Round[5*i*laneSize+j*laneSize+k] = A[i][j][k];
                end
           end
       end

end
endfunction


always @ (MESSAGE, HASH, A)
begin

RoundConstants[0]=64'h0000000000000001;
RoundConstants[1]=64'h0000000000008082;
RoundConstants[2]=64'h800000000000808A;
RoundConstants[3]=64'h8000000080008000;
RoundConstants[4]=64'h000000000000808B;
RoundConstants[5]=64'h0000000080000001;
RoundConstants[6]=64'h8000000080008081;
RoundConstants[7]=64'h8000000000008009;
RoundConstants[8]=64'h000000000000008A;
RoundConstants[9]=64'h0000000000000088;
RoundConstants[10]=64'h0000000080008009;
RoundConstants[11]=64'h000000008000000A;
RoundConstants[12]=64'h000000008000808B;
RoundConstants[13]=64'h800000000000008B;
RoundConstants[14]=64'h8000000000008089;
RoundConstants[15]=64'h8000000000008003;
RoundConstants[16]=64'h8000000000008002;
RoundConstants[17]=64'h8000000000000080;
RoundConstants[18]=64'h000000000000800A;
RoundConstants[19]=64'h800000008000000A;
RoundConstants[20]=64'h8000000080008081;
RoundConstants[21]=64'h8000000000008080;
RoundConstants[22]=64'h0000000080000001;
RoundConstants[23]=64'h8000000080008008;




// main function Keccak-F
```

```
AVector = MESSAGE;

for (r=0; r<numberOfRounds; r=r+1)
    begin
    AVector = Round(RoundConstants[r],AVector);
    end

// change AVector format to array

for (i=0; i<=4; i=i+1)
     begin
     for (j=0; j<=4; j=j+1)
         begin
         for (k=0; k<laneSize; k=k+1)
             begin
             A[i][j][k] = AVector[5*i*laneSize+j*laneSize+k] ;
             end
         end
     end


// now produce bits of HASH

for (j=0; j<=4; j=j+1)
    begin
    for (i=0; i<=4; i=i+1)
        begin
        for (k=0; k<laneSize; k=k+1)
            begin
            if (i*laneSize+k+j*laneSize*5 < hashSize) HASH[i*laneSize+k+j*laneSize*5]=A[i][j][k];
            end
        end
    end

end
endmodule
```