DOI: https://doi.org/10.31341/jios.44.2.10 UDC 004.056.55
Open Access Original Scientific Paper

Complete SAT based Cryptanalysis of RC5 Cipher

Artur Soboń *artur.sobon@gmail.com*
Developers Division
HSBC Service Delivery (Polska) Sp. z o.o., Kraków, Poland

Mirosław Kurkowski *m.kurkowski@uksw.edu.pl*
Institute of Computer Science
Card. St. Wyszynski University, Warsaw, Poland

Sylwia Stachowiak *sstachowiak@swps.edu.pl*
Department of Computer Science
SWPS University of Social Sciences and Humanities, Warsaw, Poland

Abstract

Keeping the proper security level of ciphers used in communication networks is today a very important problem. Cryptanalysts ensure a constant need for improvement complexity and ciphers' security by trying to break them. Sometimes they do not instantly try to break the strongest version of the cipher, but they are looking for weaknesses by splitting it and independently checking all algorithm components. Often cryptanalysts also attempt to break cipher by using its weaker version or configuration. There are plenty of mechanisms and approaches to cryptanalysis to solve those challenges. One of them is SAT-based method, that uses logical encoding. In this article, we present our wide analysis and new experimental results of SAT-based, direct cryptanalysis of the RC5 cipher. To perform such actions on the given cipher, we initially create a propositional logical formula, that describes and represents the entire RC5 algorithm. The second step is to randomly generate key and plaintext. Then we determine the ciphertext. In the last step of our computations, we use SAT-solvers. They are particularly designed tools for checking the satisfiability of the Boolean formulas. In our research, we make cryptanalysis of RC5 cipher in the case with plaintext and ciphertext. To get the best result, we compared many SAT-solvers and choose several. Some of them were relatively old, but still very efficient and some were modern and popular.

Keywords: Symmetric ciphers, satisfiability, SAT-based cryptanalysis, RC5 Cipher

1. Introduction

Satisfiability problem is one of the NP-complete problems that allow investigations about the computational complexity of many algorithms. In this case, SAT can be successfully used for solving problems that can be encoded as the propositional Boolean formula [2]. The main problem in this area is that usually, the formulas

obtained have a very huge size, and in the whole case, solving satisfiability is very hard. However, there are many examples, where conducted experiments show that the satisfiability of formulas with hundreds or thousands of variables can be done quite quickly. Considering such problems, we can say that almost all of the implemented algorithms for SAT computations are modified and/or optimized versions of the DPLL algorithm [8, 9]. For several decades, for solving satisfiability special programs/tools, called SAT-solvers, have been developed and successfully used.

It is obvious that none of such solvers can check all considered, big formulas that sometimes consist of thousands or more variables. The reason is that in the worst-case searching of quite a big binary tree with depth over hundreds or thousands is needed. An interesting observation is that these tools can often answer the question about boolean satisfiability, even for big formulas [14, 15]. SAT-solvers takes as input the conjunctive normal form of the formula investigated (CNF). It is a conjunction of clauses, where a clause is a disjunction of literals, and a literal is a propositional variable or its negation.

SAT procedures among others are successfully used for cryptanalysis of some cryptographic algorithms or their parts/modifications, especially symmetric ciphers. This idea supplements other classic methods of cryptanalysis [5, 6, 13, 20, 21]. In the literature can be still found new papers devoted to this topic [7, 10, 11, 16, 17, 18, 19, 22, 24].

The studies described in this work relate to generating a propositional logic formula that encodes the whole RC5 algorithm included encryption and key expansion. Here we follow for some ideas introduced in [7, 10], where the efficiency of SAT-based cryptanalysis of the Feistel Network and some modifications of the DES cipher have been showed. This work is based on our previous paper [23], where we investigated a restricted version of RC5 cipher. Here we try to check our encoding methods and SAT solutions for the full, complete the RC5 cipher. We also checked and compared how several new SAT solvers work in this case.

That logic formula is converted into a conjunctive normal form (CNF). It is a conjunction of clauses, where a clause is a disjunction of literals, and a literal is a propositional variable or its negation. SAT-solvers usually takes as input this CNF form. The generated formula besides the implemented algorithm idea also has included bits evaluations describing the plaintext and the key. This resulting formula in the form of CNF can be tested by using the SAT-Solver. As a result, we should get an evaluation of all bits included the resulting evaluation of ciphertext bits.

Once we have ciphertext evaluation, we add it to our formula by removing key bits' valuations. It is good to remember about the update count of clauses in DIMACS header. Usually, it will be the number of lines in CNF file, without counting the header itself. Such modified formula can be now run in SAT-Solver. The cipher is now being broken. By that, we can understand an attempt to solve a complicated logic formula and find the right evaluation of key bits.

Obtaining a ciphertext should be a relatively quick operation regardless of the cipher's parameters, it should take maximally a few seconds. While breaking the cipher and recovering the secret key is a much longer operation depending on the

parameters and complexity of the cipher. It can take from several seconds to many days. There is also a possibility that valuations may not be found because the formula turns out to be too complicated to solve.

The rest of this paper is organized as follows. In Section 2, we present all needed, basic information on RC5 ciphers. We do this, to the extent necessary for explaining our ideas and boolean encoding. Section 3 contains information about cryptanalysis procedure and an algorithm of a direct, boolean encoding of the ciphers investigated. In Section 4, we present experimental results as we have obtained. At the end of the paper, we will give some conclusion and future directions connected to our research.

2. RC-5 cipher

Here we present, published in 1994, the RC5 cipher [13, 21]. It is an interesting example of ciphers because it can be parameterized. When designing the algorithm, Rivest set himself the following goals [13, 21].

- Implementation of the algorithm should be simple regard to hardware and software.
- The algorithm should be fast enough to give a result almost instantly. That is the reason why all operations are performed on a bits-blocks with a reference to the size of the word length in terms of hardware.
- Parameters used in the algorithm should be adjustable in order to have control over the relation between the algorithm's efficiency and its security level.
- The algorithm shouldn't be complicated so that the cryptographic strength of RC5 can be quickly determined.
- Memory and hardware requirements for algorithm should be undemanding to have possibility implement it on e.g. on smart cards or minicomputers.
- The security level provided by the algorithm should be strong enough.

Below are presented the main algorithm components.

2.1. Key Expansion

Key Expansion is the first stage before starting encryption or decryption. From the secret key, the array S is generated. It is filled with the generated subkey strings. The size of S depends on the number of rounds. Each subkey from S is used only once in encryption or decryption algorithm. Following the naming scheme of the original paper, the following variable names are used:

$w - The\ length\ of\ a\ word\ in\ bits.$
$Allowed\ values: 16, 32\ or\ 64.$
$u = w/8 - The\ length\ of\ a\ word\ in\ byte.$
$b - The\ length\ of\ the\ key\ in\ bytes.$
$Allowed\ range: 0 - 255\ bytes.$
$K[\,] - The\ key, considered\ as\ an\ array\ of\ bytes.$
$c = b/u - The\ length\ of\ the\ key\ in\ words\ (or\ 1, if\ b\ =\ 0).$

$L[\,] - A\ temporary\ array\ used\ during\ key\ scheduling.$
$Initialized\ to\ the\ key\ in\ words.$
$r - The\ number\ of\ rounds\ to\ use\ when\ encrypting\ data.$
$Allowed\ range: 0 - 255.$
$t = 2(r+1) - The\ number\ of\ subkeys\ required.$
$S[\,] - The\ round\ subkeys\ words.$
$P_w - The\ first\ magic\ constant.$
$Q_w - The\ second\ magic\ constant.$

P_w is defined as $Odd((e-2) * 2^w$, where Odd is the nearest odd integer to the given input, e is the base of the natural logarithm, and w is defined above. For allowed values of w, the associated values of P_w will be as follows:

$For\ w = 16: 0xB7E1$
$For\ w = 32: 0xB7E15163$
$For\ w = 64: 0xB7E151628AED2A6B$

Similarly Q_w is defined as $Odd((\phi-1) * 2^w$, where Odd is the nearest odd integer to the given input, ϕ is the golden ratio, and w is defined above. For allowed values of w, the values of Q_w will be as follows:

$For\ w = 16: 0x9E37$
$For\ w = 32: 0x9E3779B9$
$For\ w = 64: 0x9E3779B97F4A7C15$

2.2. Converting the Secret Key from Bytes to Words

The first algorithmic step of key expansion is to copy the secret key $K[0 \ldots b-1]$ into an array $L[0 \ldots c-1]$, where b and c is defined above. This operation is performed in a logical way, by taking consecutive key bytes of array K, to fill up each successive word in array L, low-order byte to high-order byte. Any unfilled byte positions of array L are zeroed.

On little-endian bytes order machines, the above operation can be done literally by zeroing the array L, and then copying the string K directly into the memory positions representing array L. The following pseudo-code obtains the same effect, assuming that all bytes are unsigned and that array L is initially zeroed:

$c = [max(b,1)/u];$
$\boldsymbol{for}\ i = b-1\ \boldsymbol{downto}\ 0\ \boldsymbol{do}$:
$L[i/u] = (L[i/u] \ll 8) + K[i];$

As we can see, as a result of a left bits shift by a fixed value of 8 (number of bits in a byte) and addition a consecutive key byte of array K we get filled the L array with bytes from the array K, starting from the low-order byte and finish with the high-order byte. Any empty L byte positions are zeroed. For a better understanding of the above algorithm, we can provide a few examples:

$RC5-16/1/3$
$Key = [AB, CD, EF]$
$L = [CDAB, 00EF]$

$RC5-32/1/6$
$Key = [AB, CD, EF, AA, BB, CC]$
$L = [AAEFCDAB, 0000CCBB]$

$RC5-16/1/16$
$Key = [AB, CD, EF, AA, BB, CC, DD, 11, 22, 33, 44, 55, 66, 77, 88, 99]$
$L = [CDAB, AAEF, CCBB, 11DD, 3322, 5544, 7766, 9988]$

Let's consider in detail one of them:

$RC5-16/1/3$
$Key = [AB, CD, EF] - bits\ allocated\ in\ memory:$
$\{33, 34, \dots, 56\},$
$bits\ \{57, 58, \dots, 64\}\ will\ be\ zeroed.$
$Expected\ L = [CDAB, 00EF] -\ bits\ allocated\ in\ memory:$
$\{65, 66, \dots, 96\}$

In this case, the key needs only 3 bytes (24 bits) to save it. However, the formula will be much simpler to generate if we allocated additional bits filled with 0 as a supplement of the key. The key size should be multiple of the word. We can present this logic by the following pseudo-code:

$c = [max(b, 1)/u];$
$keyBitPosition = 1;$
$arrayLBitPosition = keyBitPosition + (c * w);$
for $i = 1$ **to** c **do:**
$q = i * w;$
for $j = q$ **downto** $j - w$ **step** -8 **do:**
for $k = 8$ **downto** 0 **do**:
$L[\text{arrayLBitPosition }] = K[\text{keyBitPosition} + \text{j} - \text{k }];$
$arrayLBitPosition = arrayLBitPosition + 1;$

The conducted tests confirmed the correctness of the generated formula and the result was as expected. By development this algorithm for transforming the K key into the auxiliary array L, we have avoided a lot of operations related to the bits shift and addition operation.

2.3. Initialization of the key-independent, pseudo-random array S

The second algorithmic step of key expansion is to initialize array S to a particular fixed (key-independent) pseudo-random bit pattern. For this we will using the magic constants P_w and Q_w. First element of the array S is filled by P_w value. Then, each

next element is filled by result of addition variable Q_w and previous element of the array S. This logic can be presented in the form of the following pseudo-code:

$S[0] = P_w;$
$\boldsymbol{for}\ i = 1\ to\ t-1\ \boldsymbol{do}$
$\quad S[i] = S[i-1] + Q_w;$

Let's take an example:

$RC5-16/1/3$
$P_w = B7E1 - bits\ allocated\ in\ memory : \{97, 98, \dots, 112\}$
$Q_w = 9E37 - bits\ allocated\ in\ memory : \{113, 114, \dots, 128\}$
$Expected\ S = [B7E1, 5618, F44F, 9286] -\ bits\ allocated\ in\ memory:$
$[\{129, 130, \dots, 144\}, \{241, 242, \dots, 256\},$
$\{353, 354, \dots, 368\}, \{465, 466, \dots, 480\}]$

The conducted tests confirmed the correctness of the generated formula and the result was as expected. We can see that for a relatively small result array and just one simple mathematical operation (addition), we got a large formula. The reason for that is because to perform the addition in the binary system we need to additionally allocate 6 auxiliary variables, each of them with a word size - more information is described in the chapter about arithmetic operations used in the algorithm.

2.4. Mixing arrays S and L

Arrays mixing process consists in performing the operations of circular left shift and addition. For this algorithm, besides arrays S and L, also are used two auxiliary variables A and B. This logic can be presented in the form of the following pseudo-code:

$i = j = A = B = 0;$
$\boldsymbol{do}\ 3 * max(t, c)\ \boldsymbol{times}:$
$\quad A = S[i] = (S[i] + A + B) \lll 3;$
$\quad B = L[j] = (L[j] + A + B) \lll (A + B);$
$\quad i = (i + 1) mod(t);$
$\quad j = (j + 1) mod(c);$

The conducted tests confirmed the correctness of the generated formula and the result was as expected. We can see that we needed about 4,000 declared variables to perform all circular shift operations and addition. The reason of that is because to perform the addition in the binary system we need additional auxiliary variable, like mentioned before. However, the circular shift needs only a lot of clauses for each possible case, which enlarges the whole formula, but does not generate any auxiliary variables - more information is described in the chapter about arithmetic operations used in the algorithm.

The whole key extension algorithm is quite clearly called one-way and it is difficult to determine the K key on the basis of the received array S. This difficulty can be seen in times that SAT-solvers need to find the key.

3. Cipher's encoding and SAT based cryptanalysis

Cryptanalysis is a domain of knowledge and research dealing with methods of breaking ciphers containing possible violations of all assumed security features. The cipher is breakable if it is possible to obtain plaintext or the key based on ciphertext, or key based on plaintext and ciphertext. In our case, we try to find the key by possessing plaintext and ciphertext. For this purpose, we use the SAT-based method. This method is the propositional satisfiability problem, determining whether for a given logical formula is such evaluation of variables when the formula is satisfiable. To obtain that result we examine the formula by special tools named SAT-solvers.

3.1. Main idea

Follow a method presented in [7, 10, 23] we can transfer the whole algorithm into a propositional boolean formula. In the case of round-ciphers we encode each round of cipher and join these formulas into a one.

Let assume, that we deal with a cipher with 64-bit plaintext block and j rounds. Let denote by $(p_1, p_2, \dots, p_{64})$ plaintext bits and by $(c_1, c_2, \dots, c_{64})$ a ciphertext. Additionally let $(p_1^k, p_2^k, \dots, p_{64}^k)$ and $(c_1^k, c_2^k, \dots, c_{64}^k)$ input and output strings of k-th round. Observe that of course output of k round is an input of $k+1$ round, for $k = 1, \dots, j-1$.

Having such expressions, we can write the formula, that encode the whole j rounds of the cipher considered:

$$\varphi(p_1, p_2, \dots, p_{64}, c_1, c_2, \dots, c_{64}) \equiv$$
$$\equiv \bigwedge_{k=1}^{j} \varphi^k(p_1^k, p_2^k, \dots, p_{64}^k, c_1^k, c_2^k, \dots, c_{64}^k) \wedge \bigwedge_{k=1}^{j-1} \bigwedge_{s=1}^{64} (p_k^s \Leftrightarrow c_{k+1}^s).$$

In such formula, $\varphi(p_1, p_2, \dots, p_{64}, c_1, c_2, \dots, c_{64})$ denote the whole algorithm formula, dependent of the plaintext bits $p_1^k, p_2^k, \dots, p_{64}^k$ and a cipher ones $c_1, c_2, \dots, c_{64}$. $\varphi^k(p_1^k, p_2^k, \dots, p_{64}^k, c_1^k, c_2^k, \dots, c_{64}^k)$ is the formula that encodes k-th round (dependent of $(p_1^k, p_2^k, \dots, p_{64}^k)$, and $(c_1^k, c_2^k, \dots, c_{64}^k)$). As we said before the conjunction $p_k^s \Leftrightarrow c_{k+1}^s$ $(s = 1, \dots, 64)$ denote that each of output bit of k-th round is an input of $k+1$-th round. Using this we can encode each cipher's fragment as simple equivalences of bits dependences.

Here we show such equivalences that represent small parts of the whole algorithm.

3.2. Cryptanalysis procedure

For our research, we do the following work. Firstly having a given cipher, we built a propositional logical formula that encodes the whole RC5 algorithm. Next, we

randomly generate plaintext and key bits. Then, using SAT solvers, specially designed tools for checking about the satisfiability of the Boolean formulas, we compute ciphertext bits. At the end of our computations, we once more use SAT-solvers trying to compute key bits having only plaintext and ciphertext bits. That means that in our experiments, we explore RC5 properties in the case of cryptanalysis with plaintext and ciphertext. In our investigations, we use and compare several SAT-solvers: a few new ones and a few rather old but still efficient and popular.

The studies described in this work relate to generating a propositional logic formula that encodes the whole RC5 algorithm, including encryption and key expansion. That logic formula is converted into conjunctive normal form (CNF). It is a conjunction of clauses, where a clause is a disjunction of literals, and a literal is a propositional variable or its negation. SAT-solvers take this CNF form as an input data. They are specific build tools for checking the satisfiability of the Boolean formulas. To obtain reliable results, we compared many SAT-solve and choose a few. All of them were very efficient, some relatively old, some modern and popular.

The generated formula besides the implemented algorithm idea also has included bits evaluations describing the plaintext and the key. This resulting formula in the form of CNF can be run by using the SAT-Solver. As a result, we should get an evaluation of all bits included the resulting evaluation of ciphertext bits.

Once we have ciphertext evaluation, we add it to our formula by removing key evaluation. It is good to remember about update the count of clauses in DIMACS header. Usually, it will be the number of lines in CNF file, without counting the header itself. Such modified formula can be now run in SAT-Solver. The cipher is now being broken. By that, we can understand an attempt to solve a complicated logic formula and find the right evaluation of key bits.

Obtaining a ciphertext should be a relatively quick operation regardless of the cipher's parameters. It should take maximally a few seconds. While breaking the cipher and recovering the secret key is a much longer operation depending on the parameters and complexity of the cipher. It can take from several seconds to many days. There is also the possibility that evaluations may not be found when the formula turns out to be too complicated to solve.

4. Boolean encoding of RC5 Cipher

First, we present propositional logical formulas that encode the most popular in ciphers operations: a permutation and a bit addition.

4.1. Permutations and bit addition

Let consider permutation θ of a 64-bit vector $p_1, p_2, \dots, p_{64}$. As before we denote by $c_1, c_2, \dots, c_{64}$ bits obtained after a permutation considered. For such functions we have the following formula:

$$\Lambda_{l=1}^{64}(c_l \Leftrightarrow p_{\theta(l)}).$$

The second popular operation in RC5 cipher is a binary addition modulo 32. In the case of Boolean encoding of such division we can observe that binary addition follows the same principle as adding a decimal system except that instead of moving the remainder 1 when the added values are $(10)_2$, we move it when the result of the addition is $(2)_{10}$. To better illustrate this, we give an example below:

$1 + 0 = 0,$
$0 + 1 = 1,$
$1 + 0 = 1,$
$1 + 1 = 0, carry\ over\ the\ 1.$

For the purpose of further discussion of the algorithms and notation of addition, let's assume that:

$a - a\ first\ bit,$
$b - a\ second\ bit,$
$p - a\ result\ of\ addition\ operation,$
$q, r, s, t, u - auxiliary\ variables.$

$\oplus - XOR\ operation,$
$\wedge - conjunction,$
$\vee - disjunction.$

We can denote the formula for an addition operation as follows:

$p \Leftrightarrow (a \oplus b) \oplus c,$
$p \Leftrightarrow (a \wedge b) \vee (a \wedge c) \vee (b \wedge c).$

Note that, using well-known logical laws, we can convert these formulas to conjunction with the auxiliary factors in the following way:

$q \Leftrightarrow (a \oplus b),$
$p \Leftrightarrow (q \oplus c),$
$r \Leftrightarrow (a \wedge b),$
$s \Leftrightarrow (a \wedge c),$
$t \Leftrightarrow (b \wedge c),$
$u \Leftrightarrow (r \vee s),$
$c \Leftrightarrow (u \vee t).$

4.2. Circular bit shift

Looking at the circular bit shift from the decimal number system point of view, it might seem complicated and difficult to encode. However, if we consider it from the binary system point of view, it looks simplest and logical. The formula proposal comes to create an appropriate condition from the bits describing the shift value. This condition will be implied by assigning the old bits to the new ones considering the shift value.

As an example, assume the right circular shift and:

$p = [1,1,0,0] = (1100)_2 = (12)_{10} - a\ number\ to\ shift$
$q = [0,1] = (01)_2 = (1)_{10} - a\ shift\ value$
$s = [0,1,1,0] = (0110)_2 = (6)_{10} - a\ result$

We can construct the encoding formula in this case as follows:

$\neg q_1 \wedge \neg q_2 \Rightarrow (p_0 \Leftrightarrow s_0) \wedge (p_1 \Leftrightarrow s_1) \wedge (p_2 \Leftrightarrow s_2) \wedge (p_3 \Leftrightarrow s_3)$,
$\neg q_1 \wedge q_2 \Rightarrow (p_0 \Leftrightarrow s_1) \wedge (p_1 \Leftrightarrow s_2) \wedge (p_2 \Leftrightarrow s_3) \wedge (p_3 \Leftrightarrow s_0)$,
$q_1 \wedge \neg q_2 \Rightarrow (p_0 \Leftrightarrow s_2) \wedge (p_1 \Leftrightarrow s_3) \wedge (p_2 \Leftrightarrow s_0) \wedge (p_3 \Leftrightarrow s_1)$,
$q_1 \wedge q_2 \Rightarrow (p_0 \Leftrightarrow s_3) \wedge (p_1 \Leftrightarrow s_0) \wedge (p_2 \Leftrightarrow s_1) \wedge (p_3 \Leftrightarrow s_2)$.

Note that we can convert these formulas to conjunction in the following way:

$\neg q_1 \wedge \neg q_2 \Rightarrow (p_0 \Leftrightarrow s_0)$,
$\neg q_1 \wedge \neg q_2 \Rightarrow (p_1 \Leftrightarrow s_1)$,
$\neg q_1 \wedge \neg q_2 \Rightarrow (p_2 \Leftrightarrow s_2)$,
$\neg q_1 \wedge \neg q_2 \Rightarrow (p_3 \Leftrightarrow s_3)$,

$\neg q_1 \wedge q_2 \Rightarrow (p_0 \Leftrightarrow s_1)$,
$\neg q_1 \wedge q_2 \Rightarrow (p_1 \Leftrightarrow s_2)$,
…….
$q_1 \wedge q_2 \Rightarrow (p_2 \Leftrightarrow s_1)$,
$q_1 \wedge q_2 \Rightarrow (p_3 \Leftrightarrow s_2)$.

4.3. Encryption

The basic assumption in RC5 Cipher is the operation on words with a length of 16, 32 or 64 bits. The plaintext is given as two blocks A and B, where each of them has a word length. Let's assume, that the array S has been already created. The idea of encryption can be presented in the form of the following pseudo-code:

$A = A + S[0]$
$B = B + S[1]$;
$\boldsymbol{for}\ i = 1\ \boldsymbol{to}\ r\ \boldsymbol{do}$:
  $A = \big((A \oplus B) \lll B\big) + S[2 * i]$;
  $A = \big((B \oplus A) \lll A\big) + S[2 * i + 1]$;

The conducted tests confirmed the correctness of the generated formula and the result was as expected. We can notice that to perform all circular shift and addition operations, we needed only about 1500 declared variables to obtain the ciphertext. This is just around 37% of that what we needed for mixing array S. The reason for this is that in encryption we have just 2 addition operations, while for mixing array S there are 5 addition operations. The rest operations like XOR or circular shift don't need so many auxiliary variables like addition.

5. SAT-solvers and experimental results

In this section we present and compare features of SAT-solvers used for testing. We also show our experimental results.

5.1. SAT-solvers

For our investigation we use some new and a few rather old but still very efficient and popular SAT-solvers:

1. MiniSAT – ver. 2.2,
2. PrecoSat – ver. 576-7e5e66f-120112,
3. PicoSAT – ver. 960,
4. Glucose – ver. 4.1,
5. RSat 2.01 – ver. 2.01,
6. Rsat Race08 – ver. 3.01,
7. COMiniSATPS Pulsar,
8. Lingeling – ver. bcj-78ebb86-180517,
9. Glu_VC,
10. Maple LCM,
11. CaDiCaL – ver. 1.0.3-cb89cbf,
12. CaDiCaL Agile – ver. sc17,
13. Syrup – ver. 4.1,
14. Plingeling – ver. bbe-sc2017,
15. Painless MapleCOMSPS.

All SAT-solvers take the first parameter as the input CNF file name. For most solvers, it is sufficient to run and see the results. Almost all display the running time by default. Some of them need to use additional parameters to display results. All differences were described below.

PrecoSat and **PicoSAT** do not display information about the running time. Therefore, for the purposes of check and verify this time we recommend using the "time" command when running those SAT-solvers. This command displays process running time when process is finished [1,3].

RSat 2.01 and **RsatRace08** do not display results by default. You must provide the run "-s" option to see the outcome.

MiniSAT, COMiniSATPS Pulsar, Glucose, Glu_VC and **Syrup** also do not present results by default. You can provide a second run parameter (an output file name) to save the outcome into a file. For **Glucose, Glu_VC** and **Syrup** in order to display results instead of saving, you can use the run "-model" option [2, 12].

Syrup, Plingeling and **Painless MapleCOMSPS** are parallel SAT-solvers. Can work multithreaded and options for that are fully configured. However, for the purposes of our research we didn't have to change it, and we stayed with the default settings [4].

5.2. Experimental results

Below are presented tables with results of times individual problems for different SAT-solvers and information about files size, count of clauses and variables for those problems.

The following notation has been used:

$$RC5 - w/r/b$$

where,

$w - The\ length\ of\ a\ word\ in\ bits.$
$Allowed\ values: 16, 32\ or\ 64.$
$r - The\ number\ of\ round\ to\ use\ when\ encrypting\ data.$
$Allowed\ range: 0 - 255.$
$b - The\ length\ of\ the\ key\ in\ bytes.$
$Allowed\ range: 0 - 255\ bytes.$

Also, you can find notation like $RC5 - w/r/b - n$, where n is the number of evaluated bits added to CNF formula. Let's take an example: $RC5 - 16/1/5 - 16$.

It means, that for 40 bits (5 bytes) key we have added 16 evaluated bits and try to find only 24 left bits (3 bytes).

An interesting observation is the fact, that the odd and even length of the key has the same number of variables and clauses. The reason of this is that the generated formula filling the "missing" even part of the key with zeros. That part of logic was described in detail in the section of converting the secret key from bytes to words. However, breaking the cipher for the 4-bytes key will be much harder than for the 3-bytes key. Regardless of the fact that we have the same number of variables and clauses in generated formula.

Version	File size [MB]	Variables	Clauses
RC5-16/1/3	0,56	6784	28034
RC5-16/6/3	2,40	23744	101362
RC5-16/12/3	4,60	44096	189346
RC5-32/1/3	1,90	13504	70418
RC5-32/6/3	7,80	47424	258338
RC5-32/9/3	11,40	67776	371066

Table 1. Information about variables and clauses for 3-bytes key.

Assumptions about fast obtain the ciphertext bits evaluations from SAT-solver, were confirmed during the tests. For each case, regardless of the cipher parameters or the size of the file contains CNF formula, it took literally fractions of a second. Long time needed for obtain the key bits evaluations while breaking the cipher have also been confirmed and presented in previous chapter.

Studies have shown that in each case the generated formula returned the expected ciphertext evaluation. However, during the stage of finding the key evaluation, the results were variety.

Version	File size [MB]	Variables	Clauses
RC5-16/1/5-16	0,56	6816	28090
RC5-16/1/5-12	0,56	6816	28086
RC5-16/1/5-8	0,56	6816	28082
RC5-16/1/4	0,56	6784	28034
RC5-16/1/5	0,56	6816	28074
RC5-32/1/4	1,90	13504	70450

Table 2. Information about variables and clauses for more than 3-bytes and custom key.

The initial assumptions were to examination RC5 Cipher with following parameters: word size: 16 and 32 bits, rounds number: 1, 3, 6, 9 and 12, and a key size: 1, 2, 3, 4 and 5 bytes.

Version	Time [s] MiniSAT	Time [s] PrecoSat	Time [s] PicoSat	Time [s] Glucose	Time [s] Rsat_2.01
RC5-16/1/2	1,37	20,99	26,25	9,52	11,76
RC5-16/6/2	37,60	98,52	103,52	10,85	100,49
RC5-16/12/2	91,25	85,40	474,02	130,50	386,99
RC5-32/1/2	31,85	47,07	40,90	33,19	15,49
RC5-32/6/2	99,36	409,09	11,52	52,36	968,60
RC5-32/12/2	467,57	482,69	23,87	329,68	167,32
RC5-32/24/2	101,78	235,61	49,44	376,89	961,74
RC5-32/48/2	1216,27	426,75	2946,13	1876,09	4793,57
Version	**Time [s] Rsat_Race**	**Time [s] COMiniSAT**	**Time [s] Lingeling**	**Time [s] Glu_VC**	**Time [s] Maple**
RC5-16/1/2	35,96	23,12	24,90	5,04	20,51
RC5-16/6/2	160,42	97,02	139,18	36,20	Error
RC5-16/12/2	177,67	40,95	350,4	67,32	Error
RC5-32/1/2	31,48	48,42	7,69	5,45	5,68
RC5-32/6/2	192,04	35,45	145,20	214,15	Error
RC5-32/12/2	832,03	140,77	1156,63	172,35	Error
RC5-32/24/2	1518,74	197,45	369,05	822,02	Error
RC5-32/48/2	3147,42	1779,48	4340,35	2206,03	Error
Version	**Time [s] CaDiCal**	**Time [s] CDC_Agile**	**Time [s] Syrup***	**Time [s] PlingeL***	**Time [s] Painless***
RC5-16/1/2	9,51	9,51	0,86	9,80	71,62
RC5-16/6/2	79,4	60,04	29,08	12,25	42,52
RC5-16/12/2	184,19	184,19	94,83	120,38	227,02
RC5-32/1/2	24,99	22,83	9,27	4,95	18,92
RC5-32/6/2	323,70	297,65	165,98	5,08	137,78
RC5-32/12/2	54,42	375,54	174,25	15,64	124,71
RC5-32/24/2	121,67	797,23	587,35	360,56	Error
RC5-32/48/2	476,96	2639,22	97,88	643,28	Error

Table 3. Results for 2-bytes key.

After started tests there was no problem to break the 1-byte key. It took literally fractions of a second. Regardless of the number of rounds as well. While raising parameters could be noticed a significant increase for the time needed to break the 2-bytes key. The same situation was with bigger keys, like 3, 4 and 5-bytes. At some point we came to a stage where further waiting for results became too much time-consuming, given the limited hardware resources.

To be able to compare more results and better understanding cryptanalysis of the RC5 cipher we abandoned calculations, that were persisted long time. Some of the tests were interrupted and emphasis like red numbers with the grave accent as a prefix. This number is the time after which the process was interrupted. After got considerable count of results we were able to see some pattern of which SAT-solvers are more credible and which are not. That was the main reason that some of tests was not performed.

It turns out that the 2-bytes key is not difficult to break down even for a large number of rounds. This was confirmed by additional tests with a significantly larger number of rounds than originally assumed. The sizes of the CNF files, the number of variables and clauses for $RC5-32/24/2$ and $RC5-32/48/2$ case look impressive. SAT-solvers working times look fine and are not surprising. The exception is the result of the Syrup program for 48 rounds case. Which found the key quickly, faster than all other solvers and faster than itself for 24 rounds case.

Version	**Time [s] MiniSAT**	**Time [s] PrecoSat**	**Time [s] PicoSat**	**Time [s] Glucose**	**Time [s] Rsat_2.01**
RC5-16/1/3	78,36	3846,00	9996,72	`67540,00	`26040,00
RC5-16/3/3	11342,50	N/A	N/A	N/A	N/A
RC5-16/6/3	`30137,00	N/A	N/A	N/A	N/A
RC5-16/12/3	46800,00	N/A	N/A	N/A	N/A
RC5-32/1/3	874,57	7640,59	N/A	N/A	N/A
Version	**Time [s] Rsat_Race**	**Time [s] COMiniSAT**	**Time [s] Lingeling**	**Time [s] Glu_VC**	**Time [s] Maple**
RC5-16/1/3	N/A	`44658,00	N/A	609,05	N/A
RC5-32/1/3	N/A	N/A	N/A	294,60	Error
Version	**Time [s] CaDiCal**	**Time [s] CDC_Agile**	**Time [s] Syrup***	**Time [s] PlingeL***	**Time [s] Painless***
RC5-16/1/3	7387,42	4465,68	44,74	4811,48	Error
RC5-32/1/3	N/A	`2400,00	1063,90	7309,10	N/A

Table 4. Results for 3-bytes key.

As you can see in the results for the 3-bytes key, obtain results for just 1 round was for most SAT-solvers pretty straightforward and usually fast. However, for 3 and more rounds efficiency has fallen dramatically, from seconds to hours. Some of the calculations worked enough long to be interrupted, so there was no point in to run tests for other rounds in order to get more data for analyse.

Version	Time [s] MiniSAT	Time [s] PrecoSat	Time [s] PicoSat	Time [s] Glucose	Time [s] Rsat_2.01
RC5-16/1/5-16	842,59	N/A	N/A	N/A	N/A
RC5-16/1/5-12	7202,57	N/A	N/A	N/A	N/A
RC5-16/1/5-8	139883,44	N/A	N/A	N/A	N/A
RC5-16/1/4	29073,90	20693,90	N/A	N/A	N/A
RC5-16/1/5	`579600,00	`579600,00	N/A	N/A	N/A
RC5-32/1/4	`550800,00	N/A	N/A	N/A	N/A
Version	**Time [s] Rsat_Race**	**Time [s] COMiniSAT**	**Time [s] Lingeling**	**Time [s] Glu_VC**	**Time [s] Maple**
RC5-16/1/5-16	N/A	58,27	N/A	`19583,00	N/A
RC5-16/1/5-12	N/A	849711,7	N/A	N/A	N/A
Version	**Time [s] CaDiCal**	**Time [s] CDC_Agile**	**Time [s] Syrup***	**Time [s] PlingeL***	**Time [s] Painless***
RC5-16/1/5-16	153,20	3164,13	18,50	1683,91	Error
RC5-16/1/5-12	29412,92	2925,78	`63273,00	N/A	N/A
RC5-16/1/5-8	N/A	N/A	770599,64	N/A	N/A
RC5-16/1/4	N/A	N/A	`54764,00	N/A	N/A

Table 5. Results for more than 3-bytes and custom key.

Very unusual results have obtained from $RC5-16/1/4$ case. MiniSAT after around 8 hours of work has returned results, but key bits evaluations were not like expected. Similar situation was with PrecoSat. After around 5 hours of work has returned different results than MiniSAT and key bits evaluations were not like expected as well. However, after testing both different key bits evaluations substituted for the formula, we have got the same and like expected results of ciphertext bits evaluations.

Analogous situation was captured for $RC5-16/1/5-8$ case, there was also searched the 4-bytes key, like in $RC5-16/1/4$ example. MiniSAT after around 38 hours of work has returned results, but key bits evaluations were not like expected. We have met the same situation as above after checking these results. However, Syrup after around 214 hours of work returned expected key bits evaluations.

The conclusion that come to mind is that the RC5 cipher may in some extreme cases return the same ciphertext for a different key, which confirms the above studies. This case will be investigated more precisely in future.

Legend:

N/A – Not applicable.
Error – Segmentation Fault / Error / Killed.
`time – Killed after the given time – no results.
* – Parallel SAT-solvers.

To avoid glitches, all SAT-solvers was running sequentially. Heavier loaded the processor cores by simultaneously running SAT-solvers, caused longer problem

solving. Below you can see difference in times, when **MiniSAT** try to solve the problem and was run simultaneously for $RC5-16/12/2$ case:

Running MiniSAT simultaneously	Loaded CPU cores	Time [s]
4	4	172,00
3	3	148,00
2	2	127,00
1	1	91,00

Table 6. Difference in times while running SAT-solver simultaneously.

Like we can see the difference between one and four loaded cores is almost twice.

All experiments have been done on the same 4-cores machine. Below are presented information about hardware and software of the used machine: Architecture: x86_64 system Standard PC (i440FX + PIIX, 1996), CPU op-mode(s): 32-bit, 64-bit bus Motherboard, Byte Order: Little Endian memory 96KiB BIOS, Address sizes: 40 bits physical, 48 bits virtual processor Common KVM processor, CPU(s): 4 memory 6GiB System Memory, On-line CPU(s) list: 0-3 memory 6GiB DIMM RAM, Thread(s) per core: 1 bridge 440FX - 82441FX PMC. *Linux 4.19.0-5-amd64 #1 SMP Debian 4.19.37-5 (2019-06-19) x86_64 GNU/Linux*

6. Conclusion

In this paper, we have presented our experimental results for SAT-based, direct cryptanalysis of the RC5 cipher. We have shown and compared results obtained from several efficient SAT-solvers. We have checked how the solvers work in the case of cryptanalysis of the RC5 cipher.

The results of individual SAT-solvers are not clearly deterministic. While a particular SAT-solver seems to be faster than the others, then in some cases was much slower. The conclusions are that heuristics and algorithms implemented in SAT-solvers are extremely diverse and very sensitive according to the complexity and type of the problem. However, is worth highlighting old but powerful MiniSat and multithread Syrup. Both of them return results quite fast in most cases.

During our research, we managed to get the results for 16 and 32-bits word size with 1, 2, 3 and 4-bytes key and for the range of rounds amount mainly from 1 to 12. Just for 2-bytes key maximally a number of rounds were 48. It seems to be achievable to break stronger versions of the RC5 cipher. It can be done by using parallel or cloud computing with a big reserve of resources.

In our next work, we will try to apply our experience for the SAT cryptanalysis of several other ciphers like Blowfish, Twofish, AES, or hash functions.

References

[1] A. Biere, *PicoSAT Essentials*. Journal on Satisfiability, Boolean Modeling and Computation (JSAT), vol. 4, pp. 75 – 97, Delft University,2008.

[2] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, vol. 185 of Frontiers in Artificial Intelligence and Applications. IOS Press, 2009.

[3] A. Biere, *Lingeling, Plingeling, Picosat and Precosat at SAT Race 2010*.Technical Report FMV Reports Series 10/1, Institute for Formal Modelsand Verification, Johannes Kepler University, Linz, Austria, 2010.

[4] A. Biere, *Lingeling, Plingeling and Treengeling Entering the SAT Competition 2013*. In Proceedings of SAT Competition 2013, A. Balint, A. Belov, M. Heule, M. Jarvisalo (editors), vol. B-2013-1 of Department of Computer Science Series of Publications B, pages 51-52, University of Helsinki, 2013.

[5] E. Biham and A. Shamir, *Differential cryptanalysis of DES-like cryptosystems*. J. Cryptology, 4(1):3–72, 1991.

[6] N. Courtois and G. V. Bard, *Algebraic cryptanalysis of the Data Encryption Standard*. In S.D. Galbraith, editor, IMA Int. Conf., volume4887 of Lecture Notes in Computer Science, pages 152–169. Springer, 2007.

[7] M. Chowaniec, M. Kurkowski, and M. Mazur, *New Results in Direct SAT-Based Cryptanalysis of DES-Like Ciphers*. In Proc. of ACS'18. AISC, vol. 889, pp. 282-294, Springer, Cham.

[8] M. Davis and H. Putnam, *A computing procedure for quantification theory*. J. ACM, 7(3):201–215, 1960.

[9] M. Davis, G. Logemann, and D. W. Loveland, *A machine program for theorem-proving*. Commun. ACM, 5(7):394–397, 1962.

[10] P. Dudek, M. Kurkowski, and M. Srebrny, *Towards Parallel Direct SAT-based Cryptanalysis*, in PPAM'11 Proceedings, pp. 266-275, vol. 7203of LNCS, Springer Verlag, 2012.

[11] A. D. Dwivedi, et al., *SAT-based Cryptanalysis of Authenticated Ciphers from the CAESAR Competition*, in Proc. of the 14th International Joint Conference on e-Business and Telecommunications (ICETE 2017) - vol.4: SECRYPT, pp. 237 – 246, 2017.

[12] https://github.com/arminbiere/cadical (accessed on 15th of Jan. 2020).

[13] Cz. Kościelny, M. Kurkowski, M. Srebrny, *Modern Cryptography Primer*, Springer Verlag, 2013.

[14] M. Kurkowski, W. Penczek, *Applying timed automata to model checking of security protocols*, in ed. J. Wang, Handbook of Finite State Based Models and Applications, pp. 223-254, Chapman and Hall/CRC Press, 2013.

[15] M. Kurkowski, W. Penczek, *Verifying Timed Security Protocols via Translation to Timed Automata*, Fundamenta Informaticae, vol. 93 (1-3), pp. 245-259, IOS Press, 2009.

[16] F. Lafitte, L. Lerman, O. Markowitch, and D. van Heule, *SAT-based cryptanalysis of ACORN*, IACR Cryptology ePrint Archive, pp. 521, vol. 2016, 2016.

[17] F. Lafitte, et.al., *Applications of SAT Solversin Cryptanalysis: Finding Weak Keys and Preimages*, JSAT, vol. 9, pp.1–25, 2014.

[18] F. Massacci, *Using Walk-SAT and Rel-SAT for cryptographic key search*. In T. Dean, editor, IJCAI, pages 290–295. Morgan Kaufmann, 1999.

[19] F. Massacci, L. Marraro, *Logical Cryptanalysis as a SAT Problem*, Journal of Automated Reasoning, pp. 165 – 203, 24: 165, 2000.

[20] M. Matsui, *The first experimental cryptanalysis of the Data Encryption Standard*. In Y. Desmedt, editor, CRYPTO, volume 839 of Lecture Notes in Computer Science, pages 1–11. Springer, 1994.

[21] A. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1996.

[22] P. Morawiecki, M. Srebrny, *A SAT-based preimage analysis of reduced Keccak hash functions*. Inf. Process. Lett. 113(10-11): 392-397, 2013.

[23] A. Soboń, M. Kurkowski, S. Stachowiak, *Towards Complete SAT-based Cryptanalysis of RC5 Cipher*, In Proc. of 2019 IEEE 15th International Scientific, Conference on Informatics, pp. 369-374, IEEE Press, 2019.

[24] M. Soos, K. Nohl, and C. Castelluccia, *Extending SAT Solvers to Cryptographic Problems*, Theory and Applications of Satisfiability Testing SAT 2009, In Proc. of 12th Int. Conf., SAT 2009, pp. 244 – 257, 2009.