Converting Symmetric Cryptography to SAT Problems Using Model Checking Tools

Pedro Carlos da S. Lara[1], **Felipe da R. Henriques**[1], **Fábio B. de Oliveira**[2]

[1]Centro Federal de Educação Tecnológica Celso Suckow da Fonseca – CEFET/RJ

[2]Laboratório Nacional de Computação Científica – LNCC

{pedro.lara,felipe.henriques}@cefet-rj.br, borges@lncc.br

Abstract. *Algebraic attack is a recent technique used to break symmetric cryptography including Crypto-1, which is used in smart NFC cards around the world. The main method is to convert a cryptography algorithm to a system of equations over a finite field, to translate it into a SAT problem and to use a good solver. The main advantage of converting cryptography problems into SAT is that we can use a lot of well know tools for SAT solving and test it in an instance. However, the high degree labor intensiveness on converting cryptography to SAT instances is a difficult task for advancement of research. To pursue this problem, this work presents a technique to quickly translate symmetric cryptography into SAT problems using Bounded Model Checking (BMC) tools. Computational experiments are performed, converting the RC4 and AES algorithms into a SAT problem.*

1. Introduction

In general, algebraic attacks aim to convert the problem of breaking a cryptographic algorithm into an algebraic system of equations, usually Boolean, and try to solve it through some known method [Courtois and Pieprzyk 2002, Courtois and Meier 2003]. There is a controversy in this type of attack, since solving these equations, in general, are also considered a hard problem. However, in this kind of attack, it is possible to see the cryptographic algorithm in the most primitive way possible: through Boolean functions. Thus, it may be possible to analyse and to propose design criteria for the development of new cryptographic algorithms and trapdoors [Courtois 2005]. Analyzing an algorithm in such a primitive way allows the identification and the avoidance of algebraic relations that are not so interesting for a cryptographic algorithm. In addition, in some cases, the only possible attack is an algebraic attack.

Among the algebraic formats usually used, the satisfaction problem (SAT) format has an important highlight, since SAT solvers have been successfully used in several applications from academic to industry [Eén and Sörensson 2003]. For example, SAT solvers can be used in professional sports league schedule where a lot of complex constraints, established by teams, television, stadiums, and others, are needed [Horbach et al. 2012]. In addition, competitions and conferences have been proposed primarily to accelerate progress in SAT solvers research.

Usually, authors convert algebraic formats into the Conjunctive Normal Form (CNF) representation through other Boolean representations. For example, on can convert Boolean functions into the Algebraic Normal Form (ANF) and then, finally, the ANF is

converted into the CNF representation (see [Jovanovic 2010]). These algebraic attacks are also extended to asymmetric algorithms. Patarin [Patarin 1995] was the first researcher that developed an algebraic attack against the asymmetric algorithm based on multivariate quadratic (MQ) equations. Doing so, it is also possible to analyze any asymmetric algorithm on the optics of an attack using SAT solvers. Since this important work, other attacks have been proposed against algorithms based on MQ [Mohamed et al. 2009].

This work differs from the previous works found in the literature by using model checking software to achieve a CNF representation from a `C` program. The advantage of our proposal is that we can achieve a very wide variety of algorithms in a generic way. In this case, we can include cryptographic hash, symmetric (block or stream), and asymmetric algorithms. Basically, we just need to know the source code of the algorithm and introduce the `asserts` correctly. An important disadvantage of this approach is that we need to "clean up" the final CNF, since the model checking software generates Boolean equations that are not interesting for the attack but are interesting in the source code. Throughout this paper, we can show how to deal with the problem of removing these unwanted equations.

In this work, we will focus on converting this problem into the CNF format, and we will provide all the details for the conversion of a cryptographic algorithm into a SAT problem. We will also show the results for the conversion of the AES block algorithm and the RC4 stream cipher.

This paper is structured as follows. Section 2 presents the fundamentals of the bounded model checking techniques. In Section 3, the conversion of symmetric cryptography into a CNF SAT instance is introduced. Section 4 presents the method considered in the conversion of symmetric cryptography algorithms to a SAT problem. Computational results are discussed in Section 5. Concluding remarks are presented in Section 6.

2. Bounded Model Checking

Model Checking techniques [Clarke and Emerson 1982] aim to verify finite state transition systems. Such approach models the design to be verified as a finite state machine, and the specifications of the referred system are formalized by temporal logic properties. Model checking tools analyze some behaviors of the system to be verified, searching for problems in the logic properties. When a given property fails, a counterexample is generated as a sequence of states. Doing so, this technique intends to verify whether the specification of a system is correct.

Symbolic Model Checking [Burch et al. 1991] treats sets of states as Boolean functions. These functions can be efficiently manipulated by Binary Decision Diagrams (BDDs) [Bryant 1986]. However, memory is an important bottleneck on these methods, because the number of Boolean functions required to represent all the possible states that define a given system.

In this work, we consider a class of model checkers that deals with this bottleneck, the Bounded Model Checking (BMC) [Biere et al. 2003] [Given-Wilson et al. 2017]. The fundamental idea of BMC is to limit the search for a counterexample, doing this procedure in executions whose length is bounded by an integer k [Biere et al. 2003]. If the checker does not find any problem, k is incremented until it reaches a superior bound, known as *Completeness Threshold*.

The BMC is a SAT–based model checking, instead of using BDDs. According to [Biere et al. 2003], if k is small enough (up to 80 cycles), BMC outperforms the BDD–based techniques. In this work, we explore this idea to convert symmetric cryptography and hash functions to SAT problems, reducing complex algorithms into basic Boolean functions. The BMC is used to check for problems in these functions, searching for bugs, and then, improving reliability to the analyzed algorithms.

3. Preparing Attack Against Symmetric Cryptography

In this section, we introduce the fundamental ideas to convert symmetric cryptography into a CNF (Conjunctive Normal Form) SAT instance. We conduct the problem into know plaintext attack in block and stream cipher. We use a SAT instance based on a counterexample generated by the CBMC (Bounded Model Checker for C and C++) software by using `asserts`. We show how to use `asserts` in the CBMC software to translate it into a breaking cipher problem [Kroening and Tautschnig 2014]. Finally, we also discuss some variation of concepts hereafter.

Let $E_k(m) = c$ be a symmetric algorithm, where k is a key, m is the message, and c is the ciphertext. Let $E_k^{-1}(c) = m$ be the decryption algorithm. Given some pairs (m_i, c_i) of plain and ciphertext, we can attack the algorithm E creating a SAT instance such that the solution is associated to the key. The CBMC software searches for a counterexample based on `asserts` declared in the `C` code. That is, the CBMC will generate a SAT so that the solution is a counterexample to the statement given in the assertions. This practice is very common in software verification. Then, we can create a `C` code, such as the example depicted in Figure 1. In this case, we have a plaintext `m`, a ciphertext `c`, and a key `k`. One can see in Figure 1, that we have not initialized the key.

```
int main() {
    unsigned char  k;
    unsigned char  m   = 0x23;
    unsigned char  c   = 0xF6;
    assert( (k ^ m) != c );
    return 0;
}
```

Figure 1. Example of a simple C program.

The asserts in this work are placed in such a way that the CBMC, when outputting a counterexample, can find the key. Thus, in Figure 1, one can notice that we created an assert with the following logical expression

$$m \oplus k \neq c. \tag{1}$$

From this logical expression, the CBMC will assembly a SAT in CNF format whose solution (if any) satisfies

$$m \oplus k = c. \tag{2}$$

In this case, we can extract the bits of the key `k` that was uninitialized in source code. For the program (example) declared in Figure 1, CBMC generated the SAT instance

presented in Figure 2. Figure 2 shows the SAT instance in CNF (Conjunctive Normal Form) format, also known as DIMACS format. In this format, each integer (except zero) represents a literal. If this value is negative, it will represent the respective literal preceded by a Boolean NOT. The clauses are joined by Boolean ANDs, and in each clause, literals are joined by the Boolean ORs. In this format, zero means the end of the clause. The first three lines of Figure 2 represent the following Boolean expression:

$$l_1 \wedge (l_2 \vee \neg l_{26}) \wedge (\neg l_3 \vee \neg l_{26}) \tag{3}$$

We modified the CBMC to display which literals correspond to each variable in the C program. Then, the 8 bits of the key, in this example, are represented by the literals $2, 3, \ldots, 9$. The remaining variables, for example 26, are generated internally by the CBMC. In our example, these surplus variables have a total dependency on the variables that contain the key. It is worth saying that we are only interested in this set of variables.

```
p cnf 34 24
1 0
2 -26 0
-3 -26 0
4 -26 0
-5 -26 0
6 -26 0
-7 -26 0
8 -26 0
9 -26 0
-2 3 -4 5 -6 7 -8 -9 26 0
26 0
-27 -29 -31 0
27 29 -31 0
-27 29 31 0
27 -29 31 0
-28 -30 -32 0
28 30 -32 0
-28 30 32 0
28 -30 32 0
-31 -33 0
-32 -33 0
31 32 33 0
-1 33 0
1 -33 0
```

Figure 2. CNF file generated from program declared in Figure 1.

In Figure 3, the vertices represent variables in the CNF instance showed in Figure 2, and the edges represent dependency in the Boolean expression. There is an edge between two variables if they appear on the same line. This graph is known in the literature as *primal graph* [Samer and Szeider 2010]. The line below

```
28 30 -32 0
```

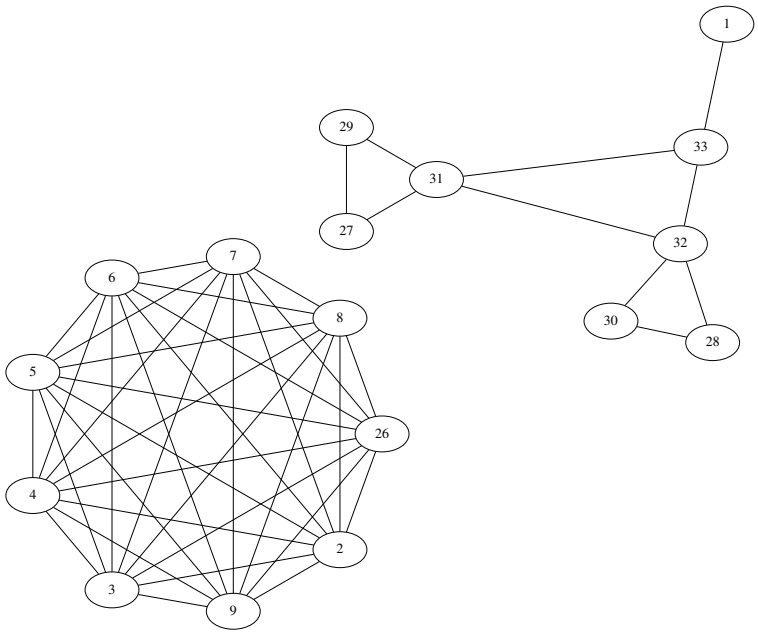

Figure 3. Example of the *primal graph* generated by variables in the CNF instance presented in Figure 2.

indicates that the variables (nodes) 28, 30, and 32 are connected to each other. The value zero (0) indicates the end of the expression.

For instance, the assignment of variable 7 do not depend on variable 29. In general, this graph generated from the source code is disconnected (one can see this disconnection between variables 7 and 29). In Figure 3, the sub-graph in right side should be disregarded because the key bits are completely in left side graph. To clean the final CNF, we can use the depth-first search (DFS) algorithm [Tarjan 1972] to remove clauses that do not have dependency on the bit keys. Basically, we give a single bit of the key as input to the DFS, and it finds all the vertices that matter for our work.

The algorithm E can be cryptographic algorithm. In the program depicted in Figure 4, we declare a variable `c_constant` that correspond to a precomputed ciphertext from a message `m` using some key. We also declare the `c` array that we used in algorithm `E` as ciphertext. CBMC software will search for a counterexample in SAT by using uninitialized variables. Note that the only variable that has not been initialized was the array `k`, that corresponds to the key. This means that the only variable with freedom in the SAT, that will be generated, corresponds to the bits of the key `k`.

In other words, CBMC generates a SAT instance for which a solution is a `k` that becomes `c[i]` equal to `c_constant[i]`, for all `i=0,...,TEXT_SIZE-1`. Then, we modified the CBMC code to identify how variables in SAT correspond the key variable in C code. After that, we can use the DFS algorithm starting from variables of k to remove undesirable clauses generated by CBMC, and finally, we can proceed by cleaning the final CNF. This process can be exemplified by the simple program presented in Figure 4.

```
void E( unsigned char  * c,
        unsigned char  * k,
        unsigned char  * m,
        int key_size,
        int text_size ) {
...
}

int main(int argc, char * argv) {
    unsigned char  k[KEY_SIZE];
    unsigned char  c[TEXT_SIZE] = { 0, 0, 0, ... };
    unsigned char  c_constant[TEXT_SIZE] = { ... };
    unsigned char  m[TEXT_SIZE] = { ... };
    E(c, k, m, KEY_SIZE, TEXT_SIZE);
    assert( c[0] != c_constant[0] ||
            c[1] != c_constant[1] ||
            c[2] != c_constant[2] ||
            ...
            c[TEXT_SIZE-1] != c_constant[TEXT_SIZE-1] );
    return 0;
}
```

Figure 4. Example of C program used in CBMC software.

4. Method Workflow

In this section, we explain all details related to the conversion of symmetric cryptographic algorithms into a SAT problem. Figure 5 can synthesize our analysis. In this figure, rectangles are files, and diamonds are processes. It is a general process, in other words, and we can use this procedure to analyze other cryptographic algorithms and even hash functions as well as other applications, like Message Authentication Code (MAC).

In the first part of the workflow, we have a `C` program that implements the known plaintext attack using `asserts`, as showed in the program presented Figure 1. We hard encode the plaintext and the ciphertext in the `asserts` to generate a SAT instance that, if solved, would retrieve the key. The goal is to declare `asserts` using ciphertext so that the generated SAT instance, if solved, would return a counterexample that, in this case, is the cryptographic key. We also provide varying sizes plaintext (and ciphertext), that can generate some knowledge in the experiments. In this step, we could also simulate the knowledge of a part of the key bits and eventually check the impact of this comprehension when trying to solve the generated SAT instance. Moreover, we can estimate the computational time to fully recover the key by declaring parts of the key and trying to recover the remain. For example, for AES-128, we could declare 96 bits of the key and try to retrieve the other 32 bits. Then, in other experiment, we could declare less bits, for example 88 bits and try to retrieve the other 40 bits. Thus, with these computational times, we could try to estimate the time required to properly break the full algorithm. Declaring part of the key could be done as follows: after the CBMC generates the CNF, we can append clauses with just one literal. This literal represents the declared key bit. If the respective bit of the

key is equal to 1, then we declare it as a positive literal, if this bit is equal to 0 we declare this literal with a negative sign.

Following the workflow, we use the modified CBMC software to output a SAT instance (using CNF format) of `C` program. We basically modified the CBMC to conveniently display the SAT variables, so that it can be possible to associate the SAT variables with the `C` program variables. Since we are interested in the variables corresponding to the cryptographic key, in the association between the SAT variables and the variables in the C program, then we have to pay attention to the format that is used in CBMC, which can be either little or big endian.

The next step of the workflow is to simplify the SAT instance generated by the CBMC software. The goal of this procedure is to select only the variables that are relevant for the analysis of the encryption algorithm. Another goal of this step is to try to normalize the outputs (CNF), to provide a fairer comparison.

Finally, after simplification, we can see the parameters and try to estimate the security of the algorithms. If possible, one can try to solve the SAT instance, if part of the key was previously declared.

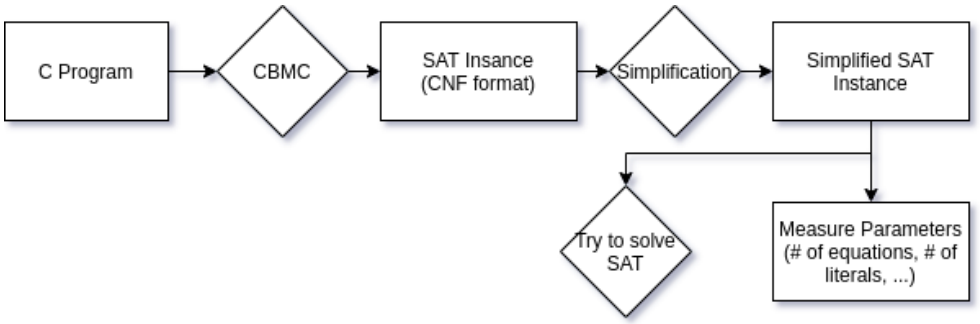

Figure 5. Workflow for SAT conversion and analysis.

5. Results

In this section, we describe the computational experiments considered to evaluate our proposal and the obtained results. We use the workflow described in Figure 5 to generate the results. In all experiments, we compare the AES, using Cipher Block Chaining (CBC), and the RC4.

We considered plaintext sizes of $\{16, 32, 64, 128\}$ bytes, and key sizes of $\{128, 192, 256\}$ bits. For AES-CBC we used tiny-AES library[1]. This library is described by authors as *"... a small and portable implementation of the AES ECB, CTR, and CBC encryption algorithms written in C"*. We did not use any IV nor PAD. In all experiments, the size of the plaintext is multiple of the block size (16 bytes). Due to the simplicity, we have implemented RC4, with less than 50 code lines in C language.

Figures 6 to 11 display the number of variables and clauses generated to attack RC4 and AES, considering 128, 192, and 256 bits after and before simplification. The

[1] Available at https://github.com/kokke/tiny-AES-c.

sizes of plaintexts, in each Figure, are $16, 32, 64$ and 128 bytes. We use the known plaintext attack, similar to that shown in Figure 4. As a simplification process, we run the pre-processor of `cryptominisat` [Soos et al. 2009]. Figure 6 presents the comparison of the number of variables and clauses, for RC4 using a key size of 128; Figure 7 shows the same comparison, for AES; Figure 8 presents the comparison of the number of variables and clauses, for RC4 with a key size of 192; Figure 9 presents the same comparison, for AES; Figure 10 presents the comparison of the number of variables and clauses, for RC4 with a key size of 256; finally, Figure 11 shows the same comparison, for AES.

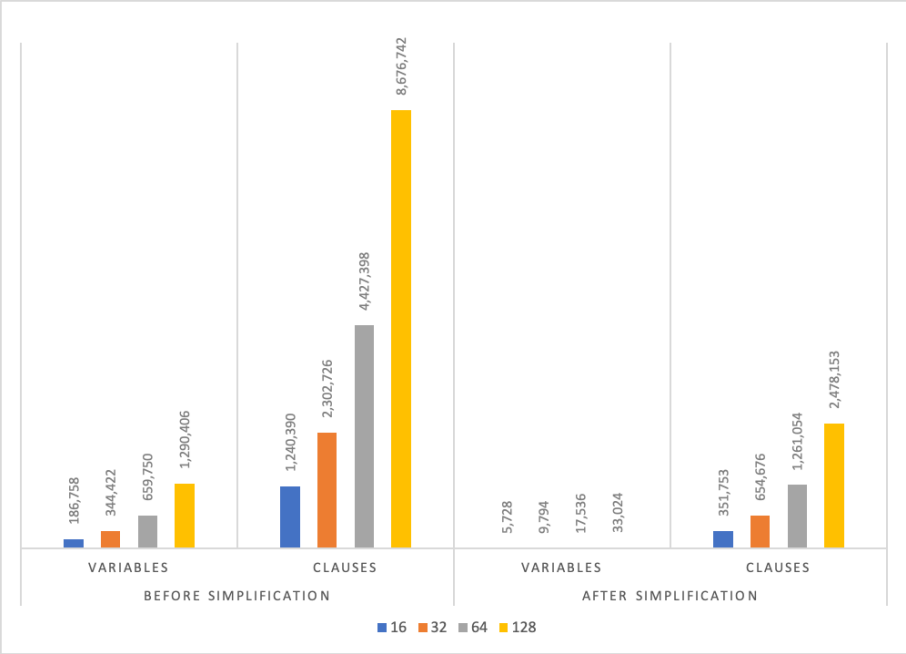

Figure 6. Comparison of variables and clauses for RC4 using key size of 128.

The comparison between RC4 and AES must be done carefully. After all, they follow different symmetric encryption models. In general, we have seen that the RC4, despite being much simpler, generated a relatively larger number of clauses and variables, when compared to the AES. However, when the key size was augmented, in the case of RC4, one could not see a significant increase in the number of variables and clauses. In the case of AES, the opposite behavior was verified, i.e., there was a significant increase in the number of variables and clauses, when the size of the key was augmented. A commonly used empirical measure for the hardness of a SAT instance is the following ratio:

$$r = \frac{\#\text{clauses}}{\#\text{variables}}. \tag{4}$$

For a random 3-SAT instance, a ratio value near of $r^* = 4.26$ indicates a hardest SAT random instance region [Nudelman et al. 2004]. Converting our SAT instances to 3-SAT (using 128 key bits and 16 bytes of plaintext) we obtain the following ratios:

$$\begin{aligned} r_{\text{RC4}} &= 1.81; \\ r_{\text{AES}} &= 1.19. \end{aligned}$$

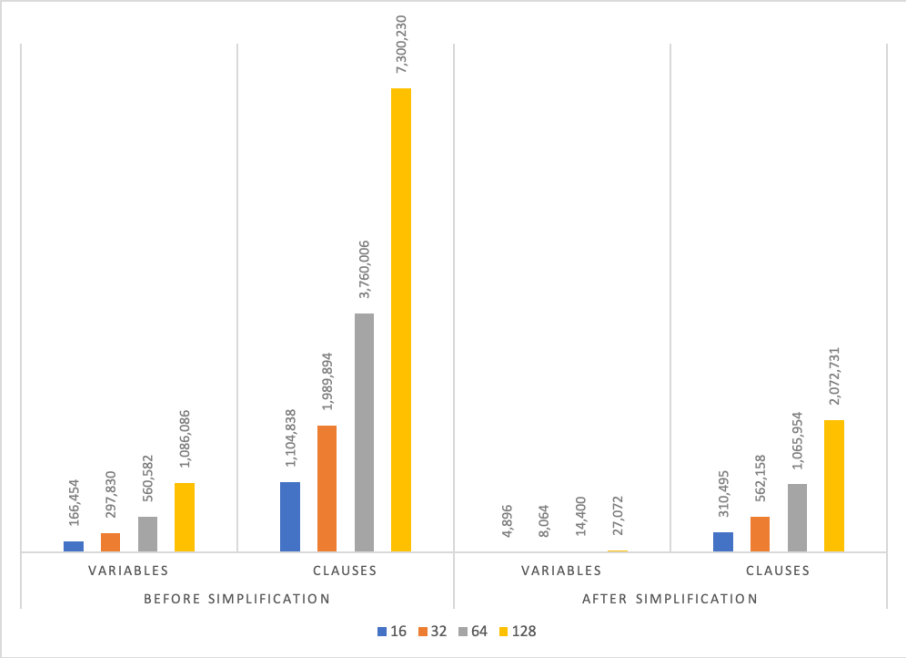

Figure 7. Comparison of variables and clauses for AES using key size of 128.

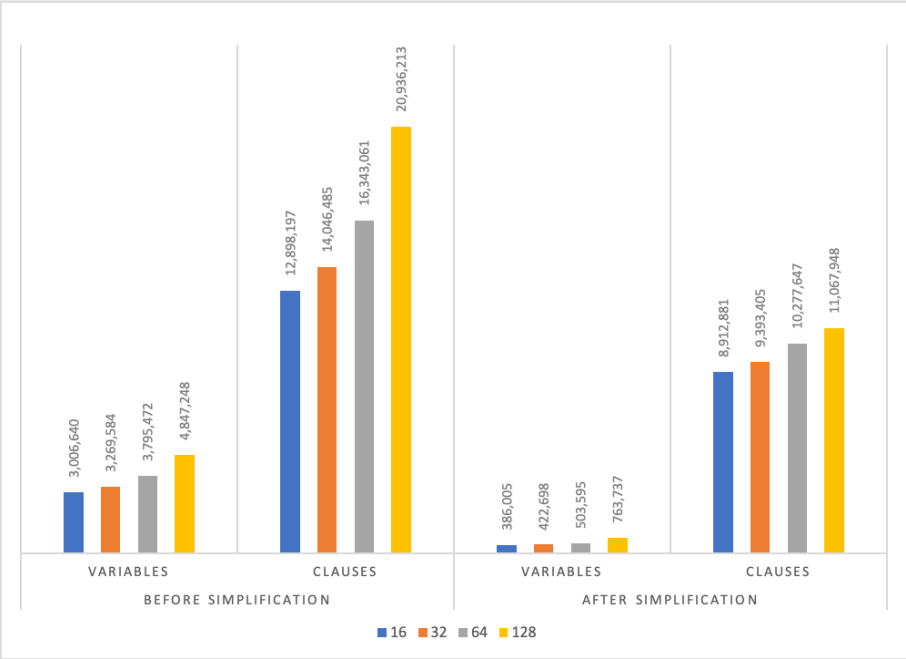

Figure 8. Comparison of variables and clauses for RC4 using key size of 192.

As the ratio r_{RC4} is slightly closer to r^*, it indicates, for this case, that resolving the instance generated by RC4 is more difficult than the AES instance. If an algorithm based on DPLL (very common among SAT solvers) [Davis and Putnam 1960] was used, the complexity would be given by [Ansótegui et al. 2008]:

$$O(n4^{w \log n}), \tag{5}$$

where n is the number of variables in a SAT instance, and w is the treewidth of tree

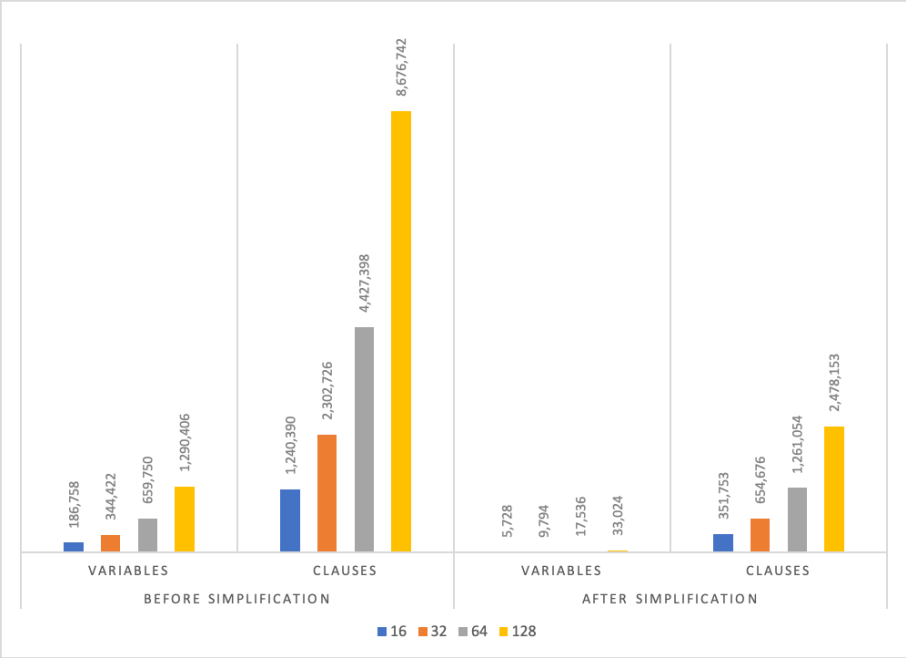

Figure 9. Comparison of variables and clauses for AES using key size of 192.

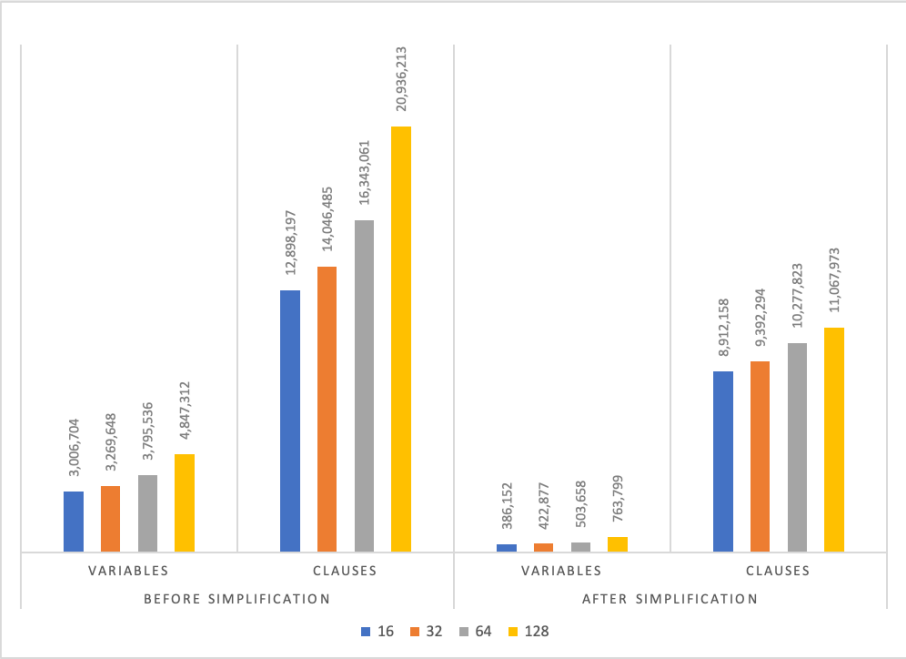

Figure 10. Comparison of variables and clauses for RC4 using key size of 256.

decomposition of the primal graph. Informally, treewidth measures how close to a tree a given graph is. For more details on how to compute treewidth from a SAT instance, see [Samer and Veith 2009].

6. Conclusions

This paper proposes a scheme to convert the problem of breaking symmetric cryptography into a SAT-type problem. In this case, we employ the CBMC software, used for Model

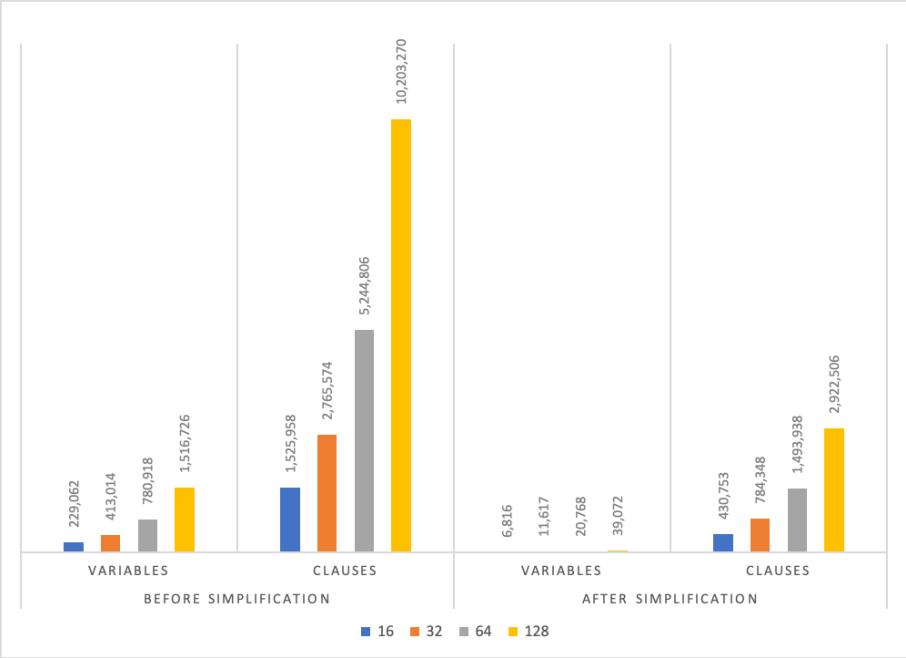

Figure 11. Comparison of variables and clauses for AES using key size of 256.

Checking, to transform a symmetric cryptographic source code into a set of clauses in CNF format, ready to be solved by a SAT solver. We use these techniques to convert the RC4 and AES algorithms to a SAT problem. As a result, the number of clauses and variables generated after the conversion, and simplification was evaluated and compared. Through the results presented in Figures 6 to 11, the number of variables and clauses for RC4 was substantially higher than AES; in addition, the ratio r_{RC4} is more closer to r^*. However, one could not see a significant increase in clauses nor variables, in the case of RC4, when the size of the key was incremented, i.e., it still is almost constant. Finally, we conclude that the ideas shown in this work can easily be extended to new attacks and security analysis. As future work, we can also calculate the treewidth of a SAT instance for a more accurate measure of the difficulty of solving the SAT using an algorithm based on DPLL.

References

[Ansótegui et al. 2008] Ansótegui, C., Bonet, M. L., Levy, J., and Manyà, F. (2008). Measuring the hardness of sat instances. In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 1*, AAAI'08, page 222–228. AAAI Press.

[Biere et al. 2003] Biere, A., Cimatti, A., Clarke, E. M., Strichman, O., and Zhu, Y. (2003). Bounded model checking.

[Bryant 1986] Bryant, R. E. (1986). Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691.

[Burch et al. 1991] Burch, J. R., Clarke, E. M., and Long, D. E. (1991). Symbolic model checking with partitioned transition relations. pages 49–58. North-Holland.

[Clarke and Emerson 1982] Clarke, E. M. and Emerson, E. A. (1982). Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, UK. Springer-Verlag.

[Courtois and Pieprzyk 2002] Courtois, N. and Pieprzyk, J. (2002). Cryptanalysis of block ciphers with overdefined systems of equations. volume 2002, page 44.

[Courtois 2005] Courtois, N. T. (2005). General principles of algebraic attacks and new design criteria for cipher components. In Dobbertin, H., Rijmen, V., and Sowa, A., editors, *Advanced Encryption Standard – AES*, pages 67–83, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Courtois and Meier 2003] Courtois, N. T. and Meier, W. (2003). Algebraic attacks on stream ciphers with linear feedback. In Biham, E., editor, *Advances in Cryptology — EUROCRYPT 2003*, pages 345–359, Berlin, Heidelberg. Springer Berlin Heidelberg.

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

[Eén and Sörensson 2003] Eén, N. and Sörensson, N. (2003). An extensible sat-solver. In Giunchiglia, E. and Tacchella, A., editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer.

[Given-Wilson et al. 2017] Given-Wilson, T., Jafri, N., Lanet, J. L., and Legay, A. (2017). An automated formal process for detecting fault injection vulnerabilities in binaries and case study on present. In *2017 IEEE Trustcom/BigDataSE/ICESS*, pages 293–300.

[Horbach et al. 2012] Horbach, A., Bartsch, T., and Briskorn, D. (2012). Using a sat-solver to schedule sports leagues. *J. Scheduling*, 15:117–125.

[Jovanovic 2010] Jovanovic, P. (2010). Algebraic attacks using sat-solvers. *Groups Complexity Cryptology*, 2:247–259.

[Kroening and Tautschnig 2014] Kroening, D. and Tautschnig, M. (2014). *CBMC – C Bounded Model Checker*, pages 389–391. Springer Berlin Heidelberg, Berlin, Heidelberg.

[Mohamed et al. 2009] Mohamed, M. S. E., Ding, J., Buchmann, J., and Werner, F. (2009). Algebraic attack on the mqq public key cryptosystem. In Garay, J. A., Miyaji, A., and Otsuka, A., editors, *Cryptology and Network Security*, pages 392–401, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Nudelman et al. 2004] Nudelman, E., Leyton-Brown, K., Hoos, H. H., Devkar, A., and Shoham, Y. (2004). Understanding random sat: Beyond the clauses-to-variables ratio. In Wallace, M., editor, *Principles and Practice of Constraint Programming – CP 2004*, pages 438–452, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Patarin 1995] Patarin, J. (1995). Cryptanalysis of the matsumoto and imai public key scheme of eurocrypt'88. In Coppersmith, D., editor, *Advances in Cryptology — CRYPT0' 95*, pages 248–261, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Samer and Szeider 2010] Samer, M. and Szeider, S. (2010). Algorithms for propositional model counting. *Journal of Discrete Algorithms*, 8(1):50 – 64.

[Samer and Veith 2009] Samer, M. and Veith, H. (2009). Encoding treewidth into sat. In Kullmann, O., editor, *Theory and Applications of Satisfiability Testing - SAT 2009*, pages 45–50, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Soos et al. 2009] Soos, M., Nohl, K., and Castelluccia, C. (2009). Extending SAT solvers to cryptographic problems. In Kullmann, O., editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257. Springer.

[Tarjan 1972] Tarjan, R. (1972). Depth first search and linear graph algorithms. *SIAM JOURNAL ON COMPUTING*, 1(2).