Journal of Applied Mathematics and Computational Mechanics 2023, 22(4), 64-78
www.amcm.pcz.pl p-ISSN 2299-9965
DOI: 10.17512/jamcm.2023.4.06 e-ISSN 2353-0588

PARTIAL MATHEMATICAL MODELING AND ANALYSIS OF THE AES SYSTEM

***Sylwia Stachowiak*[1], *Mirosław Kurkowski*[2]**

[1] *Department of Computer Science, University SWPS*
Warsaw, Poland
[2] *Institute of Computer Sciences, Cardinal Stefan Wyszyński University Warsaw, Poland*
sstachowiak@swps.edu.pl, m.kurkowski@uksw.edu.pl

Received: 31 July 2023; Accepted: 8 October 2023

Abstract. Many types of decision problems can be solved using mathematical modeling and analysis. Such techniques are also developed on the border of mathematical logic and computer science. A good example is the translation of the issues examined into the Satisfiability Problem (SAT) of a logical propositional formula. Unfortunately, this method is not always practical, considering the high computational complexity of solving the SAT problem. It often happens that in the studied cases, the encoding formulas contain even hundreds of thousands of clauses and propositional variables. However, even in these cases, modern SAT solvers can sometimes successfully solve these problems. This approach can be used to cryptanalyze some symmetric ciphers or parts/modifications. In this case, the encryption algorithm is first translated into a boolean formula. Then additional formulas are created to encode randomly selected plaintext and the key bits. Using the SAT solver; we can count the values of the ciphertext bits. Then, using the SAT solver again, we proceed to the cryptanalysis of the cipher with the selected plaintext and proper ciphertext, looking for the bits of the encryption key. In this paper, we will present the new results of how SAT techniques behave against representative fragments of the AES cipher, the current standard for symmetric encryption. We also compare the results obtained in this case by several SAT solvers. In addition, we present the results of the SAT-solver CryptoMiniSat obtained during the attack on the 1st round of the AES-128 cipher.

MSC 2010: *03B70, 94A60*
Keywords: *mathematical modeling, AES, symmetric ciphers, satisfiability, SAT-based cryptanalysis*

1. Introduction

The SAT problem is a classical problem in the NP class, meaning it can be solved in polynomial time on a nondeterministic Turing machine. Another way to define an NP problem is that checking a solution provided by an external source has polynomial complexity. SAT is the first problem proven to be NP-complete [1]. It is important to note that any other problem in NP can be reduced to SAT in polynomial time, making it a complete problem in the NP class.

The SAT problem determines whether a given propositional formula is satisfiable. Thus, the satisfiability problem focuses on finding such an evaluation of the variables in the formula that the value of the formula is 1. If there is at least one such evaluation, then the formula is called satisfiable (SAT). Otherwise, the determination formula is unsatisfiable (UNSAT).

The problem of the satisfiability of logical formulas is decidable. The simplest method of solving this problem is the method consisting in considering all possible evaluations of variables occurring in the logical formula. If the number of variables in the formula is n, then we get 2^n of such substitutions. Thus, this method of solving the SAT problem has an exponential computational complexity [2].

Boolean and SAT encoding, among other mathematical methods of software systems specification and verification [3, 4], can be used to solve problems in many fields [5–10]. This approach usually consists in transforming the studied problems into the SAT problem, solving it, and then mapping the obtained model to the solution of the source problem.

SAT solvers have been used to analyze cryptographic algorithms for many years. For example, Courtois and Pieprzyk continued the work described in [11]. In 2006, they performed algebraic attacks on the DES cipher using SAT solvers, using the polynomial equivalence of two NP-hard problems [2].

In the case of the AES cipher, Gwynne and Kullmann published a paper that describes the encoding of the transformation `SubBytes()` and the multiplication $\bullet$ of a polynomial from the field $GF(256)$ by x and $x+1$ using various heuristics. The authors also presented an AES to CNF translation, but unfortunately, they have not tested it, and there is no certainty that it is equivalent to the AES cipher [12].

This paper presents a different approach to the cryptanalysis of encryption algorithms using SAT techniques. In our method, we do not describe the algorithm using algebraic equations nor do we encode them into boolean formulas. Instead, we use the direct boolean encoding of the encryption algorithm, as was done in the works of [13–16]. In this article, we describe how the SAT techniques for the AES cipher behave. To the best of our knowledge, a complete encoding of an AES cipher using direct boolean encoding has not been previously published. The AES encoding formula we presented has been tested and is equivalent to the AES cipher.

The rest of the article is organized as follows. Section 2 contains all the information you need about the AES cipher. We do this to the extent necessary to explain our approach to the cryptanalysis of a cipher using SAT coding. Section 3 describes the method of encoding the AES actions into a boolean formula. Section 4 describes the conversion of boolean formulas to conjunctive normal form and then to the DIMACS format. Section 5 presents our experimental results. At the end of the article, there are conclusions and directions for further research that we are conducting.

2. The AES cipher

In 2001, the NIST (National Institute of Standards and Technology) adopted the current Advanced Encryption Standard (AES) as FIPS-197 [17], thus replacing the DES standard. The winner of the competition announced in 1997 by NIST was Rijndael. Rijndael is a family of ciphers with different key lengths and different block sizes. The name Rijndael comes from the names of the creators of the cipher, Joan Daemen and Vincent Rijmen. AES includes three algorithms from the Rijndael family, each of which can process 128-bit blocks of data using a 128-bit, 192-bit, or 256-bit encryption key, depending on the selected algorithm. In 2016, Ashokkumar C., Ravi Prakash Giri and Bernard Menezes presented a side-channel attack on AES implementations that can recover the complete 128-bit AES key in just 6-7 blocks of plaintext/ciphertext, which is a substantial improvement over previous works that require between one hundred and a million encryptions [18].

2.1. AES specification

When the AES algorithm runs, transformations are performed on a two--dimensional array of bytes called the *state* [17]. The state consists of four rows of bytes; each row, for the AES standard, consists of four bytes. The state table is denoted by the letter s, and each byte in it has two indices: r and c, where r is the row number and $0 \leq r \leq 3$, and c is the column number and $0 \leq c \leq 3$.

A block of input data $input_0 input_1 input_2 \ldots input_{127}$ with a length of 128 bits is stored in the byte array in such a way that: $in_0 = input_0, input_1, \ldots, input_7$, $in_1 = input_8, input_9, \ldots, input_{15}, \ldots, in_{15} = input_{120}, input_{121}, \ldots, input_{127}$. Longer sequences, such as 192-bit and 256-bit keys, are placed in the state according to the following formula: $in_n = input_{8n} input_{8n+1}, \ldots, input_{8n+7}$. When starting encryption or decryption, the algorithm copies the input $in_0, in_1, \ldots, in_{15}$ to the state according to the scheme: $s_{r,c} = in_{r+4c}$, for $0 \leq r \leq 3$ and $0 \leq c \leq 3$. It then performs encryption or decryption on that state to finally copy its final value into the output byte array as follows: $out_{r+4c} = s_{r,c}$, for $0 \leq r \leq 3$ and $0 \leq c \leq 3$.

The number of rounds (Nr) performed in AES depends on the key size. AES does ten rounds for a 128-bit key, 12 for a 192-bit key, and 14 for a 256-bit key.

The AES algorithm uses a Substitution Permutation Network (SPN) structure, where the substitution is performed by SubBytes(), and for the permutation combining the transformations `ShiftRows()` and `MixColumns()`.

2.2. AES transformations

The `SubBytes()` transformation is non-linear and operates independently on each state byte using a substitution table (S-box). The S-box, written in hexadecimal, used in the `SubBytes()` transformation is presented in paper [17]. This S-box is built based on two transformations: the multiplicative inverse in the finite field $GF(256)$, additionally assuming that the multiplicative inverse of the `0x00` element is itself and the following affine transformation over GF (2): $b'_i = b_i \oplus b_{(i+4)mod8} \oplus b_{(i+5)mod8} \oplus b_{(i+6)mod8} \oplus b_{(i+7)mod8} \oplus c_i$ for $0 \leq i \leq 7$, where b_i is the i byte bit, and c_i is the ith bit by c, which has a value of `0x63` or `01100011`.

`ShiftRows()` cycles the bytes left in the second, third, and fourth rows of the state one, two, and three places, respectively. The first line remains unchanged. The state is modified according to the equation $s'_{r,c} = s_{r,(c+shift(r,4))mod4}$, for $0 < r < 4$ and $0 \leq c \leq 4$, where the shift value $shift(r,4)$ depends on the line number r it is: $shift(0,4) = 0; shift(1,4) = 1; shift(2,4) = 2; shift(3,4) = 3$.

`MixColumns()` uses data from all state columns and then mixes them to create new columns by doing the following:

$$\begin{aligned}
s'_{0,c} &= (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c},\\
s'_{1,c} &= s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c},\\
s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c}),\\
s'_{3,c} &= (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c}),
\end{aligned}$$

where $\bullet$ is a polynomial multiplication in the field $GF(256)$ modulo, the specially chosen irreducible polynomial of degree eight over $GF(2)$: $x^8 + x^4 + x^3 + x + 1$ (see for details [17]).

The procedure `AddRoundKey()` adds a modulo two round key to the state. Each round key consists of 4 words that are placed in the key schedule, and these words are added to the stat columns as follows: $[s'_{0,c}, s'_{1,c}, s'_{2,c}, s'_{3,c}] = [s_{0,c}, s_{1,c}, s_{2,c}, s_{3,c}] \oplus [w_{round*4+c}]$ for $0 \leq c \leq 4$, where $[w_i]$ is the key words included in the key schedule, and *round* is between $0 \leq round \leq Nr$. The `AddRoundKey()` transformation is called before the first use of the round function – the initialization key is scanned into the state when $round = 0$. The `AddRoundKey()` routine is used in Nr rounds of the AES cipher when $1 \leq round \leq Nr$.

A round key set is a one-dimensional array of 4-byte words created by the `Key Expansion` procedure. This algorithm retrieves the K cipher key and generates $4(Nr+1)$ words. An initial set of 4 words is required for the algorithm to work, and each of the Nr rounds requires 4 keywords. The resulting set of keys consists of a linear array of 4-byte words, labeled $[w_i]$, where $i \in [0, 4(Nr+1))$. In the case of a 128-bit key, this table has the form: $[w_0, w_1, ..., w_{43}]$. Extending the input key to a set of keys follows the scheme presented in [17].

3. Boolean encoding

In this part of the article, we will present the transformation of the functions included in the AES algorithm to boolean formulas. For this purpose, we use the direct boolean encoding method.

3.1. Boolean encoding of `SubBytes()` transformations

The `SubBytes()` transformation is used in the AES algorithm in the main round function as well as for creating the key table. The S-box used in AES is a square matrix of 16×16, each element being a different eight-bit vector. These vectors are represented in hexadecimal form as digraphs. Hence, we can consider S-box as a function of type $S_{box} : \{0,1\}^8 \to \{0,1\}^8$. For the sake of simplicity, denote by $\bar{x}$ the vector $(x_1, \ldots, x_8)$ and by $S^k_{box}(\bar{x})$ the k-th coordinate of $S_{box}(\bar{x})$ for $k = 1, 2, \ldots, 8$. We can code the S-box as the following boolean formula:

$$\phi_{S_{box}} : \bigwedge_{\bar{x} \in \{0,1\}^8} \left(\bigwedge_{i=1}^{8} (\neg)^{1-x_i} p_i \Rightarrow \bigwedge_{j=1}^{8} (\neg)^{1-S^j_{box}(\bar{x})} q_j \right),$$

where $(p_1, \ldots, p_8)$ is the input vector to the S-box and $(q_1, \ldots, q_8)$ is the output vector. Additionally, we denote p and $\neg p$ by $(\neg)^0 p$ and $(\neg)^1 p$, respectively.

3.2. Boolean encoding of `MixColumns()` transformations

The `MixColumns()` procedure transforms the successive columns of the state table, assuming that we start numbering the columns from zero. By $(p_1, p_2, \ldots, p_{128})$, where $p_i \in \{0,1\}$, for $i = 1, 2, \ldots, 128$ let's denote the vector representing the input data stored in the state table and by $(q_1, q_2, \ldots, q_{128})$, where $q_i \in \{0,1\}$, for $i = 1, 2, \ldots, 128$, let's denote the vector representing the output after the `MixColumns()` operation. The boolean encoding of the `MixColumns()` transformation can be written as follows:

$$s'_{0,c} = (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c}$$

for the first state column written using a boolean formula is:

$$\begin{aligned}
q_i &\Leftrightarrow p_{i+1} \oplus \left(p_{(i+8)+1} \oplus p_{i+8}\right) \oplus p_{i+16} \oplus p_{i+24} \\
q_i &\Leftrightarrow p_{i+1} \oplus p_1 \oplus \left(p_{(i+8)+1} \oplus p_9 \oplus p_{i+8}\right) \oplus p_{i+16} \oplus p_{i+24} \\
q_i &\Leftrightarrow p_{i+1} \oplus \left(p_{(i+8)+1} \oplus p_{i+8}\right) \oplus p_{i+16} \oplus p_{i+24} \\
q_i &\Leftrightarrow p_{i+1} \oplus p_1 \oplus \left(p_{(i+8)+1} \oplus p_9 \oplus p_{i+8}\right) \oplus p_{i+16} \oplus p_{i+24} \\
q_i &\Leftrightarrow p_1 \oplus (p_9 \oplus p_{i+8}) \oplus p_{i+16} \oplus p_{i+24}.
\end{aligned}$$

for $i = 1, \ldots, 3$, $i = 4, 5$, $i = 6$, $i = 7$, $i = 8$, respectively. The second equation describing the operation of `MixColumns()`

$$s'_{1,c} = s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c}$$

applied to a column indexed $c = 0$ can be written as follows:

$$
\begin{aligned}
&q_i \Leftrightarrow p_{i-8} \oplus p_{i+1} \oplus \left(p_{(i+8)+1} \oplus p_{i+8}\right) \oplus p_{i+16}\\
&q_i \Leftrightarrow p_{i-8} \oplus p_{i+1} \oplus p_9 \oplus \left(p_{(i+8)+1} \oplus p_{17} \oplus p_{i+8}\right) \oplus p_{i+16}\\
&q_i \Leftrightarrow p_{i-8} \oplus p_{i+1} \oplus \left(p_{(i+8)+1} \oplus p_{i+8}\right) \oplus p_{i+16}\\
&q_i \Leftrightarrow p_{i-8} \oplus p_{i+1} \oplus p_9 \oplus \left(p_{(i+8)+1} \oplus p_{17} \oplus p_{i+8}\right) \oplus p_{i+16}\\
&q_i \Leftrightarrow p_{i-8} \oplus p_9 \oplus (p_{17} \oplus p_{i+8}) \oplus p_{i+16}
\end{aligned}
$$

for $i = 9,\dots,11$, $i = 12,13$, $i = 14$, $i = 15$, $i = 16$.

Applying boolean to the third equation $s'_{2,c} = s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c})$ we get:

$$
\begin{aligned}
&q_i \Leftrightarrow p_{i-16} \oplus p_{i-8} \oplus p_{i+1} \oplus \left(p_{(i+8)+1} \oplus p_{i+8}\right)\\
&q_i \Leftrightarrow p_{i-16} \oplus p_{i-8} \oplus p_{i+1} \oplus p_{17} \oplus \left(p_{(i+8)+1} \oplus p_{25} \oplus p_{i+8}\right)\\
&q_i \Leftrightarrow p_{i-16} \oplus p_{i-8} \oplus p_{i+1} \oplus \left(p_{(i+8)+1} \oplus p_{i+8}\right)\\
&q_i \Leftrightarrow p_{i-16} \oplus p_{i-8} \oplus p_{i+1} \oplus p_{17} \oplus \left(p_{(i+8)+1} \oplus p_{25} \oplus p_{i+8}\right)\\
&q_i \Leftrightarrow p_{i-16} \oplus p_{i-8} \oplus p_{17} \oplus (p_{25} \oplus p_{i+8}).
\end{aligned}
$$

for $i = 17,\dots,19$, $i = 20,21$, $i = 22$, $i = 23$, $i = 24$, respectively.

The last equation describes the changes by the `MixColumns()` function in the last line of the state:

$$s'_{3,c} = (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c})$$

applied to the first column takes the form of a conjunction of the following formulas:

$$
\begin{aligned}
&q_i \Leftrightarrow \left(p_{(i-24)+1} \oplus p_{i-24}\right) \oplus p_{i-16} \oplus p_{i-8} \oplus p_{i+1},\\
&q_i \Leftrightarrow \left(p_{(i-24)+1} \oplus p_1 \oplus p_{i-24}\right) \oplus p_{i-16} \oplus p_{i-8} \oplus p_{i+1} \oplus p_{25},\\
&q_i \Leftrightarrow \left(p_{(i-24)+1} \oplus p_{i-24}\right) \oplus p_{i-16} \oplus p_{i-8} \oplus p_{i+1},\\
&q_i \Leftrightarrow \left(p_{(i-24)+1} \oplus p_1 \oplus p_{i-24}\right) \oplus p_{i-16} \oplus p_{i-8} \oplus p_{i+1} \oplus p_{25},\\
&q_i \Leftrightarrow (p_1 \oplus p_{i-24}) \oplus p_{i-16} \oplus p_{i-8} \oplus p_{25}.
\end{aligned}
$$

for $i = 25,\dots,27$, $i = 28,29$, $i = 30$, $i = 31$, $i = 32$, respectively.

Similarly, assuming that the state columns are denoted by c and the first column is indexed 0, we can write a boolean formula encoding the `MixColumns()` operation for the subsequent state columns.

3.3. Boolean encoding of `AddRoundKey()` transformations

The operation of the `AddRoundKey( )` transformation is to XOR two strings of bits. We will assume that $(p_1, p_2, \dots, p_{128})$ and $(v_1, v_2, \dots, v_{128})$ will be the vectors representing the input and $p_i \in \{0.1\}$i $v_i \in \{0.1\}$, for $i = 1,2,\dots,128$ and let the vector $(q_1, q_2, \ldots, q_{128})$, where $q_i \in \{0,1\}$, for $i = 1,2,\ldots,128$ will represent the result of `AddRoundKey( )`. Therefore, the boolean encoding of `AddRoundKey( )` can be represented as follows:

$$\bigwedge \begin{cases} q_1 & \Leftrightarrow \quad p_1 \oplus v_1 \\ q_2 & \Leftrightarrow \quad p_2 \oplus v_2 \\ & \ldots \\ q_{127} & \Leftrightarrow \quad p_{127} \oplus v_{127} \\ q_{128} & \Leftrightarrow \quad p_{128} \oplus v_{128}. \end{cases}$$

The transformations that are parts of AES encoded in this way allow us to create a formula that encodes any number of rounds of the AES algorithm.

4. Translating formulas to CNF and SAT-based cryptanalysis

Below we present the conversion of sentence formulas describing the operation of individual functions in the AES algorithm, obtained as a result of direct boolean encoding to conjunctive normal form (CNF).

4.1. `SubBytes()` transformation

By using the `SubBytes()` encoding described in 3 for one byte of input data represented by a vector $(p_1, \ldots, p_8)$ and one byte of output $(q_1, \ldots, q_8)$, the following formula can be written:

$$\bigwedge \begin{cases} (p_1 \wedge p_2 \wedge p_3 \wedge p_4 \wedge p_5 \wedge p_6 \wedge p_7 \wedge p_8) \Rightarrow q_1 \\ (p_1 \wedge p_2 \wedge p_3 \wedge p_4 \wedge p_5 \wedge p_6 \wedge p_7 \wedge p_8) \Rightarrow q_2 \\ (p_1 \wedge p_2 \wedge p_3 \wedge p_4 \wedge p_5 \wedge p_6 \wedge p_7 \wedge p_8) \Rightarrow q_3 \\ (p_1 \wedge p_2 \wedge p_3 \wedge p_4 \wedge p_5 \wedge p_6 \wedge p_7 \wedge p_8) \Rightarrow q_4 \\ (p_1 \wedge p_2 \wedge p_3 \wedge p_4 \wedge p_5 \wedge p_6 \wedge p_7 \wedge p_8) \Rightarrow q_5 \\ (p_1 \wedge p_2 \wedge p_3 \wedge p_4 \wedge p_5 \wedge p_6 \wedge p_7 \wedge p_8) \Rightarrow q_6 \\ (p_1 \wedge p_2 \wedge p_3 \wedge p_4 \wedge p_5 \wedge p_6 \wedge p_7 \wedge p_8) \Rightarrow q_7 \\ (p_1 \wedge p_2 \wedge p_3 \wedge p_4 \wedge p_5 \wedge p_6 \wedge p_7 \wedge p_8) \Rightarrow q_8. \end{cases}$$

Applying the appropriate logical laws to the above formula, we transform it to CNF and obtain the equivalent formula in the given form:

$$\wedge \begin{cases} (\neg p_1 \vee \neg p_2 \vee \neg p_3 \vee \neg p_4 \vee \neg p_5 \vee \neg p_6 \vee \neg p_7 \vee \neg p_8 \vee q_1) \\ (\neg p_1 \vee \neg p_2 \vee \neg p_3 \vee \neg p_4 \vee \neg p_5 \vee \neg p_6 \vee \neg p_7 \vee \neg p_8 \vee q_2) \\ (\neg p_1 \vee \neg p_2 \vee \neg p_3 \vee \neg p_4 \vee \neg p_5 \vee \neg p_6 \vee \neg p_7 \vee \neg p_8 \vee q_3) \\ (\neg p_1 \vee \neg p_2 \vee \neg p_3 \vee \neg p_4 \vee \neg p_5 \vee \neg p_6 \vee \neg p_7 \vee \neg p_8 \vee q_4) \\ (\neg p_1 \vee \neg p_2 \vee \neg p_3 \vee \neg p_4 \vee \neg p_5 \vee \neg p_6 \vee \neg p_7 \vee \neg p_8 \vee q_5) \\ (\neg p_1 \vee \neg p_2 \vee \neg p_3 \vee \neg p_4 \vee \neg p_5 \vee \neg p_6 \vee \neg p_7 \vee \neg p_8 \vee q_6) \\ (\neg p_1 \vee \neg p_2 \vee \neg p_3 \vee \neg p_4 \vee \neg p_5 \vee \neg p_6 \vee \neg p_7 \vee \neg p_8 \vee q_7) \\ (\neg p_1 \vee \neg p_2 \vee \neg p_3 \vee \neg p_4 \vee \neg p_5 \vee \neg p_6 \vee \neg p_7 \vee \neg p_8 \vee q_8). \end{cases}$$

To write a formula in the DIMACS format that encodes the transformation of one byte by the `SubBytes()` transformation, 16 variables and 2048 clauses are needed. And writing the formula encoding the transformation of a 16-byte state table by the `SubBytes()` transformation requires writing over 30,000 clauses using 256 variables. [8] estimates that approximately 4800 clauses and 900 variables are needed to encode the S-box used in the AES cipher.

This work has used the CryptLogVer tool to generate the formula.

By transforming the formula encoding the `MixColumns()` function to the conjunctive normal form, we introduced additional variables representing the results of partial operations included in the `MixColumns()` transformation.

For example, *boolean encoding* of the $02 \bullet r$ operation, where r is the input byte variable denoted by $r = (r_1, \ldots, p_8)$, where $u_i \in \{0,1\}$, for $i = 1,2,\ldots,8$, and the vector $(p_1, \ldots, p_8)$, where $p_8 \in \{0,1\}$, for $i = 1,2,\ldots,8$ represents the output byte, has the form:

$$\wedge \begin{cases} p_1 & \Leftrightarrow & r_2 \\ p_2 & \Leftrightarrow & r_3 \\ p_3 & \Leftrightarrow & r_4 \\ p_4 & \Leftrightarrow & r_5 \oplus r_1 \\ p_5 & \Leftrightarrow & r_6 \oplus r_1 \\ p_6 & \Leftrightarrow & r_7 \\ p_7 & \Leftrightarrow & r_8 \oplus r_1 \\ p_8 & \Leftrightarrow & r_1. \end{cases}$$

Additionally, *boolean encoding* of the $03 \bullet r$ operation, where similarly to the above r is a vector representing the input data, and the sequence $(t_1, \ldots, t_8)$, where $t_i \in \{0,1\}$, for $i = 1,2,\ldots,8$ represents the output byte, has the form:

$$\bigwedge\begin{cases} t_1 & \Leftrightarrow & r_2 \oplus r_1 \\ t_2 & \Leftrightarrow & r_3 \oplus r_2 \\ t_3 & \Leftrightarrow & r_4 \oplus r_3 \\ t_4 & \Leftrightarrow & (r_5 \oplus r_1) \oplus r_4 \\ t_5 & \Leftrightarrow & (r_6 \oplus r_1) \oplus r_5 \\ t_6 & \Leftrightarrow & r_7 \oplus r_6 \\ t_7 & \Leftrightarrow & (r_8 \oplus r_1) \oplus r_7 \\ t_8 & \Leftrightarrow & r_1 \oplus r_8. \end{cases}$$

Hence, transforming one bit by `MixColumns()` can be written as the following formula:

$$q_1 \Leftrightarrow p_1 \oplus t_1 \oplus w_1 \oplus z_1.$$

Then the conjunctive normal form of the above formula is as follows:

$$\bigwedge\begin{cases} (\neg p_1 \vee \neg t_1 \vee \neg w_1 \vee \neg z_1 \vee \neg q_1) \\ (\neg p_1 \vee \neg t_1 \vee \neg w_1 \vee z_1 \vee q_1) \\ (\neg p_1 \vee \neg t_1 \vee w_1 \vee \neg z_1 \vee q_1) \\ (\neg p_1 \vee \neg t_1 \vee w_1 \vee z_1 \vee \neg q_1) \\ (\neg p_1 \vee t_1 \vee \neg w_1 \vee \neg z_1 \vee q_1) \\ (\neg p_1 \vee t_1 \vee \neg w_1 \vee z_1 \vee \neg q_1) \\ (\neg p_1 \vee t_1 \vee w_1 \vee \neg z_1 \vee \neg q_1) \\ (\neg p_1 \vee t_1 \vee w_1 \vee z_1 \vee q_1) \\ (p_1 \vee \neg t_1 \vee \neg w_1 \vee \neg z_1 \vee q_1) \\ (p_1 \vee \neg t_1 \vee \neg w_1 \vee z_1 \vee \neg q_1) \\ (p_1 \vee \neg t_1 \vee w_1 \vee \neg z_1 \vee \neg q_1) \\ (p_1 \vee \neg t_1 \vee w_1 \vee z_1 \vee q_1) \\ (p_1 \vee t_1 \vee \neg w_1 \vee \neg z_1 \vee \neg q_1) \\ (p_1 \vee t_1 \vee \neg w_1 \vee z_1 \vee q_1) \\ (p_1 \vee t_1 \vee w_1 \vee \neg z_1 \vee q_1) \\ (p_1 \vee t_1 \vee w_1 \vee z_1 \vee \neg q_1). \end{cases}$$

We can see that by converting the formula obtained above to the DIMACS format, we get 16 clauses. Moving on and undergoing surgery `MixColumns()` 1 byte of data, we get 128 clauses. Saving in the DIMACS format the encoding of the transformation of the entire internal state by this function, by our method, requires 2048 clauses.

Next, we transformed the boolean formula encoding the function `AddRoundKey()` to conjunctive normal form. Thus, by boolean encoding of individual transformations occurring in AES, using the method described above, and their conversion to CNF, we obtain a set of clauses corresponding to the operation of the entire AES algorithm.

We conducted tests that showed the correctness of this encoding. We tested the encoding of both individual AES operations and the encoding of the entire AES cipher. The input and output values for these tests were determined using the test vectors given in [17].

4.2. SAT solvers

SAT solvers are programs designed to practically solve the SAT problem for formulas containing a large number of variables as well as a large number of clauses. The SAT solver searches for, if any, an evaluation of the variables that satisfiability the boolean formula under examination. Many SAT solvers are based on an algorithm developed by Davis, Putnam, Logemann and Loveland called the DPLL algorithm. It is one of the most popular algorithms for automatic checking of formulas [19]. To be able to use it, the studied formula should be presented in the CNF.

4.3. Cryptanalysis procedure

Starting the cryptanalysis with the selected plaintext of the cipher, we first write a boolean formula that encodes the operation of the entire cipher in the way, which we presented in the Section 3 [13, 14, 20]. We convert the obtained formula to the DIMACS format. We then generate two strings of bits representing the plaintext and the key, respectively. We write them in the form of clauses in such a way that each bit corresponds to one clause consisting of a single literal. The plaintext and key bits represented in this way are added to the previously prepared formula. In the next step, we look for an evaluation that meets the prepared formula.

For this, we use the SAT solver. We receive the evaluation of the propositional variables that make up the ciphertext. Finally, using the formula encoding the operation of the cipher and a set of clauses describing the plaintext and the ciphertext, we search for the value of key bits and thus perform a cryptanalytic attack.

5. Experimental results

With time, successive progress is made in AES cryptanalysis. The best results until 2009, were reported by [21, 22], and it was a 7-round AES-128 attack. The first was slightly faster than an exhaustive search, and the time complexity of the second was 2^{120}. Subsequent works [23, 24] published attacks against 10 rounds of AES-192 and 10 rounds of AES-256 with 10, 12, and 14 rounds respectively. In 2009, Biryukov and others in [25] presented a key-related attack against the full version of AES-256 with a time complexity of 2^{96} for one of 2^{35} keys. They also described practical attacks against AES-256 [26].

In addition, Biryukov and his team in [27] presented the first related key attack against AES-256, which works for all keys, with a better time complexity of $2^{99.5}$, and the first attack of this kind against the full version of the AES-192 cipher.

In this part of the article, we present the obtained experimental results of the SAT cryptanalysis of the AES cipher in the version with a 128-bit key. For our experiments, the test platform was a laptop with a Hexa-core Intel Core i7-10750H processor working at a base frequency of 2.60 GHz with Hyper-V support. The computer had a system with 32 GB of RAM and worked under Windows 10 Professional. We used the built-in statistics for Linux to measure the time. Given the computational complexity of the SAT, we did not expect all test instances to be processed in a reasonable amount of time.

For the research, we have developed a tool that generates a logical formula that models the operation of subsequent transformations used in the AES algorithm. Finally, we received a formula that models the operation of the entire AES cipher in the 128-bit key version. The resulting formula is in DIMACS format. An undoubted advantage of this solution is the lack of the need to perform additional conversions of formulas and related activities to check the correctness of these transformations.

To carry out the cryptographic analysis with the selected plaintext of the AES cipher in the 128-bit key version using the method that we presented, we used SAT-solvers, which in recent years occupied leading positions, e.g. in the *SAT Competition 2020* competition, i.e. CaDiCal, Cryptominisat, Kissat and Plingeling [28]. In addition, we reused the stable SAT-solver which is Minisat, and the SAT-solver lingeling – a single-threaded version of Plingeling. First, we attempted to crack 1 round of AES-128. We set the time limit at one week. None of the SAT solvers used at that time found an evaluation that met the given formula, and thus did not find the bits of the key. We changed the logical formula and added to it, in the appropriate form, successively 64, 40, 32, 24, and finally 16 initial key bits. We set a time limit of 96 hours for this study. The results obtained by individual SAT-solvers are presented in the Tables 1 and 2 (in the absence of a result after 24 hours of calculations, the experiments were stopped, which was marked with ”–”).

Table 1. Results for SAT solvers on one round of AES with added initial key bits

Number of added key bits	MiniSat 2.2.1 Time [s]	CryptoMiniSat 5.8.0 Time [s]	lingeling bcp2020 Time [s]
64	0.205	0.031	0.112
40	276	15.2	9.67
32	10	289	5514
24	–	29799	–
16	–	6484	332387

When we supplemented the formula with half of the key bits, all used SAT solvers which quickly calculated the valuation of the variables that represent the missing key bits. Similarly in the next trial, where we added the initial 32 bits of the key, all SAT solvers used calculated the missing key bit evaluations. The next step was to add 24 bits of the key. The time needed for individual SAT solvers to provide the value of variables representing missing key bits increased significantly again, except for CryptoMiniSat SAT-solver. In addition, two of the used SAT solvers: MiniSat and lingeling failed to complete the calculations in the given time. Given the initial 16 bits of the SAT key, the solvers needed from about 18 hours – CryptoMiniSAT to over 92 hours – lingeling to search for the remaining 112 bits. In addition, MiniSat, Kissat, and Plingeling failed to complete the calculations within the time allowed.

Table 2. Results for SAT solvers on one round of AES with added initial key bits

Number of added key bits	CaDiCal 1.4.1/2021 Time [s]	Kissat 2.0.0 Time [s]	Plingeling bcp/2020 Time [s]
64	0.127	0.136	0.093
40	10.5	13.1	24.6
32	189	81.8	467
24	1728	6929	6368
16	81780	–	–

SAT-solver CryptoMiniSat was selected to perform further tests. The study was conducted in two variants. In the first variant, as in the previous experiment, to the boolean formula modeling the operation of the AES-128 cipher in the first round, a fixed number of initial key bits was added, and in the second variant, the final key bits. The time limit was set at 12 hours. In the first stage of the study, 32, 30, 28, and 26 initial or final key bits were added, respectively. The results of this stage of the experiment are presented in the Table 3.

Table 3. The results of the CryptoMiniSat SAT-solver were obtained when trying to break one round of AES Part. 1

	Number of added key bits			
	32	30	28	26
Initial bits Time [s]	278	21.5	219	69.2
Final bits Time [s]	25.2	12.2	17	714

In the first three cases, the SAT-solver CryptoMiniSat calculated the missing leading key bits faster, and in the last attempt, it took less time to get the remaining 102 trailing key bits than the 102 trailing key bits.

Table 4. The results of the CryptoMiniSat SAT-solver were obtained when trying to break one round of AES Part. 2

	Number of added key bits			
	24	22	18	16
Initial bits Time [s]	28.9	–	–	6564
Final bits Time [s]	75.6	448	–	–

Moving on to the second stage of the study, 24, 22, 18, and 16, respectively, the initial or final key bits were added. The results of this experiment are presented in the Table 4. As we can see in the conducted experiment, the SAT-solver calculates the remaining bits of the key faster with the initial bits added. Additionally, in neither of the two variants of this study did CryptoMiniSat calculate the missing bits, and in the last attempt, it took over 18 hours to get the final bits of the key. In the variant with 16 trailing key bits added, the SAT-solver failed to provide the trailing key bits within the time limit.

We also performed tests by adding a smaller and smaller number of initial (final) key bits, but the SAT-solver CryptoMiniSat was unable to evaluate the remaining key bits within 12 hours.

In the next study, an attempt was made to break one round of AES-128 by adding key bits in such a way that 100, 102, 104, 106, 108, and 110 bits were removed from the middle of the key bit sequence, leaving the AES-128 cipher in the first round respectively 28, 26, 24, 22, 20, and 18 added bits. The time limit set for the calculations by the SAT-solver CryptoMiniSat was set to 12 hours. The obtained results are presented in the Table 5. In the case of a one-round AES-128 attack with key bits added as described above, it was successful even when only 18 key bits were added – the SAT-solver successfully found the evaluation of the remaining key bits in the given time. Continuing the test and adding fewer and fewer key bits (in DIMACS format) in the formula encoding the AES-128 operation in the first round, CryptoMiniSat's SAT-solver failed to calculate the remaining key bits within the time limit of 12 hours.

Table 5. The results of the CryptoMiniSat SAT-solver were obtained when trying to break one round of AES Part. 3

	The number of middle key bits removed					
	100	102	104	106	108	110
Time [s]	437	135	13566	1774	322	9740

The conducted research shows that adding the initial 16 bits of the key to the formula encoding the AES-128 action in the first round causes some SAT-solvers selected for this task to calculate the remaining key bits in less than 12 hours. On the other hand, adding 16 final or 8 initial and 8 final (version with 112 middle key bits cut) key bits to this formula did not allow the SAT-solver CryptoMiniSat to find the evaluation of the remaining key bits within the set time limit.

6. Conclusion

We developed a boolean encoding of the AES cipher and obtained a CNF fully equivalent to the operation of AES, which we confirmed with tests. We transformed the selected-plaintext cryptanalysis problem of the AES algorithm with a 128-bit key into a SAT problem. Experiments have shown that the problem encoded in this way can be solved using selected SAT solvers in a reasonable time in a version reduced to one round with the addition of at least 16 key bits. In future work, we plan to reduce the number of clauses in the CNF describing the operation of the AES cipher and investigate the capabilities of SAT solvers for the newly formulated SAT problem.

References

[1] Cook, S.A. (1971). The complexity of theorem-proving procedures. *Proceedings of the Third Annual ACM Symposium on Theory of Computing - STOC -71*.

[2] Cook, S., & Mitchell, D. (1997). Finding hard instances of the satisfiability problem: A survey. *Satisfiability Problem: Theory and Applications*, 1-17.

[3] Steingartner, W., Polakova, A., Praznak, P., & Novitzka, V. (2015). Linear logic in computer science. *Journal of Applied Mathematics and Computational Mechanics*, *14*(1), 91-100.

[4] Steingartner, W., Radakovic, D., Valkosak, F., & Macko, P. (2016). Some properties of coalgebras and their role in computer science. *Journal of Applied Mathematics and Computational Mechanics*, *15*(4), 145-156.

[5] Andrzejczak, M., & Dudzic, W. (2019). SAT attacks on ARX ciphers with automated equations generation. *Infocommunications Journal*, *11*(4), 2-7.

[6] Dwivedi, A.D., Kloucek, M., Morawiecki, P., Nikolic, I., Pieprzyk, J., & Wójtowicz, S. (2017). SAT-based cryptanalysis of authenticated ciphers from the caesar competition. *Proceedings of the 14th International Joint Conference on E-Business and Telecommunications*.

[7] Dudzic, W., & Kanciak, K. (2020). Using SAT solvers to finding short cycles in cryptographic algorithms. *International Journal of Electronics and Telecommunications*, *66*(3), 443-448.

[8] Morawiecki, P., & Srebrny, M. (2013). A SAT-based preimage analysis of reduced Keccak hash functions. *Information Processing Letters*, *113*(10-11), 392-397.

[9] Lee, H., Kim, S., Kang, H., Hong, D., Sung, J., & Hong, S. (2016). Efficient differential trail searching algorithm for ARX block ciphers. *Journal of the Korea Institute of Information Security and Cryptology, 26*(6), 1421-1430.

[10] Sun, L., Wang, W., & Wang, M. (2021). Accelerating the search of differential and linear characteristics with the SAT method. *IACR Transactions on Symmetric Cryptology*, *1*, 269-315.

[11] Courtois, N.T., & Pieprzyk, J. (2002). Cryptanalysis of block ciphers with overdefined systems of equations. *Lecture Notes in Computer Science*, 267-287.

[12] Gwynne, M. (2014). *Attacking AES via Sat*. Academia.edu. https://www.academia.edu/2594954/Attacking_AES_via_SAT

[13] Stachowiak, S., Kurkowski, M., & Soboń, A. (2021). SAT vs. substitution boxes of DES like ciphers. *2021 IEEE 30th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*.

[14] Stachowiak, S., Kurkowski, M., & Soboń, A. (2021). SAT-based cryptanalysis of Salsa20 cipher. *Progress in Image Processing, Pattern Recognition and Communication Systems*, 252-266.

[15] Chowaniec, M., Kurkowski, M., & Mazur, M. (2018). New results in direct SAT-based cryptanalysis of DES-like ciphers. In: *Advances in Soft and Hard Computing*, 282-294.

[16] Soboń, A., Kurkowski, M., & Stachowiak, S. (2019). Towards complete SAT-based cryptanalysis of RC5 cipher. *2019 IEEE 15th International Scientific Conference on Informatics*, 397-402.

[17] Dworkin, M.J., Barker, E.B., Nechvatal, J.R., Foti, J., Bassham, L.E., Roback, E., & Dray jr., J.F. (2001). Advanced Encryption Standard (AES). NIST. https://www.nist.gov/publications/advanced-encryption-standard-aes

[18] Ashokkumar, C., Giri, R.P., & Menezes, B. (2016). Highly efficient algorithms for AES key retrieval in cache access attacks. *2016 IEEE European Symposium on Security and Privacy*.

[19] Davis, M., Logemann, G., & Loveland, D. (1962). A machine program for theorem-proving. *Communications of the ACM, 5*(7), 394-397.

[20] Soboń, A., Kurkowski, M., & Stachowiak, S. (2020). Complete SAT based cryptanalysis of RC5 cipher. *Journal of Information and Organizational Sciences, 44*(2), 365-382.

[21] Ferguson, N., Kelsey, J., Lucks, S., Schneier, B., Stay, M., Wagner, D., & Whiting, D. (2001). Improved cryptanalysis of Rijndael. *Fast Software Encryption*, 213-230.

[22] Gilbert, H., & Minier, M. (2000). A Collision Attack on 7 Rounds of Rijndael. *The Third Advanced Encryption Standard Candidate Conference*, 230-241.

[23] Biham, E., Dunkelman, O., & Keller, N. (2005). Related-key boomerang and rectangle attacks. *Lecture Notes in Computer Science*, 507-525.

[24] Kim, J., Hong, S., & Preneel, B. (2007). Related-key rectangle attacks on reduced AES-192 and AES-256. *Fast Software Encryption*, 225-241.

[25] Biryukov, A., Khovratovich, D., & Nikolic, I. (2009). Distinguisher and related-key attack on the full AES-256. *Advances in Cryptology – CRYPTO 2009*, 231-249.

[26] Biryukov, A., Dunkelman, O., Keller, N., Khovratovich, D., & Shamir, A. (2010). Key recovery attacks of practical complexity on AES-256 variants with up to 10 rounds. *Advances in Cryptology – EUROCRYPT 2010*, 299-319.

[27] Biryukov, A., & Khovratovich, D. (2009). Related-key cryptanalysis of the full AES-192 and AES-256. *Advances in Cryptology – ASIACRYPT 2009*, 1-18.

[28] Committee, SAT Competition Organization (2022). *The International SAT Competition Web Page*. SAT Competitions. http://satcompetition.org/