University of Rennes, ISTIC
Supervisor: Patrick Derbez

# Research project report

Samuel Mauger, Erwann Le Juif,
William Pignon, Hilan Meyran, Joffrey Hauw

February-May 2024

**Abstract**

This research delves into the security of the Advanced Encryption Standard (AES) cryptographic protocol. Despite its robustness, vulnerabilities exist, particularly in its round-reduced versions. Our study focuses on Meet-In-The-Middle Attacks, analyzing their application to AES with reduced rounds. Building upon prior work by Patrick Derbez (2012), we implement approach for automatically searching for attacks on round-reduced AES and explore its practical implications. This report has been written to allow for anyone to implement the presented program.

# Contents

# 1 Introduction

The research project we had to work on was about the AES protocol, which is used in our everyday life to secure our communications and make them private. AES is a robust protocol which cannot be broken in a reasonable time (as of today), the best complexity found in an full key-recovering attack is $2^{126.0}$ for AES-128, $2^{189.9}$ for AES-192 and $2^{254.3}$ for AES-256. Part of the research, called Meet-In-The-Middle Attacks, is focused on round-reduced protocols.

AES is very resilient against statistical attacks. This particular security property is achieved by a complete diffusion after 2 rounds, altering a single bit of the input results in a completely different output. Consequently, redundant bits are effectively obscured. Moreover, good differential properties are achieved for lower bounds on 4 rounds. Meaning that even within this limited scope, the efficiency of attacks remains significantly low, ensuring a decent level of security. In some cases, the full AES algorithm in 10 rounds which could seems to be overkill and we can have the idea to use a reduced round version of the protocol to win in efficiency on application where it is deemed necessary.

Based on this paper "*Automatic Search of Attacks on round-reduced AES and Applications*" written by Charles Bouillaguet, Patrick Derbez, and Pierre-Alain Fouque and published in 2012 [1]. We aimed to re-implement program that can find algorithms to solve equations systems. **For this work, we consider only AES with key of size 128 bits.**

## 1.1 Choice of the language

The developed program performs a lot of algebras operations and matrix computation, meaning it is very compute intensive. What this means is that we need a language with low level performance and high level abstractions to make developing easier for us. While it was proposed to do the project in C++, we decided against it because of both a lack of experience and a fear of being hindered by memory-related problems(eg: memory leakage). So we decided to go with the **Rust** language because it meets every criteria we imposed ourselves. While most of us had no prior experience with said language, we had course alongside this research project to bring us up to speed on Rust.

# 2 Literature Review

The paper[1] proposes two tools that follow different attack approaches:

- The first tool, uses the *guess and determine* method, involves guessing a portion of the key and using this known part to determine the rest. However, it is described as restricted and fails to take into account some important properties of the S-Box.
- The second tool, designed for *meet-in-the-middle* attacks, is highlighted as a more effective approach for finding attacks according to the paper.

Thanks to theses tools, researchers managed to improve some existing attacks in the cryptanalysis literature, at the time.

# 3 Methodology

We will first come back on AES basics needed to understand this project (advanced reader can skip 3.1 section).
AES is a cryptosystem adopted by the NIST in 2001. The algorithm works on block of fixed size and use keys of size: 128, 192 or 256 bits.

## 3.1 Description of a matrix in AES

AES works on a $4 \times 4$ matrix ordered by columns, where each element represents a byte. The algorithm begins with a step called Key Expansion (also known as the key schedule), which generates the round keys (10 for the 128-bit key size version). We denote the master key as $k_0$ and the derived keys as $k_i$.

Following this key generation, the AES algorithm performs rounds, they are composed of the steps listed below :

- **SubBytes (SB)**: This step replaces each byte with a value from a substitution table (S-Box). This introduces non-linearity into the algorithm, protecting against linear cryptanalysis by ensuring that a change in a single bit of the key results in a non-linear change in the output.
- **ShiftRows (SR)**: This step involves shifting the bytes in each row to the left by an amount corresponding to the row number. This step provides diffusion, which helps protect against differential cryptanalysis by ensuring that differences in the plaintext result in complex, widespread differences in the ciphertext.
- **MixColumns (MC)**: In this step, each column's four bytes are combined to further enhance diffusion, spreading the influence of each byte over the entire state.
- **AddRoundKey (ARK)**: Each byte of the state is combined with a byte of the round key using the XOR operation, integrating the round key into the state.

The previous described steps could be seen in the 1 :

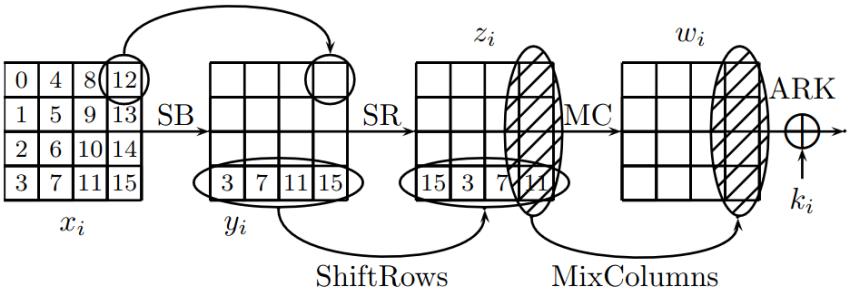


Figure 1: One AES round

## 3.2 AES as an equations system

In 2002, the researchers Murphy and Robshaw[2] demonstrated that we could represent AES as algebraic operations in the field $GF(2^8)$. This discovery allowed AES to be expressed as a very sparse, over-determined multivariate quadratic system. According to this, in our representation, AES equations will be represented by an equations system. These equations have all their right part to 0, and so later it will not be represented.

Key-Expansion equations:

$$KS_i : \begin{cases} k_i[j] + k_{i-1}[j-4] + k_{i-1}[j] = 0, \\ k_i[0] + k_{i-1}[0] + S(k_{i-1}[13]) + RCON_i = 0 \\ k_i[1] + k_{i-1}[1] + S(k_{i-1}[14]) = 0 \\ k_i[2] + k_{i-1}[2] + S(k_{i-1}[15]) = 0 \\ k_i[3] + k_{i-1}[3] + S(k_{i-1}[12]) = 0 \end{cases} , \quad j = 4, \ldots, 15 \tag{1}$$

And one round is represented as this equations system :

$$Ri : \begin{cases} y_i + S(x_i) = 0 \\ w_i + \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \times \begin{pmatrix} yi[0] & yi[4] & yi[8] & yi[12] \\ yi[5] & yi[9] & yi[13] & yi[1] \\ yi[10] & yi[14] & yi[2] & yi[6] \\ yi[15] & yi[3] & yi[7] & yi[11] \end{pmatrix} = 0 \\ x_{i+1} + w_i + k_{1+i} = 0 \end{cases} \tag{2}$$

We can denote :

- $x_i$ : the step before SubBytes
- $y_i$ : the step between SubBytes and ShiftRows
- $w_i$ : the state after MixColumns
- $k_i$ : the subkey used for each operation

We will represent the 256 elements used in AES by the finite field $F_{2^8}$ and the Sub-Bytes operation (substitution SBOX) by the function :

$$S : F_{2^8} -> F_{2^8}$$

In the next sections, it may seem unusual or outright wrong to apply linear algebra to the matrix representing AES equations, given that the SubBytes operation is non-linear. However, this is not a significant issue because of the used approach. To remind, we don't directly solve the system. But instead, we will employ heuristics to construct an algorithm that solves it as efficiently as possible. Thus, the S-Box can be treated as a black box, allowing us to perform linear algebra operations on the system.

## 3.3 Parsing Input Equations

In our approach, the equations, as mentioned earlier, are given as input and form a system describing the AES algorithm. This system of equations will be provided to our program in the form of a file, in order to be parsed and to extract the corresponding matrix from it.

### 3.3.1 Description of parser

The objective of the parser is to create, from the provided system of equations as input, the corresponding matrix. This matrix contains the coefficients of the terms present in the system of equations, with the variable names on the columns (accessible through a hashmap), and the equations represented on the lines. Below are the main steps of our parser:

- Search for section name
- Iterate into each line, if it's an equation : create new line entry
  - Iterate into each term
    - extract name of variable, if it's new variable : create new column entry
    - extract coefficient
- Stop at the end of the file or if a new section name is encountered.

The parser ensures that each equation contains at most five terms and that each term is present in at most five equations.

### 3.3.2 Description of grammar

To parse our file accordingly, we have established the following rules:

- Definition of a section:

```
--mysection:
```

- Definition of an equation:

```
TERM + TERM + TERM
```

- Definition of a term:

```
NAME * COEFFICIENT

NAME

COEFFICIENT
```

- Possibility of commenting in the file

```
# my commentary on line or at the end of line

# my commentary inside equation #
```

In our output, we get a matrix which is represented in our code in the form of number of rows, number of columns, and a single vector of data (As in Rust with 2D vectors caching cannot be done efficiently due to the two layers of indirection).
The second output of our parsing is a dictionary(Hashmap) that will establish the correspondence between the variables and the columns to which they are associated.

## 3.4 Operations on the matrix

This section will present the operations of AES as described in the NIST AES specification [3]. While our code has been build generically to allow other fields that AES one. We remain focus on it.

### 3.4.1 Addition

In a finite field of characteristic 2, such as $GF(2^8)$, both addition and subtraction correspond to a logical XOR. Meaning that, in our case the addition and substraction between two bytes is a XOR represented by $\oplus$. If we represent three numbers A, B and C as a byte such as $A \oplus B = C$. Then we have:

$$\begin{aligned} A &= \{a_7, a_6, a_5, a_4, a_3, a_2, a_1, a_0\}, \\ B &= \{b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0\}, \\ C &= \{c_7, c_6, c_5, c_4, c_3, c_2, c_1, c_0\} \text{ such as } c_i = a_i \oplus b_i. \end{aligned}$$

Here an example, in a polynomial form :
$A = x^4 + x^3 + x^2 + 1$
$B = x^5 + x^3 + x^2 + 1$
give us
$C = x^5 + x^4$, where $x^i$ represents the bit number $i$

### 3.4.2 Multiplication

The multiplication is also in the finite field $GF(2^8)$, so if we multiply an element by another, the result will stay into the field (as multiplication is an internal operation). Its representation is black point as $\cdot$ (the AND operation). The irreducible polynomial of the AES is $(x^8 + x^4 + x^3 + x + 1)$ that will be denote by $m(x)$ in the rest of the section.
The operation of multiplication is done by multiplying each polynomial term by the other and applying the modulo $m(x)$. As for the addition, we take two numbers A and B :
$A = x^6 + x^4 + x^2 + x + x^1$
$B = x^7 + x + 1$
And have C such as :

$$\begin{aligned} A \cdot B &= x^{13} + x^{11} + x^9 + x^8 + x^7 + x^7 + x^5 + x^3 + x^2 + x + x^6 + x^4 + x^2 + x + 1 \\ &= x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1 \\ C &= x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1 \\ &\quad \bmod (x^8 + x^4 + x^3 + x + 1) \\ &= x^7 + x^6 + 1 \end{aligned} \tag{3}$$

### 3.4.3 Invert

As the multiplication is associative, 1 is the identity. We can put the following formula:

$$b(x)a(x) + m(x)c(x) = 1$$

From this formula we can find the multiplicative inverse of $b(x)$ named $b^{-1}(x)$ (from the extended Euclidean algorithm)

## 3.5 Linearly Occurring Variables

After the parsing of our equation system representing the first round of AES, we get a matrix of 117 variables (represented by the columns) and 64 equations (represented by the lines).

Our initial system ($E$), is not completely linear (in the sense of the variables involved):
Some variables appear linearly and under the SBox and some others appear only linearly. If a variable $x$ occurs only linearly, it can be eliminated from all the equations except one by taking linear combinations of the equations. Then after doing this we can take apart the equation left containing $x$ to obtain a new simplified system $E'$. The apparition order of the linear variables doesn't change the properties of the computed algorithm. Then, we can safely run the search procedure on $E'$ and add at the end the removed equations in the algorithm.

In our system, the more equations we have, the more lines we will have in the matrix. Therefore, by eliminating linear variables, we reduce the size of the matrix permitting to have a better computation time.

To remove linear variables, we will first look for variables that do not appear under the SBox. For each of these, we place them on the left side of the matrix and perform Gaussian elimination on the variable column. We will obtain only zeros under the pivot, the variable is a combination of others. We can now remove the equation by removing the corresponding line in the matrix.

**Algorithm 1** remove alone variable

```
function DELETE_ALONE_VARIABLES
    variables_alone ← get_not_sboxed_variables
    while variables_alone ≠ ∅ do
        x ← variables_alone.pop()
        remove x from the matrix (scale and delete the row)
        update variables_alone to contain only the remaining alone variables
    end while
end function
```

After deleting the linear variables, we check that an "Sboxed" variable does not exist linearly with its variable. Then if it is the case we can remove the equation after writing the variable under a linear combination of others.

**Algorithm 2** removal of linear variables under the sbox

```
while variable_sboxed ≠ ∅ do
    (x, sx) ← variable_sboxed.pop()
    matrix ← get_matrix_generated_by([x, sx])
    if rank(matrix) = 1 then
        delete_row(0)
    end if
end while
```

### 3.6 Recursive construction

To construct a Solver we will use a recursive construction. This construction is possible because of the properties of the equations in the system.
The equations are "separable" in the meaning that by denoting $X$, the set of the variables of the $f$ equation. We can then write $f = f_1 + f_2$ with $X = X_1 \cup X_2$ and $f_1 \in V(X_1), f_2 \in V(X_2)$, $f_1$ is a linear combination of variables of $X_1$ (respectively for $f_2$).

We can easily extract $E$ of our system of equations, a subsystem with only the variables of $X_1$ by computing the intersection $E \cap V(X_1)$ let us denote it by, $E(X_1)$. The above property on the equations permit us to consider two Solvers $A_1$ and $A_2$ respectively finding all the solutions of $E(X_1)$ and $E(X_2)$. A solution of $E$ is a solution of $E(X_1)$ and $E(X_2)$ (converse not true), then we found all the solutions of $E$.
The only step left is to test all elements in $S_1 \times S_2$ , the Cartesian product of the solutions of $A_1$ by solutions of $A_2$ on equations of $E$.

With this we can make a recursive algorithm to find all solutions of E. At this step, we can summarize the program as in 3

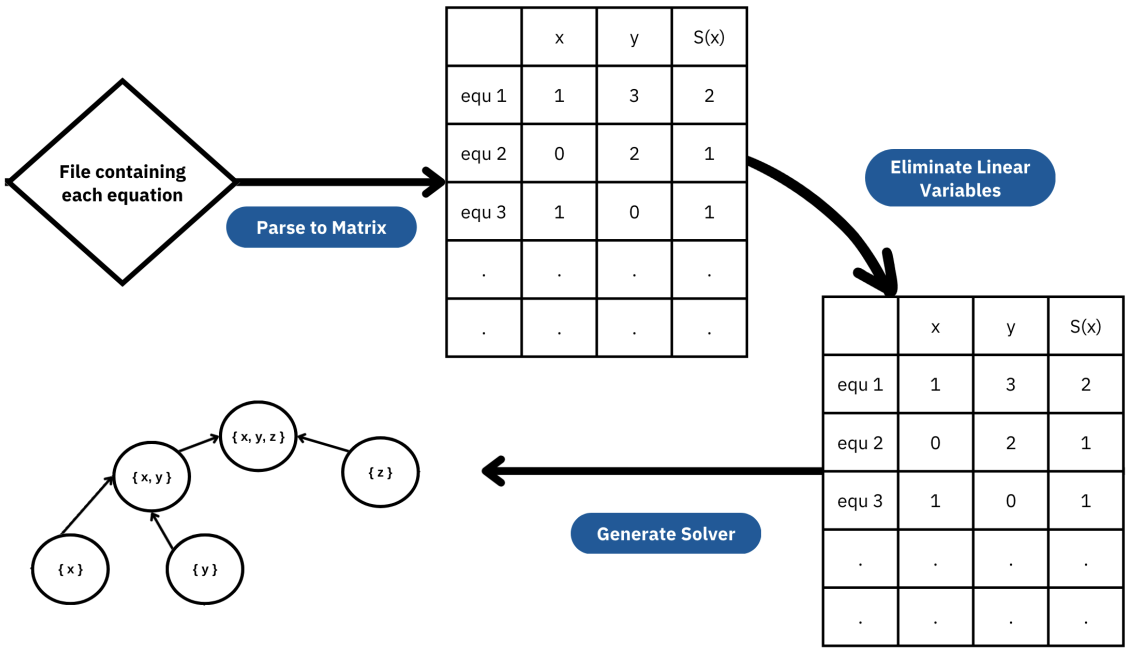


Figure 2: Program Workflow

## 3.7 Base case

To build an efficient solver, we break down the problem into smaller and smaller subsystems and recombine them. This recombination forms a tree where each leaf is a base solver and each node is a solver with two children.

Each base solver is associated with a variable $x \in X$ that cannot be further decomposed and is handled in two ways:

- Either we cannot determine if the variable $x$ is constrained by the equation, in which case the set of solutions is the field $F_{2^8}$
- Or we find at least one equation involving only $x$ and $S(x)$. In this case, $x$ can only take a few possible values.

Our base solver have the following properties:

- $\log_{256}(\text{Number of solutions}) = 1$
- Time complexity $= 2^8$
- Memory Complexity $= 1$

## 3.8 Properties of a Solver

To compare two solvers, we need to evaluate their properties, primarily by examining their time complexity.

### 3.8.1 Time Complexity

These properties are computed at each fusion. The time complexity of a Solver is given by this formula that is an approximation:

$$T(A_1 \bowtie A_2) = max\{T(A_1), T(A_2), O(A_1 \bowtie A_2)\}$$

We see in this formula that its number of solutions cannot be greater than its time complexity. Then we define the number of solutions.

### 3.8.2 Number of solutions

The number of solutions is given by the following heuristic:

$$\log_{256} O(\mathbb{E}(X)) \approx \mid X \mid - \dim \mathbb{E}(X)$$

At each merge, we compute the number of solutions of the new solver. We denote $X$ the variables of the new solver (the union of the variables of their sons). Let $Y$ the variables of the equations, then we sort the matrix $E$ by $Y - X$ and $X$ and echelon on $Y - X$. Then we count the number of zero rows on $E(Y - X)$.

**Algorithm 3** number_solutions

```
1: function NUMBER_SOLUTIONS(vars_set)
2:     not_vars_set ← matrix.get_all_variables() − vars_set
3:     matrix.scale_on(not_vars_set)
4:     if |vars| = number variables in matrix then
5:         return 0
6:     end if
7:     nb_eq ← get_number_of_zero_rows_from_bottom(|vars|)
8:     return |vars| − nb_eq
9: end function
```

### 3.8.3 Memory Complexity

The memory complexity of a Solver is defined by the following formula:

$$M(A_1 \bowtie A_2) = \max\{M(A_1), M(A_2), \min(\mathcal{O}(A_1), \mathcal{O}(A_2))\}$$

## 3.9 Comparison of Solvers

At some point we need to compare solvers to decide which is the most efficient, to do so the paper gave us two solutions to compare algorithms.

### 3.9.1 Compare 1

*compare1* is the first comparison showed and is defined by:

$$\mathcal{A}_1 \geq_1 \mathcal{A}_2 \iff \begin{cases} V(\mathcal{A}_1) = V(\mathcal{A}_2) \\ T(\mathcal{A}_1) \leq T(\mathcal{A}_2) \end{cases}$$

It is a quasi-order as we only compare time if they have the same set of variables.

### 3.9.2 Compare 2

The problem with *compare1* is that while it is a quasi-order, we often end up with incomparable solvers due to them not having the same exact set of variables. This problem tries to be solved by defining *compare2*.
The idea behind *compare2* is that if a solver find more solutions on a subset of variables in less time then the solver is better.

$$\mathcal{A}_1 \geq_2 \mathcal{A}_2 \iff \begin{cases} T(\mathcal{A}_1) \leq T(\mathcal{A}_2) \\ V(\mathcal{A}_1) \supseteq V(\mathcal{A}_2) \\ O(\mathcal{A}_1) \leq O(\mathcal{A}_2) \end{cases}$$

The problem is that this comparison makes the fusion of two solvers incompatible.

$$\mathcal{A}_1 \geq_2 \mathcal{A}_2 \not\Longrightarrow \mathcal{A}_1 \bowtie \mathcal{A}_3 \geq_2 \mathcal{A}_2 \bowtie \mathcal{A}_3$$

This incompatibility makes the second comparison unusable in the search algorithms.

## 3.10 Finding the best solver

### 3.10.1 Exhaustive Search

One approach to identify the optimal solver for our equation system is through an exhaustive search. Although this method promises to discover the **best solver**, it comes at the expense of computation time, as it necessitates enumerating every conceivable combination of solvers.

Moreover, employing an exhaustive search with an increasing number of AES rounds poses challenges. As the number of rounds grows, so do the variables, equations and base solvers. Consequently, the scalability of an exhaustive search diminishes rapidly under such circumstances.

In our code this algorithm is represented by the following pseudo-code.

**Algorithm 4** Exhaustive search

```
1: function EXHAUSTIVE_SEARCH(x, time_complexity)
2:     g ← generate_all_base_solver(x)
3:     p ← generate_all_algo_pairs(g)
4:     while p ≠ ∅ do
5:         (a1, a2) ← choose_pair_at_random(p)
6:         remove_pair_from_p((a1, a2))
7:         c ← merge_two_algo(a1, a2, x)
8:         if c.get_time_complexity() ≤ time_complexity then
9:             update_queue(g, p, c)
10:        end if
11:    end while
12:    return g
13: end function
```

**Algorithm 5** Update Queue

```
1: function UPDATE-QUEUE(G, P, A)
2:     for all A' ∈ G do
3:         if A' does not dominate A then
4:             Continue
5:         else
6:             return
7:         end if
8:     end for
9:     G ← Keep all algo that are better or equal to A
10:    G.add(A)
11:    P ← Keep all pair that are better or equal to A
12:    P ← Form new pairs with the algos such as the variables of one are not a subset of the other
       return (G, P)
13: end function
```

This exhaustive search consumes memory really fast, even for a one round AES, so it doesn't permit us to generate the optimal solver. We then try to create a new one with heuristics.

### 3.10.2 Adding heuristics

The heuristic presented is obtained by generating pairs called base strategies. These pairs are 16 and are formed as $(X\_0[0, a], K\_0[0, a])$.

**Algorithm 6** Generate all pairs such as (X_0[0,a], K_0[0,a])

```
1: function EXHAUSTIVE_SEARCH(x, time_complexity) p ←
2:     for a1 ∈ base_solvers do
3:         for a2 ∈ base_solvers do
4:             if a1.var such as X_0[0, a] AND a2.var such as K_0[0, a] then
5:                 p.insert(a1, a2)
6:             end if
7:         end for
8:     end for
9: end function
```

We add this heuristics in a new algorithm and try to generate all Solver with number of solutions growing. Initially, the algorithm generates all possible base solvers based on the equations variables. Next, we generate all pairs following the heuristics. The algorithm merges each pair to create new solvers, then adds these newly created solvers back into the set of base solvers. We initialize the search by generating pairs of solvers to merge from the base solvers. Then it starts from these two sets to generate all solvers containing number of solutions at most 1 then 2, etc. When a solver with 20 variables is generated we stop.

**Algorithm 7** Heuristics search

```
1: function HEURISTICS_SEARCH(x, time_complexity)
2:     g ← generate_all_base_solver(x)
3:     p ← generate_base_strategy_pairs(g)
4:     Delete from g the variables appearing in p
5:     Merge the pairs to create solvers
6:     Add new solvers to g
7:     p ← generate_all_algo_pairs(g)
8:     for max_sol in 1..20 do
9:         Reset g and p
10:        while p ≠ ∅ do
11:            (a1, a2) ← choose_pair_at_random(p)
12:            remove_pair_from_p((a1, a2))
13:            c ← merge_two_algo(a1, a2, x)
14:            if c.number_solutions ≤ max_sol then
15:                update_queue(g, p, c)
16:            end if
17:        end while
18:        Get from g Solver with 20 variables
19:        If exists return it
20:    end for
21:    return g
22: end function
```

This algorithm is not growing memory consumption fast but doesn't permit to generate a Solver in a reasonable time.

# 4 Results

The output of our program is a tree graph that represents a solver, obtain with random search.

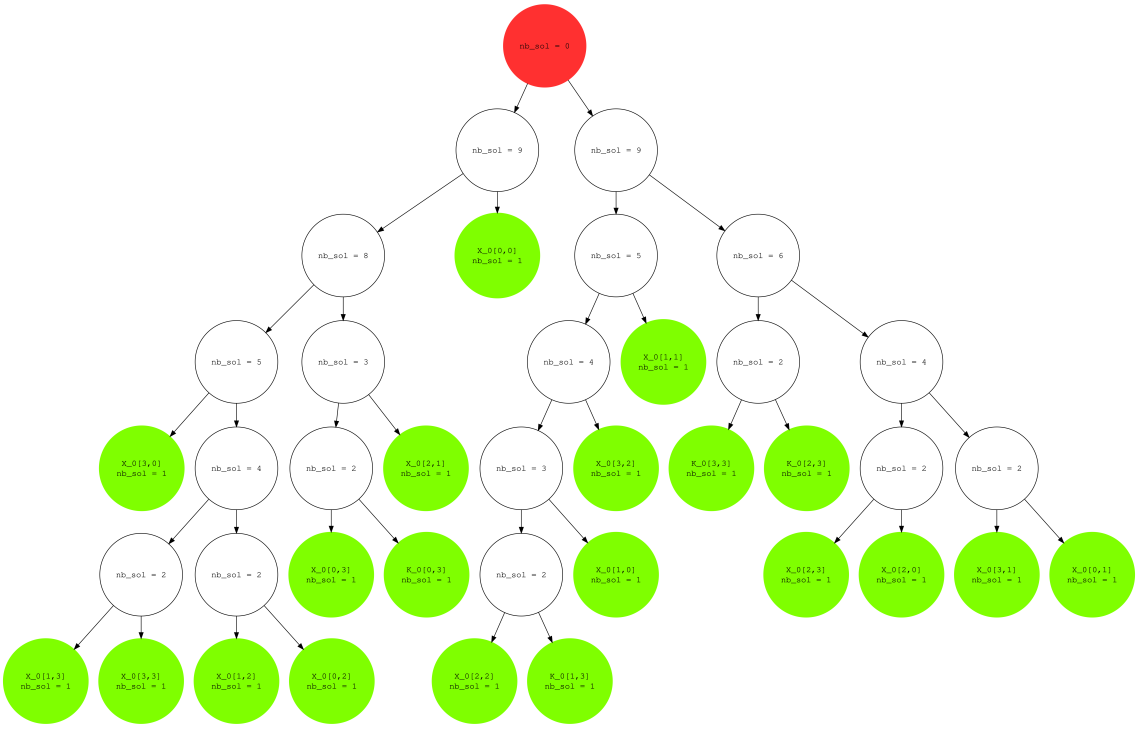


Figure 3: Output Graph

We test our program on only one round of AES in its specific field with a random search. It could be deemed interesting to improve the search algorithms and to modify the program to make it works for more rounds and maybe to combine it with the work of our supervisor on finding the solutions of AES equations on several rounds and other cryptography algorithms than AES.

# 5 Conclusion

To conclude, the results in the research paper [1] have demonstrated the possibility of finding and implementing attacks more quickly than if they were carried out directly by a human. This advancement, however, presents two major limitations: it requires human intervention to find an optimal solution. And the second is that for some given problem when only one solution is sufficient, it leads to generate solver to costly in terms of time and resources because the program only generate solver finding the entire set of solutions to the system of equations.

## 5.1 Acknowledgments

We want to thank Mr. Patrick Derbez for the knowledge he as shared with is us on the subject and for always being available when needed.

# References

[1] C. Bouillaguet, P. Derbez, and P.-A. Fouque. Automatic search of attacks on round-reduced aes and application. 2012.

[2] S. Murphy and M. J. B. Robshaw. Essential algebraic structure within the aes. in moti yung, editor, crypto, volume 2442 of lecture notes in computer science. 2442:1–16, 2002.

[3] NIST. Announcing the advanced encryption standard (aes). 2001.