Ca' Foscari
University
of Venice

Master's Degree programme

In Computer Science

Final Thesis

Using SAT solvers and brute force approach to recover AES key from partial key schedule images

Supervisor
Riccardo Focardi

Graduand
Nicolas Pietro Martignon
870034

Academic Year
2021 / 2022

Abstract

Recovering an AES key exploiting the redundancy of key material inherent in the AES key schedule is a topic widely discussed in the scientific literature. Specifically, there are many papers that analyze the aforementioned problem starting from a corrupt key schedule, this scenario is very frequent if a cold boot attack is carried out in which a corrupt dump of the ram memory is performed. In these articles, different techniques are used to derive the AES key including the use of SAT solver and MAX SAT solver, tools that aim to solve the problem of Boolean satisfiability. In this thesis, on the other hand, we want to analyze a more general case, i.e., starting from a partial key schedule and obtaining its AES key. Different techniques will be analyzed and combined including the use of SAT solver tools, and brute force algorithm, with the aim of minimizing the time for key recovery.

Keywords: Key schedule · AES · SAT solver · Brute force · Partial information

Contents

1 Introduction **6**

2 AES and SAT **7**
2.1 AES 7
2.2 SAT 13

3 Previous work **16**

4 Preliminaries **18**
4.1 Creation of input file 18
4.2 Hardware and software used 21

5 Key recovery on partial key schedule **22**
5.1 Banning solution approach 23
5.2 Brute force approach 26
5.3 Mixed approach (brute force and banning solution) 34
5.4 Random bit instancing 37

6 Comparison of different technique **40**
6.1 Comparison of all techniques in scenarios with multiple solutions . . . 40
6.2 Comparing single SAT solver instance VS brute force approach in scenarios with max one solution 41
6.3 Comparing our best approach VS Incremental SAT solver 43

7 Conclusion **45**

A Source code **49**
A.1 Python scripts to generate CNF files 49
A.2 Python scripts to conduct test on all approaches 55

List of Figures

2.1 AES Encryption Process 9
2.2 Cipher Key 10
2.3 RotWord transformation 10
2.4 S-BOX 11
2.5 SubByte transformation 11
2.6 SubByte transformation + Wi-4 + Rcon = Wi 11
2.7 First round key's word 12
2.8 Wi-1 + Wi-4 = Wi 12
2.9 Cipher key and round key 1 12
2.10 Key schedule image 13

4.1 Partial information know 19

5.1 Sub-problem generated 26
5.2 Random sub-problem generated 37

List of Tables

5.1 Run-time statistic for banning solution 24
5.2 Run-time statistic for brute force on 2 bit I 29
5.3 Run-time statistic for brute force on 2 bit II 29
5.4 Run-time statistic for brute force on 4 bit I 29
5.5 Run-time statistic for brute force on 4 bit II 29
5.6 Run-time statistic for brute force on 6 bit I 30
5.7 Run-time statistic for brute force on 6 bit II 30
5.8 Run-time statistic for bruteforce on 8 bit I 30
5.9 Run-time statistic for bruteforce on 8 bit II 30
5.10 Run-time statistic for Mixed approach on 2 bit 35
5.11 Run-time statistic for Mixed approach on 4 bit 35
5.12 Run-time statistic for Mixed approach on 6 bit 35
5.13 Run-time statistic for Mixed approach on 8 bit 35
5.14 Run-time statistic for Random bit instancing 38

6.1 Run-time statistic for single SAT solver instance (Kissat) 42
6.2 Run-time statistic for single SAT solver instance (Cryptominisat) . . 43

List of Algorithms

1 Banning solution pseudocode 23
2 Brute force pseudocode 28
3 Brute force and banning solution pseudocode 34
4 Random bit instancing pseudocode 38

Chapter 1

Introduction

The remanence of data in the RAM memory after the computer is turned off is now a well-known phenomenon and very studied in the scientific literature.
The rapid corruption of data in RAM with the passage of time from turning off or restarting the device, has given rise to many new studies on error correction, especially in the cryptographic field.
Many articles exploiting the redundancy of cipher key information present in the AES key schedule have formulated various techniques and increasingly effective error correction algorithms. Frequently there has been the use of SAT solvers, powerful programs that solve the problem of Boolean satisfiability.
This paper on the other hand wants to analyze a more general case, that is the case of knowing partial information and arriving to a total information.
More precisely, the partial information concerns the AES key schedule, where some bits are known and other bits are not, knowing in full all the bits of the key schedule will consequently also allow to obtain the cryptographic key used to generate it. Different techniques will be analyzed and combined, including the use of SAT solvers and brute force algorithms, with the aim of minimizing the time required to obtain the cryptographic key.
The thesis is organized as follows: first of all, basic notions on AES and SAT are introduced, these are necessary for the correct understanding of the following paper. This introduction also allows us to understand how these 2 elements AES and SAT can be used together.
Subsequently, in the following chapter the previous works are retraced, concerning the permanence of data in RAM and on the correction of errors in a corrupt AES key schedule. Chapter 4 introduces preliminary information to the next chapter, which introduces the work done in this paper.
In chapter 5, therefore, the various techniques and approaches developed are discussed, also many tests are conducted to have run time statistic for each technique. Chapter 6 compares all the techniques developed with the aim of decreeing the best technique for each circumstance.

Chapter 2

AES and SAT

In this chapter we will briefly introduce useful concepts for the correct understanding of the following thesis. This chapter can therefore be excluded from reading by those who have extensive knowledge of how AES and SAT solving works.

2.1 AES

Concepts introduced in this section are taken from [1, 2, 3].
Particularly, the images in this section are inspired by the following video on AES [4].

AES (Advanced Encryption Standard), better known as Rijndael is a block symmetric cryptographic algorithm (sender and receiver use the same encryption key), was developed in 1998 to address the limitations and vulnerabilities of the algorithm used up to now, namely DES (Data Encryption Standard). AES was adopted by the US federal government, and encryption using AES has become the industry standard for data security. AES is available in 128-bit, 192-bit, and 256-bit key implementations.
AES uses substitution and permutation techniques, with a variable number of rounds (cryptographic processing) depending on the length of the encryption key chosen.
According to the length of the key we will therefore have:

- 10 rounds for 128-bit key;
- 12 rounds for 192-bit key;
- 14 rounds for 256-bit key.

One round or processing cycle, consists in carrying out the following four steps in order, aimed to modifying an input data reaching an encrypted/modified output data.

1 - SubBytes: Nonlinear substitution of all bytes which are substituted according to a specific table.
2 - ShiftRows: Shift of the bytes of a certain number of positions depending on the row they belong to.

3 - MixColumns: Combining bytes with a linear operation, bytes are treated one column at a time.
4 - AddRoundKey: Each byte is combined with the session key, the session key is calculated by the key scheduler.

In each round therefore the input will be the output of the previous round, up to the last round where the output represents the actual encrypted message. In the same way it happens through the four steps introduced earlier, for example examining 2 contiguous steps, the second step will take as input the output of the first step, until it reaches the end of the round.
As previously stated, these four steps are performed in each round except the last round and an initial round which is not counted in the total number of rounds In the last round, therefore, all the steps are carried out except for the 3-MixColumns step, while in the initial round only the 4-AddRoundKey step is performed using the starting cryptographic key as the session key.
By session key we mean a different key for each round, these keys also called round keys are obtained from the primary key thanks to the Key Scheduler algorithm.

The following image summarizes what has just been said by showing the encryption process with the use of a 128-bit key, which therefore imposes the number of rounds to be performed at 10.

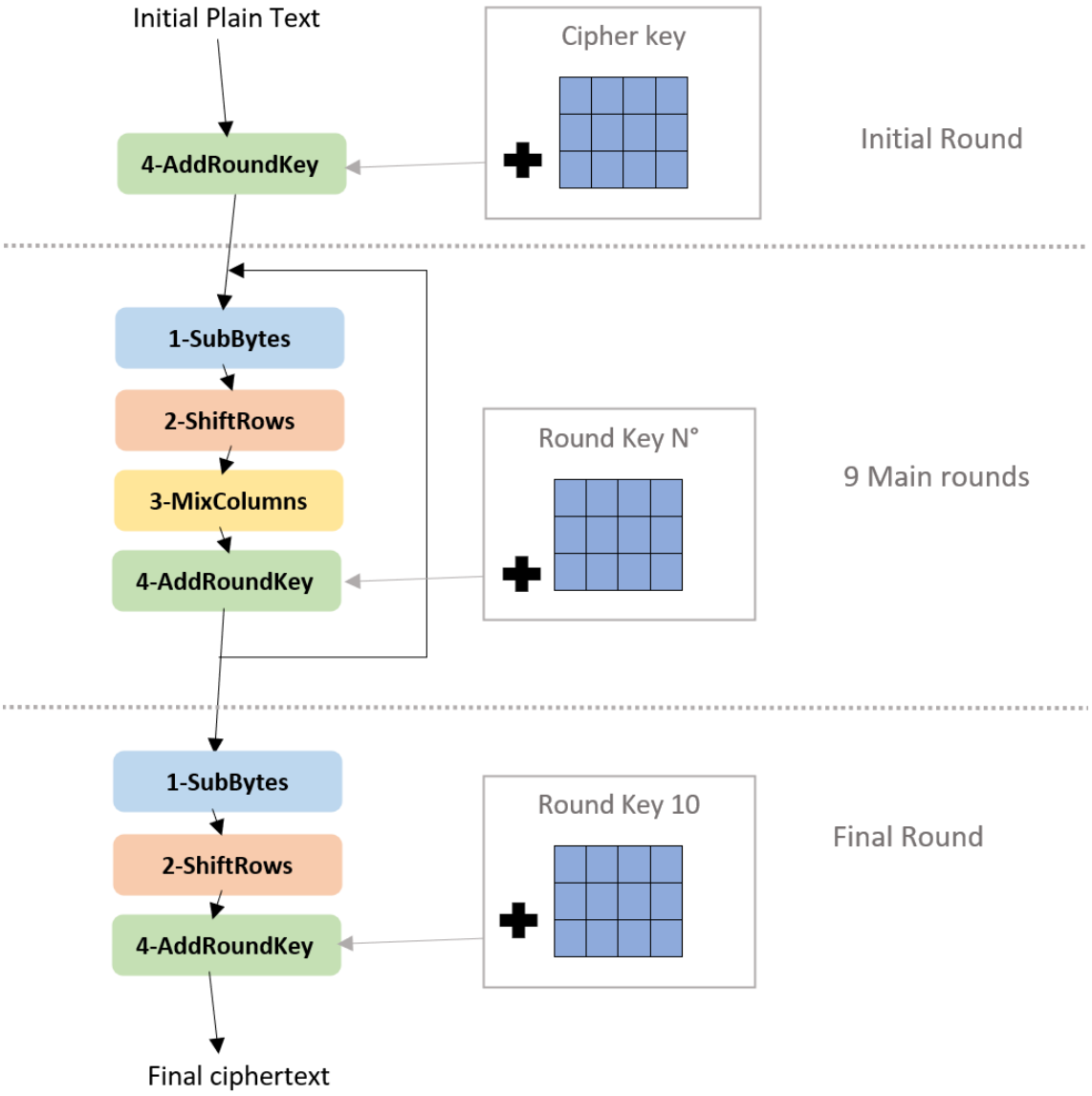


Figure 2.1: AES Encryption Process

In the next paragraph the process of creating round keys will be addressed using the key scheduler algorithm, on the contrary the 4 encryption steps will not be discussed since the understanding of the latter is not relevant for the following paper.

2.1.1 Key schedule process

In this paragraph an example of how the key scheduler algorithm works will be conducted, the aim of which is to generate session keys.
In the following example we will show its execution using a 128-bit long cryptographic key. This key can be represented by a 4X4 matrix where each cell has a size of 8 bit and each of it can be represented by 2 hexadecimal characters.

2B	28	AB	09
7E	AE	F7	CF
15	D2	15	4F
16	A6	88	3C

Figure 2.2: Cipher Key

For the calculation of all the round keys we must imagine that all the keys are adjacent and therefore form a sort of matrix with 4 rows and 44 columns. Now, therefore, to generate the first column (word) of the first round key, the last word of the starting cryptographic key is taken into consideration, which is the word preceding the column we want to calculate. A copy of this previous column is therefore made and the RotWord transformation is carried out which consists in moving each cell to the upper row, the cell that was initially in row 0 is placed in row 4.

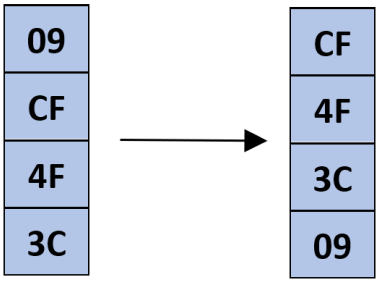


Figure 2.3: RotWord transformation

The SubBytes transformation will be performed on the resulting word, where each cell will be replaced with the content of a matrix called S-BOX. The procedure is as follows: the value of a cell of the resulting word is replaced with the value of a cell of the S-BOX matrix. The cell that will replace the value is represented by the row having the value of the first hexadecimal digit of the cell to be replaced, and column having the value of the second hexadecimal digit of the cell to be replaced.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Figure 2.4: S-BOX

So, in the case of the first cell of the word in question (CF) the new value it will replace will be found in row C and column F of the S-BOX matrix, this value is therefore 8A. This step is performed for each cell of the word.

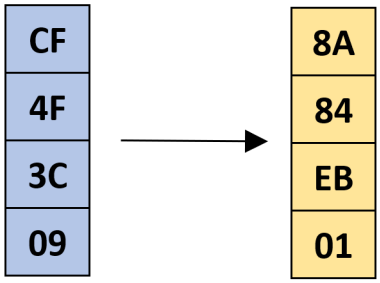


Figure 2.5: SubByte transformation

Now to this new resulting word will be added (XOR) the word found 4 columns before. Let us remember that we are calculating the first word of the first round key, so the column that is 4 words before turns out to be the first word of the starting key. In addition, a constant Rcon (Round constant) will also be added (XOR) which is different for each round, where in the first round it assumes a value of 1 in the first row and at each subsequent round it doubles its value.

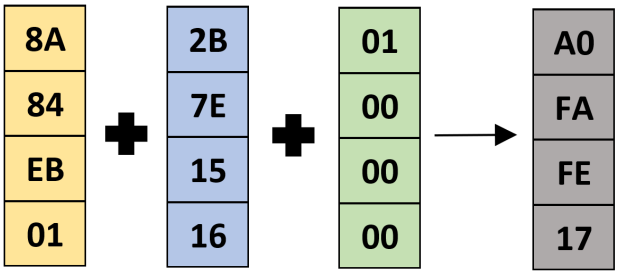


Figure 2.6: SubByte transformation + Wi-4 + Rcon = Wi

We will then get the first word of the first round key.
This step SubByte + Wi-4 + Rcon - Wi has to be done for the calculation of each word that appears to be in a multiple position of 4. In our case, having chosen 128

bit as the key size, the words that appear to be in multiple positions of 4 are always the first word of each round key.

2B	28	AB	09	A0
7E	AE	F7	CF	FA
15	D2	15	4F	FE
16	A6	88	3C	17

Figure 2.7: First round key's word

Now to calculate the remaining words of the same round key, the word in position -1 of the column we are calculating will be taken and the column in position -4 of the word we are calculating will be added (XOR) to it.

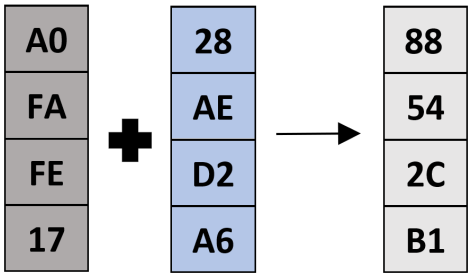


Figure 2.8: Wi-1 + Wi-4 = Wi

We then repeat this step until we have calculated all the words of the Round Key in question.

2B	28	AB	09	A0	88	23	2A
7E	AE	F7	CF	FA	54	A3	6C
15	D2	15	4F	FE	2C	39	76
16	A6	88	3C	17	B1	39	05
Cipher key				Round key 1			

Figure 2.9: Cipher key and round key 1

By iterating all the steps just made out for each round key, we then obtain all the session keys, fundamental for encryption. This highlights how all subsequent keys depend on the starting key, there is indeed a redundancy of information of the main key throughout the key schedule.

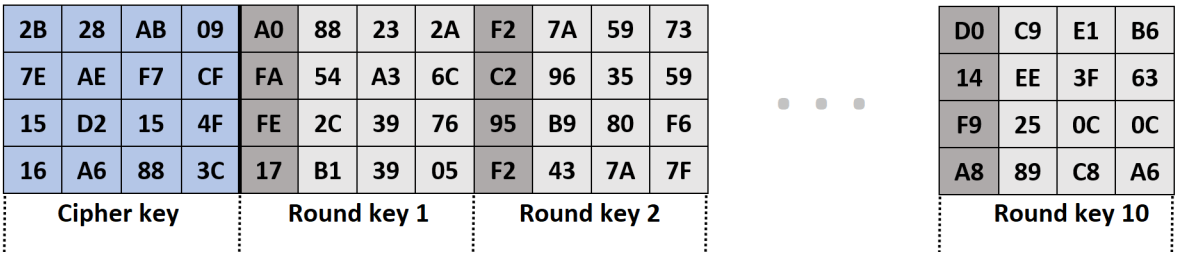


Figure 2.10: Key schedule image

2.2 SAT

Boolean satisfiability, or propositional satisfiability or SAT, is the problem of determining whether a Boolean formula is satisfiable or unsatisfiable. The formula is said to be satisfiable (SAT) if the variables can be assigned so that the formula evaluates to true. On the contrary, it is said to be unsatisfiable (UNSAT) if this assignment does not exist (therefore, the function expressed by the formula is false).

$$(\mathbf{x} \vee \mathbf{y} \vee \neg\mathbf{z}) \wedge (\neg\mathbf{y} \vee \mathbf{z})$$

In complexity theory the problem of boolean satisfiability (SAT) is a decision problem, whose instance is a boolean expression formed by operators AND, OR, NOT, "(", ")" applied to variables and sets of variables. The question is this: given an expression, is there any assignment of TRUE and FALSE values that make the whole expression true? A formula of propositional logic is said to be satisfiable if such an assignment exists. Without loss of generality, the problem can be traced back to the particular case in which the formula is expressed in conjunctive normal form (CNF), that is, as an AND of clauses each formed by an OR of literals. In fact, by applying De Morgan's Theorems, the formula can be lead back to the case in which the NOT operators are applied only to variables, and not to expressions. The Cook-Levin theorem shows that the problem of determining if any propositional formula is satisfiable is NP-complete [5], as the size of the problem to be solved increases, there is an exponential/sub-exponential increase in the time necessary for the computation of the result (SAT/UNSAT).

2.2.1 SAT Solvers

Given the complexity of the SAT problem and its great use being of central importance in many areas of computer science, including theoretical computer science, the study of algorithms, hardware design, there are many programs, called SAT Solvers which, through the help of different algorithms, they aim to solve Boolean satisfiability problems in the shortest possible time. There is also an annual international competition in which many different Solvers are tested with the aim of decreeing the best solver of the year [6]. In this competition there are different categories in which solvers can participate, these categories differ from each other based on the hardware resources available in the machine where the solver is tested, and the number of machines used.

The categories present in the SAT competition 2021 were:

- Main Track: Intel Xeon E5-2609 4 core and 128 GB RAM;
- Incremental Library Track: 2 x Intel Xeon E5430 4-Core, 24 GB RAM, only incremental SAT solvers can participate in this category*;
- Parallel Track: AWS m4.16xlarge, 64 virtual cores and 256 GB RAM;
- Cloud Track: 100 x AWS m4.4xlarge, for a total of 1600 virtual cores and 6.4 TB RAM.

*Incremental SAT solvers are particular solvers that can find more than one solution for an instance efficiently. After having found a solution, they use the acquired knowledge, such as the clauses learned, with the goal of finding other solutions. Traditional solvers, on the other hand, stop at the first solution found.

2.2.2 DIMACS Format

Concepts introduced on DIMACS format are taken from [7].

The DIMACS format is a standard interface for SAT solvers, it represents a common format to represent the Boolean formula in CNF format as input to the SAT solver. DIMACS is a text format, starting with a header line of the following form: "p cnf <variables> <clauses>". Where <variables> and <clauses> are replaced with the number of variables of the Boolean formula and the number of clauses of the formula respectively. Any line starting with the "c" character is treated as a comment. Each line of the DIMACS format that does not start with the character "c" or "p" therefore represents a clause formed by an OR of variables, each of these clauses (lines) is therefore in AND with the other clauses (lines) of the file. To represent a variable, the following notation is used: a unique integer number is assigned to each variable of the formula, a positive number such as 2 therefore represents a positive occurrence of variable 2. A negative number such as -3 represents a negative occurrence of variable 3. The number 0 is treated in a special way, it does not represent a variable, but is used to indicate the end of the clause.

For example, the following formula $(\mathbf{x} \vee \mathbf{y} \vee \neg\mathbf{z}) \wedge (\neg\mathbf{y} \vee \mathbf{z})$ can be written in DIMACS format as follows:

```
p cnf 3 2
1 2 -3 0
-2 3 0
```

In addition, the DIMACS format also specifies a common format for the output returned by the SAT solver, the line starting with the "s" character contains the result of the calculation performed by the SAT solver.
This result can be of 2 types "SAT", therefore there is an assignment of the boolean variables that allows the formula evaluates to true, or "UNSAT" if there is NO assignment of the boolean variables that allows the formula evaluates to true.

The line that begins with the "v" character contains the assignment of the variables that allow the formula given in input to evaluates true.

For example, the output for the formula seen above is:

s SATISFIABLE
v -1 -2 -3 0

In addition, there are many variations and extensions of the DIMACS format, one in particular which is very useful in cryptography, makes use of XOR clauses. However, not all SAT solvers accept this variant and therefore it will be necessary to convert the DIMACS with the XOR clauses into a CNF DIMACS. This procedure is quite simple and will be deepened in the next chapters as it is necessary for the use of certain SAT solvers that do not support XOR clauses.
An XOR clause is represented by a line that begins with the character x and subsequently presents the variables that are XOR together, obviously as happens with the other clauses, the XOR clauses are also placed in AND with the other clauses (rows) of the file.
The following example depicts an XOR clause between 3 boolean variables.

x1 -2 3 0

Chapter 3

Previous work

In this chapter we will retrace previous work related to remanence of data in the RAM memory and recovering an AES key from a corrupt key schedule. This chapter appears to be necessary to fully understand the following paper, and to understand what the following thesis adds to the literature.

It has been known since 1970s that the contents of DRAM memory cells can survive for a certain time at room temperature (memory remanence). This time can increase, decreasing the temperature of the ram cells. In 1978 an experiment [8] showed that a power-disconnected DRAM bank did not lose any data for a week when cooled with liquid nitrogen.
Chow, Pfaff, Garfinkel and Rosenblum [9] suggested that modern DRAM content can survive a Cold Boot, that is the activity of booting a computer through the power button. Petterson [10] suggested that remanence of data in cold boot could be exploited to conduct forensic analysis in the memory image and thus obtain encryption keys.
Alderman [11] show that at room temperature generally, there is a low percentage of corrupted bits for a few seconds without powering the DRAM memory, followed by a period of rapid decay where the percentage of errors grows exponentially. Alderman also observed that the majority of the corrupted bits decay into the "ground states" that correspond to binary value 0, while only a very small part make the binary transition from 0 to 1 (reverse flipping error). Alderman also introduces the following notation "$\delta 0$" to indicate the percentage of corrupted bits that make the binary transition from 1 to 0, while he uses the following notation "$\delta 1$" to indicate the percentage of corrupted bits that perform the inverse binary transition, that is from 0 to 1. This notation will be used extensively in subsequent papers. Alderman simply using a compressed air spray upside down was able to bring the temperature of the RAM bank to -50° degrees and obtained on average less than 1% corrupted bits after 10 minutes without powering the RAM. Alderman repeated the experiment by submerging the same bank of ram in liquid nitrogen and noticed a decay of only 0.17% bit after an hour without power.
Alderman presents 3 attack variants to get the RAM memory image. The simplest is to reboot the machine and boot a modified Kernel that dumps the RAM memory. Another more advanced method is to shut down the computer forcibly, directly removing the power supply, then proceed to restore the power supply and

boot a modified kernel that dumps the RAM memory as before. The latter variant is more effective as it does not allow the operating system to clear the memory before shutting down. The third type appears to be the best, it consists in disconnecting the power supply, then removing and installing the DRAM modules on a second computer prepared by the attacker, and from this computer dump the RAM memory image as previously done. Unlike the first 2 variants, this attack deprives the victim's PC's BIOS and hardware of any possibility of wiping memory during boot. Alderman explains that if the attacker were forced to wait a long time from shutdown to dump, the data in the ram will most likely start to corrupt, so it will be necessary to cool the RAM to decrease this process or implement error correction algorithms. Gutmann [12, 13] suggested that encryption keys should not be stored in RAM for more than a few minutes. Alderman explains however that for performance reasons this does not happen because many applications pre-calculate the round keys and save them in RAM for future use.
Alderman then presents an automatic and complete algorithm for identifying encryption keys in the RAM image.
Alderman develops an unpublished algorithm that can reconstruct cryptographic keys even in the presence of many errors, taking for example AES or DES the developed algorithm does not work directly on the starting key but operates on the data deriving from it such as round keys, the latter provide a high degree of redundancy and therefore, allow to obtain the main key. By applying this algorithm Alderman states to be able to reconstruct within a few seconds, almost all AES 128-bit keys starting from a key schedule image with a corruption of 10% $\delta 0$ and 0.1% $\delta 1$, while starting from a corrupt key schedule image with 30% $\delta 0$ and 0.1% $\delta 1$, the time required is less than 20 minutes in half of the cases.
Tsow [14] continued in alderman's error correction study, and developed a heuristic algorithm, capable of solving all cases that presented a corruption of 50% $\delta 0$ in the key schedule in less than half a second, while with a corruption of 60% $\delta 0$ in the key schedule the worst case time was 35 seconds, with an average test time of 0.17 seconds.
Kamal [15] said that Tsow's algorithm is complex and difficult to develop and refine, so, he decided to use another approach. Kamal noted that the relations that must be satisfied by different round keys in the AES key schedule, can be easily formulated as Boolean satisfiability problems (SAT), and therefore solvable through the use of SAT solvers. Kamal through the use of the SAT solver Cryptominisat, solved with an average time of 1.2 seconds, key schedule with a corruption of 70% $\delta 0$ while Tsow's algorithm for the same percentage of corruption took an average time of 300 seconds.
Until now the only algorithm that respected the realistic assumptions, that is, that also considered the reverse flipping error ($\delta 1$), was the Alderman algorithm which turned out to be very underperforming. On the other hand, Tsow's algorithm and kamal's SAT approach turned out to be very fast but only took into account a corruption from binary value 1 to 0 ($\delta 0$).
Xiaojuan Liao [16] took a step forward, adopting a MAX-SAT approach that also took reverse flipping error into consideration. He then got an excellent result solving with an average time of 300 seconds, starting from a corrupt key schedule with 70% $\delta 0$ and 0.1% $\delta 1$.

Chapter 4

Preliminaries

From the previous chapter, we note that there are many studies that have contributed to the development of new error correction techniques starting from a corrupted AES key schedule image (corrupted information). On the other hand, this thesis wants to deepen a more general case, that is the case of partial information, where some bits are known and others are not. The known bits can take value 1 or 0, a scenario that could not happen in the articles mentioned in chapter 2 where only the bits with value 1 are known since corruption occurs with a higher probability, from 1 to 0 and not vice versa. This scenario with partial information can occur by making an incomplete dump, or by having only a ram module available in which there is only a part of the key schedule image to be analyzed. In this thesis, therefore, starting from this partial information, we want to analyze and test some techniques with the aim of finding the fastest and most effective. We will therefore make use of well-known SAT solvers combined with brute force techniques.

4.1 Creation of input file

For simplicity, it was decided to generate the input files, which represent the partial key schedule necessary for the tests that will be conducted, starting from 128-bit AES keys, this choice is not binding in any way, with few changes therefore, all the steps that will be shown from now on can be adapted to longer keys.
The tests were conducted on CNF files in DIMACS format, this allows to extend the various tests on many other solvers. Each of these CNFs appropriately created, is formatted in the following way: line zero presents the file header, from line 1 to line 87081 there are the necessary clauses to respect the relationships between different round keys of the AES key schedule and finally from line 87042 up to the end of the file there are the binary assignments of the known bits, in other words the information we know. If we notice well, in fact, from line 87042 onwards there is only one variable for each line, which represents the known information of that bit.

The example below shows the formatting of the CNF file just described.

p cnf 1728 83200 //riga 0
1568 1696 -1728 0 //riga 1
⋮
-1537 -1538 -1539 -1540 -1541 -1542 -1543 -1544 -1600 0 //riga 87041
-161 0 //riga 87042
-162 0

These CNFs were generated by a python script interpreted by Jupiter Notebook (a web-based interactive computing platform) inside a DOCKER image. The DOCKER image contains the SAGE 9.5 mathematical library, necessary for the creation of the boolean clauses inherent to the AES key schedule, the explanation of how these clauses are constructed can be found in [15]. As for the assignment of the known bits, there is the use of an external library [17] which, starting from the initial key (generated randomly), calculates all the round keys, it will then be up to us to decide the quantity and location of each single bit to be inserted. as partial information within the CNF.
It is therefore obvious that in each CNF of an AES-128 bit key the first 87041 are the same since the clauses inherent to the relationships between different round keys will be the same, what changes from file to file are therefore the known bits of the round keys.
The script inside has a parameter that allows you to set the number of CNF files we want to generate and the number of known bits we want to insert inside the CNF. In this thesis it was decided to assign the known partial information, in the most disadvantageous way possible, so as to implement a very bad case. To do this, the known information is assigned at the beginning of the first round key and at the end of the tenth round key, the initial cipher key is therefore excluded from the known information since it is precisely what we want to calculate. The figure represents the known information assigned in the CNF, 64 bit in the left part of the round key 1 and 64 bit in the right part of the round key 10, the asterisks indicate the bits whose value is not known.

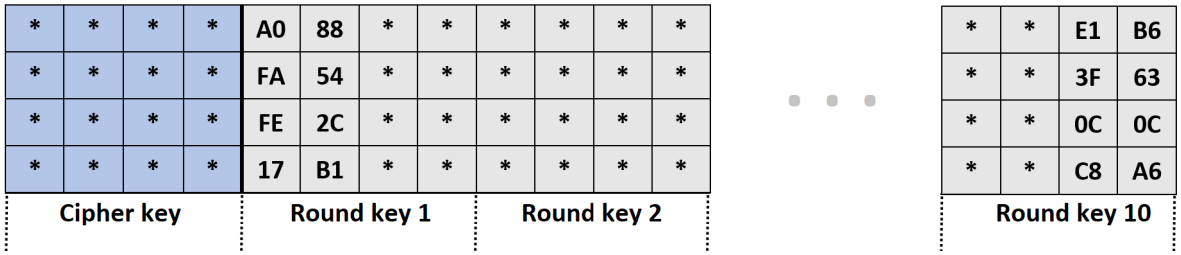


Figure 4.1: Partial information know

Also, in this case the choice of which known information to assign is not binding in any way, in all the tests conducted in this thesis different computation techniques

are always compared on the same known information, in order to have an honest comparison between the different techniques.
Originally this script generated a DIMACS CNF with xor clauses, it was therefore necessary to convert these clauses to test the CNF file on solvers that do not support this DIMACS variant with XOR clauses. To do this, it was sufficient to convert each occurrence of an XOR clause with a series of CNF clauses. In the file appeared 2 types of XOR clauses, the first type presents the third variable negated, while the second type presents all 3 variables in their positive form.
For the first type of XOR clause: **"x1 2 -3 0"** the following CNF clauses correspond:

-1 -2 3 0
-1 2 3 0
1 -2 3 0
1 2 3 0

For the second type of XOR clause: **"x1 2 3 0"** the following CNF clauses correspond:

-1 -2 3 0
-1 2 -3 0
1 -2 -3 0
1 2 3 0

The python script will then generate both files, a CNF file with the XOR clauses and a CNF file without the XOR clauses.

4.2 Hardware and software used

An ACER Aspire V Nitro laptop with Intel Core i7 4720HQ 3.6 Ghz 4 core 8 thread processor, with 16 GB of DDR3 memory, was used to conduct the tests.
The following SAT solver was used to perform the tests:

- Kissat 2.0.1 [18]: "Kissat is a Keep It Simple and clean bare metal SAT Solver written in C. It is a port of CaDiCaL back to C with improved data structures, better scheduling of inprocessing and optimized algorithms and implementation". Single-thread SAT solver, winner of Main Track 2021.

The following SAT solver was used as a comparative in some of our results:

- CryptoMiniSat 5.8.0 [19]: "An advanced incremental SAT solver", winner of the Incremental Track 2020.

The solvers have been executed in the various approaches developed through the use of python scripts, this language very easily allows you to read and write from standard output and on files. Furthermore, always with great simplicity, python has made it possible to create a thread pool using the "multiprocessing.dummy" library, necessary for conducting some tests.
The Kissat SAT solver, for convenience, was performed in a virtual environment with 7 virtual cores available. Cryptominisat, on the other hand, runs in a native windows 10 environment with 8 available cores, that correspond to all the logical cores of the machine.

Chapter 5

Key recovery on partial key schedule

In this chapter the techniques and approaches developed will be shown, they will also be tested with the aim of acquiring statistics on processing times. In the articles mentioned in chapter 2, the fastest approach was the one that found the only solution in the shortest time, in this case instead in many instances we will have the possibility of having more correct solutions that satisfy the constraints, the best approach will therefore be the one that will allow us to find all the solutions in the shortest possible time. This difference arises because the computational power in the periods in which the papers in chapter 2 were written was relatively much lower than today, and therefore only allowed to analyze instances in which corruption was relatively low to enable obtaining a single solution. In 1 out of 7 experiments conducted by Kamal [15] with a corruption of 80% $\delta 0$ the key returned by the SAT solver was different from the original key, the correct key was therefore found by re-running the sat solver and banning the previously found key. In addition to the increase in computational power, many steps forward have also been made in the development of SAT solvers, which are able to solve very complex instances with less and less time. In our case, therefore, we have an increasingly optimized computational power and SAT solvers that allow us to analyze instances in which the known information is relatively low so that we can have more solutions.
Having many candidate solutions and finding the correct one turns out to be a very simple and efficient problem, just test the keys found on intercepted clear/encrypted texts. Given its simplicity, this problem will not be dealt with in the following paper.
The techniques developed in this thesis will be presented in the next sections. For each approach it is present a detailed high level explanation, a pseudocode and the statistics of the various tests conducted, necessary for chapter 6 in which several comparisons will be made.
There will be the use of graphs with logarithmic scale in the Y axis to have a clearer report.
For a more truthful comparison, the same CNF files are used in all approaches tested and also between the different solvers tested.

5.1 Banning solution approach

This technique turns out to be the simplest and most intuitive approach to implement. The developed python program starts by launching the solver with the generated instance (file CNF), later, after the solver has found the solution, the written script will proceed by relaunching the solver excluding the previously found solution. To do this, it is sufficient to add a constraint (clause) that negates the solution just found, so that the solver looks for other solutions. In this case we would therefore have many successive invocations of the solver in which a constraint (clause) is added each time to exclude the key just found. The python script stops relaunching the solver when it says there are no more solutions (UNSAT). To carry out the banning operation of the solution just found, we must insert a new clause in the CNF file, this clause will be composed of the solution just found by the solver with the variables with the inverted binary value.
An example will be shown below on a dummy instance.
Solver output:

s SATISFIABLE
v 1 2 -3 0

Clause to be inserted in the CNF file to ban the solution just found:

-1 -2 3 0

The pseudocode of the algorithm just described is shown below.

Algorithm 1 Banning solution pseudocode

```
while TRUE do
    Launch solver with CNF
    if Solver return SAT then
        Save solution
        Modifying the CNF banning the solution just found
    else
        Display all solution, and elapsed time
        break
    end if
end while
```

5.1.1 Tests result

The tests are conducted with CNFs with different known number of bits.
Since this technique is designed to find all the solutions of a given CNF, its use on instances that have only 1 solution turns out to be very unfavourable in terms of time. This is because the solver is re-executed after it has found the only solution, in order to find others. So, in instances with many known bits where it is probabilistically known that the solution will be only one, it is therefore advisable to perform

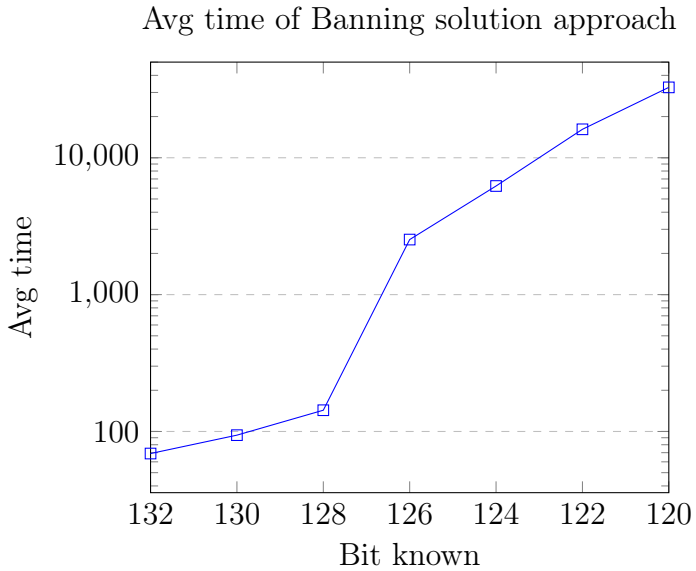
the SAT solver directly with the CNF, without the aid of this approach.
In sight of this, it was decided to start from instances that produced an average of 1 solution up to instances that produced several candidate solutions.
By known bits we mean the total of known information, which as previously mentioned is divided between the part allocated in the first round key and the part allocated in the tithe round key.

The statistics on the execution times of the various tests are shown below.

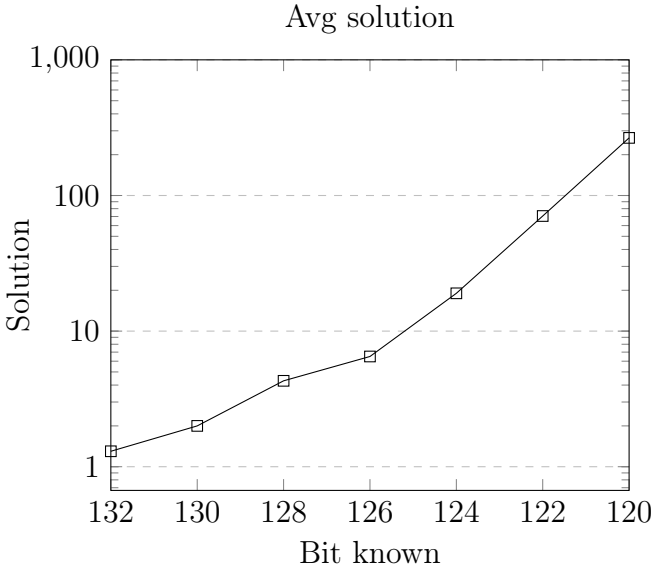
Bit known		132	130	128	126	124	122	120
Avg Solutions		1.3	2	4.3	6.5	19	70.6	266
Kissat	Min	47.4	36.4	65.1	889.0	6807	14811	22030
	Max	92.8	150.5	240.5	6494	6886	16984	43721
	Avg	69.3	94.4	143.3	2525	6218	16142	32684
	St.Dev	12.0	31.5	50.8	1546	889.4	952.8	8859
	Med.	70.9	97.25	147.5	1965	6807	16633	32303
	# Tests	10	10	10	10	3	3	3

Table 5.1: Run-time statistic for banning solution

The graph below shows the trend of the average times as a function of the known bits.



The graph below shows the trend of the candidate solutions as a function of the known bits.



As we can see from both graphs we have a sub-exponential increase in average times and candidate solutions. Through these results we can verify what has been said previously, that is, as the known information decreases, we will have an increasing number of candidate solutions. In the graph that portrays the feasible solutions we note how the increase is very predictable and smooth, as regards the graph that portrays the average times, on the other hand, we have a less predictable trend. The most notable increase is from 128 to 126 bit known where there is a growth of 1665%.
In all the other contiguous columns we have at most an increase which is less than 200%. These graphs are very important, because they show us how this problem is computationally difficult given that the complexity turns out to be sub-exponential.

5.2 Brute force approach

This technique adopts the Divide et impera approach, since the initial instance that is given as input is divided into subproblems that will be addressed individually by the SAT solvers. More precisely, this technique consists in doing a brute force of a few bits before running the SAT solver on the instance.
Once you have chosen the cardinality of the bits you want to brute force, this approach proceeds to generate all the combinations of these bits and then to create the sub-problems with the different combinations. It was decided to instantiate these bits close to the known bits, so for example if we decide to brute force on 2 bits, 1 bit will be instantiated close to the known bits of round key 1 and the other bit will be instantiated in the same way close to the known bits of the round key 10. In chapter 5.4 further tests will be carried out, changing the location of the instantiated bits.
So, suppose to do the brute force on 2 bits, since with 2 bits we have 4 combinations, we will then have 4 subproblems of the main instance to be solved individually through an execution of the SAT solver. The following image depicts the following scenario, the asterisks indicate the bits of which the value is not known while the red bits are the instantiated bits (in which we do brute force), the number of known and unknown bits represented in the image is not significant.

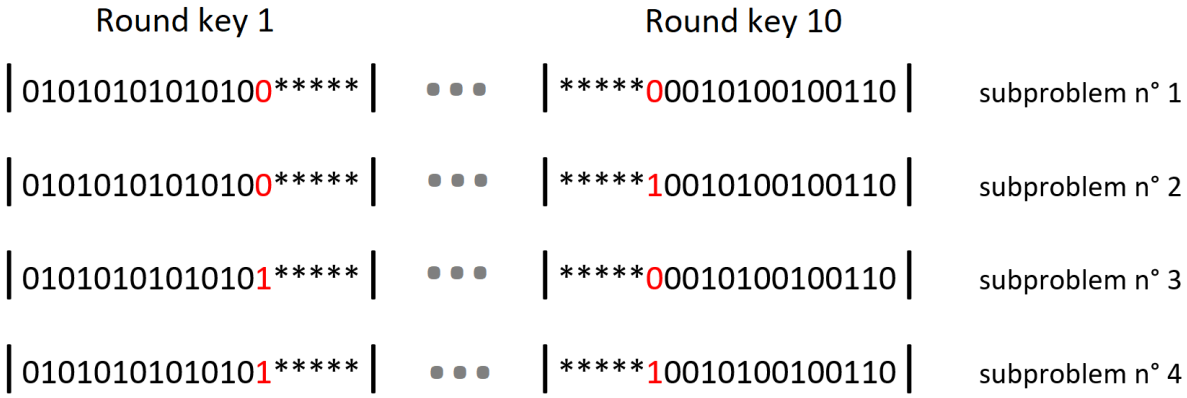


Figure 5.1: Sub-problem generated

Therefore, for each subproblem a clone of the starting CNF will be made with the addition of the instantiated bits, in this case we will therefore have further 4 CNF files.
From a coding point of view, a solution was initially implemented which consisted in asynchronously launching each call to the SAT solver with the CNFs generated, but doing so, however, in many cases the CPU was overloaded with many threads to manage. Suppose the case of having instantiated 5 bits we will then have 32 instances of execution of the SAT solver, each of this instance will compete with the others to obtain the use of the CPU, since we have only 7 logical cores available. In addition, the CPU itself will waste a lot of time by making the context switch between one process and another. It was therefore decided to use a python thread pool through the "multiprocessing.dummy" library, it will have the aim of

scheduling the various instances in such a way as not to overload the CPU. A task is created for each SAT solver execution instance that will be executed at the appropriate time from the thread pool. In this way we will potentially always have 7 instances running on the 7 logical cores, when an instance terminates another one is instantly started until all the calls to the SAT solver are terminated.
In some cases we may not use all 7 logical cores, this is because being Kissat a single thread solver, 7 instances are performed simultaneously only where the number of combinations is greater than or equal to 7. For example, if we decide to brute force on 2 bits we would have 4 CNF files that each contain a different combination. By doing so we would therefore have 4 instances of the solver running at the same time, thus using 4 logical cores out of 7 available. We will note, however, that this is not as limiting as it seems since Kissat is a solver developed to be performing with only 1 core available.
From a practical point of view, this approach has the big limitation that not in all cases it is able to find all the solutions. This is because starting from the initial problem, N subproblems are created, but each of these subproblems is solved through a simple call to the SAT solver, which returns at most one solution. To overcome this problem, we can think of increasing the cardinality of N in such a way as to have more known information in each subproblem, which is equivalent to having a smaller number of solutions. However, we will notice from the test results that many times this solution is not sufficient to obtain all the solutions and in some cases it turns out to be very inefficient.
Maximum number of solutions obtainable in function of the number of instantiated bits:

- 2 bit: 4 solutions
- 4 bit: 16 solutions
- 6 bit: 64 solutions
- 8 bit: 256 solutions

In the next section another approach will be shown that brings together the 2 techniques seen so far and allows to obtain all the solutions of the starting CNF.

The pseudocode of the brute force technique just described is shown on the next page.

Algorithm 2 Brute force pseudocode

```
TaskList = []
C = Calculate and Store all combination with N bit instanced
while C is not empty do
    Create Task = {
        1) Pick one combination from C
        2) Create CNF accordingly to the combination taken
        3) Launch SAT solver with the generated CNF
    }
    Add Task to TaskList
end while
for each Task in TaskList do
    Print solution of Task
end for
```

5.2.1 Tests result

As in the Banning solution approach, the tests are conducted with CNFs with different known number of bits, with the addition that for each CNF different tests are conducted with a different number of instantiated bits.

The tests carried out were divided into 2 groups:

I) Tests carried out from 490 to 140 known bits, in these tests given the high number of known bits, we will have only one solution in output from the solver. The results of these tests will be compared in the next chapter with the average times of a single call to the SAT solver;

II) Tests performed from 132 to 120 known bits, in these tests more than one solution is returned. The results of these tests will be compared in the next chapter with the average times of the other techniques developed.

In each table belonging to group II of the tests, there is an avg. solution row, this row contains the average solutions found with these known bits through the Banning Solution technique previously explained.

The times collected in the tests carried out also include the time needed to create the CNF files containing the subproblems, this could lead to a lot of overhead especially if many bits are instantiated and if the time to solve each subproblem is relatively low as happens in the test group 1. To overcome this complication, one could think of generating all the files first (offline) and launching the solvers on them, however this solution will not be explored in this paper;

The statistics on the execution times of the various tests as a function of the number of instantiated bits are shown in the next page, while the results obtained will be compared and discussed in the next subsection.

2 bit instanced

Bit known		490	420	350	280	210	140
Kissat	Min	.29	.24	.25	.28	.33	28.2
	Max	.35	.37	.39	.64	.44	208.6
	Avg.	.31	.32	.32	.35	.37	84.9
	St.Dev	.023	.034	.036	.10	.033	53.7
	Med.	.31	.32	.32	.32	.36	70.9
	# Tests	10	10	10	10	10	10

Table 5.2: Run-time statistic for brute force on 2 bit I

Bit known		132	130	128	126	124	122	120
Avg sol.		1.3	2	4.3	6.5	19	70.6	266
Avg sol. founds		1.1	1.6	1.9	2.5	3.4	4	4
Kissat	Min	38.2	41.1	39.9	15.9	343.3	511.7	194
	Max	58.7	64.1	284.6	133.4	1489	832.5	431
	Avg.	45.6	50.4	70.2	47.6	692.0	676.6	334
	St.Dev	6.8	7.1	75.4	43.7	401.0	131.0	100
	Med.	43.9	52.3	47.2	30.7	516.3	681.1	356
	# Tests	10	10	10	10	10	4	4

Table 5.3: Run-time statistic for brute force on 2 bit II

4 bit instanced

Bit known		490	420	350	280	210	140
Kissat	Min	.88	.89	.90	.94	1.15	26.6
	Max	1.03	.93	1.04	1.31	1.73	107.0
	Avg.	.94	.91	.94	1.05	1.26	49.9
	St.Dev	.054	.011	.042	.10	.17	22.5
	Med.	.92	.90	.92	1.03	1.21	46.6
	# Tests	10	10	10	10	10	10

Table 5.4: Run-time statistic for brute force on 4 bit I

Bit known		132	130	128	126	124	122	120
Avg sol.		1.3	2	4.3	6.5	19	70.6	266
Avg sol. founds		1.2	1.8	3.3	4.1	8.0	12.75	16
Kissat	Min	104.9	91.5	110.7	129.2	83.4	1723	1616
	Max	165.8	162.5	210.1	267.5	355.2	2461	2310
	Avg.	130.2	129.3	150.0	166.8	153.6	2093	2042
	St.Dev	22.4	27.0	27.3	41.6	85.4	327.6	260.3
	Med.	123.7	130.2	145.6	154.4	110.2	2094	2122
	# Tests	10	10	10	10	10	4	4

Table 5.5: Run-time statistic for brute force on 4 bit II

6 bit instanced

Bit known		490	420	350	280	210	140
Kissat	Min	3.52	3.55	3.54	3.60	4.54	97.1
	Max	7.05	3.89	3.99	3.77	5.33	172.3
	Avg.	4.72	3.64	3.64	3.68	5.01	128.7
	St.Dev	1.50	.097	.138	.05	.27	20.0
	Med.	3.62	3.61	3.59	3.69	4.98	122.5
	# Tests	10	10	10	10	10	10

Table 5.6: Run-time statistic for brute force on 6 bit I

Bit known		132	130	128	126	124	122	120
Avg sol.		1.3	2	4.3	6.5	19	70.6	266
Avg sol. founds		1.3	1.9	4.1	5.3	15.4	24.5	56.5
Kissat	Min	365.4	366.1	376.3	453.2	498.8	363.2	7387
	Max	516.9	539.6	568.7	608.2	667.8	449.1	10701
	Avg.	431.9	437.8	472.7	543.7	570.3	412.8	9044
	St.Dev	57.5	68.3	59.1	55.9	52.2	36.5	1657
	Med.	415.1	407.6	470.6	555.6	558.6	418.1	9044
	# Tests	10	10	10	10	10	4	2

Table 5.7: Run-time statistic for brute force on 6 bit II

8 bit instanced

Bit known		490	420	350	280	210	140
Kissat	Min	24.0	13.9	13.8	14.4	16.3	411.9
	Max	27.0	18.9	15.2	15.4	18.6	603.3
	Avg.	25.7	14.7	14.2	15.0	17.3	478.6
	St.Dev	1.07	1.51	.39	.29	.69	53.3
	Med.	26.0	14.1	14.0	15.0	17.3	477.0
	# Tests	10	10	10	10	10	10

Table 5.8: Run-time statistic for bruteforce on 8 bit I

Bit known		132	130	128	126	124	122	120
Avg sol.		1.3	2	4.3	6.5	19	70.6	266
Avg sol. founds		1^a	1.75^a	4.3	6.2	19	37	115
Kissat	Min	2397	1330	1565	1634	1783	2187	1368
	Max	2860	1805	1771	2105	2197	2459	1616
	Avg.	2629	1506	1656	1797	2009	2279	1521
	St.Dev	193.5	209.3	81.2	193.5	128.9	123.4	93.9
	Med.	2630.0	1445	1644	1727	2022	2235	1551
	# Tests	4	4	5	5	7	4	4

Table 5.9: Run-time statistic for bruteforce on 8 bit II

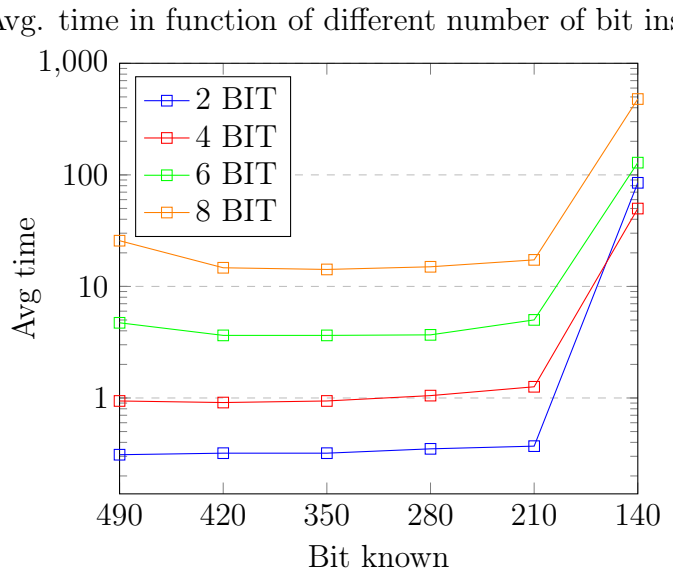
a. I am highly confident that by carrying out 10 tests, the solutions found are equal to the expected solutions.

5.2.2 Comparing tests result

In this subsection the results obtained will be compared and discussed.

From 490 to 140 known bits

The graph below shows the trend of the average times as a function of the known bits.

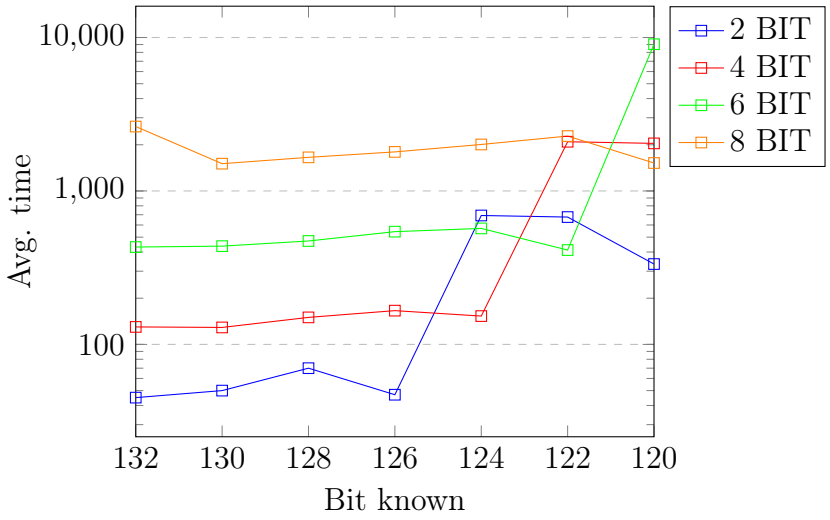


The superiority in instantiating only 2 bits can be noticed very easily, this configuration is inferior only in the case of 140 known bits, in which it is faster to brute force on 4 bits.
These results are most likely due to the fact that the times for the calculation of the solutions are very low, (<1 sec with known bits> 140), and therefore by instantiating many bits, the time necessary for the creation of the copy file prevails.
In these cases therefore, having a relatively high cardinality of the known bits, if we decide to use the brute force approach we can conclude that it is better to brute force with as few bits as possible.

From 132 to 120 known bits

The graph in the next page shows the trend of the average times as a function of the known bits.

Avg. time in function of different number of bit instanced

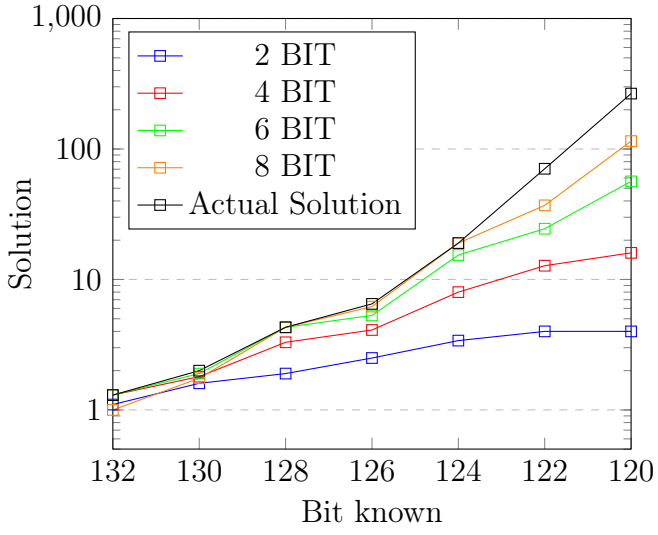


From the overlay graph we can see that the average times are very overlapping, in contrast to what was found in the previous test group. In the Find Ban technique we noticed how there was a large increase in the average times necessary for the computation from 128 to 126 known bits, we can verify this increase in the average times also in this graph. To be precise, this increase occurs when the sum between instantiated and known bits is equal to 126.

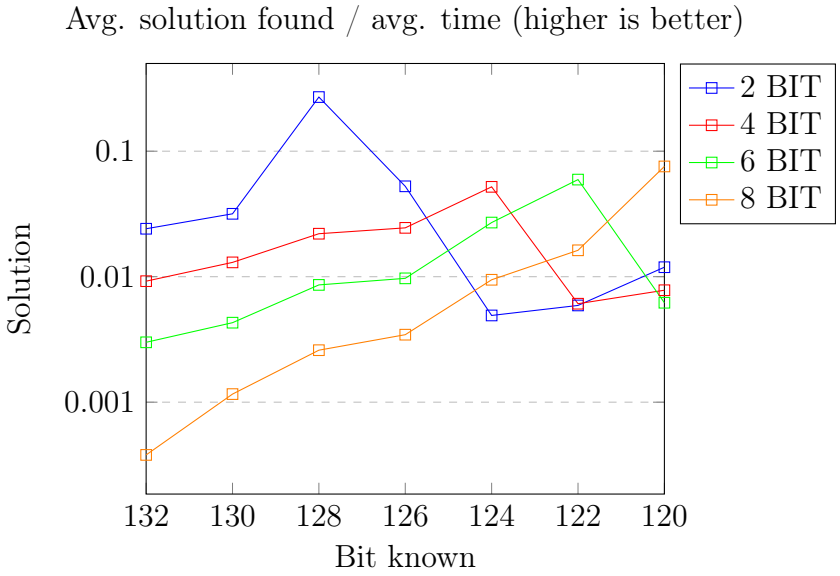
- 2 BIT -> 124+2
- 4 BIT -> 122+4
- 6 BIT -> 120+6

The average times alone, however, are not a valid metric in deciding which is the best number of bits to instantiate. This is because, as shown in the graph below, the solutions found vary considerably as the number of instantiated bits varies.

Avg. solution found vs Avg solution



It is therefore advisable to use a metric that takes into consideration the solutions found and the average time required for the computation. Despite this, it must be borne in mind that in some configurations of instantiated bits, we can get very few solutions compared to the real number of possible solutions e.g. (2 instantiated bits, 120 known bits, 4 solutions found, 266 avg. total solutions)
The graph below shows the ratio between solutions found and average time to find such solutions for each configuration of instantiated bits.



Through this graph we can therefore conclude that there is no optimal instantiated bit configuration in any known bit situation. Based on the objective to be achieved: calculate all the solutions; find as many solutions as possible; find few solutions; it may therefore be advisable to instantiate a certain number of bits instead of another.

5.3 Mixed approach (brute force and banning solution)

As already anticipated in the previous section, this technique consists in putting together the 2 previous approaches seen.
The python script then starts generating all the combinations of the instantiated bits (brute force) as in the previous approach, the difference is that the generated CNFs are not executed through an execution of the SAT solver, but rather they are executed by an execution of the Banning solution script seen in section 5.1. By doing so, this technique is therefore able to find all the solutions of the starting CNF.

The pseudocode of the Hybrid technique just described is shown below.

Algorithm 3 Brute force and banning solution pseudocode

```
TaskList = []
C = Calculate and Store all combination with N bit instanced
while C is not empty do
    Create Task = {
        1) Pick one combination from C
        2) Create CNF accordingly to the combination taken
        3) Launch Banning solution script with the generated CNF
    }
    Add Task to TaskList
end while
for each Task in TaskList do
    Print solution of Task
end for
```

5.3.1 Tests result

As in the previous approach, the tests are conducted with CNFs with different known number of bits and with a different number of instantiated bits.
In the table the avg. sol and avg. sol. founds columns have been omitted, as this approach always finds all possible solutions.

The following page shows the statistics on the execution times of the various tests as a function of the number of instantiated bits.

2 bit instanced

Bit known		132	130	128	126	124	122	120
Kissat	Min	57.0	55.1	129.4	98.5	2500	8159	21825
	Max	83.9	110.1	175.8	186.0	6119	12455	26006
	Avg.	71.4	78.6	149.7	133.3	3782	9950	23915
	St.Dev	7.3	17.5	19.6	37.2	1651	2235	2090
	Med.	70.4	77.5	146.8	124.3	3255	9235	23915
	# Tests	10	7	4	4	4	3	2

Table 5.10: Run-time statistic for Mixed approach on 2 bit

4 bit instanced

Bit known		132	130	128	126	124	122	120
Kissat	Min	116.2	122.8	175.5	182.2	194.5	8605	29151
	Max	176.0	180.8	210.4	254.4	548.6	12317	37162
	Avg.	138.1	147.8	194.6	217.4	339.1	10796	33156
	St.Dev	22.6	20.8	15.9	30.8	149.6	1944	4005
	Med.	131.5	142.5	196.3	216.5	306.7	11465	33156
	# Tests	10	7	4	4	4	3	2

Table 5.11: Run-time statistic for Mixed approach on 4 bit

6 bit instanced

Bit known		132	130	128	126	124	122	120
Kissat	Min	348.4	362.2	500.3	486.6	711.1	1003	32676
	Max	474.1	532.2	579.0	709.2	1021	16740	36822
	Avg.	410.3	410.6	543.4	563.0	807.2	5169	34749
	St.Dev	45.0	55.5	34.0	103.2	143.7	6097	2073
	Med.	403.1	399.9	547.1	528.0	748.2	1086	34749
	# Tests	10	7	4	4	4	5	2

Table 5.12: Run-time statistic for Mixed approach on 6 bit

8 bit instanced

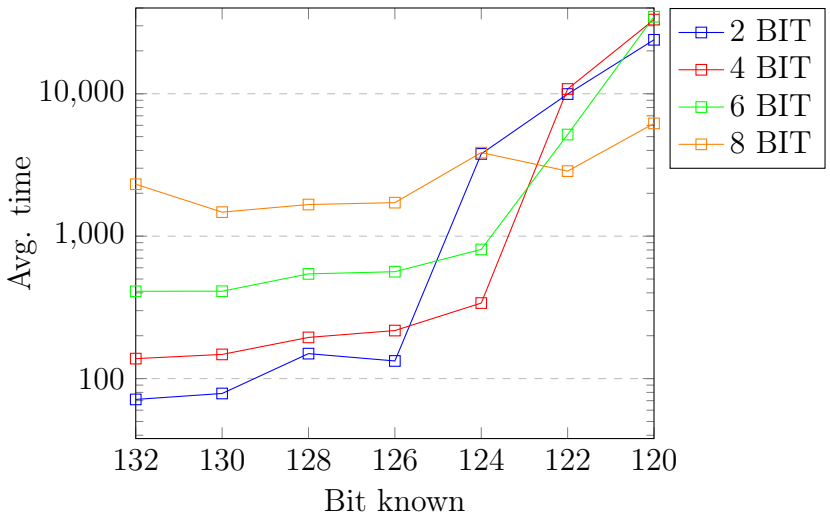
Bit known		132	130	128	126	124	122	120
Kissat	Min	2021	1288	1600	1643	2168	2510	3064
	Max	2732	1799	1750	1886	8508	3272	27342
	Avg.	2315	1475	1668	1719	3861	2862	6189
	St.Dev	228.1	181.8	66.9	113.0	3100	247.92	6950
	Med.	2370	1429	1661	1673	2384	2796	3992
	# Tests	10	7	4	4	4	5	10

Table 5.13: Run-time statistic for Mixed approach on 8 bit

5.3.2 Comparing avg run-time

The graph below shows the trend of the average times as a function of the known bits.

Avg. time in function of different number of bit instanced



Analyzing the above graph, from 132 bits to 126 bits it is advisable to brute force only on 2 bits, this suggests that in this range further subdividing the problem into more than 4 subproblems does not bring any advantage, indeed it does not bring any advantage, indeed it only increases the time needed.
Continuing with the analysis, from 124 known bits onwards, as can be seen from the graph, it is advisable to increase the cardinality of the bits on which brute force is performed, for example with 124 known bits it is faster to brute force on 4 bits while with 122 known bits it is faster to instantiate 8 bits for brute force.
In the last known bits circumstance, namely 120, by instantiating 8 bits we obtain an average time of 6189 seconds with a decrease of 74% (higher is better) with respect to the analogous case by instantiating 2 bits. This improvement allows us to believe that even in cases more extremes in which less than 120 bits are known, it will be better to brute force by increasing the cardinality of the instantiated bits more and more.
In the previous approaches we noticed an increase in terms of time when the sum between instantiated bits and known bits is equal to 126, also in this technique this increase can be found in the same circumstances.

5.4 Random bit instancing

This technique consists in carrying out the brute force on N bit as in the previously seen approaches, the difference is that the bits are not instantiated close to the known information as it happened in the previous approaches, but rather happens randomly. When the combinations are generated, N bits are then randomly chosen over the entire Key schedule (obviously excluding the bits already known and the bits that are part of the starting key), and the CNF files are generated by populating these bits with the generated combinations. Therefore, for each CNF generated, proceed as in the previous approach by launching an execution of the Banning solution script discuss in section 5.1.
This technique through randomism aims to flatten the complexity/computation time of each generated subproblem, however the effectiveness of this approach is not taken for granted and will be determined in the testing phase.
The following image depicts the following scenario, the asterisks indicate the bits whose value is unknown while the red bits are the randomly instantiated bits (which are brute-force), the number of known and unknown bits, represented in the image is not significant.

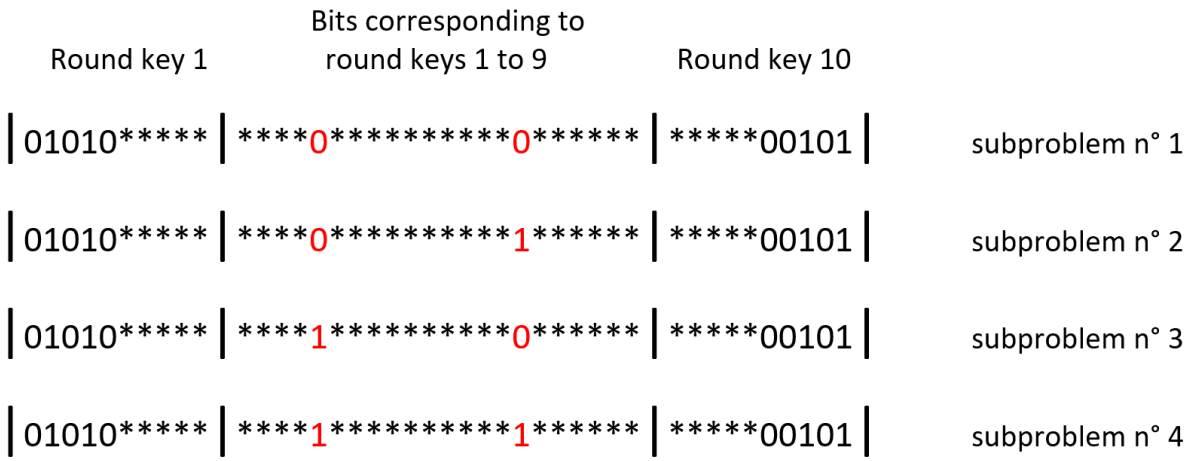


Figure 5.2: Random sub-problem generated

The next page shows the pseudocode of the Random technique just described.

Algorithm 4 Random bit instancing pseudocode

```
TaskList = []
C = Calculate and Store all combination with N bit instanced
D = Choose at random the location of N bit to be instantiated
while C is not empty do
    Create Task = {
        1) Pick one combination from C
        2) Create CNF accordingly to the combination taken and to D
        3) Launch Banning solution script with the generated CNF
    }
    Add Task to TaskList
end while
for each Task in TaskList do
    Print solution of Task
end for
```

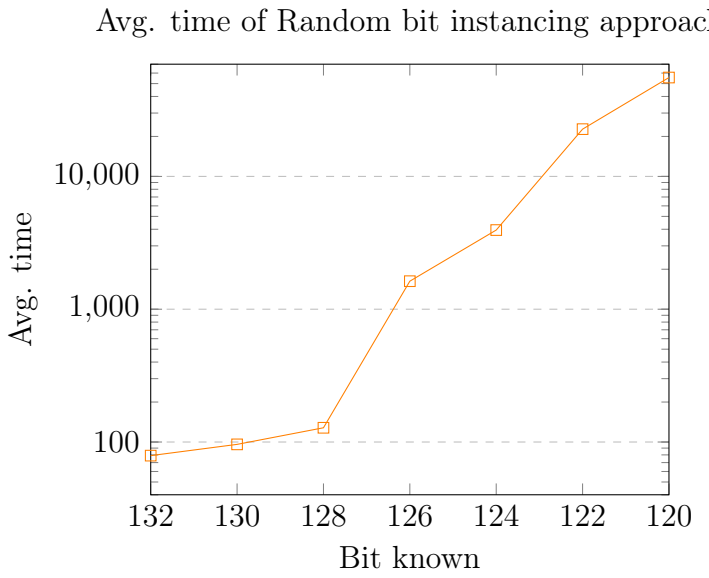
5.4.1 Tests result

In the tests conducted in the mixed approach we were able to observe which is the best number of bits to instantiate in each circumstance of known bits. In conducting these tests, therefore, we will limit ourselves to using this technique only by instantiating the number of bits that was faster in the tests carried out on the mixed approach, i.e., the following configurations will be used:
132 2 bit, 130 2 bit, 128 2 bit , 126 2-bit, 124 4-bit, 122 8-bit, 120 8-bit.
This has the aim of verifying in the next chapter if through this technique there is a further improvement in the time taken by the solver.

Bit known		132	130	128	126	124	122	120
Bit instanced		2	2	2	2	4	8	8
Kissat	Min	42.2	64.48	62.2	351.0	1704	3637	54728
	Max	141.7	146.7	169.59	2802	6045	55097	56352
	Avg	79.66	96.81	128.3	1626	3946	22698	55540
	St.Dev	28.71	28.54	42.2	1055	1601	20301	812
	Med.	78.50	84.6	130.3	1998	4326	17519	55540
	# Tests	10	7	5	5	5	5	2

Table 5.14: Run-time statistic for Random bit instancing

The next page shows the trend of the average times as a function of the known bits.



The graph is very similar to the one drawn for the banning solution approach. In fact, in the same way the largest increase is from 128 to 126 known bit where there is an increase of 1167 %, the second increase in order of magnitude is from 124 to 122 known bits in which there is an increase in average times of 475%.

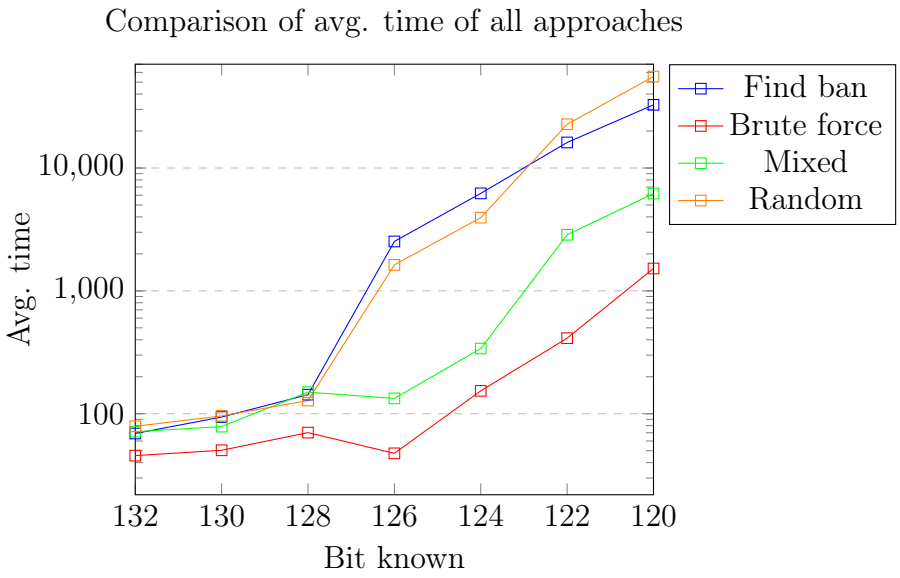
In the next chapter the average times of this approach will be compared with the statistics of the other developed techniques, this has the aim of understanding if the following approach is more or less effective than the other developed approaches.

Chapter 6

Comparison of different technique

6.1 Comparison of all techniques in scenarios with multiple solutions

In this section a comparison is made between all the approaches seen previously, in this comparison the tests carried out from 132 to 120 known bits will be taken into consideration, range in which many solutions are admissible. This comparison aims to highlight which is the fastest method to obtain all candidate solutions.
The graph in the next page shows the comparison of the average times between all the various approaches introduced in the previous chapter, the following approaches: brute force and mixed approach, are represented in the graph in their best configuration. By best configuration we mean, use the number of instantiated bits/brute force that allow to obtain the lowest time in the case of the mixed approach, while for the brute force approach we mean use the number of instantiated bits/brute force that allows us to obtain the highest value in the avg solution found/avg time metric introduced in the following paragraph 5.2.2
Noting the graph Avg. solution found/avg. time the best configuration for the brute force approach is therefore:
132 2 bit, 130 2 bit, 128 2 bit, 126 2 bit, 124 4 bit, 122 6 bit, 120 8 bit.
Noting the graph Avg. time in function of different number of bit instanced the best configuration for the mixed approach is therefore:
132 2 bit, 130 2 bit, 128 2 bit, 126 2 bit, 124 4 bit, 122 8 bit, 120 8 bit.



From the above graph as expected, the fastest technique in all cases of known bits turns out to be the brute force technique. However, remember that it does not find all possible solutions, but a subset of them, which is strictly related to the number of bits instantiated in the brute force.
Moving on to the comparison between the mixed approach and the random approach, we can see how the expected improvement in using the random technique did not occur. It therefore seems that the randomism component introduced did not produce the hoped improvements, moreover, compared to the mixed approach, the standard deviation has increased on average, making the average times of the tests very different from each other. We can therefore conclude that the best approach that allows to find all the feasible solutions with the shortest time is therefore the mixed approach, where from 126 to 120 known bits demonstrates its best effectiveness compared to the Banning solution approach.
More precisely, using the mixed approach we get a decrease in avg time of 94%, 94%, 82%, 81% (higher is better), in scenarios with 126, 124, 122, 120 known bits with respect to the analogous scenario using banning solution approach.

6.2 Comparing single SAT solver instance VS brute force approach in scenarios with max one solution

This section compares the brute force approach introduced in chapter 5.2 with a simple call to the Kissat SAT solver with the generated CNF.
This comparison has the aim of highlighting which is the fastest method to obtain the expected solution.
As already anticipated by the title of the section in this comparison the tests will be carried out from 490 to 140 known bits, range in which only one solution is admissible.

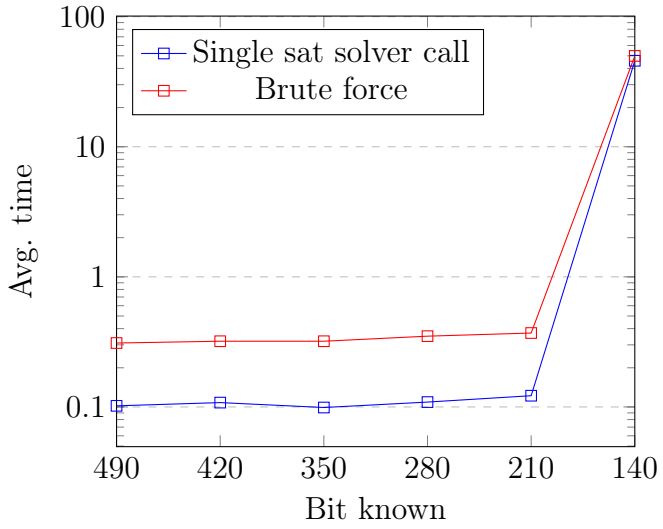
In the table below there are therefore the statistics of the tests conducted through a simple call to the solver, 10 tests were performed for each known bit scenario.

Bit known		490	420	350	280	210	140
Kissat	Min	.076	.077	.064	.079	.088	16.8
	Max	.194	.159	.140	.169	.151	83.9
	Avg	.102	.108	.099	.109	.122	45.8
	St.Dev	.035	.024	.022	.027	.022	23.9
	Med.	.091	.103	.097	.102	.129	43.2
	# Tests	10	10	10	10	10	10

Table 6.1: Run-time statistic for single SAT solver instance (Kissat)

The graph below depicts the comparison of the average times between the brute force approach (in its best configuration) and a single call to the solver. For best configuration we mean, as in the previous cases, to use the number of instanced/brute force bits that allow to obtain the shortest time.
Noting the graph Avg. time in function of diferrent number of bit instanced the best configuration is therefore:
490 2 bit, 420 2 bit, 350 2 bit, 280 2 bit, 210 2 bit, 140 4 bit.

Comparison between brute force approach and single sat solver call



Through this comparison we can therefore conclude that in the range from 490 to 140 known bits a single call to the sat solver is faster than using the brute force approach.
We have to keep in mind the clarification made earlier on the possible overhead due to the creation of the copy files.

6.3 Comparing our best approach VS Incremental SAT solver

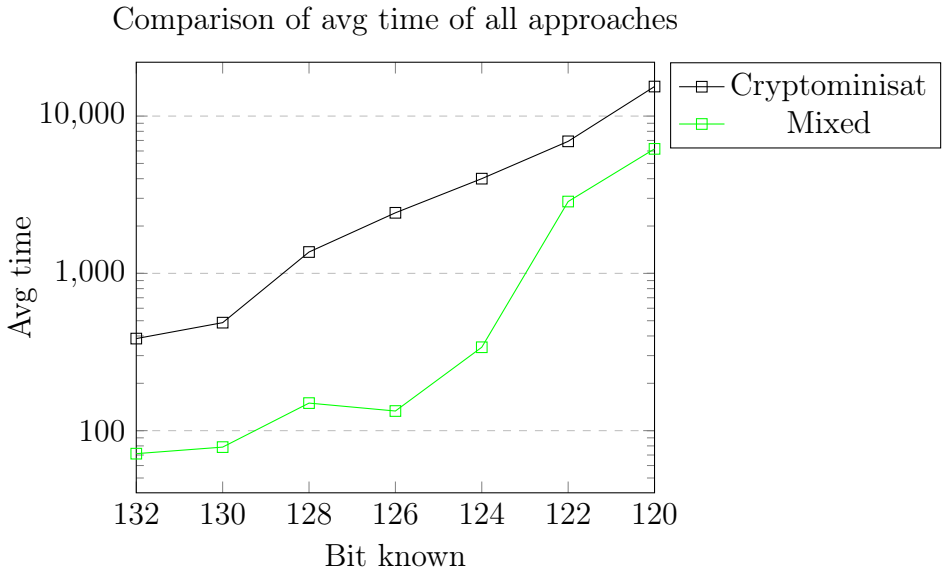
In this section, the average times of the Mixed approach, which turns out to be the most effective, will be compared with an incremental SAT solver: Cryptominisat. As previously mentioned, an incremental SAT solver is a particular type of SAT solver that is able to find more than one solution for an instance, efficiently.
So, to carry out the tests with Cryptominisat just specify the CNF file as a parameter and indicate to Cryptominisat that we want to find all the possible solutions.
The goal of this comparison is therefore to understand if it is more effective in a scenario where part of an AES key schedule is known, to use an incremental SAT solver or the mixed approach developed in this paper.
Since Cryptominisat supports XOR clauses, the corresponding CNFs used in the tests of the previous approaches are used for the current tests with the XOR clauses.

The table below shows the statistics of the tests conducted through a simple call to the Cryptominisat solver.

Bit known		132	130	128	126	124	122	120
Kissat	Min	221.1	306.1	997.0	1036	1245	6058	11516
	Max	1115	865.9	1802.8	3883	5787	7601	19312
	Avg	385.8	486.4	1367	2427	4001	6910	15413
	St.Dev	274.3	181.9	352.5	1248	1777	783.7	5512
	Med.	270.6	422.6	1307	2974	4382	7070	15413
	# Tests	10	7	5	5	5	3	2

Table 6.2: Run-time statistic for single SAT solver instance (Cryptominisat)

The graph below shows the comparison of the average times between the Mixed approach (in its best configuration) and a single call to the Cryptominisat solver.



From the above graph we can see that in this range of known bits the mixed approach developed in this paper turns out to be more effective in terms of time than Cryptominisat, the best incremental SAT solver, winner of the Incremental track 2020.
More precisely, the gap is greater in the range from 132 to 124 known bits, and decreases in the tests where the known bits are 122 and 120 bits. The percentage decrease in avg. times for Mixed approach with respect to Cryptominisat time is: 81% with 132 bit, 83% with 130 bit, 89% with 128 bit, 94% with 126 bit, 91% with 124 bit, 58% with 122 bit, 59% with 120 bit (higher is better).
This trend of the percentages would suggest that with a number of known bits less than 120, the gap of the average times of the 2 techniques can be reduced more and more until they are equivalent in time, but this cannot be stated with certainty because the number of instantiated bits could also be increased to 10 or 12 and thus be able to maintain the relatively large gap. To remove any doubt, further tests would be required on scenarios in which the known bits are less than 120.

Chapter 7

Conclusion

This chapter illustrates the conclusions concerning the topics addressed in the various chapters of this thesis.
In the first chapter after the introduction, a brief presentation is made on the functioning of the AES key schedule and on the SAT problem, this introduction is necessary to understand how these 2 elements can be used together.
In the next chapter the previous works are retraced, concerning data remanence of RAM and on the correction of errors in a corrupt AES key schedule. This chapter is useful to understand what the following thesis wants to deepen and add to the literature.
Chapter 4 introduces preliminary information on the creation of CNF files and on the hardware and software used for conducting the Tests.
In the next chapter, the techniques developed in this thesis work are presented. For each approach there is a detailed high-level explanation, a pseudocode and the run-time statistic of the various tests conducted. These statistics are necessary to make the various comparisons that will take place in chapter 6, moreover in some approaches the tests have been necessary to understand the best configuration in each known and instantiated bit scenario. In the tests carried out in each approach it was noted that the execution time increases in a sub-exponential way as a function of the known bits. Furthermore, in the Banning solution approach, a large increase in the average times from 126 to 124 known bits was noted, this increase is also visible in the other techniques that make use of brute forcing, with the difference that this increase can be noticed in the configurations in which the sum between instantiated bits and known bits is equal to 126. From the tests carried out we can conclude that the subdivision of a problem into several subproblems is effective.
Continuing to the next chapter, in chapter 6 first of all a comparison is made between all the techniques developed, this is to understand which is the most performing approach among those introduced. From this comparison it emerges that the best performing approach that allows to obtain all the admissible keys is the Mixed approach, which in the more complex scenario (120 known bits) decreases the time required for computation by 81% compared to the Banning Solution technique.
Secondly, again in the same chapter, a comparison is made between a single call to the SAT solver Kissat and the brute force approach in scenarios where there is a maximum of one solution, from the graph it is quickly understood that the single call to the SAT solver is more efficient. We can therefore conclude by saying that in

scenarios with many known bits, in which only one solution is admissible, it is not convenient to divide the starting problem into several subproblems. Recall again that the processing times collected also include the creation of the copy files, this could create a lot of overhead, in scenarios where the processing time is very low (<1 sec). In view of this, a future job could be to repeat the tests from 490 to 140 bits known without considering the time needed to create the files, which could take place offline that is before making the calls to the SAT solver.
Thirdly, the comparison is made between a single call to the incremental SAT solver Cryptominisat and the mixed approach, from the comparison it is evident that the Mixed approach is more performing than Cryptominisat. This comparison establishes how the Mixed approach developed in this paper is more effective than the best incremental SAT solver, in a scenario where a part of an AES key schedule is know and want to get the complete information. More precisely in the more complex scenario (120 known bits), there is a 59% decrease in the average times necessary for the computation of the solutions, using the Mixed approach instead of the SAT solver Cryptominisat.
In conclusion, this thesis therefore presents new techniques that can however be developed and combined in many other different ways: position of instantiated bits, combined approaches, etc, giving light to many different types and scenarios. Also given the nature of CNF files in DIMACS format, they could be tested on many other different SAT solvers and on different machines.

Bibliography

[1] J. Daemen and V. Rijmen, "The block cipher rijndael," vol. 1820, pp. 277–284, 01 1998.

[2] J. Daemen and V. Rijmen, *The Design of Rijndael: AES - The Advanced Encryption Standard (Information Security and Cryptography)*. Springer, 1 ed., 2002.

[3] N. I. of Standards and Technology, "Advanced encryption standard," *NIST FIPS PUB 197*, 2001.

[4] AES Rijndael Cipher explained as a Flash animation. https://www.youtube.com/watch?v=gP4PqVGudtg&t=48s&ab_channel=AppliedGo.

[5] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, (New York, NY, USA), p. 151–158, Association for Computing Machinery, 1971.

[6] The international SAT Competitions web page. Available at. http://www.satcompetition.org/.

[7] Dimacs format. http://beyondnp.org/static/media/uploads/docs/satformat.pdf.

[8] W. LINK and H. MAY, "Eigenschaften von mos-eintransistorspeicherzellen bei tiefen temperaturen," Archiv fur Elektronik und Ubertragungstechnik 33, pp. 229-235, June 1979.

[9] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum, "Shredding your garbage: Reducing data lifetime through secure deallocation," in *14th USENIX Security Symposium (USENIX Security 05)*, (Baltimore, MD), USENIX Association, July 2005.

[10] PETTERSON, "T. cryptographic key recovery from linux memory dumps," in Presentation, Chaos Communication Cam, August 2007.

[11] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: Cold-boot attacks on encryption keys," *Commun. ACM*, vol. 52, p. 91–98, may 2009.

[12] P. Gutmann, "Secure deletion of data from magnetic and Solid-State memory," in *6th USENIX Security Symposium (USENIX Security 96)*, (San Jose, CA), USENIX Association, July 1996.

[13] P. Gutmann, "Data remanence in semiconductor devices," in *10th USENIX Security Symposium (USENIX Security 01)*, (Washington, D.C.), USENIX Association, Aug. 2001.

[14] A. Tsow, "An improved recovery algorithm for decayed aes key schedule images," in *Selected Areas in Cryptography* (M. J. Jacobson, V. Rijmen, and R. Safavi-Naini, eds.), (Berlin, Heidelberg), pp. 215–230, Springer Berlin Heidelberg, 2009.

[15] A. A. Kamal and A. M. Youssef, "Applications of sat solvers to aes key recovery from decayed key schedule images," in *2010 Fourth International Conference on Emerging Security Information, Systems and Technologies*, pp. 216–220, 2010.

[16] X. Liao, H. Zhang, M. Koshimura, H. Fujita, and R. Hasegawa, "Using maxsat to correct errors in aes key schedule images," in *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*, pp. 284–291, 2013.

[17] External script used to generate round key. `https://github.com/fanosta/aeskeyschedule`.

[18] Kissat web page. Available at. `http://fmv.jku.at/kissat/`.

[19] Cryptominisat5 web page. Available at. `https://www.msoos.org/cryptominisat5/`.

[20] The Sage Developers, *SageMath, the Sage Mathematics Software System (Version 9.5)*, 2022. `https://www.sagemath.org`.

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

[22] S. P. Skorobogatov, "Low temperature data remanence in static ram," 2002.

[23] T. Eibach, E. Pilz, and G. Völkel, "Attacking bivium using sat solvers," in *Theory and Applications of Satisfiability Testing – SAT 2008* (H. Kleine Büning and X. Zhao, eds.), (Berlin, Heidelberg), pp. 63–76, Springer Berlin Heidelberg, 2008.

[24] T. Vidas, "The acquisition and analysis of random access memory," *Journal of Digital Forensic Practice*, vol. 1, no. 4, pp. 315–323, 2007.

Appendix A

Source code

A.1 Python scripts to generate CNF files

Below is the source code of the python scripts used to generate the CNF files.

GenerateRandomCNF.py

```
from sage.crypto.sboxes import AES
import random
import io
from aeskeyschedule2 import getSchedule
import os
import binascii
from random import randbytes
from platform import release

varcounter=1
outfile="outputXor.cnf"
kwowbit = 60

def newvars():
    global varcounter
    n = [j for j in range(varcounter,varcounter+32)]
    varcounter += 32
    return n

def AssignRoundKey(Words,KeyValue,KeyRound,Start,End):

    baseKey = Words[KeyRound*N][0]
    KeyWords = []
    for w in Words[KeyRound*N:(KeyRound+1)*N]:
        KeyWords += w

    for i in range(Start,End):
        assert(baseKey+i == KeyWords[i])

        if KeyValue[i] == '1':
            Model2.append(f'{baseKey+i} 0')
        else:
            Model2.append(f'-{baseKey+i} 0')


Words = []
Model = []
Model2 = []
N = 4
R = 10

Bcon = [0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80,0x1B,0x36]
Rcon = ['']+[f'{x:08b}'+'0'*24 for x in Bcon]

```

```
45 for i in range((R+1)*N):
46     if i<N:
47         Words.append(newvars())
48     elif i%N == 0:
49 
50         rot = Words[-1][8:]+Words[-1][:8]
51         n1 = newvars()
52         n2 = newvars()
53         for j in range(N):
54             Model += (AES.cnf(xi=rot[j*8:(j+1)*8],yi=n1[j*8:(j+1)*8],format='
    dimacs_headless').split('\n')[:-1])
55 
56         for b in range(32):
57             if Rcon[i//N][b]=='1':
58                 Model.append(f'x{Words[-N][b]} {n1[b]} {n2[b]} 0')
59             else:
60                 Model.append(f'x{Words[-N][b]} {n1[b]} -{n2[b]} 0')
61         Words.append(n2)
62     else:
63         n2=newvars()
64         for b in range(32):
65             Model.append(f'x{Words[-N][b]} {Words[-1][b]} -{n2[b]} 0')
66         Words.append(n2)
67 
68 
69 for i in range(10):
70     print("File " + str(i) + " ...")
71     Model2 = []
72 
73     Bytekey = randbytes(16)
74     keyschedule = getSchedule(0, Bytekey)
75 
76     Bkey = bin(int(keyschedule[0], 16))[2:]
77     Bkey = Bkey.zfill(128)
78 
79     Bkey1 = bin(int(keyschedule[1], 16))[2:]
80     Bkey1 = Bkey1.zfill(128)
81 
82     Bkey10 = bin(int(keyschedule[10], 16))[2:]
83     Bkey10 = Bkey10.zfill(128)
84 
85     AssignRoundKey(Words,Bkey1,1,0,kwowbit)
86     AssignRoundKey(Words,Bkey10,10,128-kwowbit,128)
87 
88     f =  open( "test/" +str(i) + outfile  , 'w', encoding='utf-8')
89     f.write(f'p cnf {varcounter-1} {len(Model)}\n')
90     for c in Model:
91         f.write(c+'\n')
92     for c in Model2:
93         f.write(c+'\n')
94 
95     f.close()
96 
97     f = io.open("test/" + str(i) + "outputXor.cnf", "r",  newline='\n')
98     allOfIt = f.readlines()
99     f.close()
100 
101     cnfClause = []
102     xorClause = []
103 
104     for line in allOfIt:
105         if line.find("x") != -1:
106             xorClause.append(line)
107         else:
108             cnfClause.append(line)
109 
110 
111     for line in xorClause:
112         line = line.replace("x", "")[:-2]
113 
114         if line.find("-") != -1:
```

```
            appo = line.split(" ")
            clause= "-" + appo[0] + " -" + appo[1] + " " + appo[2] + " 0\n"
            cnfClause.insert(1,clause)
            clause= "-" + appo[0] + " " + appo[1] + " " + appo[2][1:] + " 0\n"
            cnfClause.insert(1,clause)
            clause= "" + appo[0] + " -" + appo[1] + " " + appo[2][1:] + " 0\n"
            cnfClause.insert(1,clause)
            clause= "" + appo[0] + " " + appo[1] + " " + appo[2] + " 0\n"
            cnfClause.insert(1,clause)

        else:
            appo = line.split(" ")
            clause= "-" + appo[0] + " -" + appo[1] + " " + appo[2] + " 0\n"
            cnfClause.insert(1,clause)
            clause= "-" + appo[0] + " " + appo[1] + " -" + appo[2] + " 0\n"
            cnfClause.insert(1,clause)
            clause= "" + appo[0] + " -" + appo[1] + " -" + appo[2] + " 0\n"
            cnfClause.insert(1,clause)
            clause= "" + appo[0] + " " + appo[1] + " " + appo[2] + " 0\n"
            cnfClause.insert(1,clause)


    f = open("test/" + str(i)+"output.cnf", 'w', newline='\n')
    for x in cnfClause:
        f.write(x)
    f.close()

    #for print solution of CNF
    #f = open(str(i)+"solution.txt", 'w', newline='\n')
    #f.write(Bkey)
    #f.close()

    #in case you don't need xor cnf de-comment the next line
    #os.remove(str(i) + "outputXor.cnf")

print("Done!")
```

aeskeyschedule.py

File belonging to the external library [17], it is used to generate the keyschedule starting from a cryptographic key.

```
from typing import List
from functools import reduce

sbox = [
    0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0
    xd7, 0xab, 0x76,
    0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0
    xa4, 0x72, 0xc0,
    0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0
    xd8, 0x31, 0x15,
    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0
    x27, 0xb2, 0x75,
    0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0
    xe3, 0x2f, 0x84,
    0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0
    x4c, 0x58, 0xcf,
    0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0
    x3c, 0x9f, 0xa8,
    0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0
    xff, 0xf3, 0xd2,
    0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0
    x5d, 0x19, 0x73,
    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0
    x5e, 0x0b, 0xdb,
    0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0
    x95, 0xe4, 0x79,
    0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0
    x7a, 0xae, 0x08,
```

```
17      0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0
        xbd, 0x8b, 0x8a,
18      0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0
        xc1, 0x1d, 0x9e,
19      0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0
        x55, 0x28, 0xdf,
20      0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0
        x54, 0xbb, 0x16,
21  ]
22
23  inv_sbox = [
24      0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81, 0
        xf3, 0xd7, 0xfb,
25      0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4, 0
        xde, 0xe9, 0xcb,
26      0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42, 0
        xfa, 0xc3, 0x4e,
27      0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0
        x8b, 0xd1, 0x25,
28      0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0
        x65, 0xb6, 0x92,
29      0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7, 0
        x8d, 0x9d, 0x84,
30      0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8, 0
        xb3, 0x45, 0x06,
31      0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01, 0
        x13, 0x8a, 0x6b,
32      0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0
        xb4, 0xe6, 0x73,
33      0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0
        x75, 0xdf, 0x6e,
34      0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0
        x18, 0xbe, 0x1b,
35      0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0
        xcd, 0x5a, 0xf4,
36      0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0
        x80, 0xec, 0x5f,
37      0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0
        xc9, 0x9c, 0xef,
38      0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0
        x53, 0x99, 0x61,
39      0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63, 0x55, 0
        x21, 0x0c, 0x7d,
40  ]
41
42  rcon = [x.to_bytes(4, 'little') for x in [ 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40
        , 0x80, 0x1B, 0x36, ]]
43
44  def xor_bytes(*arg: bytes) -> bytes:
45      assert len({len(x) for x in arg}) == 1 # all args must have the same length
46      xor_fun = lambda x, y : x ^ y
47      return bytes(reduce(xor_fun, byt3s) for byt3s in zip(*arg))
48
49  def rot_word(word: bytes) -> bytes:
50      '''
51      apply the RotWord transformation to a bytes object of length 4
52      '''
53      assert len(word) == 4
54      return bytes((word[(i + 1) % 4] for i in range(4)))
55
56  def inv_rot_word(word: bytes) -> bytes:
57      '''
58      apply the inverse of the RotWord transformation to a bytes object of length 4
59      '''
60      assert len(word) == 4
61      return bytes((word[(i - 1) % 4] for i in range(4)))
62
63  def sub_word(word: bytes) -> bytes:
64      '''
65      apply the AES S-Box to each of the bytes of the 4-byte word
66      '''
```

```
    assert len(word) == 4
    return bytes((sbox[w] for w in word))

def inv_sub_word(word: bytes) -> bytes:
    '''
    apply the inverse of the AES S-Box to each of the bytes of the 4-byte word
    '''
    assert len(word) == 4
    return bytes((inv_sbox[w] for w in word))


def reverse_key_schedule(round_key: bytes, aes_round: int):
    '''
    reverse the AES-128 key schedule, using a single round_key.
    '''
    assert len(round_key) * 8 == 128
    for i in range(aes_round - 1, -1, -1):
        a2 = round_key[0:4]
        b2 = round_key[4:8]
        c2 = round_key[8:12]
        d2 = round_key[12:16]

        d1 = xor_bytes(d2, c2)
        c1 = xor_bytes(c2, b2)
        b1 = xor_bytes(b2, a2)
        a1 = xor_bytes(a2, rot_word(sub_word(d1)), rcon[i])

        round_key = a1 + b1 + c1 + d1

    return round_key

def key_schedule(base_key: bytes) -> List[bytes]:
    '''
    calculate the expanded AES key given the base key.
    Depending on the length of the base key 11, 13 or 15 round keys are returned
    for AES-128, AES-192 and AES-256 respectively.
    '''
    assert len(base_key) * 8 in {128, 192, 256}

    # length of the key in 32 bit words
    N = {128: 4, 192: 6, 256: 8}[len(base_key) * 8]

    # number of round keys needed
    R = {128: 11, 192: 13, 256: 15}[len(base_key) * 8]

    # the 32 bits words of the expanded key
    W = [None for _ in range(R * 4)]

    for i in range(N):
        W[i] = base_key[i * 4 : (i + 1) * 4]

    for i in range(N, 4 * R):
        if i % N == 0:
            W[i] = xor_bytes(W[i - N], sub_word(rot_word(W[i - 1])), rcon[i // N  -
 1])
        elif N > 6 and i % N == 4:
            W[i] = xor_bytes(W[i - N], sub_word(W[i - 1]))
        else:
            W[i] = xor_bytes(W[i - N], W[i - 1])

    keys = [b''.join(W[i * 4 + j] for j in range(4)) for i in range(R)]
    return keys
```

aeskeyschedule2.py

File belonging to the external library [17], it is used to generate the keyschedule starting from a cryptographic key.

```
import argparse
from binascii import hexlify, unhexlify
```

```
from aeskeyschedule import key_schedule, reverse_key_schedule

import sys

try:
    import colorama
    colorama.init()
    __highlight = colorama.Style.BRIGHT
    __reset = colorama.Style.RESET_ALL
except ImportError:
    __highlight = ''
    __reset = ''


def aes_round(value: str) -> int:
    aes_round = int(value)
    if not 0 <= aes_round <= 10:
        raise argparse.ArgumentError('the aes round must satisfy 0 <= r <= 10')
    return aes_round

def aes_key(value: str) -> bytes:
    if value.startswith('0x'):
        value = value[2:]
    try:
        key = unhexlify(value)
    except TypeError:
        raise argparse.ArgumentError('invalid hex bytes in aes key')
    if len(key) * 8 not in {128, 192, 256}:
        raise argparse.ArgumentError('''
            AES key must be 128, 192 or 256 bits long (is {} bits)
            '''.strip().format(len(key) * 8))
    return key


def getSchedule(aes_round: int, round_key: bytes) -> None:
    if len(round_key) * 8 != 128 and aes_round != 0:
        print("reversing the AES-{} key schedule is not supported".format(len(
    base_key) * 8, file=sys.stderr))
        sys.exit(1)


    if aes_round != 0:
        base_key = reverse_key_schedule(round_key, aes_round)
    else:
        base_key = round_key
    keys = key_schedule(base_key)

    assert keys[aes_round] == round_key or len(base_key) * 8 != 128

    total = []

    for i, key in enumerate(keys):
        if i == aes_round:
            print(__highlight, end='')
        total.append(hexlify(key).decode())
        if i == aes_round:
            print(__reset, end='')
    return total

print(getSchedule(0, b'00000000000000000000000000000000' ))
```

A.2 Python scripts to conduct test on all approaches

Below is the source code of the python scripts used to run the tests.
Each script contains absolute paths to CNF files and sat solver programs, these paths must be modified for the script to work correctly on other machines.
The scripts are designed to perform the calculation on multiple CNF files present in the same folder.
Some scripts only return the time necessary to calculate the solutions, if it is also necessary to know the value of the solutions, the script must be modified.
Other scripts return also the solution, in some cases the solution may contain non-printable asci characters. When these solutions are written on the file that contains the solutions/times necessary for processing, they could corrupt it, it is therefore recommended to open the solution file using notepad++ which allows you to open even corrupt text files.
It is also recommended for the correct understanding of the operating logic of the scripts to run them in debug mode.

BanningSolution.py

```
from ast import While
import subprocess
from platform import release
import shutil
import time
import os
from os.path import exists

def toBinaryAux(clause):
    if '-' in clause:
        return 1
    else:
        return 0

def toBinary(list):
    tot = ""

    for i in list:
        tot = tot + str(toBinaryAux(i))

    return tot


totalStart = time.time()

for cont in range(0,1000):

    Nsol =0

    #Timer variable
    start = time.time()

    #File input variable
    fileIn = "/mnt/hgfs/Tesi/Test/TestTesiNuovi/120/"+ str(cont) +"output.cnf"
    if(not exists(fileIn)):
        break
  #File in wich clause will be added
    fileBak = "/mnt/hgfs/Tesi/Test/TestTesiNuovi/120/"+ str(cont) +"output"+"2"+".
    cnf"

    print("Create copy of the cnf file...")
    shutil.copyfile(fileIn, fileBak)
    file_object = open(fileBak, 'a',  newline='\n')
```

```
43  #file in which time are printed
44    solutionFile = open('/mnt/hgfs/Tesi/Test/TestTesiNuovi/120/'+ str(cont) +'
      solution2.txt', 'w',  newline='\n')
45
46    print("Start working on file "+ str(cont))
47
48    while(True):
49        ########################################################## KISSAT
50        result  = subprocess.run(['/mnt/hgfs/Tesi/Test/Ubuntu/kissat-master/build/
      kissat', fileBak,  "--relaxed", "-q"], stdout=subprocess.PIPE).stdout.decode('
      utf-8')
51        ################################################################
52
53        try:
54            index = result.index("v ")
55        except Exception:
56            break
57
58        Nsol = Nsol +1
59        print("Solution N" + str(Nsol) + " found..." +  " {" +str(time.time() -
      start)+"}" )
60
61        # ADD NEW CLAUSE TO CNF
62        result = result[result.index("v") : result.index("1728")+4]
63        result = result.strip()
64        result = result.replace("v", "")
65        result = result.replace("\n", "")
66        result = result.replace("\r", "")
67        result = result.replace(" -", "+")
68        result = result.replace(" ", " -")
69        result = result.replace("+", " ")
70
71      # WRITING SOLUTION TO FILE
72        file_object.write(result[1:] + " 0\n")
73
74        try:
75            solution = result[:result.index("-129 ")]
76        except Exception:
77            solution = result[:result.index(" 129 ")]
78
79        solutionFile.write("SOLUTION N"+ str(Nsol) +  " [" +str(time.time() - start
      )+"]\n")
80        solution = solution.strip()
81        # BINARY
82        tot = toBinary(solution.split())
83        solutionFile.write(tot+"\n")
84        print(tot)
85        #HEX
86        decimal_representation = int(tot, 2)
87        hexer = hex(decimal_representation)
88        hexer = hexer[2:]
89        for j in range(len(hexer), 32):
90            hexer = "0" + hexer
91        solutionFile.write(hexer+"\n")
92        print(hexer)
93
94        #ASCI STRING
95        bytes_object = bytes.fromhex(hexer)
96        ascii_string = bytes_object.decode('utf-8', 'ignore')
97        solutionFile.write(ascii_string+"\n\n")
98        print(ascii_string + "\n" )
99
100
101   end = time.time()
102   appo = end - start
103   print("\nTime for computing solution on file "+ str(cont) + " : "+ str(appo) +
      " s\n")
104
105   solutionFile.write("Total time: "+ str(appo) + " s \n")
106
107   file_object.close()
```

```
    solutionFile.close()

    os.remove(fileBak)


totalEnd = time.time()
final = totalEnd - totalStart

print("Total time: " + str(final))
print("Done!")
```

Bruteforce.py

```
from ast import While
import subprocess
from platform import release
import shutil
import time
import os
import math
import threading
from os.path import exists
from multiprocessing.dummy import Pool as ThreadPool
from statistics import mean, median, stdev
import sys


################################################
## BINARY, CONBINATION
################################################

def single_task(j):
    global sol

    shutil.copyfile(fileIn, fileIn[:-4]+ str(j) + ".cnf")
    file_object = open(fileIn[:-4]+ str(j) + ".cnf", 'a',  newline='\n')
    for k in range(SBkey1, int(SBkey1+BitToInfer/2)):
        if(combination[j][k-SBkey1] == '0'):
            file_object.write("-" + str(k) + " 0\n" )
        else:
            file_object.write(str(k) + " 0\n" )

    for k in range(SBkey10, int(SBkey10+BitToInfer/2)):
        if(combination[j][int(BitToInfer/2) + k - SBkey10] == '0'):
            file_object.write("-" + str(k) + " 0\n" )
        else:
            file_object.write(str(k) + " 0\n" )
    file_object.close()

    output = subprocess.run(['/mnt/hgfs/Tesi/Test/Ubuntu/kissat-master/build/kissat
    ', fileIn[:-4]+ str(j) + ".cnf" ,  "--relaxed", "-q"], stdout=subprocess.PIPE).
    stdout.decode('utf-8')
    os.remove(fileIn[:-4]+ str(j) + ".cnf")

    try:
        output.index("s SATISFIABLE")
    except Exception:
        return

    output = output[output.index("v"):]
    output = output.replace("v", "")
    output = output.replace("\n", "")
    output = output.replace("\r", "")

    try:
        solution = output[:output.index("-129 ")]
    except Exception:
        solution = output[:output.index(" 129 ")]

    solutionFile.write("SOLUTION N"+ str(sol+1)+ "\n")
```

```
56      print("SOLUTION N"+ str(sol+1)+ "    [" +str(j) + "/" + str(Nfile)  +"]\n")
57      solution = solution.strip()
58      # BINARY
59      tot = toBinary(solution.split())
60      solutionFile.write(tot+"\n")
61      print(tot)
62      #HEX
63      decimal_representation = int(tot, 2)
64      hexer = hex(decimal_representation)
65      hexer = hexer[2:]
66      for j in range(len(hexer), 32):
67          hexer = "0" + hexer
68      solutionFile.write(hexer+"\n")
69
70      print(hexer)
71
72      #ASCI STRING
73      bytes_object = bytes.fromhex(hexer)
74      ascii_string = bytes_object.decode('utf-8', 'ignore')
75      solutionFile.write(ascii_string+"\n\n")
76      print(ascii_string + "\n" )
77      sol = sol+1
78
79
80  def toBinaryAux(clause):
81      if '-' in clause:
82          return 0
83      else:
84          return 1
85
86  def toBinary(list):
87      tot = ""
88
89      for i in list:
90          tot = tot + str(toBinaryAux(i))
91
92      return tot
93
94  def per(n):
95      appo = []
96      for i in range(1<<n):
97          s=bin(i)[2:]
98          s='0'*(n-len(s))+s
99          appo.append(list(s))
100     return appo
101
102
103 ################################################################
104 ##  BRUTEFORCE
105 ################################################################
106
107 #bit_to_infer
108 BitToInfer = int(sys.argv[2])
109 KwowingBit = int(sys.argv[1])
110 folder = KwowingBit*2
111
112 SBkey1= 161 + KwowingBit
113 SBkey10 = 1729 - KwowingBit - int(BitToInfer/2)
114
115 #thread variable
116 Nthread = 7
117 pool = ThreadPool(Nthread)
118 task = []
119
120 Nfile = int(math.pow(2, BitToInfer))
121 print("Total Combination: "+ str(Nfile))
122 print("Start computing solution... [" + str(Nthread)   + "]\n")
123
124 combination = per(BitToInfer)
125 totalsol=0
126 times = []
```

```
127
128 solutionFile = open("/mnt/hgfs/Tesi/Test/TestTesiNuovi/"+ str(folder) +"/BF"+str(
    BitToInfer)+".txt", "w",  newline='\n')
129
130 for cont in range(1,1000):
131     fileIn = "//mnt/hgfs/Tesi/Test/TestTesiNuovi/"+ str(folder) +"/"+ str(cont) +"
    output.cnf"
132
133     sol = 0
134
135     if(not exists(fileIn)):
136         break
137
138     start = time.time()
139
140     print("Start working on file "+ str(cont))
141
142     for i in range(0 , Nfile):
143         task.append(pool.apply_async(single_task, args=(i,)))
144
145     for x in task:
146         x.get()
147
148     totalsol += sol
149
150     end = time.time()
151     appo = end - start
152
153     times.append(appo)
154     solutionFile.write("Total time: "+ str(appo) + " s \n")
155
156     print("\nTime for computing solution on file "+ str(cont) + " : "+ str(appo) +
    " s\n")
157
158
159 print("Tot sol: " + str(totalsol))
160 print("Mean: " + str(mean(times)))
161 print("Median: " + str(median(times)))
162 print("St.Dev: " + str(stdev(times)))
163 print("Min: " + str(min(times)))
164 print("Max: " + str(max(times)))
165
166 solutionFile.write("\ntot solution: " + str(totalsol));
167 solutionFile.write("\nMean: " + str(mean(times)))
168 solutionFile.write("\nMedian: " + str(median(times)))
169 solutionFile.write("\nSt.Dev: " + str(stdev(times)))
170 solutionFile.write("\nMin: " + str(min(times)))
171 solutionFile.write("\nMax: " + str(max(times)))
172
173 solutionFile.close()
174 print("Done!")
```

MixedApproach.py

This technique makes a call to a slightly modified Banning Solution script named "BanningSolution2.py", this script is located after this script.

```
1  from ast import While
2  import subprocess
3  from platform import release
4  import shutil
5  import time
6  import os
7  import math
8  import threading
9  from os.path import exists
10 from multiprocessing.dummy import Pool as ThreadPool
11 import sys
12 from statistics import mean, median, stdev
13
```

```
14
15 ################################################
16 ## BINARY, CONBINATION
17 ################################################
18
19 def single_task(j):
20     global sol
21
22     shutil.copyfile(fileIn, fileIn[:-4]+ "[" + str(j)+ "].cnf")
23     file_object = open(fileIn[:-4]+ "[" + str(j)+ "].cnf", 'a',  newline='\n')
24     for k in range(SBkey1, int(SBkey1+BitToInfer/2)):
25         if(combination[j][k-SBkey1] == '0'):
26             file_object.write("-" + str(k) + " 0\n" )
27         else:
28             file_object.write(str(k) + " 0\n" )
29
30     for k in range(SBkey10, int(SBkey10+BitToInfer/2)):
31         if(combination[j][int(BitToInfer/2) + k - SBkey10] == '0'):
32             file_object.write("-" + str(k) + " 0\n" )
33         else:
34             file_object.write(str(k) + " 0\n" )
35     file_object.close()
36
37     output = subprocess.run(["python3", "/mnt/hgfs/Tesi/Test/Ubuntu/Script/
       Bruteforce/BanningSolution2.py", fileIn[:-4]+ "[" + str(j)+ "].cnf" ], stdout=
       subprocess.PIPE).stdout.decode('utf-8')
38
39     print("#Combination N " + str(j) + "     [" +str(combination[j]) + "]" )
40     print(output)
41
42     output = output[output.index("Tot solution found: "):]
43     sol = sol + int(output[output.index("{")+1:output.index("}")])
44
45     os.remove(fileIn[:-4]+ "[" + str(j)+ "].cnf")
46
47
48 def toBinaryAux(clause):
49     if '-' in clause:
50         return 0
51     else:
52         return 1
53
54
55 def toBinary(list):
56     tot = ""
57
58     for i in list:
59         tot = tot + str(toBinaryAux(i))
60
61     return tot
62
63
64 def per(n):
65     appo = []
66     for i in range(1<<n):
67         s=bin(i)[2:]
68         s='0'*(n-len(s))+s
69         appo.append(list(s))
70     return appo
71
72
73 ################################################################
74 ##  BRUTEFORCE
75 ################################################################
76
77 #bit_to_infer
78 KwowingBit = int(sys.argv[1])
79 BitToInfer = int(sys.argv[2])
80 folder = KwowingBit*2
81
82 SBkey1= 161 + KwowingBit
```

```
SBkey10 = 1729 - KwowingBit - int(BitToInfer/2)

#thread variable
Nthread = 7
pool = ThreadPool(Nthread)
task = []

Nfile = int(math.pow(2, BitToInfer))
print("KwowingBit: " + str(KwowingBit) )
print("Instanced bit: " + str(BitToInfer) )
print("Total Combination: "+ str(Nfile))
print("Start computing solution... [" + str(Nthread)   + "]\n")

totalStart = time.time()
combination = per(BitToInfer)

totalsol=0
times = []

solutionFile = open("/mnt/hgfs/Tesi/Test/TestTesiNuovi/"+ str(folder) +"/"+str(
    BitToInfer)+".txt", "w",  newline='\n')

for cont in range(3,1000):
    fileIn = "/mnt/hgfs/Tesi/Test/TestTesiNuovi/"+str(folder)+"/"+ str(cont) +"
    output.cnf"

    sol = 0

    if(not exists(fileIn)):
        break

    start = time.time()

    print("START WORKING ON FILE: "+ str(cont) + "
    **********")

    for i in range(0 , Nfile):
        task.append(pool.apply_async(single_task, args=(i,)))

    for x in task:
        x.get()

    appo = time.time() - start

    totalsol += sol
    times.append(appo)
    solutionFile.write("Total time: "+ str(appo) + " s \n")
    print("\nTime for computing solution on file "+ str(cont) + " : "+ str(appo) +
    " s")
    print("TOTAL SOLUTION FOR FILE: "+ str(cont) + " IS: " + str(sol)+ "\n")


final = time.time() - totalStart

print("Total time: " + str(final))
print("Tot sol: " + str(totalsol))
print("Mean: " + str(mean(times)))
print("Median: " + str(median(times)))
print("St.Dev: " + str(stdev(times)))
print("Min: " + str(min(times)))
print("Max: " + str(max(times)))

solutionFile.write("\ntot solution: " + str(totalsol));
solutionFile.write("\nMean: " + str(mean(times)))
solutionFile.write("\nMedian: " + str(median(times)))
solutionFile.write("\nSt.Dev: " + str(stdev(times)))
solutionFile.write("\nMin: " + str(min(times)))
solutionFile.write("\nMax: " + str(max(times)))

solutionFile.close()
print("Done!")
```

BanningSolution2.py

```
from ast import While
import subprocess
from platform import release
import shutil
import time
import os
import sys
import math

def toBinaryAux(clause):
    if '-' in clause:
        return 1
    else:
        return 0

def toBinary(list):
    tot = ""

    for i in list:
        tot = tot + str(toBinaryAux(i))

    return tot


#Number solution found
Nsol =0

#Timer variable
start = time.time()

#File input variable
fileIn = sys.argv[1]

file_object = open(fileIn, 'a',  newline='\n')

while(True):
    ############################################################ KISSAT
    result  = subprocess.run(['/mnt/hgfs/Tesi/Test/Ubuntu/kissat-master/build/
    kissat', fileIn,  "--relaxed", "-q"], stdout=subprocess.PIPE).stdout.decode('
    utf-8')
    ################################################################

    try:
        index = result.index("v ")
    except Exception:
        break

    Nsol = Nsol +1
    print("Solution N" + str(Nsol) + " " +str(math.trunc(time.time() - start)))

    # ADD NEW CLAUSE TO CNF
    result = result[result.index("v") : result.index("1728")+4]
    result = result.strip()
    result = result.replace("v", "")
    result = result.replace("\n", "")
    result = result.replace("\r", "")
    result = result.replace(" -", "+")
    result = result.replace(" ", " -")
    result = result.replace("+", " ")

    file_object.write(result[1:] + " 0\n")

    # WRITING SOLUTION TO FILE
    try:
        solution = result[:result.index("-129 ")]
    except Exception:
        solution = result[:result.index(" 129 ")]

    solution = solution.strip()
```

```
    # BINARY
    tot = toBinary(solution.split())

    print(tot)
    #HEX
    decimal_representation = int(tot, 2)
    hexer = hex(decimal_representation)
    hexer = hexer[2:]
    for j in range(len(hexer), 32):
        hexer = "0" + hexer
    print(hexer)

    #ASCI STRING
    bytes_object = bytes.fromhex(hexer)
    ascii_string = bytes_object.decode('utf-8', 'ignore')
    print(ascii_string + "\n" )


end = time.time()
appo = end - start
print("Tot solution found: "+ " {"+str(Nsol)+"}")
print("Tot time: " + str(math.trunc(appo)) + " s")

file_object.close()
```

RandomInstancing.py

Like the mixed approach this technique also makes a call to the modified Banning Solution script "BanningSolution2.py".

```
from ast import While
import subprocess
from platform import release
import shutil
import time
import os
import math
import threading
from os.path import exists
from multiprocessing.dummy import Pool as ThreadPool
import sys
import random
from statistics import mean, median, stdev


################################################
## BINARY, CONBINATION
################################################

def single_task(j):
    global sol

    shutil.copyfile(fileIn, fileIn[:-4]+ "[" + str(j)+ "].cnf")
    file_object = open(fileIn[:-4]+ "[" + str(j)+ "].cnf", 'a',  newline='\n')
    i=0
    for k in listBit:
        if(combination[j][i] == '0'):
            file_object.write("-" + str(k) + " 0\n" )
        else:
            file_object.write(str(k) + " 0\n" )
        i=i+1

    file_object.close()

    output = subprocess.run(["python3", "/mnt/hgfs/Tesi/Test/Ubuntu/Script/
    Bruteforce/BanningSolution2.py", fileIn[:-4]+ "[" + str(j)+ "].cnf" ], stdout=
    subprocess.PIPE).stdout.decode('utf-8')

    print("#Combination N" + str(j) + "     [" +str(combination[j]) + "]" )
    print(output)
```

```
39
40      output = output[output.index("Tot solution found: "):]
41      sol = sol + int(output[output.index("{")+1:output.index("}")])
42
43      os.remove(fileIn[:-4]+ "[" + str(j)+ "].cnf")
44
45
46  def toBinaryAux(clause):
47      if '-' in clause:
48          return 0
49      else:
50          return 1
51
52  def toBinary(list):
53      tot = ""
54
55      for i in list:
56          tot = tot + str(toBinaryAux(i))
57
58      return tot
59
60  def per(n):
61      appo = []
62      for i in range(1<<n):
63          s=bin(i)[2:]
64          s='0'*(n-len(s))+s
65          appo.append(list(s))
66      return appo
67
68
69  ################################################################
70  ##   BRUTEFORCE
71  ################################################################
72
73  #bit_to_infer
74  KwowingBit = int(sys.argv[1])
75  BitToInfer = int(sys.argv[2])
76  folder = KwowingBit*2
77
78  SBkey1= 161 + KwowingBit
79  SBkey10 = 1729 - KwowingBit - int(BitToInfer/2)
80
81  #thread variable
82  Nthread = 7
83  pool = ThreadPool(Nthread)
84  task = []
85
86  Nfile = int(math.pow(2, BitToInfer))
87  print("KwowingBit: " + str(KwowingBit) )
88  print("Instanced bit: " + str(BitToInfer) )
89  print("Total Combination: "+ str(Nfile))
90  print("Start computing solution... [" + str(Nthread)   + "]\n")
91
92  totalStart = time.time()
93  combination = per(BitToInfer)
94
95  totalsol=0
96  times = []
97
98  solutionFile = open("/mnt/hgfs/Tesi/Test/TestTesiNuovi/"+ str(folder) +"/"+str(
        BitToInfer)+".txt", "w",  newline='\n')
99
100 for cont in range(1000):
101     fileIn = "/mnt/hgfs/Tesi/Test/TestTesiNuovi/"+str(folder)+"/"+ str(cont) +"
        output.cnf"
102
103     sol = 0
104
105     if(not exists(fileIn)):
106         break
107
```

```
    arr = list(range(228,289)) + list(range(321, 449)) + list(range(481, 609)) + list(range(641, 769)) + list(range(801, 929)) + list(range(961, 1089)) + list(range(1121, 1249)) + list(range(1281, 1409))  + list(range(1441, 1569)) + list(range(1601, 1729))
    random.shuffle(arr)

    start = time.time()

    listBit = []
    for i in range(BitToInfer):
        listBit.append(arr.pop())

    print("Bit choosed to infer: "+ str(listBit))

    print("START WORKING ON FILE: "+ str(cont) + "   ********************")

    for i in range(0 , Nfile):
        task.append(pool.apply_async(single_task, args=(i,)))

    for x in task:
        x.get()

    appo = time.time() - start

    totalsol += sol
    times.append(appo)
    solutionFile.write("Total time: "+ str(appo) + " s \n")
    print("\nTime for computing solution on file "+ str(cont) + " : "+ str(appo) + " s")
    print("TOTAL SOLUTION FOR FILE: "+ str(cont) + " IS: " + str(sol)+ "\n")

final = time.time() - totalStart

print("Total time: " + str(final))
print("Tot sol: " + str(totalsol))
print("Mean: " + str(mean(times)))
print("Median: " + str(median(times)))
print("St.Dev: " + str(stdev(times)))
print("Min: " + str(min(times)))
print("Max: " + str(max(times)))

solutionFile.write("\ntot solution: " + str(totalsol));
solutionFile.write("\nMean: " + str(mean(times)))
solutionFile.write("\nMedian: " + str(median(times)))
solutionFile.write("\nSt.Dev: " + str(stdev(times)))
solutionFile.write("\nMin: " + str(min(times)))
solutionFile.write("\nMax: " + str(max(times)))

solutionFile.close()
print("Done!")
```

SingleKissatCall.py

```
from ast import While
import subprocess
from platform import release
import shutil
import time
import os
from os.path import exists
from statistics import mean, median, stdev
import sys

times = []

folder = sys.argv[1]

solutionFile = open("/mnt/hgfs/Tesi/Test/TestTesiNuovi/Unasoluzione/"+ str(folder)+"/OnlyKissat.txt", "w",  newline='\n')

```

```
for cont in range(0,10):

    fileIn = "//mnt/hgfs/Tesi/Test/TestTesiNuovi/Unasoluzione/"+ str(folder) +"/"+
    str(cont) +"output.cnf"

    start = time.time()
    subprocess.run(['/mnt/hgfs/Tesi/Test/Ubuntu/kissat-master/build/kissat', fileIn
    ,  "--relaxed", "-q"], stdout=subprocess.PIPE).stdout.decode('utf-8')
    end = time.time()
    appo = end - start

    times.append(appo)
    print("\nTime for computing solution on file "+ str(cont) + " : "+ str(appo) +
    " s\n")


print("Mean: " + str(mean(times)))
print("Median: " + str(median(times)))
print("St.Dev: " + str(stdev(times)))
print("Min: " + str(min(times)))
print("Max: " + str(max(times)))

solutionFile.write("\nMean: " + str(mean(times)))
solutionFile.write("\nMedian: " + str(median(times)))
solutionFile.write("\nSt.Dev: " + str(stdev(times)))
solutionFile.write("\nMin: " + str(min(times)))
solutionFile.write("\nMax: " + str(max(times)))

solutionFile.close()

print("Done!")
```

SingleCryptominisatCall.py

```
from ast import While
import subprocess
from platform import release
import shutil
import time
import os
from os.path import exists
from statistics import mean, median, stdev
import sys

times = []

folder = int(sys.argv[1])*2

solutionFile = open("E:/Scuola/UNIve/MAGISTRALE/Tesi/Test/TestTesiNuovi/"+ str(
    folder) +"/OnlyCryptominisat.txt", "w",  newline='\n')

for cont in range(0,10):

    fileIn = "E:/Scuola/UNIve/MAGISTRALE/Tesi/Test/TestTesiNuovi/"+ str(folder) +"/
    "+ str(cont) +"outputXor.cnf"

    if(not exists(fileIn)):
        break

    start = time.time()
    stri = subprocess.run(['E:/Scuola/UNIve/MAGISTRALE/Tesi/Solvers/Cryptominisat/
    cryptominisat5.exe',  fileIn, "-t8", "--maxsol=1000"], stdout=subprocess.PIPE).
    stdout.decode('utf-8')
    end = time.time()
    appo = end - start
    print(stri)

    times.append(appo)
    print("\nTime for computing solutions on file "+ str(cont) + " : "+ str(appo) +
     " s\n")
```

```
32
33 print("Mean: " + str(mean(times)))
34 print("Median: " + str(median(times)))
35 print("St.Dev: " + str(stdev(times)))
36 print("Min: " + str(min(times)))
37 print("Max: " + str(max(times)))
38
39 solutionFile.write("\nMean: " + str(mean(times)))
40 solutionFile.write("\nMedian: " + str(median(times)))
41 solutionFile.write("\nSt.Dev: " + str(stdev(times)))
42 solutionFile.write("\nMin: " + str(min(times)))
43 solutionFile.write("\nMax: " + str(max(times)))
44
45 solutionFile.close()
46
47 print("Done!")
```