Integral attacks on the PRESENT block cipher with practical complexity

Tuesday 25th March, 2025 - 10:00

Ivaylo Krumov
University of Luxembourg
Email: ivaylo.krumov.001@student.uni.lu

This report has been produced under the supervision of:
Baptiste Lambin
University of Luxembourg
Email: baptiste.lambin@uni.lu

Abstract

The purpose of this Bachelor Semester Project is to determine the extent to which the PRESENT block cipher is vulnerable to cryptanalytical attacks utilizing efficient mathematical techniques. To find this out, the project focuses on the domain of integral cryptanalysis. In order to produce concrete results, a number of keywords associated with the project's topic have been defined in detail, establishing the scientific context around which the project is created. Furthermore, several programs representing attacks on the PRESENT block cipher have been developed with the aim of showing the limit of these attacks in practice. All accomplished results of the project are thoroughly presented in this report. The ultimate goal of the report and the project as a whole is to give useful insight into the utilization of mathematical tools to efficiently conduct cryptanalytical attacks on block ciphers.

1. Plagiarism statement

I declare that I am aware of the following facts:

- As a student at the University of Luxembourg I must respect the rules of intellectual honesty, in particular not to resort to plagiarism, fraud or any other method that is illegal or contrary to scientific integrity.
- My report will be checked for plagiarism and if the plagiarism check is positive, an internal procedure will be started by my tutor. I am advised to request a pre-check by my tutor to avoid any issue.
- As declared in the assessment procedure of the University of Luxembourg, plagiarism is committed whenever the source of information used in an assignment, research report, paper or otherwise published/circulated piece of work is not properly acknowledged. In other words, plagiarism is the passing off as one's own the words, ideas or work of another person, without attribution to the author. The omission of such proper acknowledgement amounts to claiming authorship for the work of another person. Plagiarism is committed regardless of the language of the original work used. Plagiarism can be deliberate or accidental. Instances of plagiarism include, but are not limited to:
  1) Not putting quotation marks around a quote from another person's work
  2) Pretending to paraphrase while in fact quoting
  3) Citing incorrectly or incompletely
  4) Failing to cite the source of a quoted or paraphrased work
  5) Copying/reproducing sections of another person's work without acknowledging the source
  6) Paraphrasing another person's work without acknowledging the source
  7) Having another person write/author a work for oneself and submitting/publishing it (with permission, with or without compensation) in one's own name ('ghost-writing')
  8) Using another person's unpublished work without attribution and permission ('stealing')
  9) Presenting a piece of work as one's own that contains a high proportion of quoted/copied or paraphrased text (images, graphs, etc.), even if adequately referenced

  Auto- or self-plagiarism, that is the reproduction of (portions of a) text previously written by the author without citing that text, i.e. passing previously authored text as new, may be regarded as fraud if deemed sufficiently severe.

2. Introduction

In the digital era we all currently live in, the rate of exchange of information in all forms has been exponentially increasing. With the various technological developments that have been occurring for the past few decades, communication of information has evolved substantially in regard to the methods used to securely convey it from one party to another. But so have the techniques of attackers, who maliciously exploit this progress in the technological field to obtain this information without authorization for their own purposes.

To transfer all kinds of digital data, cryptography is widely used as a means to protect it from third party threats. Block ciphers in particular have become a topic of interest for cryptographers due to the seemingly impenetrable defense they offer with the rigorous algorithms used to encrypt data so that it is unreadable to a human but once decrypted can be read with no issue. But even with these complex algorithms, all it takes is a single oversight for a block cipher that is thought to be secure to be exploited so that all encrypted messages are made readable to an attacker.

Oftentimes, cryptanalysis is used to look for flaws in a block cipher that can lead to breaking it entirely without knowing all the information about it. Nowadays, there exist numerous sophisticated techniques and types of cryptanalytical attacks on block ciphers, which is why before a block cipher is implemented in practice it is important to know exactly how resistant it is to these kinds of threats.

This report presents a thorough response to the question of how vulnerable a block cipher, in particular the PRESENT block cipher, is when considering practical constraints. To determine this, a handful of cryptanalytical attacks have been conducted on the block cipher utilizing efficient mathematical exploits on it. Additionally, a scientific text provides the context for these attacks and for the project's topic overall. The presented information in this report is expected to give beneficial insight into the utilization of mathematical tools to efficiently conduct cryptanalytical attacks on block ciphers.

3. Project description

3.1. Domains

3.1.1. Scientific.

- **Cryptography.** Cryptography is a method to ensure secure communication of information between 2 parties. It does so by utilizing different techniques for encryption and/or decryption of data, e.g. ciphers, in such a way that a potential third party would be unable to retrieve and understand the data. The concept of applying cryptography to securely transmit messages has existed since ancient times, albeit with rather simple approaches to hide information by today's standards. In modern times, cryptography has evolved substantially so that it uses computers alongside more complex mathematical algorithms in the encryption and decryption of information to guarantee its protection from the ever-increasing amount of third party threats in the digital era.
- **Cryptanalysis.** Cryptanalysis deals with closely studying cryptographic systems in order to obtain and decipher encrypted information without having initial access to the secret key. The techniques used very often rely on mathematically finding weaknesses in a system's algorithm that can allow a third party to retrieve the unknown key and decrypt the encrypted message with it. By discovering such weaknesses cryptanalysis can be used to understand how secure a cryptographic algorithm is and potentially improve its security. At the same time, however, this can be exploited in a malicious way such that an attacker is able to obtain and read encrypted data without authorization.

3.1.2. Technical.

- **Python.** Python is the programming language of choice for the realization of the technical part of the project. It has been one of the most popular programming languages for years due to it being relatively easy to get into and the sheer versatility it offers through its various libraries. Still, as a high-level language, it suffers from slower performance when compared to lower-level languages, such as C++. Nevertheless, this issue does not affect this particular project in a significant way and Python remains a reliable language to use in the technical portion, because it supports the Gurobi package, used in solving mixed integer linear programming problems.
- **Mixed integer linear programming (MILP).** MILP is a type of mathematical optimization problem, where some of the variables in an inequality are constrained to be integers, while others can take any value within a given range. Most commonly, the goal of solving such a problem is to optimize a given function by finding the minimum or maximum value within the provided variable constraints. This concept is useful in cryptanalysis, particularly in conducting integral attacks, as it can allow an attacker to identify specific patterns in the encrypted output given some specific input, reducing the resources needed for a successful attack.

3.2. Targeted Deliverables

3.2.1. Scientific deliverables. The produced scientific deliverable attempts to find an answer to the question "How many rounds of the PRESENT block cipher can be attacked using integral attacks with practical complexity?". With the aid of other scientific sources, it thoroughly examines the question's most important keywords and offers concise definitions and overviews for each one. A definitive answer to the scientific question is then developed using this understanding. Finally, there is a brief discussion to assess the extent to which all requirements for the deliverable have been satisfied.

3.2.2. Technical deliverables. The produced technical deliverable is represented by multiple programs, each corresponding to a certain number of rounds attacked on the PRESENT block cipher, as well as a single program that is the cipher itself. Moreover, an additional helper program has been created with the purpose of using MILP to identify patterns in the encrypted data that can ease the work needed for attacks on higher numbers of rounds. All of this is presented by a thorough description of the key algorithms in the individual programs. For most n-round attacks a mention of the modifications in the code needed to do an attack on $n+1$ rounds is also included, based on the findings from the helper program.

In the end, there is a short section assessing the work done and what improvements could have been made given more time and resources.

4. Pre-requisites

The complete realization of this project, both the scientific and the technical part, takes advantage of several skills and competencies that the author already is in possession of before the start of the project research process. These pre-requisites are respectively identified in the 2 subsections that follow.

4.1. Scientific pre-requisites

The author is familiar with the proper writing and organizational techniques for a piece of writing with a scientific theme. Additionally, as long as the terminology used is at a reasonable level of complexity, he is fully capable of understanding meaningful information from scientific sources. Last but not least, he has some general knowledge of the scientific fields involved in the project,namely the application of block ciphers in cryptography and a few techniques used to attack ciphers with cryptanalysis.

4.2. Technical pre-requisites

The author has prior experience in programming in several languages. As a result, he is well aware of the possibility of running into technical challenges while developing the technical deliverable and is ready to take the necessary steps to overcome them. The author also has good proficiency with Python programming, is able to install and use a variety of Python packages, and is able to apply his technical skills in the area of cryptography and cryptanalysis.

5. How many rounds of the PRESENT block cipher can be attacked using integral attacks with practical complexity?

5.1. Requirements

The scientific deliverable of this project aims to satisfy several requirements, which are presented in this section. In order to provide a better quality for the structure of their presentation, they are split into two different categories, namely functional and non-functional requirements. The functional requirements are described first, followed by a description of the non-functional ones.

Functional requirements:
The project's scientific question, "How many rounds of the PRESENT block cipher can be attacked using integral attacks with practical complexity?" should be fully addressed in the scientific deliverable. It should be able to offer concise explanations of the key terms relevant to the subject of the question for this purpose. Additionally, this terminology must be used in the deliverable in order to logically reach a conclusive answer to the question. When appropriate, it should also draw support from outside scientific sources and any relevant information they may contain.

Non-functional requirements:
The ideas in the scientific deliverable should also be expressed in a way that makes it possible for a non-professional reader to be able to understand them, provided that the necessary scientific terms are well defined. Regarding the overall structure of the deliverable, its contents should make sense when read as a whole and should all be somehow related to the main subject of the question. All information must be crucial to achieving the final objective of the scientific deliverable, which is to provide a conclusive answer to the scientific question. Furthermore, citations from other scientific sources should always include acknowledgment to the original author and the work being referenced.

5.2. Design

The scientific deliverable is a text that aims to provide context to the ideas and implementation of the technical deliverable. As a result, the scientific deliverable has received less attention because the project's overall success depends more on the technical deliverable's outcomes. Due to this, the research for the scientific deliverable took only about 3 weeks to complete, which is much shorter than what might have been anticipated. Nevertheless, it is still considered a crucial part to the overall project, as it provides the necessary definitions required to fully understand the work done for the technical deliverable.

The structure of the scientific deliverable is straightforward - it is divided into a number of smaller sections, each of which focuses on defining a specific term or subtopic associated with the scientific question. Once all of these definitions are made, the text concludes with the formulation of the final answer to the question, which is a result of the compilation of all important information presented in the previous subsections.

Multiple scientific sources are utilized as points of reference throughout the scientific deliverable with the goal of presenting clear and specific information. These sources assist with providing the corresponding explanations in each subsection by presenting formal mathematical ways of defining them. For some of them, concrete examples are also given to more clearly show what is meant by the formal definition.

5.3. Production

To be able to give an answer to the scientific question "How many rounds of the PRESENT block cipher can be attacked using integral attacks with practical complexity?", we will need to delve deeper into the meaning of each contained term it is composed of. To that end, we will begin the search for an answer by precisely defining what is a block cipher (and

more specifically the PRESENT block cipher) and an integral attack on a cipher (again in the context of PRESENT).

As a general definition, a block cipher in cryptography is a cipher which encrypts and decrypts a certain number of bits, called blocks, at a time. In a block cipher by dividing the input plaintext into fixed-sized blocks and using a secret key to perform a series of mathematical operations on each block, the encryption process creates the corresponding output ciphertext. With the secret key a key schedule is created, which in turn generates round keys used in each round of the encryption algorithm. Block ciphers are symmetric and reversible, so to decrypt a ciphertext into a plaintext, the same key is used alongside a reversed encryption process.

With that said, we can continue by describing the specifications of the PRESENT block cipher, as presented by the original paper in which it was introduced [1]. PRESENT is designed for up to 31 rounds and is an instance of what is called a substitution-permutation network (SPN) (Fig. 1). This is a type of block cipher in which the operations done in each round of encryption are bit substitution followed by bit permutation. Usually, as is the case with PRESENT, an XOR operation is also added to each round. In the substitution stage, a predefined S-box (Fig. 2) is used for blocks of bits, defining the one-to-one mapping between every possible input value and its output substitution value. Similarly, for the permutation stage, a P-box (Fig. 3) is put into use that takes all bits as input and returns a permutation of them as output. The way PRESENT utilizes these concepts is the following - first the key scheduler generates a number of round keys equal to the specified number of encryption rounds; then for each round of encryption, excluding the last one, the current ciphertext (starting with the plaintext for the first round) is XOR-ed with the corresponding round key, its bits are substituted in blocks of 4 bits based on the S-box and the P-box is applied to permute them the bit output from the substitution stage; in the end, at the final round of encryption, the returned ciphertext computed in the previous round is only XOR-ed with the last remaining round key, producing the final ciphertext.

To gain a better understanding of the encryption process, we will briefly describe PRESENT's key scheduling, substitution stage and permutation stage. The key scheduling algorithm (Fig. 4), which is responsible for the round key generation, is intended to be used with either an 80-bit or 128-bit secret key, both of which slightly differ in key scheduling steps. Since in our project we use the version for 80-bit keys, we will only focus on that one and ignore the 128-bit key version. In the 80-bit version, for each round the scheduler takes the secret key, shifts it by 61 bits to the left, applies the S-box on the 4 left-most bits and finally the number of the current round is XOR-ed with the bits indexed from 15 to 19 incl., thus producing the corresponding round key. The result is then stored in a register and the entire process is repeated until the number of round keys in the register matches the number of rounds. As for the substitution and permutation stages, both of them are much more straightforward - in the former, the S-box (Fig. 1) is applied to the ciphertext block by block, where each block consists of 4-bits (from least significant bit to most significant bit), and in the latter, each bit's value is swapped with that of another bit based on the P-box (Fig. 2), effectively permuting the entire bit order.

We shall now move onto the definition of integral attack. In cryptanalysis, an integral attack is a type of attack on a block cipher, which aims to retrieve the secret key while only knowing the encryption algorithm by exploiting it in such a way that, given a specific pair of plaintext and ciphertext, some correlation between the two is recognized, revealing information about the key. This correlation is called an integral distinguisher. Informally, a distinguisher is a behaviour for a block cipher that a random function should not be able to exhibit. The idea behind it is that a block cipher should not reveal that it is a block cipher but rather behave as if it was a random function. If we can find a property of the block cipher that does not apply for a random function, then we obtain a distinguisher as a way to determine whether a given function is the cipher or some random function. A distinguisher in itself can be quite complicated to find. Once constructed, however, a strong distinguisher is usually enough to break the cipher and retrieve the key, though this really depends on the key size and the level of symmetry of the block cipher. Luckily, PRESENT is a near perfectly symmetric cipher, due to decryption essentially being almost exactly the same as reversed encryption and vice versa. Moreover, the key size used for this project is considered relatively small based on today's standards. All of this means that, with a good distinguisher, in theory it would be very much possible to conduct a successful integral attack on PRESENT, though the practical level of success would be limited by the number of rounds, since for higher rounds it is expected that more time and computing resources would be needed to succeed. Therefore, it would be likely that practical complexity for an integral attack would be achieved only for a very limited number of rounds.

Now that we have defined the terms contained in the scientific question and have established that what is asked is most certainly achievable, all that is left to provide a conclusive answer is to find a way to use integral attacks on as many rounds of PRESENT as practically possible. To that end, we will introduce a general method to find effective distinguishers over 4 up to n rounds for respectively attacking 5 up to $n + 1$ rounds, where $n + 1$ is the maximum number of rounds that can be practically attacked with this technique. We purposely mentioned 4 rounds as the minimum number, since the distinguishers (plaintext-ciphertext pairs) over 1, 2, 3 and 4 rounds are all the same. In essence, a distinguisher can be identified in the following way - first a set of plaintexts is generated, each plaintext having a special bit structure, where a sufficient number (depending on the number of rounds to attack) of particular bits (starting from the most significant bit and continuing with the subsequent bits to its right) take the same constant values throughout all plaintexts in the set, while the rest of the bits take all possible values allowed with the given set size; then all plaintexts are encrypted using a random

secret key and the desired number of rounds to attack, and finally all resulting ciphertexts are XOR-summed to compute a new ciphertext. In the end, the value of the XOR-sum is guaranteed to be 0, indicating that when XOR-ed together all ciphertexts always result in a ciphertext consisting of only bits with value 0. Mathematically speaking, the probability of a random function returning the exact same output (ciphertext with all 0-bits) using the exact same input (set of plaintexts with special structure) is $\frac{1}{2^N}$, where N is the number of bits of any input plaintext, as well as the output ciphertext (they are all of equal bit size). This means that if some function gives this output from this input, then it is most likely the actual PRESENT encryption scheme, with an extremely low chance of it being a false positive (for a higher number of bits, it becomes practically impossible for such a coincidence to occur).

The reason behind why this works is a mathematical property of bit vectors, called bit-based division property (or simply division property for short) and is dependent on the vector's bit-product. For the sake of completeness, we will now give both terms formal definitions, although we will not present them in detail. Let $x = (x_0, ..., x_{n-1}) \in F_n^2$ is an n-bit vector. If $u \in F_n^2$, then x^u is defined as the bit-product

$$x^u = \prod_{i=0}^{n-1} x_i^{u_i}. \text{ [2]}$$

We say that when a set $X \subset F_n^2$ has the division property D_K^n, where $K \subset F_n^2$ is a set, then we have

$$\bigoplus_{x \in X} x^u = \begin{cases} unknown & \text{if } \exists k \in K, \text{ s.t. } u \succeq k \\ 0 & \text{otherwise} \end{cases}. \text{ [3]}$$

To make use of division property in cryptanalysis, one must study how this property evolves throughout the round function of a block cipher, which can be formally defined with the following definition [4]: *"Let f_r denote the round function of an iterated block cipher. Assume the input multiset to the block cipher has initial division property $D_k^{n,m}$, and denote the division property after i-round propagation through f_r by $D_{K_i}^{n,m}$. Thus, we have the following chain of division property propagations:*

$$k := K_0 \xrightarrow{f_r} K_1 \xrightarrow{f_r} K_2 \xrightarrow{f_r} ...$$

Moreover, for any vector k_i^* in K_i $(i \geq 1)$ there must exist a vector k_{i-1}^* in K_{i-1} such that k_{i-1}^* can propagate to k_i^* by division property propagation rules. Furthermore, for $(k_0, k_1, ..., k_r) \in K_0 \times K_1 \times ... \times K_r$, if k_{i-1} can propagate to k_i for all $i \in \{1, 2, ..., r\}$ we call $(k_0, k_1, ..., k_r)$ and r-round ***division trail****.*

In practice, we are only going to use Proposition 6 described in [4]. More precisely, if the unit vector e_i is not in the last set with division property, then we know that the i-th bit of the output is balanced, i.e. it sums to 0. Therefore, all we need to do is find an input division property vector with the special bit structure, such that there are as few as possible unit vectors in the last set of the propagation chain, meaning that there should be as many as possible output bits that sum to 0. Essentially, this is going to be our distinguisher and, as suggested in [4], to discover a strong distinguisher for a specific amount of PRESENT encryption rounds, MILP can be quite a useful tool, which is what is going to be utilized for the technical portion of the project.

The presented information so far is sufficient to partially formulate an answer to the scientific question "How many rounds of the PRESENT block cipher can be attacked using integral attacks with practical complexity?". It appears that by exploiting the division property to reveal a distinguisher and use it as a means to identify the correct key, the maximum possible number of rounds that can be attacked within a reasonable amount of time is at least 5. While the actual maximum cannot be fixed without finding distinguishers for subsequent rounds, this number is a good starting point for that. As expected, the complete answer to the question will be given only after experimentally confirming how much higher this number can go with this method, which is a problem left for the technical deliverable to tackle.

5.4. Assessment

As a reminder, the scientific deliverable only aimed to establish the context for the technical deliverable's purpose by giving precise definitions for the key terms of the scientific question. In order to achieve that, several requirements were established prior to the development of the scientific deliverable itself.

The scientific deliverable was produced in accordance with all specified criteria. The deliverable's use of external scientific sources to support some of its ideas is of utmost importance and it clearly communicates the key information. These sources and their respective authors are given the proper credit where it is due. Moreover, the text's logical flow is clear enough to be taken as easily related to the main topic of the question. While the conclusive answer formulated at the end only loosely responds to the question, the information provided by it can be deemed sufficient to have a solid basis the technical deliverable.

Overall, the scientific deliverable successfully meets all of the expectations that were placed on it. The reader is prepared to explore the production of the technical deliverable by understanding the key concepts included in it and the justification for their utilization in its implementation thanks to the scientific context that has been provided for it.

6. Integral attacks on the PRESENT block cipher

6.1. Requirements

The technical deliverable of this project aims to satisfy several requirements, which are presented in this section. In

order to provide a better quality for the structure of their presentation, they are split into two different categories, namely functional and non-functional requirements. The functional requirements are described first, followed by a description of the non-functional ones.

Functional requirements:

The technical deliverable is expected to include an implementation of the PRESENT block cipher. As its main goal, the deliverable should be able to perform integral attacks on this implementation of the cipher on a number of different encryption rounds and retrieve the secret key. For this purpose, it should utilize MILP optimization in order to find a unique distinguisher for each subsequent round. When obtained, the information provided by the distinguisher should be used to greatly reduce the amount of possibilities for the value of the key. In the end, a bruteforce approach should be able to retrieve the correct key from the remaining values within a practical amount of time.

Non-functional requirements:

All of the integral attacks of the technical deliverable should be implemented in such a way so that the secret key is retrieved in a practical amount of time, taking no more than a few minutes on a modern everyday computer. Moreover, the deliverable should target as many rounds as practically possible by making use of the information given by the unique distinguisher for each round. Finally, while attempting to find a suitable distinguisher, the corresponding input and output should be printed in the terminal to easily analyze all pairs and identify the most practical one.

6.2. Design

The technical deliverable of this project is split into multiple programs, each having its own specific purpose - one is the implementation of the PRESENT block cipher, another one is the helper MILP optimization program required to find a distinguisher for the integral attacks on the cipher, while each of the rest represents an integral attack on a specific number of encryption rounds of PRESENT.

In regard to this particular implementation of PRESENT, it slightly diverges from the original specifications of the block cipher, as it has been modified to strictly generate a 64-bit secret key, instead of the usual 80-bit or 128-bit key. Given the project's implementations of the integral attacks on the cipher, this has been done so with simplicity and practicality in mind - for an 80-bit key it would have been too troublesome and not worth the extra effort to find the last 14 bits of the key, while a key composed of 128 bits would have been too large for the integral attacks to be deemed practical.

As for the range of rounds targeted for attacks, the minimum number of rounds is 5, as this was considered a solid starting point for conducting subsequent attacks (as shown in the scientific deliverable), while the maximum is dependent on the time and resources dedicated for this project. Given these constraints of the project, the maximum number of rounds that we were able to successfully attack was 7. Something of note is that to extend this range an attempt for an 8-round attack was also made, though it was deemed too impractical to succeed. Therefore, this leaves the final count of the programs representing the integral attacks in the deliverable to be 3.

The technical deliverable as a whole has been developed in the span of roughly 3 months. A substantial amount of this time was strictly dedicated to creating and refining the code of each of the programs. Only a couple of weeks have been devoted to research purposes, e.g. learning about MILP optimization. Each week, the latest progress on the technical deliverable was stored in a Github repository specifically created for organizational purposes.

6.3. Production

The technical deliverable consists of an implementation of the PRESENT block cipher for 64-bit keys, a helper program that uses MILP model optimization to reveal possible distinguishers over a certain amount of rounds of encryption with PRESENT and finally 3 programs corresponding to 5-, 6- and 7-round integral attacks on the cipher. In the text that follows, each of these components will be presented one by one in more detail.

To begin with, we will show the implementation of PRESENT with the modifications mentioned in the previous section. Since we already described how the block cipher functions in the scientific deliverable, here we will not go into detail. We have implemented PRESENT as a class that consists of a constructor function and an encryption function (Fig. 5). Whenever a new object, i.e. instance of PRESENT, of that class is created, the constructor is called, initializing it with the specified number of encryption rounds and the secret key. If there is no round number passed, the default value of 31 rounds is used to initialized the instance. Similarly, if no key is specified, then the constructor determines a random 64-bit value for the key. After initializing the instance, the constructor function is also responsible for generating the round keys with PRESENT's key scheduler, which are all stored in an array for the encrypt function to use. The key schedule (Fig. 6) has been adjusted to accommodate the change to a 64-bit secret key and differs from the original 80-bit key specification by the rotation of the bits and the application of the S-box at each round. In particular, the bit rotation is done by 45 bits instead of 61 bits and the S-box is applied on bits with indices 63 to 60 incl. instead of 79 to 76 incl. (obviously the latter range of bits does not even exist in a 64-bit key). As for the encryption function, no modifications were needed to adapt to a 64-bit key, as it was already designed to work with 64-bit round keys. It takes as an argument a plaintext and inside it we use 3 other functions - addRoundKey, sBoxLayer and pLayer (Fig. 7) - each of which corresponds to the respective step in the encryption process (XOR key with current round number, apply S-box, apply P-box). In the end, the encryption function returns the produced ciphertext.

As a prerequisite for understanding the MILP optimization program, in the scientific deliverable we already presented the definition of a division trail. From Section 3.1 in [4] we know that the four division trails of the logical AND operation are $(0,0) \xrightarrow{Xor} (0)$, $(0,1) \xrightarrow{Xor} (1)$, $(1,0) \xrightarrow{Xor} (1)$ and $(1,1) \xrightarrow{Xor} (0)$. Moreover, the paper shows that, given a division trail $(a_0, a_1) \xrightarrow{and} (b)$, we are able to represent (modelize) the division property propagation of the logical AND operation as the following set of linear inequalities:

$$\begin{cases} b - a_0 \geq 0 \\ b - a_1 \geq 0 \\ b - a_0 - a_1 \geq 0 \\ a_0, a_1, b \text{ are binaries} \end{cases}$$

In practice, our MILP optimization program contains a Gurobi linear expression model, where its constraints are represented by PRESENT's encryption operations (Fig. 8). Key addition (XOR operation) does not affect the division property in any way because the operation is always constant, so there is no need to add a constraint for it in the model. The permutation stage can be represented simply as equalities (bit i takes bit j's value). As for the substitution stage, the S-box is represented as a set of inequalities, provided by the tutor (Fig. 9). Without going into detail, these inequalities have been obtained using the information of Section 3.2 in [4].

Once we have added all corresponding constraints to the Gurobi model, we are ready to look for distinguishers over some number of rounds of PRESENT (Fig. 10). First, for each i between 1 and 64 incl. an input is generated in the format presented in the scientific deliverable, where the number i corresponds to the number of consecutive 1's in the input (constant bits) and the rest $64 - i$ bits are all 0's (non-constant bits). Then for each bit of the input, a j-bit output is generated, where j is the position of the current input bit. The current input and output are then passed in the Gurobi model, which attempts to determine whether there exists a division trail for the corresponding bit, which can reveal its value for certain given the current input-output pair. Once all 64 bits are checked, their respective division trail properties are printed as a string, which tells if a bit is identifiable in that way by representing these bits with 0 and a question mark otherwise. The idea of finding a strong distinguisher with this method is to look for an input-output pair, such that the ratio between 1's in the input and 0's in the output is as small as possible (few 1's in the input and many 0's in the output), as we are trying to find a distinguisher within practical time constraints (the input needs to be as small as possible while at the same time revealing as much output information as possible).

Finally, we will describe how we have done the integral attacks on 5, 6 and 7 rounds of PRESENT. For each of those we have used the strongest distinguisher that can be identified from the MILP optimization program for the corresponding number of rounds. Moreover, each attack has almost the exact same structure, so for the last 2 attacks we will only show the additions to the code needed to adapt the attack to the respective number of rounds. Starting off with the integral attack on 5 rounds, it consists of 3 parts: generation of a set with random plaintexts, retrieval of all possible keys using the distinguisher and a bruteforce phase to recover the exact correct key. At the start, a loop is initialized, which in each iteration generates a new set with random plaintext values and with the help of the distinguisher filters out those plaintexts that surely cannot lead to the correct key (Fig. 11). For the generation part, a *genList* function is used (Fig. 12), which randomly generates and returns a list of 2^4 64-bit input plaintexts with the aforementioned structure. In the context of the 5-round attack, the plaintexts contain only 4 non-constant bits (the first 4 least significant bits), therefore they always contain constant values (either 0 or 1) for the rest 60 bits. This split has been determined after using our MILP modeling program to find a distinguisher over 4 rounds, which clearly shows that for an input with 4 non-constant bits we are able to reveal all bits of the real key (Fig. 13). This is also the reason why we generate exactly 2^4 plaintexts. After a set of plaintexts is generated, the filtering of incorrect keys is done by a *keyRecovery* function (Fig. 14), which takes the list of plaintexts and keeps only those for which the division property applies. For that, the plaintexts are first encrypted with 5 rounds of PRESENT and then partially decrypted by 1 round by applying the reverse of the P-box and the reverse of the S-box in order to get the values that would have the division property. If there are values for which the XOR sum is 0, they are added to the list of possible keys. The loop continues the generating and filtering processes until only few keys are left unfiltered in the final key list. The number of unfiltered keys depends on how the key filtering exponentially decreases in performance. For 5 rounds specifically, we have identified that after roughly 2^{10} keys remain in the list, the *keyRecovery* function begins filtering a negligible amount of keys in each subsequent iteration, thus the number of unfiltered keys does not get reduced by a lot. This is where the loop is stopped and a bruteforce is required to check all the remaining keys that are left and identify which is the one we need (Fig. 15). To that end, each key possibility is constructed from the possible keys list by inverting the key schedule for each round. A random 64-bit plaintext is generated using *genList* to verify if a 5-round instance of PRESENT using a random key returns the same ciphertext as the same instance but using one of the key candidates we have retrieved. If both instances return the same ciphertext, then we have identified the correct key and there is no need to continue checking the rest of the candidates, so we stop the process and print the final key.

The integral attack on 5 rounds provides the code basis for subsequent attacks. All we need to know to do an attack on 6 rounds is the respective distinguisher over 5 rounds. Again, this distinguisher has been found with the help of the MILP modeling program - it turns out that this time by using an input structure with 13 constant bits, we are able to reveal all 64 key bits (Fig. 16). Knowing this, the 5-round attack can be easily transformed to a 6-round attack by simply modifying the number of rounds for the PRESENT instance from 5 to 6,

the *genList* function so that it generates plaintexts with 13 non-constant bits instead of 4 (therefore also changing the number of plaintexts from 2^4 to 2^{13}) and the limit at which the filtering loop stops. For the last one, it can be verified experimentally that when the length of the key list reaches around 2^{16} keys, the subsequent filtering is negligible, so this is a good number to terminate the loop. Everything else needed to conduct the 6-round attack remains the same as described in the 5-round attack.

For the final attack on 7-rounds, the process starts getting slightly tricky. From the search for a distinguisher over 6 rounds with the MILP modeling program it can be seen that a practical distinguisher is hard to identify, as the minimum input required to reveal all key bits contains 32 non-constant bits, which is considered too large to be practical (Fig. 17). Instead, as a distinguisher we can use the minimum input that reveals the maximum number of bits and adapt the attack to handle the case where not all bits are revealed to compute the XOR sum. In our case, the optimal distinguisher is found at input with 16 non-constant bits, revealing all bits except for those at indices 21 to 23, 29 to 31, 37 to 39, 45 to 47, 53 to 55 and 61 to 63 (Fig. 18). If we are observant, we can notice that if we split the 64-bit value into 16 blocks of 4 bits, the unrevealed bits are respectively the 3 most significant bits in each of the blocks with indices 5, 7, 9, 11, 13 and 15. Since in our *keyRecovery* function we check the XOR sum for each block of 4 bits, we can add a specific case for when we are checking these specific blocks so that we keep the value only if, after applying a bitwise AND operation between the least significant bit of that block (the only revealed bit in the block) and the current value of the XOR sum, the result is 0 (Fig. 19). This is because if it is not 0, then we know for certain that it is not possible for the XOR sum to result in 0 at the end. In this way, while we do get more candidates for the final key list in the bruteforce phase, we still manage to filter out a significant chunk of impossible keys even when the distinguisher does not give us the full information about some of the key bits. For the rest of the attack we only need to modify the same parameters as we did with the 6-round attack. The generating function generates 2^{16} plaintexts with the 16 least significant bits being non-constant, at which point it becomes noticeably slower than before due to the larger number of plaintexts that need to be generated. Lastly, to determine the number of keys left before the bruteforce, we can once more check experimentally for which values the filtering becomes too inefficient. From our tests, we have concluded that 2^{20} keys remaining in the candidate list is a good estimate to end the filtering loop, even though it might take a while to reach because of the sheer number of potential keys checked by the filtering function.

6.4. Assessment

The goal of the technical deliverable was to provide a method to conduct integral attacks within a practical amount of time on as many encryption rounds of the PRESENT block cipher as possible. The programs developed for the purposes of this deliverable and the results given by each of them most certainly fully accomplish this goal.

The technical deliverable also manages to use the information provided in the scientific deliverable as a theoretical basis so that it can provide a conclusive answer to the project's scientific question "How many rounds of the PRESENT block cipher can be attacked using integral attacks with practical complexity?". Indeed, it can be seen that the maximum number of rounds we have managed to successfully attack with our method is 7. It should be noted that there was an attempt to do an attack on 8 rounds, however while searching for a distinguisher to help us do so, it was seen that no distinguisher exists that can lead to a practical 8-round attack with our technique. Nevertheless, an experiment was conducted to try to do the attack using the 6-round distinguisher (the same one used for the attack on 7-rounds) but to no success. We will not describe this experiment here but for the sake of completeness, the code for it has been included in the project's source files.

A small aspect of most of the programs that could use some room for improvement is the code itself. While it does do its job as intended on a technical level, each integral attack program's code is slightly inconsistent with the rest in terms of visual structure as well as output clarity. With a little dedication, it can be tweaked to be more visually appealing and to more descriptively provide information whenever certain output is printed in the terminal window.

As a whole, the technical deliverable manages to satisfy the needed requirements and produce the expected results. It clearly shows the possibility to attack up to 7 rounds of PRESENT with the utility of MILP so that one is able to retrieve the secret key. The accomplishment of the main goals of the deliverable suffice to deem it as a success.

Acknowledgment

The authors would like to thank the BiCS management and education team for the amazing work done.

7. Conclusion

To sum up, the project most certainly manages to satisfy its defined requirements and live up to the expected results. Therefore, it can be considered successful in what it tries to achieve, despite the slight imperfections mentioned earlier. The most important thing is that this report gives the reader practical knowledge about the fields of cryptography and cryptanalysis, as the project manages to provide insight into the utilization of mathematical tools, such as MILP optimization problems, to efficiently conduct cryptanalytical attacks on block ciphers.

References

[1] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. PRESENT: an ultra-lightweight block cipher. In Pascal Paillier and Ingrid

Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, volume 4727 of *Lecture Notes in Computer Science*, pages 450–466. Springer, 2007.

[2] Patrick Derbez and Baptiste Lambin. Fast MILP models for division property. *IACR Trans. Symmetric Cryptol.*, 2022(2):289–321, 2022.

[3] Yosuke Todo and Masakatu Morii. Bit-based division property and application to simon family. In Thomas Peyrin, editor, *Fast Software Encryption - 23rd International Conference, FSE 2016, Bochum, Germany, March 20-23, 2016, Revised Selected Papers*, volume 9783 of *Lecture Notes in Computer Science*, pages 357–377. Springer, 2016.

[4] Zejun Xiang, Wentao Zhang, Zhenzhen Bao, and Dongdai Lin. Applying MILP method to searching integral distinguishers based on division property for 6 lightweight block ciphers. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I*, volume 10031 of *Lecture Notes in Computer Science*, pages 648–678, 2016.

8. Appendix

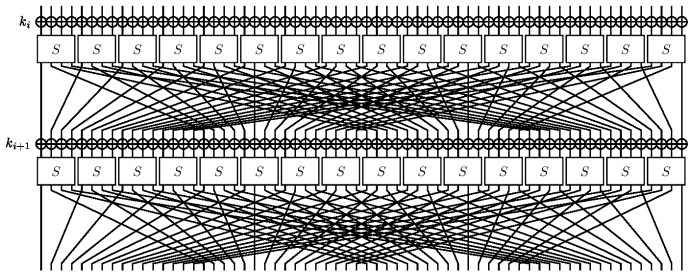

Fig. 1. Overview of PRESENT's encryption scheme, as shown in [1]

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S[x]$	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

Fig. 2. PRESENT's S-box, as shown in [1]

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$P(i)$	0	16	32	48	1	17	33	49	2	18	34	50	3	19	35	51
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$P(i)$	4	20	36	52	5	21	37	53	6	22	38	54	7	23	39	55
i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
$P(i)$	8	24	40	56	9	25	41	57	10	26	42	58	11	27	43	59
i	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
$P(i)$	12	28	44	60	13	29	45	61	14	30	46	62	15	31	47	63

Fig. 3. PRESENT's P-box, as shown in [1]

1. $[\mathtt{k}_{79}\mathtt{k}_{78}\ldots\mathtt{k}_1\mathtt{k}_0] = [\mathtt{k}_{18}\mathtt{k}_{17}\ldots\mathtt{k}_{20}\mathtt{k}_{19}]$
2. $[\mathtt{k}_{79}\mathtt{k}_{78}\mathtt{k}_{77}\mathtt{k}_{76}] = S[\mathtt{k}_{79}\mathtt{k}_{78}\mathtt{k}_{77}\mathtt{k}_{76}]$
3. $[\mathtt{k}_{19}\mathtt{k}_{18}\mathtt{k}_{17}\mathtt{k}_{16}\mathtt{k}_{15}] = [\mathtt{k}_{19}\mathtt{k}_{18}\mathtt{k}_{17}\mathtt{k}_{16}\mathtt{k}_{15}] \oplus \mathtt{round\_counter}$

Fig. 4. PRESENT's key scheduling algorithm for an 80-bit key, as shown in [1]

```python
class Present:
    def __init__(self, rounds=31, key=-1):
        if key == -1:
            key = random.randrange(1 << 64)

        self.rk = genRoundKeys(key, rounds)
        self.rounds = rounds

    def encrypt(self, x):
        for i in range(self.rounds):
            x = addRoundKey(x, self.rk[i])
            x = sBoxLayer(x)
            x = pLayer(x)
        x = addRoundKey(x, self.rk[self.rounds])
        return x
```

Fig. 5. Overview of our implementation of PRESENT as a class

```python
def genRoundKeys(key, rounds):
    rk = []
    for i in range(rounds+1):
        rk.append(key)
        key = ((key & (2 ** 19 - 1)) << 45) + (key >> 19)
        key = (sbox[key >> 60] << 60) + (key & (2 ** 60 - 1))
        key ^= (i+1) << 15
    return rk
```

Fig. 6. Key scheduling function (adjusted for a 64-bit key)

```python
def addRoundKey(x, rk):
    return x ^ rk

def sBoxLayer(x):
    res = 0
    for i in range(16):
        res += (sbox[(x >> (i * 4)) & 0xF]) << (i * 4)
    return res

def pLayer(x):
    res = 0
    for i in range(64):
        res += ((x >> i) & 0x1) << p[i]
    return res
```

Fig. 7. Functions used in PRESENT's encryption

```python
def modeling(m, x, v):
    # for i in range(len(x)):
    #     m.addVar(vtype=GRB.BINARY, name=f"x{i}")
    # m.update()

    expr = LinExpr(v[:-2], x)
    expr.addConstant(v[-2])
    if v[-1] == ">=":
        m.addConstr(expr >= 0)
    else:
        m.addConstr(expr == 0)

def addConstrPerm(m, x, y, p):
    for j in range(64):
        m.addConstr(x[p[j]] == y[j])

def addConstrSbox(m, x, y, listIneq):
    for ineq in listIneq:
        modeling(m, x+y, ineq)
```

Fig. 8. Defining the constraints for our Gurobi model

```python
ineqSbox = [[1, 1, 1, 1, -1, -1, -1, -1, 0, '>='],
            [-4, -2, -2, -2, -3, 1, 4, 1, 7, '>='],
            [-2, 0, 0, 0, 2, -1, -1, -1, 3, '>='],
            [0, -1, -1, -2, 2, 3, 3, 3, 0, '>='],
            [1, 1, 1, 1, -2, 1, -2, -2, 1, '>='],
            [1, 0, 0, 0, -1, -2, -1, 1, 2, '>='],
            [-2, -1, -1, 0, -1, 1, 0, 1, 3, '>='],
            [0, 0, 0, 0, 1, -1, 1, -1, 1, '>='],
            [0, -2, -2, 0, 2, 1, -1, 1, 3, '>='],
            [-1, 0, -1, 0, 1, 2, 2, 2, 0, '>=']]
```

Fig. 9. S-box of inequalities for the Gurobi model

```
for i in range(1, 65):
    inp = (1 << i) - 1
    inp = [int(n) for n in bin(inp)[2:].zfill(64)] #convert to integers
    inp.reverse()
    existTrail = [None for _ in range(64)]
    for j in range(64):
            output = 1 << j
            output = [int(n) for n in bin(output)[2:].zfill(64)] #convert to integers
            output.reverse()
            m, x, y = genModelPresent(nbRounds)
            for k in range(64):
                m.addConstr(x[0][k] == inp[k]) #x[0][k] == inp[k]
                m.addConstr(x[nbRounds][k] == output[k])
            m.optimize()
            if m.Status == GRB.OPTIMAL:
                existTrail[j] = True
            else:
                existTrail[j] = False
    inp = "".join(str(n) for n in inp)
    print(f"For input {inp} ({i})")
    stringOutput = [None for _ in range(64)]
    for k in range(64):
        if existTrail[k] == True:
            stringOutput[k] = "?"
        else:
            stringOutput[k] = "0"
    stringOutput = "".join(n for n in stringOutput)
    print(f"Output  : {stringOutput}")
    print()
```

Fig. 10. Function to look for a distinguisher over $nbRounds$ rounds

```
while(math.prod(len(x) for x in keyList) > (1 << 10)):
    #Generate a set of plaintexts with the proper structure
    plaintextList = genList(4)
    keyRecovery(present, plaintextList, keyList)
```

Fig. 11. Loop for filtering out wrong keys

```
def genList(aCount):
    valueList = []
    c = ""
    for i in range(64-aCount):
        c += str(random.randint(0,1))
    aList = list(itertools.product([0,1], repeat=aCount))
    for i in range(len(aList)):
        a = ''.join(map(str, aList[i]))
        value = c + a
        valueList.append(int(value, base=2))
    return valueList
```

Fig. 12. Function generating a set of random plaintexts with a special structure ($aCount$ indicates the number of non-constant bits)

```
For input 1111000000000000000000000000000000000000000000000000000000000000 (4)
Output  : 0000000000000000000000000000000000000000000000000000000000000000
```

Fig. 13. Distinguisher over 4 rounds of PRESENT, used to attack 5 rounds

```
def keyRecovery(present, plaintextList, keyList):
    """
    From a set of plaintext plaintextList, filter out some wrong keys that are still in keyList
    """
    ciphertextList = []
    for plaintext in plaintextList:
        ciphertext = present.encrypt(plaintext)
        ciphertextList.append(ciphertext)

    ciphertextListPrime = []
    for ciphertext in ciphertextList:
        res = 0
        for i in range(64):
            res += ((ciphertext >> i) & 0x1) << p_inverse[i]
        ciphertextListPrime.append(res)

    for i in range(16):
        temp = []
        for k in keyList[i]:
            xorSum = 0
            for y in ciphertextListPrime:
                x = sbox_inverse[((y >> (i * 4)) & 0xF) ^ k]
                xorSum ^= x
            if xorSum == 0:
                temp.append(k)
        keyList[i] = temp
```

Fig. 14. Function to retrieve possible candidates for the key

```
plain = genList(4)[0]
candidates = list(itertools.product(*keyList))
for c in candidates:
    kPrime = 0
    for i in range(16):
        # kPrime += c[i] * 16**(i)
        kPrime |= c[i] << (4*i)
    k = 0
    for i in range(64):
        k += ((kPrime >> i) & 0x1) << p[i]

    if k == present.rk[-1]:
        print(f"k5 = {hex(k)}")

    for i in range(5,0,-1):
        k ^= i << 15
        k = (sbox_inverse[k >> 60] << 60) | (k & (2 ** 60 - 1))
        k = ((k & (2 ** 45 - 1)) << 19) | (k >> 45)
    c1 = Present(rounds=5, key=k).encrypt(plain)
    c2 = present.encrypt(plain)
    if c1 == c2:
        print("Found key:", k)
        break
```

Fig. 15. Bruteforce for finding the exact key from all candidates

```
For input 1111111111110000000000000000000000000000000000000000000000000000 (13)
Output  : 0000000000000000000000000000000000000000000000000000000000000000
```

Fig. 16. Distinguisher over 5 rounds of PRESENT, used to attack 6 rounds

```
For input 11111111111111111111111111111111000000000000000000000000000000000 (32)
Output  : 0000000000000000000000000000000000000000000000000000000000000000
```

Fig. 17. Distinguisher over 6 rounds of PRESENT revealing all key bits but at the cost of larger input size, making it impractical to use for the 7-round attack

```
For input 1111111111111111000000000000000000000000000000000000000000000000 (16)
Output  : 0000000000000000000???00000???00000???00000???00000???00000???
```

Fig. 18. A more practical distinguisher over 6 rounds of PRESENT, used to attack 7 rounds

```
listFullZero = [0,1,2,3,4,6,8,10,12,14]
listOnlyOneZero = [5,7,9,11,13,15]

ciphertextList = []
for plaintext in plaintextList:
    ciphertext = present.encrypt(plaintext)
    ciphertextList.append(ciphertext)

ciphertextListPrime = []
for ciphertext in ciphertextList:
    res = 0
    for i in range(64):
        res += ((ciphertext >> i) & 0x1) << p_inverse[i]
    ciphertextListPrime.append(res)

for i in range(16):
    temp = []
    for k in keyList[i]:
        xorSum = 0
        for y in ciphertextListPrime:
            x = sbox_inverse[((y >> (i * 4)) & 0xF) ^ k]
            xorSum ^= x
        if i in listFullZero:
            if xorSum == 0:
                temp.append(k)
        elif i in listOnlyOneZero:
            if xorSum&1 == 0: #LSB of xorSum is 0
                temp.append(k)
    keyList[i] = temp
```

Fig. 19. Modified $keyRecovery$ function for the 7-round attack