An Easy to Use Tool for Rotational-XOR Cryptanalysis of ARX Block Ciphers

Adrián Homero Ranea Robles

Thesis submitted for the degree of Master of Science in Electrical Engineering, option Electronics and Integrated Circuits

Thesis supervisors:
Prof. dr. ir. Vincent Rijmen
Tomer Ashur
Yunwen Liu

Academic year 2016 – 2017

Master of Science in Electrical Engineering
An Easy to Use Tool for Rotational-XOR Cryptanalysis of ARX Block Ciphers

Adrián Homero Ranea Robles

Thesis submitted for the degree of Master of Science in Electrical Engineering, option Electronics and Integrated Circuits

Thesis supervisors:
Prof. dr. ir. Vincent Rijmen
Tomer Ashur
Yunwen Liu

Academic year 2016 – 2017
Preface

I would like to thank my promoter Prof. Vincent Rijmen for supporting this research and given me this opportunity. Furthermore, I would especially like to thank my supervisors, Tomer Ashur and Yunwen Liu, for guiding me through the research and helping me with academic writing.

My sincere gratitude also goes to my family and friends, for their support and encouragement.

Adrián Homero Ranea Robles
Contents

Preface i

Abstract iii

List of Figures and Tables iv

List of Abbreviations and Symbols v

1 Introduction 1
   1.1 Related Work ........................................... 2
   1.2 Thesis Contributions .................................. 2
   1.3 Structure of the Thesis ................................. 3

2 Background 4
   2.1 Basics of Cryptography ................................. 4
   2.2 Block Ciphers ........................................... 5
   2.3 Cryptanalysis of ARX Block Ciphers ..................... 9
   2.4 Automatic Search for Rotational-XOR Characteristics .... 16
   2.5 Conclusion ............................................... 19

3 The ArxPy Tool 20
   3.1 Functionality and Features ............................ 20
   3.2 Usage ................................................... 20
   3.3 Implementation ......................................... 26
   3.4 Conclusion ............................................... 31

4 Conclusion 32
   4.1 Future Work ............................................ 32

A Source code of ArxPy 35

B ARX implementations 59

C Paper 63

Bibliography 70
Abstract

Due to the progress of technology, electronic devices have become so cheap and small that they are being embedded in everyday objects. Some of these new devices have extreme constraints in computational power, chip area or memory. Therefore, they cannot implement conventional cryptographic algorithms. As a result, several cryptographic primitives tailored to such constrained environments have been published recently. Some of these proposals only use modular additions, cyclic rotations and exclusive-or operations.

In this thesis, a computer tool has been developed to speed up and make easier the security evaluation of this type of proposals. The first part of the thesis starts with an introduction of cryptography. Then, the structure of these proposals is explained. Afterwards, three techniques to analyse the security of these proposals are explained along with a method for applying one of these techniques with an automatic search tool.

Our computer tool automatizes this method. The tool takes a Python implementation of a cryptographic primitive as input and automatically applies this method. As opposed to most of the automated tools, which only support a small set of primitives, our tool supports an entire class of cryptographic primitives and it is executed with a simple shell command. The last part of this thesis explains how to use our tool and the details of the implementation with several examples.
List of Figures and Tables

List of Figures

2.1 Communication over an insecure channel. .......................... 4
2.2 Encryption of an iterated cipher. ........................................ 6
2.3 Basic SPN structure. ..................................................... 6
2.4 Basic Feistel structure. .................................................. 6
2.5 The round function of SPECK. ......................................... 8
2.6 Differential (\(\alpha, \beta\)) of the encryption function \(E_k\). .......... 10
2.7 2-round differential characteristic (\(\alpha, \gamma, \beta\)). .............. 11
2.8 Rotational pairs. .......................................................... 12
2.9 RX differential. ............................................................ 15
3.1 Expression tree of \(\text{Xor} (\text{Ror}(a, 7), c)\). ...................... 30

List of Tables

2.1 Parameters of SPECK. .................................................... 8
2.2 Notation used in theorem 2.3.1. ...................................... 16
List of Abbreviations and Symbols

Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES</td>
<td>Advance Encryption Standard</td>
</tr>
<tr>
<td>ARX</td>
<td>Addition-Rotation-XOR</td>
</tr>
<tr>
<td>RFID</td>
<td>Radio-Frequency IDentification</td>
</tr>
<tr>
<td>XOR</td>
<td>Exclusive-Or</td>
</tr>
<tr>
<td>SAT</td>
<td>Boolean Satisfiability</td>
</tr>
<tr>
<td>SPN</td>
<td>Substitution-Permutation Network</td>
</tr>
<tr>
<td>DES</td>
<td>Data Encryption Standard</td>
</tr>
<tr>
<td>NIST</td>
<td>National Institute of Standards and Technology</td>
</tr>
<tr>
<td>MAC</td>
<td>Message Authentication Code</td>
</tr>
<tr>
<td>SMT</td>
<td>Satisfiability Modulo Theories</td>
</tr>
</tbody>
</table>

Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊕</td>
<td>Modular addition</td>
</tr>
<tr>
<td>⊕</td>
<td>XOR</td>
</tr>
<tr>
<td>≫</td>
<td>Left rotation</td>
</tr>
<tr>
<td>≪</td>
<td>Right rotation</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Information security is the practice of protecting information and securing communications in the presence of adversaries. Cryptography is the underlying tool used to design security protocols in a communication system. It is applied to achieve confidentiality, integrity, or authentication among other security goals.

For example, cryptography prevents unauthorized disclosure of the information by encrypting the information, that is, transforming it into apparent nonsense. In order to recover the original information, a key, some secret value only known by the authorized parties, is required. Encryption systems are classified into two main categories: symmetric cryptosystems and asymmetric cryptosystems.

In symmetric cryptosystems, the same secret key is used to transform (encrypt) and recover (decrypt) the information. On the other hand, asymmetric cryptosystems use a public key to encrypt and a private key to decrypt. One particular type of symmetric cryptosystems is the block ciphers whose best representative is the Advance Encryption Standard (AES) [1] which was selected by the US National Institute for Standards in Technology in 2001.

Cryptography is widely used in our everyday life. Every time a web browser is used to send an email, check a bank account or shop online, cryptographic algorithms are used. Cryptography is used by many types of electronic devices such as PC’s, smartphones, smart-cards (e.g. credit cards or identification cards) or WiFi access points. For example, AES is used in over two billion devices and the entire internet infrastructure [2].

Due to the progress of technology, electronic devices have become so cheap and small that they are being embedded in everyday objects. As a result, the Internet is evolving into a network of “smart objects” that communicate with each other. This “Internet of Things” will include about 20 billion devices by 2020 according to Gartner [3].

The basic function of these devices is to collect and transmit information. In some cases, sensitive information is collected such as health-monitoring or biometric data [4]. Therefore, there is a high demand to implement cryptographic algorithms in these devices.

However, some of these new devices have so extreme constrains in computational
power, chip area or memory that they are not powerful enough to use the same
cryptographic algorithms as standard PCs. Example of these types of devices are
RFID (Radio-Frequency IDentification) chips and sensor networks. For this reason,
many cryptographic algorithms tailored to such constrained environments have been
published recently.

Some of these proposals only use three types of operations: modular addition,
cyclic rotation and exclusive-or (XOR). Examples of these Addition-Rotation-XOR
(ARX) primitives are Salsa20 [5], LEA [6], Chaskey [7], SPECK [8] or SPARX [9]. ARX
primitives are among the best performers in software [10]. Before these promising
candidates are used in real-world applications, it is necessary to analyse their security
in detail.

1.1 Related Work

Biham and Shamir proposed in 1991 a technique to analyse the security of block
ciphers [11]. This technique, called differential cryptanalysis, studies how differences
in a pair of inputs can affect the resultant difference at the pair of outputs. This is
done by tracing the possible evolutions of the differences through the cipher, which
are called characteristics. Differential cryptanalysis have been applied to several
ARX primitives [12, 13, 14].

Another similar technique is rotational cryptanalysis, which was proposed as a
generic attack to ARX structures in 2010 [15]. Rotational cryptanalysis studies the
propagation through the encryption function of a rotational pair, that is, a pair of
states where the second state is the first one rotated by a certain amount. Rotational
cryptanalysis was improved in [16], where the propagation of rotational pairs through
modular addition, especially through consecutive modular additions, was studied in
more detail.

One of the main drawbacks of rotational cryptanalysis is that it can not be applied
to ARX primitives where constants are injected into the state. A combination of
differential and rotational cryptanalysis was considered in [15] and formalized as
rotational-XOR cryptanalysis in [17]. Rotational-XOR cryptanalysis removes this
restriction and so it can be applied to a bigger subset of ARX primitives than
rotational cryptanalysis.

In differential and rotational-XOR cryptanalysis, a characteristic that predict
the differences in each round with high probability is necessary for the attack to
be successful. Recently, a SAT (Booolean Satisfability) solver was used to find
characteristics with high probability in ARX primitives [12, 14]. To the best of
our knowledge, the only application of a SAT solver for finding rotational-XOR
characteristics was done in [18] for the ARX block cipher SPECK [19].

1.2 Thesis Contributions

We have implemented a computer tool for finding optimal rotational-XOR character-
istics of ARX block ciphers. Our tool, called ArxPy, takes a Python implementation
of an ARX block cipher as input and finds an optimal characteristic by using a SAT solver.

Most of the tools [18, 20, 21, 22] that search for characteristics with high probability are implemented specifically for a particular cipher or a small set of ciphers. In order to use such tools with an arbitrary cipher, it is necessary to make a significant effort in adapting the tool.

Given a Python implementation of an ARX block cipher, ArxPy is executed with a simple shell command. Therefore, the only effort to use ArxPy is implementing the ARX block cipher with the Python language. On top of that, ArxPy have been implemented with a modular architecture and is open source. Therefore, it can be easily adapted for specific needs.

1.3 Structure of the Thesis

The general outline of the thesis is summarized below:

- Chapter 2 contains the background necessary to understand the purpose of ArxPy. Symmetric cryptosystems, block ciphers and ARX block ciphers are introduced. Then, differential cryptanalysis, rotational cryptanalysis and rotational-XOR cryptanalysis are explained. Afterwards, a SAT-based method for finding optimal rotational-XOR characteristic is described.

- Chapter 3 focuses on ArxPy. The functionality and features of ArxPy are presented and its usage and implementation are explained in detail.

- Chapter 4 contains the conclusion of the thesis. A summary of the thesis and some future lines of research are described.
Chapter 2

Background

This chapter provides the complete background related to the objective of the thesis. First, an informal introduction of cryptography is explained, along with its security goals and a cryptographic scheme. Then, block ciphers and ARX block ciphers are discussed. Afterwards, three techniques to analyze ARX block ciphers are presented: differential cryptanalysis, rotational cryptanalysis and rotational-XOR cryptanalysis. Finally, a SAT-based method for finding rotational-XOR characteristics is described.

2.1 Basics of Cryptography

The following abstract scenario, depicted in Figure 2.1, will be used to explain some concepts of cryptography. One party, Alice, wants to send a message to another party, Bob, over an insecure channel in the presence of a malicious party Eve.

![Figure 2.1: Communication over an insecure channel.](image)

Cryptography is the discipline that studies methods and techniques for secure communications in the presence of malicious parties. Cryptographic schemes can provide several security goals such as confidentiality, integrity, authenticity, etc. In our abstract scenario, confidentiality is achieved if Eve is not able to obtain any information about the message Alice sent. Integrity is achieved if Bob can detect whether the original message was modified during the transmission, whereas authenticity is achieved if Bob is able to verify that the message was actually sent by Alice.
2.2 Block Ciphers

An example of a cryptographic scheme which provides confidentiality is the symmetric cryptosystem, which consists of an encryption algorithm and a decryption algorithm. The encryption algorithm takes a message, called the plaintext, and some secret information, called the key, and produces another message, called the ciphertext. The decryption algorithm takes a ciphertext and a key and recovers the plaintext using that key.

In our scenario (Figure 2.1), a symmetric cryptosystem can be used to achieve confidentiality as follows. First, Alice chooses a key and sends it to Bob over a secure channel. Second, Alice encrypts her message using the key and sends the ciphertext over the insecure channel. Finally, Bob gets the ciphertext and decrypts it to obtain the original message. Eve, who is eavesdropping the insecure channel, can only obtain the ciphertext. Since she does not have the key, she cannot recover the original message.

There is a counterpart of symmetric cryptosystems, called asymmetric cryptosystems, where instead of using the same key for encryption and decryption, the key to encrypt a plaintext and the key to decrypt such ciphertext are different but linked by a mathematical relation. These cryptosystems remove the need to use a secure channel to transfer the key, but are much slower than symmetric cryptosystems.

The next section introduces a particular type of symmetric cryptosystems: block ciphers.

2.2 Block Ciphers

A block cipher with an $n$-bit block and a $k$-bit key is a symmetric cryptosystem where the encryption takes an $n$-bit string as a plaintext and a $k$-bit string as a key and produces an $n$-bit string as the ciphertext. In other words, for a given key $k$, the encryption $E_k : \{0,1\}^n \rightarrow \{0,1\}^n$ of a block cipher is a permutation (a bijective mapping from a set to itself) over the $n$-bit strings $\{0,1\}^n$ and the decryption $D_k$ is the inverse permutation $D_k = E_k^{-1}$.

Block ciphers are one of the oldest and most-studied cryptographic primitives. They can be used to achieve confidentiality as well as to build other cryptographic primitives [23] such as stream ciphers (another type of symmetric cryptosystem) or message authentication codes (a symmetric primitive which achieves integrity and authenticity).

In 1949, Shannon suggested several design principles for secrecy systems in his pioneering paper [24]. These principles are the basis of modern block ciphers. Basically, a block cipher should achieve a high degree of confusion (the complexity of the relation between the ciphertext and the plaintext) and a high degree of diffusion (the influence of each bit of the plaintext and each bit of the key on the ciphertext). Shannon also suggested that secrecy systems should be designed to resist all known attacks, which is currently the main design principle of modern block ciphers.

Most modern block ciphers are iterated ciphers, where the encryption and decryption are based on the iteration of a fixed key-dependent function, the round function. In most cases, a key scheduling algorithm expands the key(master key) into
the round keys passing to the round functions Figure 2.2 represents the encryption of an iterated cipher.

Two common structures for the round function are the substitution-permutation network (SPN) and the Feistel structure. The basic SPN structure consists of a layer of substitution boxes and a layer of linear permutations. On the other hand, the basic Feistel structure divides the input into two halves, mixes the first half with the right half processed by the round function, and swaps both halves. Figure 2.3 shows an SPN structure with a 16-bit block and 4 substitution boxes and Figure 2.4 shows a Feistel structure.

Two well-known examples of block ciphers are the Data Encryption Standard
2.2. Block Ciphers


The next section introduces a particular type of block ciphers: the ARX block ciphers.

2.2.1 ARX block ciphers

Block ciphers which only use modular additions, cyclic rotations and exclusive-or (XOR) operations are called Addition-Rotation-XOR (ARX) block ciphers. The usage of these operations in modern block ciphers is not new; the first one dates back to 1987: the FEAL cipher [26]. The term ARX is much recent; it was proposed in 2009 by Weinmann [27].

Many ARX block ciphers have been proposed since 2009. Some examples are: LEA [6], the underlying permutation for the message authentication code Chaskey [7], SPARX [9], Speck [8], and the underlying block cipher Threefish of the hash function Skein [28].

Some important properties of the ARX block ciphers are:

• Any function over the $n$-bit strings $\{0, 1\}^n$ can be realized with modular additions, rotations, XORs and a single constant [15].

• ARX block ciphers are efficient in software. In a comparison of software implementations of block ciphers for small processors [10, 29], the most efficient ones were ARX block ciphers.

• ARX block ciphers are simple and easy to describe, which results in implementations with small code size. As shown in [10, 29], the implementations with the smallest code size were achieved by ARX block ciphers.

Removing one of the operations results in a dramatic loss of security. Without the modular addition, an XR block cipher is easily broken by describing each output bit and solving the resulting system of linear equations modulo 2 (which can be done efficiently). Without the rotation, it is also possible to break an AX block cipher with low complexity [30]. Without the XOR, an AR block cipher needs a higher number of operations than an ARX block cipher for the same level of security [15].

Next is described the ARX block cipher Speck in detail to show the simplicity of these types of block ciphers.

The Speck family of block ciphers

Speck is a family of ARX block ciphers designed by the National Security Agency [8]. The different variants of Speck are given in Table 2.1. Speck has two additional parameters $(\alpha, \beta)$ which depend on the variant: $(\alpha, \beta) = (7, 2)$ for the 32-bit block size version and $(\alpha, \beta) = (8, 3)$ for the other versions.

The encryption algorithm splits the $2n$-bit plaintext into two $n$-bit words $x_0$ and $y_0$ and applies the round function iteratively. The round function takes as input two
### Table 2.1: Parameters of Speck.

<table>
<thead>
<tr>
<th>Block size</th>
<th>Key size</th>
<th>Rounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>64</td>
<td>22</td>
</tr>
<tr>
<td>48</td>
<td>72</td>
<td>22</td>
</tr>
<tr>
<td>48</td>
<td>96</td>
<td>23</td>
</tr>
<tr>
<td>64</td>
<td>96</td>
<td>26</td>
</tr>
<tr>
<td>64</td>
<td>128</td>
<td>27</td>
</tr>
<tr>
<td>96</td>
<td>96</td>
<td>28</td>
</tr>
<tr>
<td>96</td>
<td>144</td>
<td>29</td>
</tr>
<tr>
<td>128</td>
<td>128</td>
<td>32</td>
</tr>
<tr>
<td>128</td>
<td>192</td>
<td>33</td>
</tr>
<tr>
<td>128</td>
<td>256</td>
<td>34</td>
</tr>
</tbody>
</table>

The key schedule reuses this round function. The key is split into \( m \) words of \( n \) bits \((l_{m-2}, \ldots, l_0, k_0)\) and the round keys \( k_i \) are computed as follows:

\[
\begin{align*}
  l_{i+m-1} &= (l_i \gg \alpha \oplus k_i) \oplus i \\
  k_{i+1} &= ((k_i \ll \beta) \oplus l_{i+m-1})
\end{align*}
\]

\( n \)-bit word \( x_i \) and \( y_i \) and a round key \( k_i \) and outputs two words \( x_{i+1} \) and \( y_{i+1} \) as follows:

\[
\begin{align*}
  x_{i+1} &= (x_i \gg \alpha) \boxdot y_i \oplus k_i \\
  y_{i+1} &= ((y_i \ll \beta) \oplus x_{i+1})
\end{align*}
\]

where \( \boxdot, \ll, \gg \) and \( \oplus \) denote the modular addition, left and right cyclic rotations and XOR, respectively. The round function is depicted in Figure 2.5.
2.3 Cryptanalysis of ARX Block Ciphers

While cryptography is the discipline that studies methods to build secure communications, cryptanalysis is the discipline that analyses the security of such methods by finding weaknesses and attacks. In the standard model of the cryptanalysis of symmetric cryptosystems, the attacker has access to the ciphertexts and knows all the details of the symmetric cryptosystem, except the secret key (Kerckhoffs’ principle).

The main goal of an attack on a block cipher is to obtain the secret key. However, there is another type of attacks, called distinguishing attacks, where the attacker is able to distinguish a block cipher from a random permutation by observing both of them only in terms of their inputs and outputs. Distinguishing attacks do not recover the key or the plaintext, but they exhibit some non-random behaviour of the block cipher. This non-random behaviour may further exploited to mount a key-recovery attack [11].

Cryptanalytic techniques can be classified according to the capabilities of the attacker. Some of the most important types [31] are:

- Ciphertext-only attacks. The attacker has access only to a set of ciphertexts.
- Known plaintext attacks. The attacker has access to a limited number of plaintexts and their corresponding ciphertexts.
- Chosen plaintext attacks. The attacker can choose a number of plaintexts and query their respective ciphertexts.
- Chosen ciphertext attacks. The attacker can choose a number of ciphertexts and query their respective plaintexts.

It is also possible to consider a combination of the above attacks. An example of a known plaintext attack is the exhaustive key search. Given a ciphertext and its corresponding plaintext, this attack consists of decrypting the ciphertext under all possible values of the key until the decryption matches the plaintext.

In this chapter, three chosen plaintext attacks will be described: differential cryptanalysis (Section 2.3.1), rotational cryptanalysis (Section 2.3.2) and rotational-XOR cryptanalysis (Section 2.3.3). Differential cryptanalysis has played a major role in the design of modern block ciphers [32, 33], while rotational cryptanalysis and rotational-XOR cryptanalysis are recent techniques to analyse ARX block ciphers.

2.3.1 Differential cryptanalysis

Differential cryptanalysis was proposed by Biham and Shamir in 1991 to attack DES [11]. This technique studies the propagation of differences through the encryption function of a block cipher. In [11], XOR was used as the difference operation and this will be assumed herein as well.
2.3. Cryptanalysis of ARX Block Ciphers

Definitions and notations

Let $E_k$ be the encryption function of a block cipher with key $k$, $(P, P')$ a pair of plaintexts and $(C, C')$ the pair of corresponding ciphertext, that is, $C = E_k(P)$ and $C' = E_k(P')$. The difference $\alpha$ of a plaintext pair, or input difference, is the XOR of the plaintext values $\alpha = P \oplus P'$. Similarly, the difference $\beta$ of a ciphertext pair, or output difference, is the XOR of the ciphertext values $\beta = C \oplus C'$.

A differential $(\alpha, \beta)$ is a pair of an input difference $\alpha$ and an output difference $\beta$ (see Figure 2.6). The probability of a differential $P(\alpha \to \beta)$ is the probability that a pair of plaintexts with difference $\alpha$ produces an output difference $\beta$ when the plaintext pair is sampled uniformly at random.

\[ P \oplus P' = \alpha \]
\[ E_k \]
\[ C \oplus C' = \beta \]

Figure 2.6: Differential $(\alpha, \beta)$ of the encryption function $E_k$.

Differential attack

In its basic form, differential cryptanalysis is an attack that exploits differentials with high probability. The probability of any differential for an $n$-bit random permutation is $2^{-n}$. Therefore, if a differential $(\alpha, \beta)$ for an $n$-bit block cipher is found such that $P(\alpha \to \beta) > 2^{-n}$, it is possible to distinguish the block cipher from a random permutation by encrypting pairs of plaintexts with difference $\alpha$ and counting the occurrences of $\beta$ as output difference.

On top of that, this non-random behaviour can be exploited to recover the key faster than exhaustive search, as it was shown in [11]. The higher the probability of the differential, the more efficient the key recovery attack is. For example, the number of plaintext pairs required for the key-recovery attack is a small multiple of $1/p$ [34], where $p$ is the probability of the differential used for the attack.

The search for a differential with a high probability is the core of the attack. The rest of this section will focus on differential characteristics, which can simplify significantly this search. A complete explanation of the key recovery part can be found in [34].

Differential characteristics

For an iterated cipher, the output difference $\beta$ can be obtained by propagating round-by-round the input difference $\alpha$. The sequence of intermediate differences, along with the input difference and output difference, is called a differential characteristic.
Figure 2.7 shows a characteristic with one intermediate difference for a 2-round iterated cipher with round function $f$.

$$P \oplus P' = \alpha$$

$$f$$

$$X \oplus X' = \delta$$

$$C \oplus C' = \beta$$

Figure 2.7: 2-round differential characteristic $(\alpha, \gamma, \beta)$.

The sum of the probabilities of the characteristics that share an input difference $\alpha$ and an output difference $\beta$ is exactly the probability of the differential $(\alpha, \beta)$. Furthermore, the probability of a characteristic can be computed as the multiplication of the probabilities of its single-round differentials under the assumption that the probability of transitions does not depend on the input state and the round keys are independent [35].

The search for optimal differentials, with respect to its probability, is generally a hard problem due to the size of the search space. For an $n$-bit block cipher, there are $2^{2n}$ possible differentials. Two common shortcuts are used. First, the optimal characteristic is used instead of the optimal differential [12]. While this is valid, this only provides a lower bound of the success probability and the optimal differential could have a higher probability. Second, the probability of a characteristic is computed with its single-round differentials. This is only strictly valid when the previous assumptions hold, otherwise the actual probability could be lower.

### 2.3.2 Rotational cryptanalysis

Rotational cryptanalysis was proposed as a generic attack to ARX structures by Khovratovich and Nikolic in 2010 and was applied to the block cipher Threefish [15]. This technique studies the propagation of rotational pairs through the encryption function of ARX block ciphers.

### Definitions and notation

Let $\gamma$ be a fixed rotational offset. The right cyclic rotation of a word $X$ by an amount of $\gamma$ will be denoted by $\overrightarrow{X}$, that is, $\overrightarrow{X} = X \gg\gg \gamma$. If $X$ denotes a tuple of
2.3. Cryptanalysis of ARX Block Ciphers

$n$-bit words $X = (X_1, \ldots, X_r)$, $X \gg \gamma$ denotes the rotation word-wise by $\gamma$, that is, $X \gg \gamma = (X_1 \gg \gamma, \ldots, X_r \gg \gamma)$. A pair $(X, \overset{\rightleftharpoons}{X})$ is called a rotational pair.

A pair of keys $(k, \overset{\rightleftharpoons}{k})$ is rotational if the pairs of round keys $(k_i, \overset{\rightleftharpoons}{k_i})$ form rotational pairs. A rotational input is a rotational pair of plaintexts $(P, \overset{\rightleftharpoons}{P})$. If the encryption of a rotational input under a pair of related keys $(k, \overset{\rightleftharpoons}{k})$ is also a rotational pair, that is, $(E_k(P), E_{\overset{\rightleftharpoons}{k}}(P)) = (C, \overset{\rightleftharpoons}{C})$ for some ciphertext $C$, the pair $(C, \overset{\rightleftharpoons}{C})$ is called a rotational output. Figure 2.8 illustrates this notation.

![Figure 2.8: Rotational pairs.](image)

Similar to the probability of a differential, the probability of a rotational offset $\gamma$ is the probability that a rotational input with rotational offset $\gamma$ produces a rotational output with the same rotational offset when the plaintext pair is sampled uniformly at random.

The rotational offset in rotational cryptanalysis plays the same role as the difference in differential cryptanalysis. However, rotational cryptanalysis focuses on rotational inputs and outputs which have the same rotational offset, whereas differential cryptanalysis allows the input difference and output difference to be different.

**Rotational attack**

In its basic form, rotational cryptanalysis is a distinguishing attack that exploits rotational offsets with probability higher than for a random permutation. It can be extended to a key recovery attack similar to differential cryptanalysis.

The selection of the rotational offset and the computation of this probability is the main part of the attack. In the rest of the section, the propagation of rotational pairs through the ARX operations will be studied and an estimation of the success probability will be presented.

**Propagation of rotational pairs**

One important property about rotational pairs is that the rotational property is preserved after a rotation by a constant value and through the XOR of rotational...
pairs, that is:

\[ \overrightarrow{X \oplus Y} = \overrightarrow{X \oplus Y} \]
\[ \overrightarrow{X \ll r} = \overrightarrow{X \ll r} \]
\[ \overrightarrow{X \gg r} = \overrightarrow{X \gg r} \]

for any rotational pairs \((X, \overrightarrow{X})\) and \((Y, \overrightarrow{Y})\) and any constant \(r\).

However, the rotational property is not always preserved after modular addition. Given two rotational pairs \((X, \overrightarrow{X})\) and \((Y, \overrightarrow{Y})\) where \(X\) and \(Y\) are uniformly sampled, the probability \(p_\oplus\) that their modular addition produces another rotational pair is given by the following formula [36]:

\[ p_\oplus = P(\overrightarrow{X \boxplus Y} = \overrightarrow{X \boxplus Y}) = \frac{1}{4}(1 + 2^{\gamma-n} + 2^{-\gamma} + 2^{-n}). \] (2.1)

Since Equation (2.1) is a decreasing function with respect to \(\gamma\), the probability is maximized for \(\gamma = 1\).

Estimating the success probability

The modular addition is the only operation which preserves the rotational property with a probability less than 1. Assuming that the probability of the modular additions is independent and the round keys are independent as well, the probability of a rotational offset \(\gamma\) can be computed as the multiplication of the probability of each modular addition, that is, \((p_\oplus)^q\) where \(q\) is the number of modular additions of the encryption function.

The probability of any rotational offset for an \(n\)-bit random permutation is \(2^{-n}\). Therefore, if \((p_\oplus)^q > 2^{-n}\), the ARX block cipher can be distinguished from a random permutation. By fixing \(\gamma = 1\) and using the formula (2.1), the following conclusion is obtained: any ARX block cipher implemented with less than \(n/1.415\) additions is vulnerable to rotational cryptanalysis.

There are some important remarks about this analysis, which was done similarly in [15]. First, Khovratovich et al. showed that assumption about the independence of the modular additions does not hold in many cases, and the actual success probability may be lower [16]. In particular, this assumption does not hold for a chain of modular additions, that is, a sequence of modular additions where the output of a modular addition is given directly as input to the next modular addition. The longer the chain, the lower the probability of each addition. If an ARX block cipher contains chains of modular additions, the Formula 2.1 is not valid and a correct formula was provided in [16].

Second, the above analysis only holds if no constants are injected into the state. In other words, given a rotational pair \((X, \overrightarrow{X})\) and a constant \(c\) which is not rotational-invariant (i.e. \(c \neq \overrightarrow{c}\)), XOR and modular addition does not preserve in general the
rotational property, that is:
\[
(X \oplus c, X' \oplus c) \neq (X', X')
\]
\[
(X \boxplus c, X' \boxplus c) \neq (X', X').
\]

An extension of this technique was considered in [15] to deal with the non-invariant constants of Threefish. In [17], this extension was fully formalized and formulas were given to estimate the success probability. This technique, called rotational-XOR cryptanalysis, is explained in the next section.

### 2.3.3 Rotational-XOR cryptanalysis

Rotational-XOR cryptanalysis is a recent technique to analyse ARX block ciphers. Ashur and Liu formalized it and applied it to Speck [17]. This technique studies the propagation of rotational-XOR (RX) differences through the encryption function of ARX block ciphers.

#### Definitions and notation

In this thesis, a slightly different notation from [17] will be used. A pair \((X, X')\) has RX difference \((\alpha, \gamma)\) if \(X \oplus (X' \ll \gamma) = \alpha\). To simplify the notation, the rotational offset \(\gamma\) will be fixed and \(\bar{X}\) will denote \(X' \ll \gamma\). Furthermore, we will say that RX difference of a pair \((X, X')\) is \(\alpha\) if \(X \oplus \bar{X} = \alpha\).

If \(\gamma = 0\), an RX difference is just an XOR-difference and if \(\alpha = 0\), the pair \((X, X') = (X, \bar{X})\) is just a rotational pair. Therefore, rotational-XOR cryptanalysis can be seen as a generalization of differential cryptanalysis and rotational cryptanalysis.

In rotational-XOR cryptanalysis, the attacker can obtain the encryption of plaintexts under a pair of keys \((k, k^*)\) where the RX differences of the pairs of round keys \((k_i, k_i^*)\) are known to the attacker.

The RX difference \(\alpha\) of a plaintext pair \((P, P')\) is called an input RX difference and the RX difference \(\beta\) of its corresponding ciphertext \((C, C') = (E_k(P), E_{k^*}(P))\) is called an output RX difference. An RX differential \((\alpha, \beta)\) is a pair of an input RX difference and an output difference \(\alpha\) (see Figure 2.9). The probability of an RX differential \(P(\alpha \rightarrow \beta)\) is the probability that a pair of plaintexts with RX difference \(\alpha\) produces a output RX difference \(\beta\) when the plaintext pair is sampled uniformly at random.

A RX characteristic is a sequence of intermediate RX differences along with the input RX difference and RX output difference.

#### Rotational-XOR attack

Similar to the previous techniques, rotational-XOR cryptanalysis is a distinguishing attack that exploits RX differentials. To the best of our knowledge, the only
2.3. Cryptanalysis of ARX Block Ciphers

application of rotational-XOR cryptanalysis (as described in [17]) is a distinguishing attack on Speck [17, 18].

In the rest of this section, the propagation of RX differences through the ARX operations will be studied. The results about the propagation of RX differences will be used in Section 2.4 to describe a method to search for optimal RX characteristics.

Propagation of RX differences

Lemma 2.3.1 ([18]). Given two pairs \((X, \overrightarrow{X} \oplus \alpha)\) and \((Y, \overrightarrow{Y} \oplus \beta)\) with RX differences \(\alpha\) and \(\beta\) respectively and a constant \(c\), the RX differences propagate through rotation and XOR as follows:

\[
\begin{align*}
(X \gg c, (X \oplus \alpha) \gg c) &= (Z, \overrightarrow{Z} \oplus (\alpha \gg c)), \text{ for } Z = X \gg c \\
(X \ll c, (X \oplus \alpha) \ll c) &= (Z, \overrightarrow{Z} \oplus (\alpha \ll c)), \text{ for } Z = X \ll c \\
(X \oplus Y, (\overrightarrow{X} \oplus \alpha) \oplus (\overrightarrow{Y} \oplus \beta)) &= (Z, \overrightarrow{Z} \oplus (\alpha \oplus \beta)), \text{ for } Z = X \oplus Y \\
(X \oplus c, (\overrightarrow{X} \oplus \alpha) \oplus c) &= (Z, \overrightarrow{Z} \oplus (\alpha \oplus c \oplus \gamma)), \text{ for } Z = X \oplus c
\end{align*}
\]

Lemma 2.3.1 can be interpreted as follows:

- The RX difference of a rotation is the rotation of the RX difference.
- The RX difference of an XOR of pairs is the XOR of the RX differences.
- The RX difference of an XOR of a pair and a constant is the XOR of the RX difference, the constant, and the constant rotated by \(\gamma\).

In summary, the propagation of RX differences through rotations or XORs is deterministic and the resulting RX difference can be computed from the RX differences and the constant values.

The propagation of RX differences through modular addition is not deterministic, but its probability can be computed when the rotational offset \(\gamma\) is fixed to 1 [17].

Theorem 2.3.1 (Table 2.2 contains the notation used). Let \(X, Y\) and \(Z\) be \(n\)-bit words sampled uniformly at random over the \(n\)-bit strings \(\{0, 1\}^n\) and let \((X, \overrightarrow{X} \oplus \alpha),\)
2.4. Automatic Search for Rotational-XOR Characteristics

\( (Y, \overrightarrow{Y} \oplus \beta) \) and \( (Z, \overrightarrow{Z} \oplus \zeta) \) be three pairs with RX differences \( \alpha \), \( \beta \) and \( \zeta \) respectively and with rotational offset \( \gamma = 1 \). Then,

\[
P[(X \oplus Y, (X \oplus \alpha) \oplus (\overrightarrow{Y} \oplus \beta)) = (Z, \overrightarrow{Z} \oplus \zeta)]
\]

\[
= 1 \cdot 2^{\left|SHL(\delta_{\alpha} \oplus \delta_{\beta} \oplus \delta_{\zeta})\right|} \times 2^{-3}
\]

where \( \delta_{\alpha} = L(\alpha), \delta_{\beta} = L(\beta) \) and \( \delta_{\zeta} = L(\zeta) \).

| SAT(X) | \( \text{A non-cyclic left shift of } X \text{ by one bit} \) |
| \( (I \oplus \text{SHEL})(X) \) | \( X \oplus \text{SHEL}(X) \) |
| \( x \leq y \) | \( \text{A characteristic function which evaluates to 1 if } X_i \leq Y_i \text{ for all } i \) |
| \( X \mid Y \) | \( \text{The vector bitwise OR operation between } X \text{ and } Y \) |
| \( |X| \) | \( \text{The hamming weight of } X \) |
| \( L(X) \) | \( \text{The } n-1 \text{ most significant bits of } X \) |

Table 2.2: Notation used in theorem 2.3.1.

In the case that \( \alpha = \beta = \zeta = 0 \), theorem 2.3.1 predicts the probability that the rotational property is preserved after a modular addition \( P(\overrightarrow{X} \oplus \overrightarrow{Y} = \overrightarrow{X} \oplus \overrightarrow{Y}) \) (as Equation (2.1) for \( \gamma = 1 \)).

2.4 Automatic Search for Rotational-XOR Characteristics

Characteristics can be used to estimate the success probability of differential and rotational-XOR cryptanalysis. The higher the probability of the characteristic, the higher the success probability of the distinguishing attack (and the lower the complexity of the corresponding key recovery attack).

Different automatic tools have been proposed to search for characteristics of ARX systems [12, 13, 37]. The technique proposed by Mouha et al. [12] has the advantage that it finds the optimal characteristics. In [12], it was applied to search for differential characteristics of the ARX stream cipher Salsa20 [5]. An improvement of this technique was considered in [14] to search for differential characteristics of Speck and LEA. Recently, it was applied to search for RX characteristics of SPECK [18].

Basically, this technique consists of rewriting the search problem as a SAT (Boolean Satisfiability) problem. A similar approach was taken in [37], but the search problem was formulated as a Mixed-Integer Linear Program instead.

The rest of this section focuses on the SAT approach. First, the SAT problem is introduced and then a SAT-based method for finding optimal rotational-XOR characteristics is explained.
2.4. Automatic Search for Rotational-XOR Characteristics

2.4.1 The boolean satisfiability problem

A boolean formula is an expression which consists of boolean variables, which can take the values TRUE or FALSE, and the logic operators AND, OR and NOT. A boolean formula is satisfiable if there exists an assignment of the variables that makes the formula TRUE. For example the boolean formula $a \text{ AND } (\text{NOT } b)$ is satisfiable since the assignment $(a, b) = (\text{TRUE}, \text{FALSE})$ evaluates the entire formula to TRUE.

The boolean satisfiability (SAT) problem is the problem of determining whether a boolean formula is satisfiable. In general, the SAT problem is NP-complete [38], which implies that no known algorithm solves SAT in polynomial time (with respect to the number of variables). In other words, solving a SAT problem is infeasible if the number of variables is high enough. In practice, SAT solvers can handle instances with thousands (and sometimes even millions) of variables [39].

A generalization of the SAT problem is the satisfiability modulo theories (SMT) problem. Basically, SMT formulas can be expressed with richer languages (theories) than boolean formulas. In particular, a formula in the bit-vector theory can contain bit-vectors (a vector of boolean variables) and the usual operations of bit-vectors such as bitwise operations (XOR, OR, AND, etc) arithmetic operations (addition, multiplication, etc), cyclic operations and so on. A common approach in SMT solvers [40, 41] is to translate the SMT instance to a SAT instance and solve using a SAT solver.

2.4.2 A SAT-based method for searching optimal rotational-XOR characteristics

In [18], a SAT solver was used for finding optimal characteristics of the ARX block cipher SPECK. In this section, the method used in [18] will be explained in detail. First, a similar notation to [12] is introduced.

Definitions and notation

A triplet of RX differences $(\alpha, \beta, \zeta)$ is valid if $\alpha$ and $\beta$ propagate to $\zeta$ after a modular addition with non-zero probability, that is, $P(\alpha, \beta \xrightarrow{\text{mod}} \zeta) \neq 0$. Using theorem 2.3.1, the triplet $(\alpha, \beta, \zeta)$ is valid if and only if one of the following conditions holds

\[(I \oplus \text{SHL})(\delta\alpha \oplus \delta\beta \oplus \delta\zeta) \oplus 1 \leq \text{SHL}(\langle \delta\alpha \oplus \delta\zeta \rangle|\langle \delta\beta \oplus \delta\zeta \rangle)\]
\[(I \oplus \text{SHL})(\delta\alpha \oplus \delta\beta \oplus \delta\zeta) \leq \text{SHL}(\langle \delta\alpha \oplus \delta\zeta \rangle|\langle \delta\beta \oplus \delta\zeta \rangle).\]

The weight $\omega$ of a valid triplet $(\alpha, \beta, \zeta)$ is defined as

\[\omega(\alpha, \beta, \zeta) = -\log_2(P(\alpha, \beta \xrightarrow{\text{mod}} \zeta)).\]

Using Theorem 2.3.1, the weight can be calculated as follows

\[\omega(\alpha, \beta, \zeta) = \begin{cases} |\text{SHL}(\langle \delta\alpha \oplus \delta\zeta \rangle|\langle \delta\beta \oplus \delta\zeta \rangle)| + 3, & \text{if (2.2) holds} \\ |\text{SHL}(\langle \delta\alpha \oplus \delta\zeta \rangle|\langle \delta\beta \oplus \delta\zeta \rangle)| + 1.415, & \text{if (2.3) holds} \end{cases}\]
2.4. Automatic Search for Rotational-XOR Characteristics

The weight $W$ of an RX characteristic is defined as the sum of the weights of each of its modular additions. As in [12], it is assumed that the probability of a characteristic is the multiplication of the probabilities of each modular addition. In this case, the probability of an RX characteristic $p$ can be calculated as $p = 2^{-W}$.

The search

The main idea of this technique is to use a SAT solver to determine whether there exists an RX characteristic up to a certain weight $W$. If the SAT solver obtains that such problem is satisfiable, a higher weight is chosen. Otherwise, a lower weight is chosen. This is repeated until the minimal weight is found, that is, a weight $\hat{W}$ such that there exists an RX characteristic up to weight $\hat{W}$ but not up to weight $\hat{W} - \epsilon$, where $\epsilon$ is a value fixed from the start (usually $\epsilon = 1$ [12, 18]).

Different strategies have been used to obtain the minimal weight. In [12], the search starts with a low weight and it is incremented by one at a time until a satisfiable formula is found. In [42], the search starts with a high weight and it is decremented by one at a time until an unsatisfiable formula is found. In [18], a binary search strategy is used to find the minimal weight.

SAT solvers not only determine whether a formula is satisfiable or unsatisfiable, but also obtain an assignment that makes the formula TRUE if it is satisfiable. Therefore, the result of this method is a minimal weight $\hat{W}$, an RX characteristic with probability in the interval $(\hat{W} - \epsilon, \hat{W}]$ and the knowledge that there is no RX characteristic with probability lower that $\hat{W} - \epsilon$.

Since the RX differences of the round keys are necessary to propagate the RX differences through the encryption function, a pair of characteristics is actually considered: one for the key-schedule and one for the encryption, and define the encryption characteristic in terms on the key-schedule characteristic. The weight related to the key-schedule characteristic will be denoted by $W_K$ and the weight related to the encryption characteristic will be denoted by $W_E$. This modification was done in [18], where first $W_E$ was minimized with $W_K$ fixed to a high value and then $W_K$ was minimized with $W_E$ fixed to $\hat{W}_E$.

The rest of this section explains how to write the decision problem of whether there exists a pair of RX characteristic up to a certain pair of weights $(W_K, W_E)$ as an SMT problem.

The formulation of the SMT problem

The operations of an ARX cipher are performed on $n$-bit vectors, whereas the formula of a SAT problem can only contains boolean variables and the operations AND, NOT and OR. Therefore, an SMT problem in the bit-vector theory, which supports bit-vectors variables and the usual operations of bit-vectors, is used instead. Once the SMT problem is written, an SMT solver translates it into a SAT problem and solves it using a SAT solver. An example of an SMT solver is STP [40]. STP is built from a SAT solver and was used in [12, 18].

The SMT problem is written as follows:
• For every pair of $n$-bit inputs words of the key schedule and the encryption, an $n$-bit vector is used to represent the RX difference of such pair.

• Additional $n$-bit vectors are used to represent the RX difference after the addition, XOR and rotation operation when required.

• For every XOR and every bit rotation of the ARX block cipher, the equations of Lemma 2.3.1 are used to propagate properly the RX differences.

• For every modular addition of the ARX block cipher, Equations (2.2) and (2.3) are used to ensure that the RX differences are propagated with non-zero probability and Equation (2.5) is used to calculate the weight of the modular addition.

• Finally, two constraints are used to ensure that the sum of the weights related to the modular additions of the key schedule (encryption) is at most $W_K (W_E)$.

2.5 Conclusion

In this chapter, the cryptographic background related to this objective was explained. The symmetric cryptosystems, a cryptographic scheme to achieve confidentiality, were introduced. Then, a particular type of symmetric cryptosystems, the block ciphers, were studied. After showing some aspects of the design of modern block ciphers, ARX block ciphers were introduced and their properties were explained.

Then, the basic concepts of the cryptanalysis of block ciphers were defined and three cryptanalytic techniques were considered: differential cryptanalysis, rotational cryptanalysis and rotational-XOR cryptanalysis. Each one of them studies a property of the plaintext pairs which is preserved through the encryption function with a probability higher than a random permutation. These properties were explained along with the estimation of their probabilities. On top of that, the propagation of these properties through each ARX operation was analysed for rotational and rotational-XOR cryptanalysis.

After that, the SAT problem and the SMT problem, a generalization of the SAT problem, was introduced. Then, an SMT-based method to find optimal RX characteristics were described. In particular, the formulation of the search of RX characteristics as a SMT problem was explained in details.

The next chapter will explain the thesis contribution, a tool to speed up and facilitate this process by generating automatically the SMT problem from a software implementation of an ARX block cipher.
Chapter 3

The ArxPy Tool

This chapter introduces the contribution of the thesis: the ArxPy tool. First, the purpose of developing ArxPy is explained and then its usage and implementation are described.

3.1 Functionality and Features

ArxPy is a tool for finding optimal rotational-XOR characteristics of ARX block ciphers. Basically, ArxPy takes a Python implementation of an ARX block cipher as input and applies the SAT-based method described in Section 2.4.2. ArxPy is a generalization of Ashur, De Witte and Liu’s tool [18] for finding optimal RX characteristics in Speck.

Most of the automated tools [18, 20, 21, 22] searching for characteristics with high probability are implemented specifically for a particular cipher or for a small set. In order to use such tools with an arbitrary cipher, a significant effort is required.

Given a Python implementation of an ARX block cipher, ArxPy is executed with a simple shell command. Therefore, the only effort to use ArxPy is implementing the ARX block cipher in Python, which is negligible due to the low development time and code complexity of the Python programming language.

On top of that, ArxPy is open source and has a modular architecture. Therefore, it can be easily adapted for specific needs. The source code of ArxPy can be found in the appendix.

3.2 Usage

ArxPy expects a certain structure in the Python implementation of an ARX block cipher. The first part of this section will explain this structure. Then, the shell command to run ArxPy and the format of the output will be described.
3.2. Usage

3.2.1 Structure of Python implementations of ARX block ciphers

A Python implementation of an ARX block cipher following the structure required by ArxPy will be called an ARX implementation. Any iterated ARX block cipher can be considered, as long as all its operations are performed on words of the same size.

A minimal ARX implementation contains a global variable, \texttt{wordsize}, and two functions, \texttt{key\_schedule} and \texttt{encryption}. The global variable \texttt{wordsize} contains the word size (in bits) of the ARX block cipher.

The function \texttt{key\_schedule} implements the key scheduling algorithm of the ARX block cipher. This function has \( m \) arguments representing the \( m \) words of the key. It has no return value; the round keys are stored into the list-like object \texttt{round\_keys}.

The function \texttt{encryption} implements the encryption algorithm of the ARX block cipher. The arguments of this function represent the words of the plaintext. The output of each round is stored in the list-like object \texttt{rounds}, except the last one, that is used as the return value. This function can obtain the round keys from the list-like object \texttt{round\_keys}.

In order to implement the encryption and the key schedule, the Python operators \texttt{+}, \texttt{>>}, \texttt{<<} and \texttt{^} are used as the modular addition, right and left rotation and XOR, respectively. Augmented assignments, such as \texttt{+=} or \texttt{^=}, are not allowed.

Apart from the variable \texttt{wordsize} and the functions \texttt{key\_schedule} and \texttt{encryption}, additional variables and functions can be defined to improve the readability and the modularity of the implementation. Listing 1 contains an ARX implementation of SPECK with 32-bit block size (SPECK32). ARX implementations of LEA, Chaskey and the SPECK family can be found in the appendix.

The special objects \texttt{rounds} and \texttt{round\_keys}

There are several considerations about the list-like objects \texttt{rounds} and \texttt{round\_keys}:

- They are not created or declared in the ARX implementation. They are created by the parser of ArxPy.

- They only support the operator \texttt{[]} to access their elements (slices and negative indices are not supported).

- They can store either single values or lists of values, but each position can only be assigned once. Furthermore, storing a list of values must be done with one assignment and not element-wise (see Listing 2).

- Their elements should always be used over other variables. Listing 3 shows two possibilities to implement the encryption algorithm of SPECK32. In the first option, the elements of \texttt{rounds} are passed to the round function, whereas the variables \texttt{x} and \texttt{y} are passed to the round function in the second option. Although both options are equivalent, ArxPy will process the first implementation much faster than the second one.
3.2. Usage

```python
wordszie = 16
number_of_rounds = 22
alpha = 7
beta = 2

# round function
def f(x, y, k):
x = ((x >> alpha) + y) ^ k
y = (y << beta) ^ x
return x, y

def key_schedule(l2, l1, l0, k0):
l = [None for i in range(number_of_rounds + 3)]
round_keys[0] = k0
l[0:3] = [l0, l1, l2]
for i in range(number_of_rounds - 1):
l[i+3], round_keys[i+1] = f(l[i], round_keys[i], i)

def encryption(x0, y0):
rounds[0] = f(x0, y0, round_keys[0])
for i in range(1, number_of_rounds):
x, y = rounds[i - 1]
round_output = f(x, y, round_keys[i])
if i < number_of_rounds - 1:
rounds[i] = round_output
else:
return round_output
```

Listing 1: An ARX implementation of Speck32.

Adding test vectors and fixing RX differences

Apart from the functions encryption and key_schedule, an ARX implementation contains two more special functions: the function test and the function fix_differences.

Test vectors can be added to an ARX implementation by using the Python statement assert inside the function test. Listing 4 shows an example of this function for Speck32.

On the other hand, it is possible to fix the RX differences of the round keys and
3.2. Usage

```
rounds[i] = [value1, value2]
rounds[i][0] = value1
rounds[i][1] = value2
```

Listing 2: Correct way (left) and wrong way (right) to store a list of values.

```python
# option 1
def encryption(x0, y0):
    rounds[0] = f(x0, y0, round_keys[0])

    for i in range(1, number_of_rounds):
        x, y = rounds[i - 1]
        x, y = f(x, y, round_keys[i])

        if i < number_of_rounds - 1:
            rounds[i] = x, y
        else:
            return x, y

# option 2
def encryption(x0, y0):
    rounds[0] = f(x0, y0, round_keys[0])

    x, y = rounds[0]
    for i in range(1, number_of_rounds):
        x, y = f(x, y, round_keys[i])

        if i < number_of_rounds - 1:
            rounds[i] = x, y
        else:
            return x, y
```

Listing 3: Two possible options to implement the encryption function of Speck32. The first one is processed by ArxPy much faster than the second.

the outputs of each round by using the method fix_difference of round_keys and rounds inside the function fix_differences. To fix the RX difference of the i-round key to a value v, round_keys.fix_difference(i, v) is added to the function fix_differences (and similar for rounds). Listing 5 shows an example of this function for Speck32, where the RX differences of the round keys are fixed to 0.
def test():
    key = (0x1918, 0x1110, 0x0908, 0x0100)
    plaintext = (0x6574, 0x694c)
    ciphertext = (0xa868, 0x42f2)

    key_schedule(*key)
    assert ciphertext == encryption(*plaintext)

Listing 4: A test function with a test vector of Speck32.

def fix_differences():
    for i in range(number_of_rounds):
        round_keys.fix_difference(i, 0)

Listing 5: A fix_differences function for Speck32.

3.2.2 Running the program

ArxPy uses the Python library SymPy [43] and the SMT solver STP. They, together with Python3, must be installed in order to execute ArxPy.

The shell command to run ArxPy is the following:

    python3 arxpy.py <ARX_implementation> <output>

where <ARX_implementation> is the name of the file containing an ARX implementation and <output> is the name of the file where the output will be written.

3.2.3 The output

To find the optimal RX characteristic, ArxPy searches for characteristics up to many weights. During an execution of ArxPy, these intermediate characteristics are written in the output file. After the execution finishes, the last characteristic in the output file is the optimal one.

ArxPy actually searches for a pair of characteristics: the encryption characteristic, which contains the RX differences of the output of each round, and the key schedule characteristic, which contains the RX differences of the round keys. When a pair of characteristics is found by ArxPy, the following information is printed into the output file:

- The probabilities of the encryption characteristic and the key schedule characteristic.
- The upper bounds on these probabilities used to limit the search.
3.2. Usage

- The RX differences of the plaintext (\(pj\) for the \(j\)-th plaintext word) and the RX differences of the (master) key (\(mkj\) for the \(j\)-th key word).

- The RX differences of the output of each round (\(xi_{j}\) for the \(j\)-th word of the round \(i\)) and the round keys (\(ki_{j}\) for the \(j\)-th word of the \(i\)-th round key).

- The weights of the modular additions corresponding to the RX differences of the output of each round (\(wxi_{j}\)) and the round keys (\(wki_{j}\)).

- A pair of counters (\(counter1\), \(counter2\)) specifying the number of modular additions of the encryption whose RX difference was propagated according to Equation (2.2) and Equation (2.3) respectively. Similar for the pair (\(counterk1\), \(counterk2\)) and the modular additions of the key schedule.

- The total time of execution.

An example of a pair of characteristics found by ArxPy is the following optimal characteristic of Speck32 with 5 rounds obtained by ArxPy:

**Encryption characteristic:**

<table>
<thead>
<tr>
<th>Probability</th>
<th>(-log2): 7.075</th>
</tr>
</thead>
<tbody>
<tr>
<td>Upper bound on the probability</td>
<td>(-log2): 7.075</td>
</tr>
</tbody>
</table>

**RX differences:**

- \(p0\): 0x0000
- \(p1\): 0x0000
- \(x0_0\): 0x0000
- \(x0_1\): 0x0000
- \(x1_0\): 0x0000
- \(x1_1\): 0x0000
- \(x2_0\): 0x0000
- \(x2_1\): 0x0000
- \(x3_0\): 0x0000
- \(x3_1\): 0x0000
- \(x4_0\): 0x0000
- \(x4_1\): 0x0005

**Weights:**

- \(wx0_0\): 0x0000
- \(wx0_1\): 0x0000
- \(wx1_0\): 0x0000
- \(wx1_1\): 0x0000
- \(wx2_0\): 0x0000
- \(wx2_1\): 0x0000
- \(wx3_0\): 0x0000
- \(wx3_1\): 0x0000
- \(wx4_0\): 0x0000
- \(wx4_1\): 0x0000

**Counters:**

- Counter1: 25
3.3 Implementation

ARXPy have been implemented in three modules: the ARX block cipher parser, the SMT writer and the RX characteristic finder. The parser module has been written from scratch and the other two modules are based on the tool proposed in [18]. This section explains these three modules, and the parser module is described in more detail.

3.3.1 The parser module

The ARX block cipher parser module takes an ARX implementation and generates the symbolic expressions of the output values of each round and the round keys.

Importing the ARX implementation

Before importing the ARX implementation, its source code is modified dynamically. This is done by generating the Abstract Syntax Tree (AST) of the ARX implement-
3.3. Implementation

Implementation, modifying the AST to change the behaviour of the Python operators and finally translating the AST into Python code.

Python AST module is used to modify the AST in the following way. The AST of the ARX implementation is traversed and when a binary operator node is found, its binary operation (+, >>, <<, or ^) is replaced by the modular addition (implemented in the function ModAdd), the left rotation (Rol), the right rotation (Ror) or the exclusive-or (XOR) respectively.

Listing 6 shows the round function of the ARX implementation of Listing 1 before and after being parsed.

```python
def f(x, y, k):
    x = ((x >> alpha) + y) ^ k
    y = (y << beta) ^ x
    return x, y

# after being parsed
def f(x, y, k):
    x = Xor(ModAdd(Ror(x, alpha), y), k)
    y = Xor(Rol(y, beta), x)
    return x, y
```

Listing 6: The round function of Listing 1 before and after being parsed.

Generating the symbolic expressions

Once the modified AST has been compiled and imported, the functions key_schedule and encryption are symbolically executed and the return value of encryption is stored in rounds. After the execution, rounds and round_keys do not contain specific values but the symbolic expressions to compute such values.

SymPy, a Python library for symbolic mathematics, is used to generate and handle the symbolic expressions. Basically, Sympy provides symbolic variables (sympy.Symbol), symbolic functions (sympy.Function) and symbolic expressions (sympy.Expr) and these three objects are used to symbolically execute encryption and key_schedule. Listing 7 shows an example of a symbolic execution with Sympy.

Optimization of the symbolic expressions

The size of the symbolic expressions contained in rounds and round_keys grows exponentially with respect to the number of operations in the ARX block cipher. This increases significantly the running time of the symbolic execution. In order to solve this problem, a Python class, SymbolicArray, has been implemented to handle sequences of Sympy expressions efficiently. The variables rounds and round_keys are instances of SymbolicArray.
3.3. Implementation

def f(x, y, k):
    x0 = Ror(x, alpha)
    x1 = ModAdd(x0, y)
    x2 = Xor(x1, k)
    y0 = Rol(y, beta)
    y1 = Xor(y0, x2)
    return x2, y1

x, y = f(sympy.Symbol("a"), sympy.Symbol("b"), sympy.Symbol("c"))
print(x)
> Xor(ModAdd(Ror(a, alpha), b), c)
print(y)
> Xor(Rol(b, beta), Xor(ModAdd(Ror(a, alpha), b), c))

Listing 7: A symbolic execution of the round function of Speck32.

Before an expression $E$ is stored in a SymbolicArray, it is simplified using the expressions already stored. If any stored expression $E'$ is contained in $E$, the matching subexpression of $E$ is replaced by the identifier of $E'$. Listing 8 shows an example of this automatic simplification. In this example, the expression $E = \text{Rol}(\text{XOR}(a, b), 1)$ is going to be stored in $\text{rounds}[1]$. This expression contains the expression $E' = \text{XOR}(a, b)$ stored in $\text{rounds}[0]$. Therefore, $\text{rounds}[1]$ stores the expression $E$ where the subexpression $E'$ has been replaced by its identifier $x0$.

rounds[0] = Xor(sympy.Symbol("a"), sympy.Symbol("b"))
rounds[1] = Rol(rounds[0], 1)
print(rounds[1])
>>> Rol(x0, 1)

Listing 8: An example of the automatic simplification of rounds.

This automatic simplification reduces vastly the size of the symbolic expressions of rounds and rounds_keys. For this reason, it is recommended to use the elements of rounds and round_keys instead of other variables.

Printing the symbolic expressions of an ARX implementation

ArxPy has an option to print the symbolic expressions of an ARX implementation with the following shell command:

```
python3 parser.py <ARX_implementation>
```

where <ARX_implementation> is the name of the file containing an ARX implementation.
3.3. Implementation

For example, ArxPy generates the following symbolic expressions for the ARX implementation of Listing 1, where the number of rounds has been reduced to 5:

Key words:
- mk0, mk1, mk2, mk3

Round keys:
- \( k_{0,0} = mk3 \)
- \( k_{1,0} = \text{Xor}(\text{Rol}(k_{0,0}, 2), \text{Xor}(\text{ModAdd}(\text{Ror}(mk2, 7), k_{0,0}), 0)) \)
- \( k_{2,0} = \text{Xor}(\text{Rol}(k_{1,0}, 2), \text{Xor}(\text{ModAdd}(\text{Ror}(mk1, 7), k_{1,0}), 1)) \)
- \( k_{3,0} = \text{Xor}(\text{Rol}(k_{2,0}, 2), \text{Xor}(\text{ModAdd}(\text{Ror}(mk0, 7), k_{2,0}), 2)) \)
- \( k_{4,0} = \text{Xor}(\text{Rol}(k_{3,0}, 2), \text{Xor}(\text{ModAdd}(\text{Ror}(\text{Xor}(\text{ModAdd}(\text{Ror}(mk2, 7), k_{0,0}), 0), 7), k_{3,0}), 3)) \)

Plaintext words:
- p0, p1

Outputs of each round:
- \( x_{0,0} = \text{Xor}(\text{ModAdd}(\text{Ror}(p0, 7), p1), k_{0,0}) \)
- \( x_{0,1} = \text{Xor}(\text{Rol}(p1, 2), x_{0,0}) \)
- \( x_{1,0} = \text{Xor}(\text{ModAdd}(\text{Ror}(x_{0,0}, 7), x_{0,1}), k_{1,0}) \)
- \( x_{1,1} = \text{Xor}(\text{Rol}(x_{0,1}, 2), x_{1,0}) \)
- \( x_{2,0} = \text{Xor}(\text{ModAdd}(\text{Ror}(x_{1,0}, 7), x_{1,1}), k_{2,0}) \)
- \( x_{2,1} = \text{Xor}(\text{Rol}(x_{1,1}, 2), x_{2,0}) \)
- \( x_{3,0} = \text{Xor}(\text{ModAdd}(\text{Ror}(x_{2,0}, 7), x_{2,1}), k_{3,0}) \)
- \( x_{3,1} = \text{Xor}(\text{Rol}(x_{2,1}, 2), x_{3,0}) \)
- \( x_{4,0} = \text{Xor}(\text{ModAdd}(\text{Ror}(x_{3,0}, 7), x_{3,1}), k_{4,0}) \)
- \( x_{4,1} = \text{Xor}(\text{Rol}(x_{3,1}, 2), x_{4,0}) \)

3.3.2 The writer module

The SMT writer module takes the symbolic expressions generated by the parser module and writes the SMT problem. The SMT problem is written in the SMT-LIB v2 [44] language. This input format is supported by many SMT solvers such as STP, the SMT solver used by ArxPy.

This is done by extracting the sequence of ARX operations of the encryption and the key schedule and translating these operations to equations according to the steps described in Section 2.4.2. The sequence of operations is obtained from the symbolic expressions by traversing them as trees and extracting their nodes with the methods provided by Sympy. Figure 3.1 shows an example of an expression tree of a symbolic expression.

ArxPy has an option to write a test SMT problem (with fixed upper bounds) of an ARX implementation with the following shell command:

```
python3 smt_writer.py <ARX_implementation> <output>
```
3.3. Implementation

![Expression tree of Xor(Ror(a, 7), c).](image)

where `<ARX_implementation>` is the name of the file containing an ARX implementation and `<output>` is the name of the file where the SMT problem will be written.

### 3.3.3 Characteristic finder module

The third module implements the search strategy to find the optimal RX characteristic. A binary search strategy, based on [18], is used to minimize the following upper bounds.

For each SMT problem, four upper bounds are considered: the upper bounds on the probabilities of the encryption characteristic and the key schedule characteristic ($W_E$ and $W_K$ according to the notation of Section 2.4.2) and the upper bounds on $\text{counter}_1$ and $\text{counter}_k$.

The value of $\text{counter}_1$ represents the number of modular additions of the encryption whose RX difference was propagated according to Equation (2.2). On the other hand, $\text{counter}_k$ represents the number of modular additions of the key schedule whose RX difference was propagated according to Equation (2.3). The weight related to a modular addition (see Equation (2.5)) has a factor of 3 if Equation (2.3) holds and a factor of 1.415 if Equation (2.3) holds. Therefore, upper bounds are considered on $\text{counter}_1$ and $\text{counter}_k$ to find RX characteristic with more modular additions of the second type.

These four upper bounds are minimized one-by-one in the following way. First, all upper bounds except $W_E$ are fixed to a high value and $W_E$ is minimized. Then, $W_E$ is fixed to such minimum $\hat{W}_E$ and $W_K$ is minimized. The same process is repeated for the upper bounds on the counters.

At the end of this process, a pair of RX characteristic (the encryption one and the key schedule one), their probabilities and their upper bounds $W_E$ and $W_K$ are returned. This search strategy guarantees that the encryption characteristic
is optimal, that is, no encryption characteristic exists with probability lower than $W_E - \epsilon$ for $\epsilon \approx 1$.

The search strategy is implemented in the class `CharacteristicFinder`. In order to consider another search strategy, only this class needs to be modified. Therefore, different search strategies can be considered by making few modifications in ArxPy’s code.

### 3.4 Conclusion

In this chapter, the ArxPy tool was introduced. First, the purpose of developing ArxPy was discussed, along with its features. Then, the format of ArxPy’s input, a Python implementation of an ARX block cipher, was explained with several examples. Afterwards, it was described the shell command to execute ArxPy and the format of the output.

In the last section, the implementation of ArxPy was explained. It was described the generation of the symbolic expressions of the ARX block cipher, the writing of the SMT problem and the search strategy to find an optimal RX characteristic.
Chapter 4

Conclusion

Recently, several ARX block ciphers have been proposed, along with cryptanalytic techniques to analyse their security. The objective of this thesis was to provide a tool to speed up and make easier the security evaluation of these type of ciphers.

This tool, ArxPy, automatizes the search for optimal RX characteristics in ARX block ciphers. Given a Python implementation of an ARX block cipher, ArxPy finds an optimal RX characteristic with a simple shell command. Therefore, ArxPy can be used with any ARX block cipher with minimal effort.

On top of that, ArxPy can be easily adapted thanks to its modular architecture. For example, another SMT solver which supports the SMT-LIB v2 language can be used instead of STP just by modifying a few lines of code. As another example, a different search strategy can be considered by modifying only the characteristic finder module.

The generation of the SMT problems from an ARX implementation is done in a few seconds. However, a high-end computer is necessary to solve the SMT problems in a reasonable time. In our experiments with a standard PC, STP was only able to solve in reasonable time the SMT problems where the number of rounds was reduced significantly. For this reason, and due to the time constrains, we did not focus on evaluating a particular ARX block cipher with ArxPy.

4.1 Future Work

There are several ideas to continue this work. One option could be to adapt ArxPy for other types of cryptanalysis. For example, ArxPy could be combined with CryptoSMT [20], a SAT-based Python tool which applies different cryptanalytic techniques but only to some number of ciphers. The combined tool could evaluate an arbitrary ARX block cipher against several techniques with a simple shell command.

Another option could be to improve ArxPy by finding all the characteristics that share the input RX difference and the output RX difference of the optimal characteristic. This can be done with a SAT solver, as shown in [42], and the sum of the probabilities of all these characteristics provides a better estimation on the success probability.
Futhermore, ArxPy could be extended to accept software implementations in other languages. Since many authors of ARX block ciphers [6, 9, 8] draw a diagram to illustrate their ciphers, another possibility would be to use a data flow diagram as input. For example, PYTHONECT [45], a tool to translate data flow diagrams to python code and vice versa, could be used to this purpose.

ArxPy searches for RX characteristics where the rotational offset is fixed to 1 and the constants are injected through XOR. Therefore, future research could extend rotational-XOR cryptanalysis to cases where the rotational offset differs from 1, and to cases where the constants are injected through modular addition.
Appendices
Appendix A

Source code of ArxPy

ArxPy is implemented in four Python files: utilities.py, parser.py, smt_writer.py, and arxpy.py. The file utilities.py contains some auxiliary functions and the files parser.py, smt_writer.py, and arxpy.py contain the parser module, the writer module and the characteristic finder module respectively.

utilities.py

```python
import inspect

def to_list(value):
    try:
        l = list(value)
    except TypeError:
        l = [value]
    return l

def to_tuple(value):
    try:
        l = tuple(value)
    except TypeError:
        l = (value,)
    return l

def number_of_arguments(function):
    sig = inspect.signature(function)
    params = sig.parameters
    return len(params)

def flatten(l):
    return [item for sublist in l for item in sublist]
```
import ast
import copy
import itertools
import sys
import sympy
from sympy.core.sympify import sympify, SympifyError
from utilities import to_list, to_tuple, number_of_arguments, flatten

# Global variables
output_round_symbol = "x"
round_key_symbol = "k"
master_key_symbol = "mk"
plaintext_symbol = "p"
MAX_SIZE = 128  # maximum size for SymbolicArray

class ModAdd(sympy.Function):
    modulus = None
    
    @classmethod
def eval(cls, x, y):
        if x.is_Integer and y.is_Integer:
            return sympy.Integer((int(x) + int(y)) & int(cls.modulus - 1))

class Ror(sympy.Function):
    wordsize = None
    
    @classmethod
def eval(cls, value, rotation):
        def ror(v, r, size):
            mod_mask = 2**size - 1
            return ((v << (size - r)) + (v >> r)) & mod_mask

        if value.is_Integer and rotation.is_Integer:
            return sympy.Integer(ror(int(value), int(rotation), int(cls.wordsize)))

    
    @classmethod
def eval(cls, value, rotation):
        def rol(v, r, size):
            mod_mask = 2**size - 1
            return ((v >> (size - r)) + (v << r)) & mod_mask

        if value.is_Integer and rotation.is_Integer:
            return sympy.Integer(rol(int(value), int(rotation), int(cls.wordsize)))
class Xor(sympy.Function):
    @classmethod
    def eval(cls, x, y):
        if x.is_Integer and y.is_Integer:
            return sympy.Integer(int(x) ^ int(y))

def ARX_operations():
    return [ModAdd, Ror, Rol, Xor]

def parse_ast(source, filename):
    class NodeTransformer(ast.NodeTransformer):
        valid_binop = {
            ast.Add: 'ModAdd',
            ast.LShift: 'Rol',
            ast.RShift: 'Ror',
            ast.BitXor: 'Xor',
            ast.Sub: 'ModAdd',
        }
        invalid_binop = {
            ast.Mult: '*',
            ast.MatMult: '@',
            ast.Div: '/',
            ast.Mod: '%',
            ast.Pow: '**',
            ast.BitOr: '|',
            ast.BitAnd: '&',
            ast.FloorDiv: '//',
        }
        def visit_BinOp(self, node):
            if node.op.__class__ in self.valid_binop:
                id_class = self.valid_binop[node.op.__class__]
                right = self.visit(node.right)
                left = self.visit(node.left)

                if isinstance(node.op, ast.Sub):
                    right = ast.UnaryOp(op=ast.USub(), operand=right)
                    new_node = ast.Call(
                        func=ast.Name(id=id_class, ctx=ast.Load()),
                        args=[left, right],
                        keywords=[],
                        starargs=None,
                        kwargs=None
                    )

                    return new_node
            else:
                sys.exit("Error: invalid operation ({}) found.".format(  
                    self.invalid_binop[node.op.__class__]))

        def visit_AugAssign(self, self, node):
sys.exit("Augmented assignments (+=, ^=, ..) are not supported.")

node = ast.parse(source, filename, mode='exec')
node = NodeTransformer().visit(node)
return ast.fix_missing_locations(node)

class SymbolicArray(object):
    max_size = MAX_SIZE

    def __init__(self, symbol_name):
        self.list = [None for i in range(SymbolicArray.max_size)]
        self.fixed_differences = {}
        self.symbol_name = symbol_name
        self.restricted_mode = True

    def __getitem__(self, key):
        if key < 0 or key >= SymbolicArray.max_size:
            raise IndexError("Maximum size of '{}' is {}", format("SymbolicArray", SymbolicArray.max_size))

        return self.list[key]

    def __iter__(self):
        if self.restricted_mode:
            return NotImplemented
        return self.list.__iter__()

    def __len__(self):
        if self.restricted_mode:
            return NotImplemented
        return self.list.__len__()

    def enable_list_methods(self):
        self.restricted_mode = False

    def __setitem__(self, key, value):
        if key < 0 or key >= SymbolicArray.max_size:
            raise IndexError("Maximum size of '{}' is {}", format("SymbolicArray", SymbolicArray.max_size))

        elif self.list[key] is not None:
            raise ValueError("Position {} was already assigned".format(key))

        elif None in self.list[:key]:
            p = self.list.index(None)
            raise ValueError("Trying to assign position {} before pos {}!".format(key, p))

        value = to_list(value)
        for i, v in enumerate(value):
            try:
                value[i] = sympy.sympify(v, strict=True, convert_xor=False)
            except SympifyError as e:
                raise SyntaxError("{} is not a valid ARX expr".format(type(v))) from None

        # simplification between words of the same round
        seq = list(itertools.product(range(len(value)), range(len(value))))
for j, i in reversed(seq):
    if i < j:
        symbol = sympy.Symbol('{}{}_{}/quotesingle.ts1'.format(self.symbol_name, key, i))
        value[j] = value[j].subs(value[i], symbol)

    # simplification between words of previous rounds
    for i, expressions in enumerate(self.list[:key]):
        expressions = to_list(expressions)
        for j, expr in enumerate(expressions):
            symbol = sympy.Symbol('{}{}_{}/quotesingle.ts1'.format(self.symbol_name, i, j))
            value = [v.subs(expr, symbol) for v in value]

    if len(value) == 1:
        value = value[0]
    else:
        value = tuple(value)

    self.list[key] = value

def shrink(self):
    try:
        pos = self.list.index(None)
    except ValueError:
        pos = len(self.list)

    for i, value in enumerate(self.list[pos:]):
        if value is not None:
            raise ValueError("Pos {} was assigned but {} wasn't!".format(pos - 1, i))

    self.list = self.list[:pos]

def tuplify(self):
    for i, value in enumerate(self.list):
        self.list[i] = to_tuple(value)

def symbols(self, return_list_of_lists=True):
    symbol_list = []
    for i, values in enumerate(self.list):
        v = to_list(values)
        s = self.symbol_name
        symbols = [sympy.Symbol('{}{}_{}/quotesingle.ts1'.format(s, i, j)) for j in range(len(v))]

    if not return_list_of_lists:
        if len(symbols) == 1:
            symbols = symbols[0]

        symbol_list.append(symbols)

    return symbol_list

def fix_difference(self, key, value):
    if key < 0 or key >= SymbolicArray.max_size:
        raise IndexError("Maximum size of " '{}/quotesingle.ts1' is {}".format("SymbolicArray", SymbolicArray.max_size))

39
value = to_list(value)
for i, v in enumerate(value):
    if isinstance(v, int):
        value[i] = sympy.Integer(v)

for j, v in enumerate(value):
    self.fixed_differences["{}{0}_{1}\].format(self.symbol_name, key, j)] = v

def number_of_additions(self):
    additions = set()
    expressions = flatten(self.list)

    for expr in expressions:
        for node in expr.atoms(ModAdd):
            additions.add(node)

    return len(additions)

def parse_ARX_cipher(filename):
    with open(filename, "r") as filehandler:
        source = filehandler.read()

    temp_vars = {}
    exec(source, temp_vars)

    wordsize = temp_vars["wordsize"]

    tree = parse_ast(source, filename)

    ModAdd.modulus = 2**(wordsize)
    Ror.wordsize = wordsize
    Ror.wordsize = wordsize
    module = {"ModAdd": ModAdd, "Xor": Xor, "Ror": Ror, "Ror": Rol}

    exec(compile(tree, filename=filename, mode="exec"), module)

    module["rounds"] = SymbolicArray(output_round_symbol)
    if "key_schedule" in module:
        module["round_keys"] = SymbolicArray(round_key_symbol)
    else:
        module["round_keys"] = None

    module["name"] = filename[:-3]  # without ".py"

    return module

def generate_symbolic_expressions(cipher):
    if cipher["round_keys"] is not None:
        n = number_of_arguments(cipher["key_schedule"])
        masterkey = [sympy.Symbol("{}\].format(master_key_symbol, i)) for i in range(n)]
        cipher["key_schedule"](+masterkey)
        cipher["round_keys"].enable_list_methods()
        cipher["masterkey"] = masterkey
        cipher["round_keys"].shrink()
original_round_keys = cipher["round_keys"]
cipher["round_keys"] = cipher["round_keys"].symbols(return_list_of_lists=False)
n = number_of_arguments(cipher["encryption"])
plaintext = [sympy.Symbol("{}{}".format(plaintext_symbol, i)) for i in range(n)]
output_rounds = cipher["encryption"](*plaintext)
cipher["rounds"].enable_list_methods()
cipher["plaintext"] = plaintext
pos = cipher["rounds"].list.index(None)
cipher["rounds"][pos] = output_rounds
cipher["rounds"].shrink()
cipher["rounds"].tuplify()

if cipher["round_keys"] is not None:
    cipher["round_keys"] = original_round_keys
cipher["round_keys"]= tuplify()

if "fix_differences" in cipher is not None:
    fix_differences(cipher)

def print_cipher(cipher):
    if cipher["round_keys"] is not None:
        print("Key words:")
        print(" \t" + ',
             '.join([str(id) for id in cipher["masterkey"]]))

        print("Round keys: ")
        for i, key in enumerate(cipher["round_keys"]):
            for j, k in enumerate(key):
                print(" \t{}{}{} = {}".format(cipher["round_keys"].symbol_name, i, j, k))

        print()

    print("Plaintext words:")
    print(" \t" + ',
             '.join([str(id) for id in cipher["plaintext"]]))

    print("Output of each round:")
    for i, rounds in enumerate(cipher["rounds"]):
        for j, d in enumerate(rounds):
            print(" \t{}{}{} = {}".format(cipher["rounds"].symbol_name, i, j, d))

def test(cipher):
    if cipher["round_keys"] is not None:
        original_round_keys = cipher["round_keys"]
cipher["round_keys"] = [None for i in range(len(original_round_keys))]

    original_rounds = cipher["rounds"]
cipher["rounds"] = [None for i in range(len(original_rounds))]

    try:
        cipher["test"]()
    except AssertionError as e:
        sys.exit("test(): FAILED.")
    else:
print("test(): OK.")

if cipher["round_keys"] is not None:
cipher["round_keys"] = original_round_keys
cipher["rounds"] = original_rounds

def fix_differences(cipher):
    cipher["fix_differences"]()

if __name__ == '__main__':
    filename = sys.argv[1]
cipher = parse_ARX_cipher(filename)
generate_symbolic_expressions(cipher)
print_cipher(cipher)
if "test" in cipher:
    print()
test(cipher)

class SMTwriter(object):
    def __init__(self, cipher, upper_bound_data, upper_bound_data_counter,
                 upper_bound_keys, upper_bound_key_counter):
        self.cipher = cipher
        self.upper_bound_data = upper_bound_data
        self.upper_bound_keys = upper_bound_keys
        self.upper_bound_data_counter = upper_bound_data_counter
        self.upper_bound_key_counter = upper_bound_key_counter

        self.smt_code = ""

        self.wordsize = self.cipher["wordsize"]
        self.key_counters = key_counters
        self.data_counter = data_counters
        self.countersize = None
def hex(self, number, bitsize):
    assert bitsize % 4 == 0
    return "#x{0:0{1}x}".format(int(number), bitsize // 4)

def multi_binop(self, args, binop="bvadd", bitsize=None):
    """multi_binop([a, b, c]) returns (bvadd a (bvadd b c))""
    if len(args) == 0:
        return self.hex(0, bitsize)
    elif len(args) == 1:
        return str(args[0])
    sum_pattern = "(" + binop + " {0} {1})""
    sum_code = sum_pattern
    for i in range(len(args) - 2):
        sum_code = sum_code.format(args[i], sum_pattern)
    sum_code = sum_code.format(args[len(args) - 2], args[len(args) - 1])
    return sum_code

def header(self):
    return textwrap.dedent(""
(set-info :smt-lib-version 2.0)
(set-logic QF_BV)
""

def hamming_weight(self):
    smt_code = "(define-fun bvhamw ((x (_ BitVec {}))) (_ BitVec {})).format(
        self.wordsize, self.countersize)
    sum_pattern = "(bvadd (concat (_ bv0 {})).format(self.countersize - 1)
        sum_pattern += " ((_ extract {} {} (x))) {})"
    hw_code = sum_pattern
    for i in range(self.wordsize - 3):
        hw_code = hw_code.format(i, i, sum_pattern)
    hw_code = hw_code.format(self.wordsize - 3, self.wordsize - 3,
        "(concat (_ bv0 {}) ((_ extract {} {} (x)))".format(self.countersize - 1,
            self.wordsize - 2, self.wordsize - 2))
    smt_code += hw_code
    smt_code += "")"
    return smt_code

def characteristic_functions(self):
    return textwrap.dedent(""
(define-fun f1 ((d1 (_ BitVec {0})) (d2 (_ BitVec {0}))) (_ BitVec {0}))
    (d3 (_ BitVec {0}))) (_ BitVec {0})
    (bvxor (bvxor d1 (bvxor d2 d3)) (concat ((_ extract {1} 0)
        (bvxor d1 (bvxor d2 d3))) (_ bv0 1))))
    (define-fun f2 ((d1 (_ BitVec {0})) (d2 (_ BitVec {0})))
        (d3 (_ BitVec {0}))) (_ BitVec {0})
    (concat ((_ extract {1} 0) (bvor (bvxor d1 d3) (bvxor d2 d3))) (_ bv0 1))))
    """.format(self.wordsize - 1, self.wordsize - 3))

def declare_key_variables(self):
    smt_code = ""
for counter in self.key_counters:
    smt_code += "(declare-fun {} () (_ BitVec {}))\n".format(counter, self.countersize)

for key in self.cipher["masterkey"]:
    smt_code += "(declare-fun {} () (_ BitVec {}))\n".format(key, self.wordsize)

key_symbols = flatten(self.cipher["round_keys"].symbols())
for i, symbol in enumerate(key_symbols):
    smt_code += "(declare-fun {} () (_ BitVec {}))\n".format(symbol, self.wordsize)
    if str(symbol) in self.cipher["round_keys"].fixed_differences:
        value = self.cipher["round_keys"].fixed_differences[str(symbol)]
        smt_code += "(assert (= {} {}))\n".format(symbol, self.hex(value, self.wordsize))

weight_symbol = weight_prefix + str(symbol)
    smt_code += "(declare-fun {} () (_ BitVec {}))\n".format(weight_symbol, self.countersize)

smt_code = smt_code[:-1]  # remove last "\n"
return smt_code

def declare_data_variables(self):
    smt_code = ""

    for counter in self.data_counters:
        smt_code += "(declare-fun {} () (_ BitVec {}))\n".format(counter, self.countersize)

    for data in self.cipher["plaintext"]:
        smt_code += "(declare-fun {} () (_ BitVec {}))\n".format(data, self.wordsize)

    data_symbols = flatten(self.cipher["rounds"].symbols())
    for i, symbol in enumerate(data_symbols):
        smt_code += "(declare-fun {} () (_ BitVec {}))\n".format(symbol, self.wordsize)
        if str(symbol) in self.cipher["rounds"].fixed_differences:
            value = self.cipher["rounds"].fixed_differences[str(symbol)]
            smt_code += "(assert (= {} {}))\n".format(symbol, self.hex(value, self.wordsize))

    weight_symbol = weight_prefix + str(symbol)
    smt_code += "(declare-fun {} () (_ BitVec {}))\n".format(weight_symbol, self.countersize)

    smt_code = smt_code[:-1]  # remove last "\n"
    return smt_code

def key_counter_constrains(self):
    return "(assert (bvule {} {}))".format(self.key_counters[0],
                                            self.hex(self.upper_bound_key_counter, self.countersize))

def data_counter_constrains(self):
    return "(assert (bvule {} {}))".format(self.data_counters[0],
                                            self.hex(self.upper_bound_data_counter, self.countersize))
def ARX_node_to_smt(self, node, symbol, output_var_name, add_counter):
    smt_code = ""
    counter_smt_code = None
    args = [node.args[0], node.args[1]]
    for a in args:
        if not a.is_Integer and not a.is_Symbol:
            raise ValueError("Invalid arguments: {}\n\n            if node.func in [parser.Ror, parser.Rol):
                if args[0].is_Integer:
                    args[0] = self.hex(args[0], self.wordsize)
                if node.func == parser.Ror:
                    op = "rotate_right"
                elif node.func == parser.Rol:
                    op = "rotate_left"
                smt_code += "(assert (= {} (_ {} {}) {}))\n            else:
                for i in range(len(args)):
                    if args[i].is_Integer:
                        args[i] = self.hex(args[i], self.wordsize)
                    if node.func == parser.Xor:
                        for i in range(len(args)):
                            if isinstance(args[i], str):
                                cte_rotated = "(_ rotate_left 1) {0})".format(args[i])
                                args[i] = "(bvxor {0} {1})".format(args[i], cte_rotated)
                        smt_code += "(assert (= {} (bvxor {} {})))\n                elif node.func == parser.ModAdd:
                    smt_code += textwrap.dedent("\n                        (assert (or
                            (= (bvxor (f1 (_ extract {1} 1) {2}) (_ extract {1} 1) {3})
                        (bvand (bvxor (f1 (_ extract {1} 1) {2}) (_ extract {1} 1) {3})
                            (f2 (_ extract {1} 1) {3})
                            (_ extract {1} 1) {0}))
                            (_ extract {1} 1) {0}))
                            (f2 (_ extract {1} 1) {3})
                            (_ extract {1} 1) {0}))
                            """.format(output_var_name, self.wordsize - 1, args[0], args[1]))
        weight_symbol = "{}{}_{\}").format(weight_prefix, symbol, add_counter)
        weight_smt_code = textwrap.dedent("\n            (declare-fun {} () (_ BitVec {1}))
            (assert (= {} (bvhamw (concat
                (f2 (_ extract {2} 1) {3}) (_ extract {2} 1) {4})
                (_ extract {2} 1) {5})) (\n                _ extract {2} 1) {0}))\n            """\n            (declare-fun {} () (_ BitVec {1}))
            (assert (= {} (bvhamw (concat
                (f2 (_ extract {2} 1) {3}) (_ extract {2} 1) {4})
                (_ extract {2} 1) {5})) (\n                _ extract {2} 1) {0}))\n            """

45
```python
weight_smt_code = weight_smt_code.rstrip()
smt_code += weight_smt_code

counter_smt_code = textwrap.dedent(""
(\texttt{concat (\_ bv0 \{0\}) (\_ extract 0 0)
(f1 ((\_ extract \{1\} 1) (2)) ((\_ extract \{1\} 1) (3))
((\_ extract \{1\} 1) (4))))")""
(counter_smt_code - 1, self.wordsize - 1,
args[0], args[1], output_var_name))
counter_smt_code = counter_smt_code.rstrip()

return smt_code, counter_smt_code

def expr_to_smt(self, expr, symbol):

def get_last_arx_node(expr):
    node = next(sympy.preorder_traversal(expr))
    if node.func in parser.ARX_operations():
        return node
    raise ValueError("Invalid ARX expression: {}".format(expr))

def get_array_and_index(symbol):
    s = str(symbol)
    if s[0] == self.cipher["rounds"][symbol_name]:
        array = self.data_exprs
        cipher_rk = self.cipher["round_keys"]
        elif cipher_rk is not None and s[0] == cipher_rk.symbol_name:
            array = self.key_exprs
        else:
            raise ValueError("Invalid symbol: {}".format(s))
    i, j = s[1:].split(_)
    return array, int(i), int(j)

def update_array(array, i, j, node, symbol):
    for next_i in range(i, len(array)):
        for next_j in range(len(array[next_i])):
            array[next_i][next_j] = array[next_i][next_j].subs(node, symbol)

smt_code = ""
counter_smt_code = []

array, i, j = get_array_and_index(symbol)
nodes = list(sympy.postorder_traversal(expr))

if len(nodes) == 1:  # expr is a constant or a symbol (no op involved)
    node = nodes[0]
    if node.is_Integer:
        node = self.hex(int(node), self.wordsize)
    elif not node.is_Symbol:
        raise ValueError("Invalid expression: {}".format(expr))
    smt_code += "(\texttt{assert (= \{0\} \{1\})})\n".format(symbol, node)

update_array(array, i, j, expr, symbol)
```
return smt_code, counter_smt_code

add_counter = 0
op_counter = 0

while True:
    for node in sympy.postorder_traversal(expr):
        if node.func not in parser.ARX_operations():
            continue

        if node == get_last_arx_node(expr):
            output_var_name = str(symbol)
        else:
            output_var_name = "{}_".format(symbol, op_counter)
        smt_code += "(declare-fun {} () (_ BitVec {}))\n        output_var_name, self.wordsize)"
        op_counter += 1

    new_smt_code, new_counter_smt_code = self.ARX_node_to_smt(
        node, symbol, output_var_name, add_counter)

    smt_code += new_smt_code + "\n
    if new_counter_smt_code:
        counter_smt_code.append(new_counter_smt_code)
        add_counter += 1
    expr = expr.subs(node, sympy.Symbol(output_var_name))
    update_array(array, i, j, node, sympy.Symbol(output_var_name))
    break
else:
    return smt_code, counter_smt_code

def key_schedule_constraints(self):
    smt_code = ""
    counter_smt_code = []
    key_symbols = self.cipher["round_keys"].symbols()
    self.key_exprs = self.cipher["round_keys"].list[:]
    self.key_exprs = [list(key) for key in self.key_exprs]

    for i, key in enumerate(self.key_exprs):
        for j, expr in enumerate(key):
            symbol = key_symbols[i][j]
            new_smt_code, new_counter_smt_code = self.expr_to_smt(expr, symbol)
            smt_code += new_smt_code

            adds = len(new_counter_smt_code)
            if adds > 0:
                counter_smt_code.extend(new_counter_smt_code)
                weights = "{}_{i}_{j}_a\n                weight_prefix, symbol, a) for a in range(adds)"
                smt_code += "(assert (= {}_{i}_{j}_a))\n                weight_prefix, symbol, self.multi_binop(weights))"
            else:
                smt_code += "(assert (= {}_{i}_{j}))\n                weight_prefix, symbol, self.hex(0, self.countersize)"
smt_code += "\n"
return smt_code, counter_smt_code

def encryption_constraints(self):
smt_code = ""
counter_smt_code = []
data_symbols = self.cipher["rounds"].symbols()
self.data_exprs = self.cipher["rounds"].list[:]
self.data_exprs = [list(data) for data in self.data_exprs]
for i, data in enumerate(self.data_exprs):
    for j, expr in enumerate(data):
        symbol = data_symbols[i][j]
        new_smt_code, new_counter_smt_code = self.expr_to_smt(expr, symbol)
smt_code += new_smt_code + "\n"
adds = len(new_counter_smt_code)
if adds > 0:
    counter_smt_code.extend(new_counter_smt_code)
    weights = ["{}{}_{}".format(weight_prefix, symbol, a) for a in range(adds)]
smt_code += "(assert (= {}{} {}))\n".format(weight_prefix, symbol, self.multi_binop(weights))
else:
    smt_code += "(assert (= {}{} {}))\n".format(weight_prefix, symbol, self.hex(0, self.countersize))
return smt_code, counter_smt_code

def set_key_counters(self, key_counter_smt_code):
    num_additions = len(key_counter_smt_code)
    smt_code = "(assert (= {} {}))\n".format(self.key_counters[0],
        self.multi_binop(key_counter_smt_code, bitsize=self.countersize))
smt_code += "(assert (= {} (bvsub {} {})))\n".format(self.key_counters[1],
        self.hex(num_additions, self.countersize), self.key_counters[0])
return smt_code

def set_data_counters(self, data_counter_smt_code):
    num_additions = len(data_counter_smt_code)
    smt_code = "(assert (= {} {}))\n".format(self.data_counters[0],
        self.multi_binop(data_counter_smt_code, bitsize=self.countersize))
smt_code += "(assert (= {} (bvsub {} {})))\n".format(self.data_counters[1],
        self.hex(num_additions, self.countersize), self.data_counters[0])
return smt_code

def key_constraints(self):
symbols = flatten(self.cipher["round_keys"].symbols())
symbols = [weight_prefix + str(s) for s in symbols]
return textwrap.dedent("""
    (assert (bvule (bvadd (bvmul {} {})
        (bvadd (bvmul {} {}) (bvmul {} {}))) {})))"""

48
def data_constraints(self):
symbols = flatten(self.cipher["rounds"].symbols())
symbols = [weight_prefix + str(s) for s in symbols]
return textwrap.dedent("""\n(assert (bvule (bvmul {} {}) (bvadd (bvmul {} {}) {})))
""".format(self.hex(2, self.countersize), self.multi_binop(symbols),
self.hex(6, self.countersize), self.key_counters[0],
self.hex(3, self.countersize), self.key_counters[1],
self.hex(self.upper_bound_keys, self.countersize)))

def non_zero_input_data_difference(self):
symbols = [str(s) for s in self.cipher["plaintext"]]
eqs = ["(= {} {})".format(s, self.hex(0, self.wordsize)) for s in symbols]
return "(assert (not ((1))))".format(self.hex(0, self.wordsize),
self.multi_binop(eqs, binop="and"))

def footer(self):
data_symbols = flatten(self.cipher["rounds"].symbols())
data_weight_symbols = [weight_prefix + str(s) for s in data_symbols]
variables = itertools.chain(self.data_counters,
self.cipher["plaintext"], data_symbols, data_weight_symbols)
if self.cipher["round_keys"] is not None:
    key_symbols = flatten(self.cipher["round_keys"].symbols())
    key_weight_symbols = [weight_prefix + str(s) for s in key_symbols]
    variables = itertools.chain(variables, self.key_counters,
self.cipher["masterkey"], key_symbols, key_weight_symbols)
variables = [str(v) for v in variables]
smt_code = ' '.join(variables)
return textwrap.dedent("""
(check-sat)
(get-value ({}))
(exit)
""".format(smt_code))

def set_countersize(self):
if self.cipher["round_keys"] is not None:
    key_adds = self.cipher["round_keys"].number_of_additions()
else:
    key_adds = 0
data_adds = self.cipher["rounds"].number_of_additions()
if key_adds != 0:
    key_adds = int(math.ceil(math.log(key_adds, 2)))
if data_adds != 0:
    data_adds = int(math.ceil(math.log(data_adds, 2)))

margin = 2
self.countersize = max(
    key_adds + margin,
    data_adds + margin,
    int(math.ceil(math.log(self.wordsize, 2))) + margin,
    MIN_COUNTERSIZE
)

if self.countersize % 4 != 0:
    # must be multiple of 4
    self.countersize += 4 - self.countersize % 4

def write_smt_file(self, output_filename):
    self.set_countersize()

    with open(output_filename, 'w') as f:
        f.write(self.header() + '

        f.write(self.hamming_weight() + '

        f.write(self.characteristic_functions() + '

        if self.cipher['round_keys'] is not None:
            f.write(self.declare_key_variables() + '

            f.write(self.key_counter_constrains() + '

            key_schedule_smt_code, key_counter_smt_code = self.key_schedule_constraints()

            f.write(key_schedule_smt_code)
            f.write(self.set_key_counters(key_counter_smt_code) + '

            f.write(self.declare_data_variables() + '

            f.write(self.data_counter_constrains() + '

            data_schedule_smt_code, data_counter_smt_code = self.encryption_constraints()

            f.write(data_schedule_smt_code)
            f.write(self.set_data_counters(data_counter_smt_code) + '

            f.write(self.data_constraints() + '

            if self.cipher.get('non_zero_input_data_difference', False) is True:
                f.write(self.non_zero_input_data_difference() + '

        f.write(self.footer())

if __name__ == '__main__':
    cipher_filename = sys.argv[1]
    output_filename = sys.argv[2]

    cipher = parser.parse_ARX_cipher(cipher_filename)
    parser.generate_symbolic_expressions(cipher)
    parser.print_cipher(cipher)

    upper_bound_data = 2 * cipher['wordsize'] * len(cipher['plaintext'])
upper_bound_data_counter = cipher["rounds"].number_of_additions()
if cipher["round_keys"] is not None:
    upper_bound_keys = 2 * cipher["wordsize"] * len(cipher["masterkey"])
    upper_bound_key_counter = cipher["round_keys"].number_of_additions()
else:
    upper_bound_keys = None
    upper_bound_key_counter = None

writer = SMTwriter(cipher, upper_bound_data, upper_bound_data_counter,
    upper_bound_keys, upper_bound_key_counter)
writer.write_smt_file(output_filename)
print("\nSMT file generated in {}".format(output_filename))

arxpy.py
import re
import subprocess
import sys
import time
import parser
import smt_writer

class STP(object):
    def __init__(self, output_filename, cipher,
        upper_bound_data, upper_bound_data_counter,
        upper_bound_keys=None, upper_bound_key_counter=None,
        max_upper_bound_data_counter=None,
        max_upper_bound_key_counter=None):
        self.output_filename = output_filename
        self.cipher = cipher
        self.upper_bound_data = upper_bound_data
        self.upper_bound_data_counter = upper_bound_data_counter
        self.upper_bound_keys = upper_bound_keys
        self.upper_bound_key_counter = upper_bound_key_counter
        if max_upper_bound_data_counter is None:
            self.max_upper_bound_data_counter = cipher["rounds"].number_of_additions()
        else:
            self.max_upper_bound_data_counter = max_upper_bound_data_counter
        if max_upper_bound_key_counter is None:
            if cipher["round_keys"] is not None:
                self.max_upper_bound_key_counter = cipher["round_keys"].number_of_additions()
            else:
                self.max_upper_bound_key_counter = None
        else:
            self.max_upper_bound_key_counter = None
self.max_upper_bound_key_counter = max_upper_bound_key_counter

def write_smt_file(self):
    writer_object = smt_writer.SMTwriter(self.cipher,
                                           self.upper_bound_data, self.upper_bound_data_counter,
                                           self.upper_bound_keys, self.upper_bound_key_counter)

    writer_object.write_smt_file(self.output_filename)
    self.writer_object = writer_object

def solve(self):
    self.write_smt_file()
    command = "stp {} --cryptominisat --threads {}".format(self.output_filename, threads)

    try:
        self.output = subprocess.check_output(command, encoding='ascii', shell=True)
    except subprocess.CalledProcessError as e:
        log_file = self.output_filename.replace(".smt2", crash_suffix)
        with open(log_file, 'w') as f:
            f.write("An error occurred when STP was solving {}.\n\n".format(self.output_filename))

            f.write(str(e) + "\n\n")

            f.write("Current values of the bounds. \n")
            f.write("- Encryption characteristic: \n")
            f.write("  Upper bound on the probability : {}\n".format(self.upper_bound_data))
            f.write("  Upper bound on the counter: {}\n".format(self.upper_bound_data_counter))

            if self.upper_bound_keys and self.upper_bound_key_counter:

                f.write("- Key schedule characteristic:: \n")
                f.write("  Upper bound on the probability : {}\n".format(self.upper_bound_keys))
                f.write("  Upper bound on the counter: {}\n".format(self.upper_bound_key_counter))

            f.write("Time consumed: {} seconds\n".format((time.time() - start_time)))

        sys.exit("An error occurred while solving {}.\nCheck {}.".format(self.output_filename, log_file))
    else:
        print('.', end=' ', flush=True)

def is_satisfiable(self):
    return self.output.startswith("sat")

def get_data_counters(self):
    data_counters = []
    for counter_name in smt_writer.data_counters:
        pattern = r"\( \|{}\| \(_ bv(\d+) " .format(counter_name)
        data_counters.append(int(re.search(pattern, self.output).groups()[0])))
return data_counters

def get_key_counters(self):
    key_counters = []
    for counter_name in smt_writer.key_counters:
        pattern = r"\( \|{}\| \(_ bv\(d+\)\)\)".format(counter_name)
        key_counters.append(int(re.search(pattern, self.output).groups()[0]))
    return key_counters

def get_solution(self):
    if self.is_satisfiable():
        solution = "Encryption characteristic: \n"
        solution += "\tProbability (-log2): {}\n".format(self.get_data_probability())
        solution += "\tUpper bound on the probability (-log2): {}\n".format(
            self.get_upper_bound_data())
        solution += "\tRX differences:\n"
        variables = re.findall(r"\( \|\w+\| \(_ bv\(d+\)\)\)\", self.output)
        for name, value in variables:
            if name.startswith(parser.plaintext_symbol):
                size = self.cipher["wordsize"]
                hex_format = "{0:#0{1}x}".format(int(value), (size // 4) + 2)
                solution += "\t\t\t\t{}: {}\n".format(name, hex_format)
            elif name.startswith(parser.output_round_symbol):
                size = self.cipher["wordsize"]
                hex_format = "{0:#0{1}x}".format(int(value), (size // 4) + 2)
                solution += "\t\t\t\t{}: {}\n".format(name, hex_format)
        solution += "\tWeights:\n"
        for name, value in variables:
            if name.startswith(smt_writer.weight_prefix + parser.output_round_symbol):
                size = self.writer_object.countersize
                hex_format = "{0:#0{1}x}".format(int(value), (size // 4) + 2)
                solution += "\t\t\t\t{}: {}\n".format(name, hex_format)
        solution += "\tCounters:\n"
        for name, value in variables:
            if name in smt_writer.data_counters:
                size = self.writer_object.countersize
                hex_format = "{0:#0{1}x}".format(int(value), (size // 4) + 2)
                solution += "\t\t\t\t{}: {}\n".format(name, hex_format)
    if self.cipher["round_keys"] is not None:
        solution += "Key schedule characteristic: \n"
        solution += "\tProbability (-log2): {}\n".format(self.get_key_probability())
        solution += "\tUpper bound on the probability (-log2): {}\n".format(
            self.get_upper_bound_keys())
        solution += "\tRX differences:\n"
        variables = re.findall(r"\( \|\w+\| \(_ bv\(d+\)\)\)\", self.output)
        for name, value in variables:
            if name.startswith(parser.master_key_symbol):
                size = self.cipher["wordsize"]
                hex_format = "{0:#0{1}x}".format(int(value), (size // 4) + 2)
                solution += "\t\t\t\t{}: {}\n".format(name, hex_format)

53
size = self.cipher["wordsize"]
hex_format = "\(0:\#0(1)x\)".format(int(value), (size // 4) + 2)
solution += "\(t\{\}\n".format(name, hex_format)

for name, value in variables:
    if name.startswith(parser.round_key_symbol):
        size = self.cipher["wordsize"]
        hex_format = "\(0:\#0(1)x\)".format(int(value), (size // 4) + 2)
        solution += "\(t\{\}\n".format(name, hex_format)

solution += "\tWeights:
" for name, value in variables:
    if name.startswith(smt_writer.weight_prefix + parser.round_key_symbol):
        size = self.writer_object.countersize
        hex_format = "\(0:\#0(1)x\)".format(int(value), (size // 4) + 2)
        solution += "\(t\{\}\n".format(name, hex_format)

solution += "\tCounters:
" for name, value in variables:
    if name in smt_writer.key_counters:
        size = self.writer_object.countersize
        hex_format = "\(0:\#0(1)x\)".format(int(value), (size // 4) + 2)
        solution += "\(t\{\}\n".format(name, hex_format)

return solution

else:
    solution = "No solution found with the following upper bounds: \n"
solution += "Upper bound on the encryption characteristic's "
if self.cipher["probability\(-log2\): {}"].format(self.get_upper_bound_data())
    solution += "Upper bound on the key schedule characteristic's "
solution += "probability\(-log2\): {}".format(self.get_upper_bound_keys())

return solution

def get_upper_bound_data(self):
    lower_bound_ctr = self.max_upper_bound_data_counter - self.upper_bound_data_counter
    return self.upper_bound_data / 2.0 - 0.085 * lower_bound_ctr

def get_upper_bound_keys(self):
    lower_bound_ctr = self.max_upper_bound_key_counter - self.upper_bound_key_counter
    return self.upper_bound_keys / 2.0 - 0.085 * lower_bound_ctr

def get_data_probability(self):
    weights = []
    pattern = r"\( \|{\}\{\}\{\}\{\}\d+\d+\| \(_ bv\(\d+) \).format(
        smt_writer.weight_prefix, parser.output_round_symbol)
    groups = re.findall(pattern, self.output)
    weights = [int(g) for g in groups]

    log_probability = sum(weights)

    counters = self.get_data_counters()
    if sum(counters) != self.cipher["rounds"].number_of_additions():
        raise ValueError("Sum of counters ({} must be the # of ModAdd ({}).".format(
            sum(counters), self.cipher["rounds"].number_of_additions()))
log_probability += 3 * counters[0] + 1.415 * counters[1]
return log_probability

def get_key_probability(self):
    weights = []
    pattern = r"\( \|{}{}\d+\| \(_ bv(\d+) \) \).format(
        smt_writer.weight_prefix, parser.round_key_symbol)
groups = re.findall(pattern, self.output)
weights = [int(g) for g in groups]

    log_probability = sum(weights)

    counters = self.get_key_counters()
    if sum(counters) != self.cipher["round_keys"].number_of_additions():
        raise ValueError("Sum of counters ({}) must be the # of ModAdd ({}).".format( 
            sum(counters), self.cipher["round_keys"].number_of_additions()))

    log_probability += 3 * counters[0] + 1.415 * counters[1]
    return log_probability

class CharacteristicFinder(object):
    def __init__(self, cipher, smt_filename, results_filename):
        self.cipher = cipher
        self.smt_filename = smt_filename
        self.results_filename = results_filename

        w = self.cipher["wordsize"]
        self.default_upper_bound_data = 2 * w * len(self.cipher["plaintext"])
        self.default_upper_bound_data_counter = self.cipher["rounds"].number_of_additions()
        if cipher["round_keys"] is not None:
            self.default_upper_bound_keys = 2 * w * len(self.cipher["masterkey"])
            aux = self.cipher["round_keys"].number_of_additions()
            self.default_upper_bound_key_counter = aux
        else:
            self.default_upper_bound_keys = None
            self.default_upper_bound_key_counter = None

    def find_best_upper_bound_data(self, min_upper_bound_data, max_upper_bound_data):
        data_counter = None

        while min_upper_bound_data < max_upper_bound_data:
            upper_bound_data = (min_upper_bound_data + max_upper_bound_data) // 2
            stp_object = STP(self.smt_filename, self.cipher, 
                upper_bound_data, self.default_upper_bound_data_counter, 
                self.default_upper_bound_keys, self.default_upper_bound_key_counter)
            stp_object.solve()
            if stp_object.is_satisfiable():
                max_upper_bound_data = upper_bound_data
                data_counter = stp_object.get_data_counters()[0]
            else:
                min_upper_bound_data = upper_bound_data + 1

        return data_counter
solution = stp_object.get_solution()
with open(self.results_filename, 'a') as f:
    f.write(solution)
    f.write("Time consumed (from the start): {} seconds\n\n".format(time.time() - start_time))
return max_upper_bound_data, data_counter

def find_best_upper_bound_keys(self, min_upper_bound_keys, max_upper_bound_keys, best_upper_bound_data):
    key_counter = None
    while min_upper_bound_keys < max_upper_bound_keys:
        upper_bound_keys = (min_upper_bound_keys + max_upper_bound_keys) // 2
        stp_object = STP(self.smt_filename, self.cipher,  
        best_upper_bound_data, self.default_upper_bound_data_counter,  
        upper_bound_keys, self.default_upper_bound_key_counter)
        stp_object.solve()
        if stp_object.is_satisfiable():
            max_upper_bound_keys = upper_bound_keys
            key_counter = stp_object.get_key_counters()[1]
        else:
            min_upper_bound_keys = upper_bound_keys + 1
        solution = stp_object.get_solution()
    with open(self.results_filename, 'a') as f:
        f.write(solution)
        f.write("Time consumed (from the start): {} seconds\n\n".format(time.time() - start_time))
    return max_upper_bound_keys, key_counter

def find_best_upper_bound_data_counter(self, best_upper_bound_data, max_data_counter):
    if max_data_counter == 0:
        return max_data_counter

    min_data_counter = 0
    data_counter = None
    while min_data_counter < max_data_counter:
        data_counter = (min_data_counter + max_data_counter) // 2
        stp_object = STP(self.smt_filename, self.cipher,  
        best_upper_bound_data, data_counter,  
        self.default_upper_bound_keys, self.default_upper_bound_key_counter)
        stp_object.solve()
        if stp_object.is_satisfiable():
            max_data_counter = data_counter
        else:
            min_data_counter = data_counter + 1
        solution = stp_object.get_solution()
    with open(self.results_filename, 'a') as f:
        f.write(solution)
        f.write("Time consumed (from the start): {} seconds\n\n".format(time.time() - start_time))
    return max_data_counter
def find_best_upper_bound_key_counter(self, best_upper_bound_data, best_upper_bound_keys, best_upper_bound_data_counter, max_key_counter):
    if max_key_counter == 0:
        return max_key_counter
    min_key_counter = 0
    key_counter = None
    while min_key_counter < max_key_counter:
        key_counter = (min_key_counter + max_key_counter) // 2
        stp_object = STP(self.smt_filename, self.cipher,
                         best_upper_bound_data, best_upper_bound_data_counter,
                         best_upper_bound_keys, key_counter)
        stp_object.solve()
        if stp_object.is_satisfiable():
            max_key_counter = key_counter
        else:
            min_key_counter = key_counter + 1
    solution = stp_object.get_solution()
    with open(self.results_filename, 'a') as f:
        f.write(solution)
        f.write("Time consumed (from the start): {} seconds\n
        ".format(time.time() - start_time))
    return max_key_counter

def find_optimal_characteristic(self):
    start_time = time.time()
    open(self.results_filename, 'w').close()  # erase file
    min_upper_bound_data = 0
    max_upper_bound_data = 2 * self.cipher["wordsize"] * len(self.cipher["plaintext"])
    best_upper_bound_data, data_counter = self.find_best_upper_bound_data(min_upper_bound_data, max_upper_bound_data)
    if data_counter is None:
        with open(self.results_filename, 'a') as f:
            f.write("No encryption characteristic found with enough probability. \n\n")
    else:
        best_upper_bound_data_counter = self.find_best_upper_bound_data_counter(  
            best_upper_bound_data, data_counter)
        if self.cipher["round_keys"] is not None:
            min_upper_bound_keys = 0
            max_upper_bound_keys = 2 * self.cipher["wordsize"]
            max_upper_bound_keys *= len(self.cipher["masterkey"])
            best_upper_bound_keys, key_counter = self.find_best_upper_bound_keys(  
                min_upper_bound_keys, max_upper_bound_keys, best_upper_bound_data)
            self.find_best_upper_bound_key_counter(  
                best_upper_bound_data, best_upper_bound_keys,  
                best_upper_bound_data, best_upper_bound_keys,  
                best_upper_bound_data, max_key_counter)
best_upper_bound_data_counter, key_counter)

with open(self.results_filename, 'a') as f:
    f.write("Total time consumed: {} seconds\n\n".format(time.time() - start_time))

if __name__ == '__main__':
    cipher_filename = sys.argv[1]
    smt_filename = cipher_filename.replace(".py", "\nsmt2")
    results_filename = sys.argv[2]
    cipher = parser.parse_ARX_cipher(cipher_filename)
    parser.generate_symbolic_expressions(cipher)
    finder = CharacteristicFinder(cipher, smt_filename, results_filename)
    finder.find_optimal_characteristic()
Appendix B

ARX implementations

This chapter contains ARX implementations of the ARX block cipher LEA, the underlying permutation of the MAC Chaskey, and the entire Speck family.

lea128.py

"""LEA with 128-bit key""

wordsize = 32
number_of_rounds = 2
delta = [
    0xc3efe9db, 0x44626b02,
    0x79e27c8a, 0x78df30ec,
    0xc3efe9db, 0x44626b02,
    0x79e27c8a, 0x78df30ec,
    0xc3efe9db, 0x44626b02,
    0x79e27c8a, 0x78df30ec,
    0xc3efe9db, 0x44626b02,
    0x79e27c8a, 0x78df30ec,
    0xc3efe9db, 0x44626b02,
    0x79e27c8a, 0x78df30ec,
    0xc3efe9db, 0x44626b02,
    0x79e27c8a, 0x78df30ec,
]

def encryption(p0, p1, p2, p3):
    for i in range(number_of_rounds):
        if i == 0:
            x0, x1, x2, x3 = p0, p1, p2, p3
        else:
            x0, x1, x2, x3 = rounds[i - 1]

        x0 = ((x0 ^ round_keys[i][0]) + (x1 ^ round_keys[i][1])) << 9
        x1 = ((x1 ^ round_keys[i][2]) + (x2 ^ round_keys[i][3])) >> 5
        x2 = ((x2 ^ round_keys[i][4]) + (x3 ^ round_keys[i][5])) >> 3
        x3 = x0

    if i == number_of_rounds - 1:
        return x0, x1, x2, x3
else:
    rounds[i] = x0, x1, x2, x3

def key_schedule(k0, k1, k2, k3):
    for i in range(number_of_rounds):
        if i == 0:
            T0, T1, T2, T3 = k0, k1, k2, k3
        else:
            T0, T1, T2 = round_keys[i - 1][:3]
            T3 = round_keys[i - 1][4]
            T0 = (T0 + (delta[i] << i)) << 1
            T1 = (T1 + (delta[i] << (i + 1))) << 3
            T2 = (T2 + (delta[i] << (i + 2))) << 6
            T3 = (T3 + (delta[i] << (i + 3))) << 11
        round_keys[i] = T0, T1, T2, T1, T3, T1

chaskey_permutation.py

"""The underlying permutation of the MAC Chaskey"""
wordsize = 32
number_of_rounds = 8

def permutation(v0, v1, v2, v3):
    v0 = v0 + v1
    v1 = v1 << 5
    v1 = v1 ^ v0
    v0 = v0 << 16
    v2 = v2 + v3
    v3 = v3 << 8
    v3 = v3 ^ v2
    v0 = v0 + v3
    v3 = v3 << 13
    v3 = v3 ^ v0
    v2 = v2 + v1
    v1 = v1 << 7
    v1 = v1 ^ v2
    v2 = v2 << 16
    return v0, v1, v2, v3

def encryption(p0, p1, p2, p3):
    rounds[0] = permutation(p0, p1, p2, p3)
    for i in range(1, number_of_rounds - 1):
        rounds[i] = permutation(*rounds[i - 1])
    return permutation(*rounds[number_of_rounds - 2])

def test():
    assert encryption(0, 0, 0, 0) == (0, 0, 0, 0)
```python
speck_family.py

"""SPECK 2n/mn
Instances of SPECK (n, m):
(16, 4), (24, 3), (24, 4),
(32, 3), (32, 4), (48, 2), (48, 3),
(64, 2), (64, 3), (64, 4)
"""

# wordsize and number_of_rounds are set automatically according to the value of (n, m)
n = 64
m = 4

wordsie = n
if n == 16:
    number_of_rounds = 22
elif n == 24:
    if m == 3:
        number_of_rounds = 22
    elif m == 4:
        number_of_rounds = 23
elif n == 32:
    if m == 3:
        number_of_rounds = 26
    elif m == 4:
        number_of_rounds = 27
elif n == 48:
    if m == 2:
        number_of_rounds = 28
    elif m == 3:
        number_of_rounds = 29
elif n == 64:
    if m == 2:
        number_of_rounds = 32
    elif m == 3:
        number_of_rounds = 33
    elif m == 4:
        number_of_rounds = 34

if n == 16:
    alpha = 7
    beta = 2
else:
    alpha = 8
    beta = 3

def f(x, y, k):
    x = ((x >> alpha) + y) ^ k
    y = (y << beta) ^ x
    return x, y

def encryption(x0, y0):
    rounds[0] = f(x0, y0, round_keys[0])
```
for i in range(1, number_of_rounds - 1):
    rounds[i] = f(rounds[i - 1][0], rounds[i - 1][1], round_keys[i])

return f(rounds[number_of_rounds - 2][0], rounds[number_of_rounds - 2][1],
         round_keys[number_of_rounds - 1])

def _key_schedule(*args):
    l = [None for i in range(number_of_rounds + (m - 1))]

    round_keys[0] = args[-1]
    l[:m - 1] = list(reversed(args[:-1]))

    for i in range(number_of_rounds - 1):
        l[i + (m - 1)], round_keys[i + 1] = f(l[i], round_keys[i], i)

if m == 2:
    def key_schedule(l0, k0):
        return _key_schedule(l0, k0)
elif m == 3:
    def key_schedule(l1, l0, k0):
        return _key_schedule(l1, l0, k0)
elif m == 4:
    def key_schedule(l2, l1, l0, k0):
        return _key_schedule(l2, l1, l0, k0)
Appendix C

Paper
An Easy to Use Tool for Rotational-XOR Cryptanalysis of ARX Block Ciphers

Abstract—An increasing number of lightweight cryptographic primitives have been published recently. Some of these proposals are ARX primitives, which have shown a great performance in software. Rotational-XOR cryptanalysis is a statistical technique to attack ARX primitives. As opposed to rotational cryptanalysis, rotational-XOR cryptanalysis can deal with ARX primitives where constants are injected into the state. In this paper, it is shown a computer tool to speed up and make easier the security evaluation of ARX block ciphers against rotational-XOR cryptanalysis. In particular, our tool takes a Python implementation of an ARX block cipher and automatically finds an optimal rotational-XOR characteristic. Compared to most of the automated tools, which only support a small set of primitives, our tool supports any ARX block cipher and it is executed with a simple shell command.

Index Terms—Cryptography, ARX, Rotational-XOR Cryptanalysis, Automatic Search, ArXPy

I. INTRODUCTION

Due to the progress of technology, electronic devices have become so cheap and small that they are being embedded in everyday objects. As a result, the Internet is evolving into a network of “smart objects” that communicate with each other. This “Internet of Things” will include about 20 billion devices by 2020 according to Gartner [3].

The basic function of these devices is to collect and transmit information. In some cases, sensitive information is collected such as health-monitoring or biometric data [4]. Therefore, there is a high demand to implement cryptographic algorithms in these devices.

However, some of these new devices have so extreme constrains in computational power, chip area or memory that they are not powerful enough to use the same cryptographic algorithms as standard PCs. Example of these types of devices are RFID (Radio-Frequency IDentification) chips and sensor networks. For this reason, many cryptographic algorithms tailored to such constrained environments have been published recently.

Some of these proposals only use three types of operations: modular addition, cyclic rotation and exclusive-or (XOR). Examples of these Addition-Rotation-XOR (ARX) primitives are Salsa20 [5], Chaskey [7], or SPECK [8]. ARX primitives are among the best performers in software [10].

Differential cryptanalysis [11], one of the most powerful attack against block ciphers, have been applied to several ARX primitives [12]–[14]. On the other hand, rotational cryptanalysis has become very popular to analyse ARX primitives since it was proposed as a generic attack to ARX structures [15]. Rotational cryptanalysis was improved in [16], where the propagation of rotational pairs through modular addition, especially through consecutive modular additions, was studied in more detail.

One of the main drawbacks of rotational cryptanalysis is that it can not be applied to ARX primitives where constants are injected into the state. A combination of differential and rotational cryptanalysis was considered in [15] and formalized as rotational-XOR cryptanalysis in [17]. Rotational-XOR cryptanalysis removes this restriction and so it can be applied to a bigger subset of ARX primitives than rotational cryptanalysis.

Recently, automatic search tools have been used for finding characteristics with high probability. In [12], a method was proposed for finding differential characteristics in ARX primitives using a SAT solver. To the best of our knowledge, the only application of a SAT solver for finding rotational-XOR characteristics was done in [18] to attack SPECK.

We have implemented a computer tool for finding optimal rotational-XOR characteristics of ARX block ciphers. Our tool, called ArXPy, takes a Python implementation of an ARX block cipher as input and finds an optimal characteristic by using a SAT solver.

The rest of this paper is organized as follows: in Section II, ARX block ciphers and rotational-XOR cryptanalysis are introduced. In Section III, a SAT-based method for finding optimal rotational-XOR characteristic is described. The ArXPy tool is presented in Section IV, along with its usage and implementation. Lasty, Section V concludes this paper.

II. PRELIMINARIES

A. Notations

The used notations in this paper are shown in Table I.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊕</td>
<td>Modular addition.</td>
</tr>
<tr>
<td>⊕</td>
<td>XOR.</td>
</tr>
<tr>
<td>◄</td>
<td>Left rotation.</td>
</tr>
<tr>
<td>◄</td>
<td>Right rotation.</td>
</tr>
<tr>
<td>X</td>
<td>An n-bit boolean vector.</td>
</tr>
<tr>
<td>S(HL(X))</td>
<td>A non-cyclic left shift of X by one bit.</td>
</tr>
<tr>
<td>(I ⊕ S(HL)(X))</td>
<td>X ⊕ S(HL)(X).</td>
</tr>
<tr>
<td>1X ≤ Y</td>
<td>A characteristic function which evaluates to 1 if X_i ≤ Y_i for all i.</td>
</tr>
<tr>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>X</td>
</tr>
<tr>
<td>L(X)</td>
<td>The n − 1 most significant bits of X.</td>
</tr>
</tbody>
</table>

B. ARX block ciphers

Block ciphers which only use modular additions, cyclic rotations and exclusive-or (XOR) operations are called Addition-
Rotation-XOR (ARX) block ciphers. The usage of these operations in modern block ciphers is not new; the first one dates back to 1987: the FEAL cipher [26]. The term ARX is much recent; it was proposed in 2009 by Weinmann [27].

Many ARX block ciphers have been proposed since 2009. Some examples are: LEA [6], the underlying permutation for the message authentication code Chaskey [7], SPARX [9], Speck [8], and the underlying block cipher Threefish of the hash function Skein [28].

ARX block ciphers are among the best performers in software. In a comparison of software implementations of block ciphers for small processors [10], the most efficient ones were ARX block ciphers. Furthermore, ARX block ciphers are simple and easy to describe, which results in implementations with small code size. As shown in [10], the implementations with the smallest code size were achieved by ARX block ciphers.

In [15], it was shown that any function can be realized with modular additions, rotations, XORs and a single constant. Furthermore, removing one of these operations results in a dramatic loss of security. Without the modular addition, an XR block cipher is easily broken by describing each output bit and solving the resulting system of linear equations modulo 2 (which can be done efficiently). Without the rotation, it is also possible to break an AX block cipher with low complexity [15]. Without the XOR, an AR block cipher needs a higher number of operations than an ARX block cipher for the same level of security [30].

C. Rotational-XOR cryptanalysis

Rotational-XOR cryptanalysis is a recent technique to analyse ARX block ciphers. Ashur and Liu formalized it and applied it to Speck [17]. This technique studies the propagation of rotational-XOR (RX) differences through the encryption function of ARX block ciphers. First, some notation is introduced.

A pair \((X, X')\) has RX difference \((\alpha, \gamma)\) if \(X \oplus (X' \lll \gamma) = \alpha\). To simplify the notation, the rotational offset \(\gamma\) will be fixed, \(X\) will denote \(X' \lll \gamma\) and \(X\) will denote \(X' \ggg \gamma\). Furthermore, we will say that the RX difference of a pair \((X, X')\) is \(\alpha\) if \(X \oplus X' = \alpha\).

If \(\gamma = 0\), an RX difference is just an XOR-difference and if \(\alpha = 0\), the pair \((X, X') = (X, \bar{X})\) is just a rotational pair. Therefore, rotational-XOR cryptanalysis can be seen as a generalization of differential cryptanalysis and rotational cryptanalysis.

In rotational-XOR cryptanalysis, the attacker can obtain the encryption of plaintexts under a pair of keys \((k, k')\) where the RX differences of the pairs of round keys \((k_i, k_i')\) are known to the attacker.

The RX difference \(\alpha\) of a plaintext pair \((P, P')\) is called an input RX difference. The output RX difference is the RX difference \(\beta\) of its corresponding ciphertext \((C, C') = (E_k(P), E_{k'}(P'))\), where \(E_k\) denotes the encryption function using key \(k\). A RX characteristic is a sequence of intermediate RX differences along with the input RX difference and RX output difference.

Rotational-XOR cryptanalysis is a distinguishing attack that exploits input RX differences which produces output RX differences with a probability higher than a random permutation. To the best of our knowledge, the only application of rotational-XOR cryptanalysis (as described in [17]) is a distinguishing attack on Speck [17], [18].

In the rest of this section, the propagation of RX differences through the ARX operations will be studied. The results about the propagation of RX differences will be used in Section III to describe a method to search for optimal RX characteristics.

The propagation of RX differences through rotations or XORs is deterministic and the resulting RX difference can be computed as follows:

**Lemma II.1** ([18]). Given two pairs \((X, \bar{X} \oplus \alpha)\) and \((Y, \bar{Y} \oplus \beta)\) with RX differences \(\alpha\) and \(\beta\) respectively and a constant \(c\), the RX differences propagate through rotation and XOR as follows:

\[
\begin{align*}
(X \ggg c, (\bar{X} \oplus \alpha) \ggg c) &= (Z, \bar{Z} \oplus (\alpha \ggg c)) \\
(X \lll c, (\bar{X} \oplus \alpha) \lll c) &= (Z, \bar{Z} \oplus (\alpha \lll c)) \\
(X \oplus Y, (\bar{X} \oplus \alpha) \oplus (\bar{Y} \oplus \beta)) &= (Z, \bar{Z} \oplus (\alpha \oplus \beta)) \\
(X \oplus c, (\bar{X} \oplus \alpha) \oplus c) &= (Z, \bar{Z} \oplus (\alpha \oplus c \lll \gamma))
\end{align*}
\]

for some \(Z\) (different in each case).

The propagation of RX differences through modular addition is not deterministic, but its probability can be computed when the rotational offset \(\gamma\) is fixed to 1 [17].

**Theorem II.1.** Let \(X, Y\) and \(Z\) be \(n\)-bit words sampled uniformly at random over the \(n\)-bit strings \(\{0, 1\}^n\) and let \((X, \bar{X} \oplus \alpha), (Y, \bar{Y} \oplus \beta)\) and \((Z, \bar{Z} \oplus \zeta)\) be three pairs with RX differences \(\alpha\), \(\beta\) and \(\zeta\) respectively and with rotational offset \(\gamma = 1\). Then,

\[
P[X \oplus Y, (\bar{X} \oplus \alpha) \oplus (\bar{Y} \oplus \beta)] = (Z, \bar{Z} \oplus \zeta) =
\]

\[
\begin{align*}
&\frac{1}{2} \left[ \text{SHL}((\delta_\alpha \oplus \delta_\beta) \lll \delta_\zeta) \leq \text{SHL}((\delta_\alpha \oplus \delta_\beta) \lll \delta_\zeta) \right] \times 2^{-3} + \\
&\times \frac{2}{\text{SHL}((\delta_\alpha \oplus \delta_\beta) \lll \delta_\zeta)} \times \frac{2}{\text{SHL}((\delta_\alpha \oplus \delta_\beta) \lll \delta_\zeta)} \times 2^{-1.415}
\end{align*}
\]

where \(\delta_\alpha = L(\alpha), \delta_\beta = L(\beta)\) and \(\delta_\zeta = L(\zeta)\).

In the case that \(\alpha = \beta = \zeta = 0\), theorem II.1 predicts the probability that the rotational property is preserved after a modular addition \(P(\bar{X} \oplus \bar{Y}) = \bar{X} \oplus \bar{Y})\).

III. AUTOMATIC SEARCH FOR ROTATIONAL-XOR CHARACTERISTICS

Characteristics can be used to estimate the success probability of differential and rotational-XOR cryptanalysis. The higher the probability of the characteristic, the higher the success probability of the distinguishing attack (and the lower the complexity of the corresponding key recovery attack).

Different automatic tools have been proposed to search for characteristics of ARX systems [12], [13], [37]. The technique proposed by Mouha et al. [12] has the advantage that it finds
the optimal characteristics. In [12], it was applied to search for differential characteristics of the ARX stream cipher Salsa20 [5]. An improvement of this technique was considered in [14] to search for differential characteristics of SPECK and LEA. Recently, it was applied to search for RX characteristics of SPECK [18].

Basically, this technique consists of rewriting the search problem as a SAT (Boolean Satisfiability) problem. A similar approach was taken in [37], but the search problem was formulated as a Mixed-Integer Linear Program instead.

The rest of this section focuses on the SAT approach. First, the SAT problem is introduced and then a SAT-based method for finding optimal rotational-XOR characteristics is explained.

A. The boolean satisfiability problem

A boolean formula is an expression which consists of boolean variables, which can take the values TRUE or FALSE, and the logic operators AND, OR and NOT. A boolean formula is satisfiable if there exists an assignment of the variables that makes the formula TRUE. For example the boolean formula

\[(\alpha \land \lnot \beta) \land (\lnot \gamma \lor \delta)\]

is satisfiable since the assignment \((\alpha, \beta, \gamma, \delta) = (\text{FALSE}, \text{TRUE}, \text{TRUE}, \text{FALSE})\) evaluates the entire formula to TRUE.

The boolean satisfiability (SAT) problem is the problem of determining whether a boolean formula is satisfiable. In general, the SAT problem is NP-complete [38], which implies that no known algorithm solves SAT in polynomial time (with respect to the number of variables). In other words, solving a SAT problem is infeasible if the number of variables is high enough. In practice, SAT solvers can handle instances with thousands (and sometimes even millions) of variables [39].

A generalization of the SAT problem is the satisfiability modulo theories (SMT) problem. Basically, SMT formulas can be expressed with richer languages (theories) than boolean formulas. In particular, a formula in the bit-vector theory can contain bit-vectors (a vector of boolean variables) and the usual operations of bit-vectors such as bitwise operations (XOR, OR, AND, etc) arithmetic operations (addition, multiplication, etc), cyclic operations and so on. A common approach in SMT solvers [40], [41] is to translate the SMT instance to a SAT instance and solve using a SAT solver.

B. A SAT-based method for searching optimal rotational-XOR characteristics

In [18], a SAT solver was used for finding optimal characteristics of the ARX block cipher SPECK. In this section, the method used in [18] will be explained in detail. First, a similar notation to [12] is be introduced.

A triplet of RX differences \((\alpha, \beta, \zeta)\) is valid if \(\alpha\) and \(\beta\) propagate to \(\zeta\) after a modular addition with non-zero probability, that is, \(P(\alpha, \beta \rightarrow \zeta) \neq 0\). Using theorem II.1, the triplet \((\alpha, \beta, \zeta)\) is valid if and only if one of the following conditions holds

\[
(I \oplus \text{SHL})(\delta_\alpha \oplus \delta_\beta \oplus \delta_\zeta) + 1 \leq R
\]

\[
(I \oplus \text{SHL})(\delta_\alpha \oplus \delta_\beta \oplus \delta_\zeta) \leq R.
\]

where \(R = \text{SHL}((\delta_\alpha \oplus \delta_\zeta)(\delta_\beta \oplus \delta_\zeta))\).

The weight \(\omega\) of a valid triplet \((\alpha, \beta, \zeta)\) is defined as

\[
\omega(\alpha, \beta, \zeta) = -\log_2(P(\alpha, \beta \rightarrow \zeta)).
\]

Using Theorem II.1, the weight can be calculated as follows

\[
\omega(\alpha, \beta, \zeta) = \begin{cases} |R| + 3, & \text{if (1) holds} \\ |R| + 1.415, & \text{if (2) holds} \end{cases}
\]

The weight \(W\) of an RX characteristic is defined as the sum of the weights of each of its modular additions. As in [12], it is assumed that the probability of a characteristic is the multiplication of the probabilities of each modular addition. In this case, the probability of an RX characteristic \(p\) can be calculated as \(p = 2^{-W}\).

The main idea of this technique is to use a SAT solver to determine whether there exists an RX characteristic up to a certain weight \(W\). If the SAT solver obtains that such problem is satisfiable, a higher weight is chosen. Otherwise, a lower weight is chosen. This is repeated until the minimal weight is found, that is, a weight \(W\) such that there exists an RX characteristic up to weight \(W\) but not up to weight \(W - \epsilon\), where \(\epsilon\) is a value fixed from the start (usually \(\epsilon = 1\) [12], [18]).

Different strategies have been used to obtain the minimal weight. In [12], the search starts with a low weight and it is incremented by one at a time until a satisfiable formula is found. In [42], the search starts with a high weight and it is decremented by one at a time until an unsatisfiable formula is found. In [18], a binary search strategy is used to find the minimal weight.

SAT solvers not only determine whether a formula is satisfiable or unsatisfiable, but also obtain an assignment that makes the formula TRUE if it is satisfiable. Therefore, the result of this method is a minimal weight \(W\), an RX characteristic with probability in the interval \([W - \epsilon, W]\) and the knowledge that there is no RX characteristic with probability lower that \(W - \epsilon\).

Since the RX differences of the round keys are necessary to propagate the RX differences through the encryption function, a pair of characteristics is actually considered: one for the key-schedule and one for the encryption, and define the encryption characteristic in terms on the key-schedule characteristic. The weight related to the key-schedule characteristic will be denoted by \(W_K\) and the weight related to the encryption characteristic will be denoted by \(W_E\) This modification was done in [18], where first \(W_E\) was minimized with \(W_K\) fixed to a high value and then \(W_K\) was minimized with \(W_E\) fixed to \(W_E\).

An important step of this technique is the formulation of the decision problem of whether there exists a pair of RX characteristic up to a certain pair of weights \((W_K, W_E)\) as an SMT problem. The operations of an ARX cipher are performed on \(n\)-bit vectors, whereas the formula of a SAT problem can only contains boolean variables and the operations AND, NOT and OR. Therefore, an SMT problem in the bit-vector theory, which supports bit-vectors variables and the usual operations of bit-vectors, is used instead. Once the SMT problem is written, an SMT solver translates it into a SAT problem and
solves it using a SAT solver. An example of an SMT solver is STP [40]. STP is built from a SAT solver and was used in [12], [18].

The SMT problem is written as follows:

- For every pair of n-bit inputs words of the key schedule and the encryption, an n-bit vector is used to represent the RX difference of such pair.
- Additional n-bit vectors are used to represent the RX difference after the addition, XOR and rotation operation when required.
- For every XOR and every bit rotation of the ARX block cipher, the equations of Lemma II.1 are used to propagate properly the RX differences.
- For every modular addition of the ARX block cipher, Equations (1) and (2) are used to ensure that the RX differences are propagated with non-zero probability and Equation (4) is used to calculate the weight of the modular addition.
- Finally, two constraints are used to ensure that the sum of the weights related to the modular additions of the key schedule (encryption) is at most $W_K (W_E)$.

### IV. The ArxPy Tool

ArxPy is a tool for finding optimal rotational-XOR characteristics of ARX block ciphers. Basically, ArxPy takes a Python implementation of an ARX block cipher as input and applies the SAT-based method described in Section III-B. ArxPy is a generalization of Ashur, De Witte and Liu’s tool [18] for finding optimal RX characteristics in Speck.

Most of the automated tools [18], [20]–[22] searching for characteristics with high probability are implemented specifically for a particular cipher or for a small set. In order to use such tools with an arbitrary cipher, a significant effort is required.

Given a Python implementation of an ARX block cipher, ArxPy is executed with a simple shell command. Therefore, the only effort to use ArxPy is performing the ARX block cipher in Python, which is negligible due to the low development time and code complexity of the Python programming language. On top of that, ArxPy is open source and has a modular architecture. Therefore, it can be easily adapted for specific needs.

ArxPy expects a certain structure in the Python implementation of an ARX block cipher. The first part of this section will explain this structure. Then, the shell command to run ArxPy and the implementation will be described.

#### A. Structure of Python implementations of ARX block ciphers

A Python implementation of an ARX block cipher following the structure required by ArxPy will be called an ARX implementation. Any iterated ARX block cipher can be considered, as long as all its operations are performed on words of the same size.

A minimal ARX implementation contains a global variable, wordsize, and two functions, key_schedule and encryption. The global variable wordsize contains the word size (in bits) of the ARX block cipher.

```python
wordsize = 16
number_of_rounds = 22
alpha = 7
beta = 2

# round function
def f(x, y, k):
    x = ((x >> alpha) + y) - k
    y = (y << beta) - x
    return x, y

# key schedule
def key_schedule(l2, l1, l0, k0):
    l = [None for i in range(number_of_rounds + 3)]
    l[0:3] = [l0, l1, l2]
    for i in range(number_of_rounds + 3):
        round_keys[i] = [k0, l0, l1, l2]
        for i in range(number_of_rounds - 1):
            round_keys[i] = f(l[i], round_keys[i], i)
        round_output = f(x, y, round_keys[i])
        if i < number_of_rounds - 1:
            round_output = f(x, y, round_keys[i])
        else:
            return round_output

# encryption
def encryption(x0, y0):
    for i in range(1, number_of_rounds):
        rounds[i] = round_output = f(x, y, round_keys[i])

fig. 1. An ARX implementation of Speck32.
```

The function key_schedule implements the key scheduling algorithm of the ARX block cipher. This function has $m$ arguments representing the $m$ words of the key. It has no return value; the round keys are stored into the list-like object round_keys.

The function encryption implements the encryption algorithm of the ARX block cipher. The arguments of this function represent the words of the plaintext. The output of each round is stored in the list-like object rounds, except the last one, that is used as the return value. This function can obtain the round keys from the list-like object round_keys.

In order to implement the encryption and the key schedule, the Python operators $\oplus$, $\gg$, $\ll$ and $\circ$ are used as the modular addition, right and left rotation and XOR, respectively. Augmented assignments, such as $+= \text{ or } -= \text{ are not allowed.}$

Apart from the variable wordsize and the functions key_schedule and encryption, additional variables and functions can be defined to improve the readability and the modularity of the implementation. Figure 1 contains an ARX implementation of Speck with 32-bit block size (Speck32).

There are several considerations about the list-like objects rounds and round_keys:

- They are not created or declared in the ARX implementation. They are created by the parser of ArxPy.
- They only support the operator [] to access their elements (slices and negative indices are not supported).
- They can store either single values or lists of values, but each position can only be assigned once. Furthermore, storing a list of values must be done with one assignment and not element-wise.

Apart from the functions encryption and key_schedule, an ARX implementation contains two
Fig. 2. A test function and a fix_differences function for SPECK32.

more special functions: the function test and the function fix_differences.

Test vectors can be added to an ARX implementation by using the Python statement assert inside the function test. On the other hand, it is possible to fix the RX differences of the round keys and the outputs of each round by using the method fix_difference of round_keys and rounds inside the function fix_differences. Figure 2 shows an example of these functions for SPECK32, where the RX differences of the round keys are fixed to 0.

B. Running the program

ARXPy uses the Python library SymPy [43] and the SMT solver STP. They, together with Python3, must be installed in order to execute ARXPy.

The shell command to run ARXPy is the following:

```
python3 arxpy.py <ARX_implementation> <output>
```

where <ARX_implementation> is the name of the file containing an ARX implementation and <output> is the name of the file where the output will be written.

To find the optimal RX characteristic, ARXPy searches for characteristics up to many weights. During an execution of ARXPy, these intermediate characteristics are written in the output file. After the execution finishes, the last characteristic in the output file is the optimal one.

ARXPy actually searches for a pair of characteristics: the encryption characteristic, which contains the RX differences of the output of each round, and the key schedule characteristic, which contains the RX differences of the round keys. When a pair of characteristics is found by ARXPy, their RX differences are printed in the output file, along with information about their probabilities.

C. Implementation

ARXPy have been implemented in three modules: the ARX block cipher parser, the SMT writer and the characteristic finder. The parser module has been written from scratch and the other two modules are based on the tool proposed in [18]. This section explains briefly these three modules.

1) ARX block cipher parser: This module takes an ARX implementation and generates symbolic expressions of the output values of each round and the round keys. This is done by modified dynamically the source code of the ARX implementation and executing symbolically the functions key_schedule and encryption.

2) SMT writer: The writer module takes the symbolic expressions generated by the parser module and writes the SMT problem. The SMT problem is written in the SMT-LIB v2 [44] language, an input format supported by many SMT solvers such as STP, the SMT solver used by ARXPy.

3) Characteristic finder module: The third module implements the search strategy to find the optimal RX characteristic. A binary search strategy, based on [18], is used to minimize the weight of the characteristic.

V. Conclusion

Recently, several ARX block ciphers have been proposed, along with cryptanalytic techniques to analyse their security. ARXPy tool was developed to provide a tool to speed up and make easier the security evaluation of these type of ciphers.

This tool automatizes the search for optimal RX characteristics in ARX block ciphers. Given a Python implementation of an ARX block cipher, ARXPy finds an optimal RX characteristic with a simple shell command. Therefore, ARXPy can be used with any ARX block cipher with minimal effort.

On top of that, ARXPy can be easily adapted thanks to its modular architecture. For example, another SMT solver which supports the SMT-LIB v2 language can be used instead of STP just by modifying a few lines of code. As another example, a different search strategy can be considered by modifying only the characteristic finder module.

The generation of the SMT problems from an ARX implementation is done in a few seconds. However, a high-end computer is necessary to solve the SMT problems in a reasonable time. In our experiments with a standard PC, STP was only able to solve in reasonable time the SMT problems where the number of rounds was reduced significantly. For this reason, and due to the time constrains, we did not focus on evaluating a particular ARX block cipher with ARXPy.

A. Future Work

There are several ideas to continue this work. One option could be to adapt ARXPy for other types of cryptanalysis. Furthermore, ARXPy could be extended to accept software implementations in other languages. Since many authors of ARX block ciphers [6], [8], [9] draw a diagram to illustrate their ciphers, another possibility would be to use a data flow diagram as input

Another option could be to improve ARXPy by finding all the characteristics that share the input RX difference and the output RX difference of the optimal characteristic. This can be done with a SAT solver, as shown in [42], and the sum of the probabilities of all these characteristics provides a better estimation on the success probability.

ARXPy searches for RX characteristics where the rotational offset is fixed to 1 and the constants are injected through XOR. Therefore, future research could extend rotational-XOR cryptanalysis to cases where the rotational offset differs from 1, and to cases where the constants are injected through modular addition.
Bibliography


