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

Adrián Ranea[1,2], Yunwen Liu[1,3], Tomer Ashur[1]

firstname.lastname@esat.kuleuven.be

[1] imec-COSIC, K.U. Leuven, Leuven, Belgium
[2] Universitat Politècnica de Catalunya, Barcelona, Spain
[3] College of Science, National University of Defense Technology, Changsha, China

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. In this paper, a computer tool to speed up and make easier the security evaluation of ARX block ciphers against rotational-XOR cryptanalysis is shown. 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.

Keywords: ARX, rotational-XOR cryptanalysis, automatic search, ArxPy

1 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 [16].

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 [10]. Therefore, there is a high demand to implement cryptographic algorithms in these devices.

However, some of these new devices have so extreme constraints 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 [23], or SPECK [4]. ARX primitives are among the best performers in software [11].

Differential cryptanalysis [6], one of the most powerful attack against block ciphers, have been applied to several ARX primitives [7, 24, 27]. 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 [18]. Rotational cryptanalysis was improved in [19], where the propagation of rotational pairs through a sequence of modular additions, 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 [18] and formalized as rotational-XOR cryptanalysis in [1]. 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 [24], 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 [2] to attack SPECK.

In this paper, 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, generates a set of SAT/SMT problems automatically, and finds an optimal characteristic by using a SAT/SMT solver. This tool saves cipher designers the trouble of learning how to write the SAT problem from scratch, the only thing they need to do is to plug in a Python implementation of the cipher.

The rest of this paper is organized as follows: in Sect. 2, ARX block ciphers and rotational-XOR cryptanalysis are introduced. In Sect. 3, a SAT-based method for finding optimal rotational-XOR characteristic is described. The idea and process of the tool we have developed is presented in Sect. 4, along with its usage and implementation. Lastly, Sect. 5 concludes this paper.

2 Preliminaries

2.1 Notations

The notations used in this paper is shown in Table 1.

2.2 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.

Table 1. Notations used through the paper

$\boxplus$	Modular addition.
$\oplus$	XOR.
$\lll$	Left rotation.
$\ggg$	Right rotation.
X	An n-bit boolean vector.
$SHL(X)$	A non-cyclic left shift of X by one bit.
$(I \oplus SHL)(X)$	$X \oplus SHL(X)$.
$1_{X \preceq Y}$	A characteristic function which evaluates to 1 if $X_i \leq Y_i$ for all i.
$X \vert Y$	The vector bitwise OR operation.
$\vert X \vert$	The hamming weight of X.
$L(X)$	The $n-1$ most significant bits of X.

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 more recent; it was proposed in 2009 by Weinmann [30].

Many ARX block ciphers have been proposed since 2009. Some examples are: LEA [17], the underlying permutation for the message authentication code Chaskey [23], SPARX [12], SPECK [4], and the underlying block cipher Threefish of the hash function Skein [13].

ARX block ciphers are among the best performers in software. In a comparison of software implementations of block ciphers for small processors [11], 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 [11], the implementations with the smallest code size were achieved by ARX block ciphers.

In [18], 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 [18]. Without the XOR, an AR block cipher needs a larger number of rounds than an ARX block cipher for the same level of security [25].

2.3 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 [1]. This technique studies the propagation of *rotational-XOR (RX) differences* through the encryption function of ARX block ciphers. First, some notation is introduced.

Definitions and Notation. A pair of variables (X, X') has RX difference (α, γ) if

$$X \oplus (X' \lll \gamma) = \alpha.$$

To simplify the notation, the rotational offset γ will be fixed, $\overleftarrow{X}$ will denote $X \lll \gamma$ and $\overrightarrow{X}$ will denote $X \ggg \gamma$. In this case, we will say that the RX difference of a pair (X, X') is α if $X \oplus \overleftarrow{X'} = \alpha$.

If $\gamma = 0$, an RX difference is just an XOR-difference and if $\alpha = 0$, the pair $(X, X') = (X, \overrightarrow{X})$ 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 α of a plaintext pair (P, P') is called an *input RX difference.* The *output RX difference* is the RX difference β of its corresponding ciphertext $(C, C') = (E_k(P), E_{k^*}(P))$, where E_k denotes the encryption function using key k. An *RX characteristic* is a sequence of intermediate RX differences along with the input RX difference and RX output difference.

Rotational-XOR Attack. Rotational-XOR cryptanalysis is a distinguishing attack that exploits input RX differences which produce output RX differences with a probability higher than for a random permutation. To the best of our knowledge, the only application of rotational-XOR cryptanalysis (as described in [1]) is a distinguishing attack on SPECK [1, 2].

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 Sect. 3 to describe a method to search for optimal RX characteristics.

Propagation of RX Differences. The propagation of RX differences through rotation or XOR is deterministic. Table 2 contains the resulting RX differences after rotation or XOR given two pairs (X, X') and (Y, Y') with RX differences α and β respectively and a constant c.

Table 2. Propagation of RX differences through rotation and XOR

Pair	RX difference
$((X \ggg c), (X' \ggg c))$	$(\alpha \ggg c, \gamma)$
$((X \lll c), (X' \lll c))$	$(\alpha \lll c, \gamma)$
$((X \oplus Y), (X' \oplus Y'))$	$(\alpha \oplus \beta, \gamma)$
$((X \oplus c), (X' \oplus c))$	$(\alpha \oplus c \oplus (c \lll \gamma), \gamma)$

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

Theorem 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, X'), (Y, Y') and (Z, Z') be three pairs with RX differences α, β and ζ respectively and with rotational offset $\gamma = 1$. Then,*

$$
\begin{aligned}
&P[(X \boxplus Y, X' \boxplus Y') = (Z, Z')] \\
&\quad = 1_{(I \oplus SHL)(\delta_\alpha \oplus \delta_\beta \oplus \delta_\zeta) \oplus 1 \preceq SHL((\delta_\alpha \oplus \delta_\zeta) | (\delta_\beta \oplus \delta_\zeta))} \times 2^{-|SHL((\delta_\alpha \oplus \delta_\zeta) | (\delta_\beta \oplus \delta_\zeta))|} \times 2^{-3} \\
&\quad + 1_{(I \oplus SHL)(\delta_\alpha \oplus \delta_\beta \oplus \delta_\zeta) \preceq SHL((\delta_\alpha \oplus \delta_\zeta) | (\delta_\beta \oplus \delta_\zeta))} \times 2^{-|SHL((\delta_\alpha \oplus \delta_\zeta) | (\delta_\beta \oplus \delta_\zeta))|} \times 2^{-1.415}
\end{aligned}
$$

where $\delta_\alpha = L(\alpha), \delta_\beta = L(\beta)$ and $\delta_\zeta = L(\zeta)$. 这式子。。。。。怎么来的？？？

In the case that $\alpha = \beta = \zeta = 0$, Theorem 1 predicts the probability that the rotational property is preserved after a modular addition $P(\overrightarrow{X \boxplus Y} = \overrightarrow{X} \boxplus \overrightarrow{Y})$.

3 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 [7, 14, 24]. The technique proposed by Mouha et al. [24] has the advantage that it finds the optimal characteristics. In [24], it was applied to search for differential characteristics of the ARX stream cipher Salsa20 [5]. An improvement of this technique was considered in [27] to search for differential characteristics of Speck and LEA. Recently, it was applied to search for RX characteristics of Speck [2].

Basically, this technique consists of rewriting the search problem as a SAT (Boolean Satisfiability) problem. A similar approach was taken in [14], 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.

3.1 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 AND (NOT b) is satisfiable since the assignment (a, b) = (TRUE, 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 [9], 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 [31].

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 [8, 15] is to translate the SMT instance to a SAT instance and solve using a SAT solver.

3.2 A SAT-based Method for Searching Optimal Rotational-XOR Characteristics

In [2], a SAT solver was used for finding optimal characteristics of the ARX block cipher Speck. In this section, the method used in [2] will be explained in detail. First, a similar notation to [24] is introduced.

A triplet of RX differences (α, β, ζ) is *valid* if α and β propagate to ζ after a modular addition with non-zero probability, that is, $P(\alpha, \beta \xrightarrow{\boxplus} \zeta) \neq 0$. Using theorem 1, the triplet (α, β, ζ) is valid if and only if one of the following conditions holds

$$(I \oplus SHL)(\delta_\alpha \oplus \delta_\beta \oplus \delta_\zeta) \oplus 1 \preceq SHL((\delta_\alpha \oplus \delta_\zeta)|(\delta_\beta \oplus \delta_\zeta)) \tag{1}$$
$$(I \oplus SHL)(\delta_\alpha \oplus \delta_\beta \oplus \delta_\zeta) \preceq SHL((\delta_\alpha \oplus \delta_\zeta)|(\delta_\beta \oplus \delta_\zeta)). \tag{2}$$

The *weight ω of a valid triplet* (α, β, ζ) is defined as

$$\omega(\alpha, \beta, \zeta) = -\log_2(P(\alpha, \beta \xrightarrow{\boxplus} \zeta)).$$

Using Theorem 1, the weight can be calculated as follows

$$\omega(\alpha, \beta, \zeta) = \begin{cases} |SHL((\delta_\alpha \oplus \delta_\zeta)|(\delta_\beta \oplus \delta_\zeta))| + 3, & \text{if (1) holds} \\ |SHL((\delta_\alpha \oplus \delta_\zeta)|(\delta_\beta \oplus \delta_\zeta))| + 1.415, & \text{if (2) holds} \end{cases} \tag{3}$$

The *weight W of an RX characteristic* is defined as the sum of the weights of each of its modular additions. As in [24], 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 this problem is satisfiable, a lower weight is chosen. Otherwise, a higher weight is chosen. This is repeated until the *minimal weight* is found, that is, a weight $\widehat{W}$ such that there exists an RX characteristic up to weight $\widehat{W}$ but not up to weight $\widehat{W} - \epsilon$, where ϵ is a value fixed from the start (usually $\epsilon = 1$ [2, 24]).

Different strategies have been used to obtain the minimal weight. In [24], the search starts with a low weight and it is incremented by one until a satisfiable formula is found. In [20], the search starts with a high weight and it is decremented by one until an unsatisfiable formula is found. In [2], 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 $\widehat{W}$, an RX characteristic with probability in the interval $(\widehat{W} - \epsilon, \widehat{W}]$ and the knowledge that there is no RX characteristic with probability lower that $\widehat{W} - \epsilon$.

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. The RX differences predicted by the key-schedule characteristic are used in the encryption characteristic for the RX differences of the round keys. Similar to [2], the weight related to the encryption characteristic is minimized first, and then the weight related to the key-schedule characteristic is minimized.

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 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 [15]. STP is built from a SAT solver and was used in [2, 24].

The SMT problem is written as follows:

- For every pair of n-bit input 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, Table 2 is used to propagate properly the RX differences.
- For every modular addition of the ARX block cipher, (1) and (2) are used to ensure that the RX differences are propagated with non-zero probability and (3) 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).

4 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 Sect. 3.2. ArxPy is a generalization of [2] for finding optimal RX characteristics in Speck.

To the best of our knowledge all automated tools [2, 21, 28, 29] 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.

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.

4.1 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, `wordsize`, and two functions, `key_schedule` and `encryption`. The global variable `wordsize` contains the word size (in bits) of the ARX block cipher.

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 `+`, `>>`, `<<` and `^` are used as the modular addition, right and left rotation and XOR, respectively. Augmented assignments , such as `+=` or `^=`, 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 objects `rounds` and `round_keys`:

```
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

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
```

Fig. 1. An ARX implementation of SPECK32. The total length of the required code is 23 lines.

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

```
def test():
    key = (0x1918, 0x1110, 0x0908, 0x0100)
    plaintext = (0x6574, 0x694c)
    ciphertext = (0xa868, 0x42f2)

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

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

Fig. 2. A `test` function and a `fix_differences` function for SPECK32

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.

4.2 Running The Program

ARXPY uses the Python library SymPy [22] 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 an optimal RX characteristic, ARXPY searches for characteristics up to many weights. During an execution of ARXPY, these intermediate characteristics are written to 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.

4.3 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 [2]. This section explains briefly these three modules.

The parser module takes an *ARX implementation* and generates symbolic expressions of the output values of each round and the round keys. This is done by modifying the source code of the ARX implementation dynamically and executing the functions `key_schedule` and `encryption` symbolically. The Abstract Syntax Tree (AST) of the ARX implementation is used to modify the source code dynamically, whereas SymPy, a Python library for symbolic mathematics, is used to generate and handle the symbolic expressions.

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 [3] language, an input format 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 Sect. 3.2. 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.

The finder module implements the search strategy to find the optimal RX characteristic. A binary search strategy, based on [2], is used to minimize the weight of the characteristic.

5 Conclusion

Recently, several ARX block ciphers have been proposed, along with cryptanalytic techniques to analyse their security. The ArxPy tool was developed to provide a way to speed up and make easier the security evaluation of these 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 SMT problems from an ARX implementation is done in a few seconds. However, a high-end computer is required to solve the SMT problems in a reasonable time.

5.1 Future Work

There are several ideas to continue this work. One option could be to adapt ArxPy for other cryptanalytic techniques. Furthermore, ArxPy could be extended to accept software implementations in other languages. Since many au-

thors of ARX block ciphers [4, 12, 17] 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 [20], and the sum of the probabilities of all these characteristics provides a better estimation on the success probability.

References

1. T. Ashur and Y. Liu, *Rotational Cryptanalysis in the Presence of Constants*, IACR Transactions on Symmetric Cryptology, **1** (2017).
2. T. Ashur, G. D. Witte, and Y. Liu, *An Automated Tool for Rotational-XOR Cryptanalysis of ARX-based Primitives*, in Proceedings of the 38th Symposium on Information Theory in the Benelux, Delft, NL, 2017, Werkgemeenschap voor Informatie- en Communicatietheorie.
3. C. Barrett, A. Stump, and C. Tinelli, *The SMT-LIB Standard: Version 2.0*, tech. rep., Department of Computer Science, The University of Iowa, 2010. `www.SMT-LIB.org`.
4. R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers, *The SIMON and SPECK Lightweight Block Ciphers*, in Proceedings of the 52Nd Annual Design Automation Conference, DAC '15, New York, NY, USA, 2015, ACM.
5. D. J. Bernstein, *The Salsa20 Family of Stream Ciphers*, in New Stream Cipher Designs, M. Robshaw and O. Billet, eds., no. 4986 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2008.
6. E. Biham and A. Shamir, *Differential Cryptanalysis of DES-like Cryptosystems*, in Advances in Cryptology-CRYPT0' 90, Springer, Berlin, Heidelberg, 1990.
7. A. Biryukov and V. Velichkov, *Automatic Search for Differential Trails in ARX Ciphers*, in Topics in Cryptology – CT-RSA 2014, Springer, Cham, 2014.
8. R. Brummayer and A. Biere, *Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays*, in Tools and Algorithms for the Construction and Analysis of Systems, Springer, Berlin, Heidelberg, 2009.
9. S. A. Cook, *The Complexity of Theorem-proving Procedures*, in Proceedings of the Third Annual ACM Symposium on Theory of Computing, STOC '71, New York, NY, USA, 1971, ACM.
10. D. V. Dimitrov, *Medical Internet of Things and Big Data in Healthcare*, Healthc Inform Res, **22** (2016).
11. D. Dinu, Y. L. Corre, D. Khovratovich, L. Perrin, J. Grossschädl, and A. Biryukov, *Triathlon of lightweight block ciphers for the internet of things*. Cryptology ePrint Archive, Report 2015/209, 2015.
12. D. Dinu, L. Perrin, A. Udovenko, V. Velichkov, J. Grossschädl, and A. Biryukov, *Design Strategies for ARX with Provable Bounds: Sparx and LAX*, in Advances in Cryptology – ASIACRYPT 2016, Springer, Berlin, Heidelberg, 2016.
13. N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker, *The skein hash function family*, Submission to NIST (round 3), (2010).

14. K. Fu, M. Wang, Y. Guo, S. Sun, and L. Hu, *MILP-Based Automatic Search Algorithms for Differential and Linear Trails for Speck*, in Fast Software Encryption, Springer, Berlin, Heidelberg, 2016.
15. V. Ganesh and D. L. Dill, *A Decision Procedure for Bit-vectors and Arrays*, in Proceedings of the 19th International Conference on Computer Aided Verification, CAV'07, Berlin, Heidelberg, 2007, Springer-Verlag.
16. Gartner, Inc., *Gartner Says 6.4 Billion Connected "Things" Will Be in Use in 2016, Up 30 Percent From 2015*, 2015. https://www.gartner.com/newsroom/id/3165317.
17. D. Hong, J.-K. Lee, D.-C. Kim, D. Kwon, K. H. Ryu, and D.-G. Lee, *LEA: A 128-Bit Block Cipher for Fast Encryption on Common Processors*, in Information Security Applications, Springer, Cham, 2013.
18. D. Khovratovich and I. Nikolić, *Rotational Cryptanalysis of ARX*, in Fast Software Encryption, Springer, Berlin, Heidelberg, 2010.
19. D. Khovratovich, I. Nikolić, J. Pieprzyk, P. Sokołowski, and R. Steinfeld, *Rotational Cryptanalysis of ARX Revisited*, in Fast Software Encryption, Springer, Berlin, Heidelberg, 2015.
20. S. Kölbl, G. Leander, and T. Tiessen, *Observations on the SIMON Block Cipher Family*, in Advances in Cryptology – CRYPTO 2015, Springer, Berlin, Heidelberg, 2015.
21. G. Leurent, *ARXtools: A toolkit for ARX analysis*, 2012. https://who.rocq.inria.fr/Gaetan.Leurent/arxtools.html.
22. A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R. Terrel, v. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, and A. Scopatz, *Sympy: symbolic computing in python*, PeerJ Computer Science, **3** (2017).
23. N. Mouha, B. Mennink, A. V. Herrewege, D. Watanabe, B. Preneel, and I. Verbauwhede, *Chaskey: An Efficient MAC Algorithm for 32-bit Microcontrollers*, in Selected Areas in Cryptography – SAC 2014, Springer, Cham, 2014.
24. N. Mouha and B. Preneel, *Towards Finding Optimal Differential Characteristics for ARX: Application to Salsa20*, Tech. Rep. 328, 2013.
25. S. Paul and B. Preneel, *Solving Systems of Differential Equations of Addition*, in Information Security and Privacy, Springer, Berlin, Heidelberg, 2005.
26. A. Shimizu and S. Miyaguchi, *Fast Data Encipherment Algorithm FEAL*, in Proceedings of the 6th Annual International Conference on Theory and Application of Cryptographic Techniques, EUROCRYPT'87, Berlin, Heidelberg, 1988, Springer-Verlag.
27. L. Song, Z. Huang, and Q. Yang, *Automatic Differential Analysis of ARX Block Ciphers with Application to SPECK and LEA*, in Information Security and Privacy, Springer, Cham, 2016.
28. Stefan Kölbl, *CryptoSMT: An easy to use tool for cryptanalysis of symmetric primitives.* https://github.com/kste/cryptosmt.
29. V. Y. Velichkov, *YAARX: Yet another toolkit for the analysis of ARX cryptographic algorithms*, 2016. https://github.com/vesselinux/yaarx.
30. R. Weinmann, *AXR - Crypto made from modular additions, XORs and word rotations*, Dagstuhl Seminar 09031, Dagstuhl, Germany, 2009.
31. L. Zhang and S. Malik, *The Quest for Efficient Boolean Satisfiability Solvers*, in Automated Deduction—CADE-18, Springer, Berlin, Heidelberg, 2002.