SCrypt: Using Symbolic Computation to Bridge the Gap Between Algebra and Cryptography

Burçin Eröcal

Abstract. The first step in algebraic cryptanalysis is to obtain a system of polynomial equations representing a cryptosystem. Even though many different approaches for this stage have been proposed, it still remains to be a considerable obstacle. We describe the software package SCrypt, which automates this tedious step. The task of describing the structure of a cryptosystem is separated from modeling the components algebraically. Symbolic computation is used in this process to delay attaching algebraic models to components, and allowing the user to experiment with different algebraic representations of cipher components.

Keywords. symbolic computation, cryptography, algebraic cryptanalysis.

1. Introduction

The first step in the algebraic cryptanalysis of a cryptosystem is expressing it as a system of polynomial equations over a suitable finite field, where the key, plaintext, cipher-text and internal states of the cipher are variables in the polynomial ring. Then the problem of cryptanalysis is reduced to finding solutions to this system.

Even though algebraic cryptanalysis has been a popular topic in cryptography for many years and many approaches to finding algebraic representations of cryptosystems have been explored, there are only a few example cryptosystems available readily with their algebraic representation. Some examples of such systems are the implementation in Sage [11] of the small scale variants of the AES [5] by Martin Albrecht, and the systems available on Nicolas Courtois’ web page [7].

Starting from specifications of cryptosystems and obtaining systems of polynomial equations for algebraic cryptanalysis requires considerable effort. This leaves many questions in this field open, such as the effectiveness of the different approaches proposed for the passage from specifications to algebraic representations, or the performance of different methods of solving the generated systems.
The software SCrypt implements a novel approach to automate this first step in algebraic cryptanalysis by separating the steps of describing the structure of the cryptosystem, and using this description with choices made by the cryptographer to obtain a system of polynomial equations representing the cryptosystem.

SCrypt includes definitions of commonly used components in symmetric ciphers, such as addition modulo a power of 2, or multiplication in a finite field. These components can generate algebraic representations of themselves automatically over different domains specified by the user. Structural constructs such as block ciphers, stream ciphers, and hash functions are also available, allowing the user to express the structure of a cryptosystem only in a few lines.

We believe SCrypt has the potential to become a knowledge base of results in the field of algebraic cryptanalysis, and a platform for checking the vulnerability of cryptosystems against classical attacks, as well as a valuable tool for learning and experimenting with cryptosystem design.

The implementation of SCrypt will be made available under the GPL license. SCrypt is built on the algebraic framework provided by the Sage computer algebra system [11]. The symbolic computation engine uses the PolyBoRi framework [3] in the background. Computations over multivariate polynomial rings in Sage is performed using Singular [9].

This paper is intended to be a short introduction to the operation of SCrypt through examples. In the next section we present the use of symbolic expressions, followed by a demonstration of algebraic representation capabilities of SCrypt. Then we present an example definition of a cryptosystem.

2. Symbolic Expressions

In order to allow the user to choose between different algebraic representations of cipher components, and switch between them freely, SCrypt computes the internal states of a cryptosystem as symbolic expressions first.

The symbolic computation engine tries to make the least possible assumptions about the algebraic structure of the system. This allows one to separate the steps of describing the structure of the cryptosystem, and modeling it algebraically. Once the general structure of the system is expressed, using the components provided by SCrypt, the passage to an algebraic representation is possible. At this stage, the user can make choices between different base fields for the polynomial ring, such as \( \mathbb{F}_2 \) or \( \mathbb{F}_{16} \), or different ways of modeling certain components.

Symbolic computations are performed in a ring of characteristic 2, where each variable represents a vector of bits in the cipher state. Addition and multiplication over this ring correspond to component-wise logical XOR and AND operations. Other operations commonly used in cryptosystem design, such as multiplication by constant over a given field, summation modulo a power of 2, application of an SBox is denoted as a symbolic function application.
Example. To obtain an example of a symbolic expression, we use the implementation of the MiniAES toy cipher [10] in SCrypt.

```
sage: import miniaes
sage: m = miniaes.MiniAES()
sage: m.calc_states()
sage: m.states[3][0]  
FElem_1 + k0 + FMul16(FElem_2, SBox(k3 + p3)) + FMul16(FElem_3, SBox(k0 + p0)) + SBox(k3)
```

The symbolic expression displayed on the last line uses the variables k0, k3, p3, constants FElem_1, FElem_2, FElem_3 and functions FMul16 and SBox. In this case the variables denote 4-bits each. These can be viewed as elements in \( \mathbb{F}_{16} \), or a different suitable structure when generating polynomial equations. The expression k0 + p0 corresponds to the bitwise xor (or addition in \( \mathbb{F}_{16} \)) of values denoted by k0 and p0. In this example, SBox is a unary function, since the specification of MiniAES makes it convenient to denote it as such. This expression includes two applications of this function, namely SBox(k3 + p3) and SBox(k0 + p0). The function FMul16 denotes the multiplication over the field \( \mathbb{F}_{16} \), so it’s an n-ary function. We see it applied to two arguments in this example, in order to multiply the outputs of the SBox with constants.

In the case where a symbolic function has multiple outputs, i.e., when it is easier to model the functions acting on bits, such as the DES and CTC S-boxes, we introduce new variables into the system for each output of the function, and record the relation of these variables to the function in a symbolic relation.

Example. We use CTC to demonstrate the use of symbolic relations in SCrypt.

```
sage: import ctc
sage: c = ctc.CTC(1, 1)
sage: c.calc_states()
sage: c.states[2][0]  
regs: [k0 + p0, k1 + p1, k2 + p2]
rels: [(SBox(k0 + p0, k1 + p1, k2 + p2), [rr_0_0 , rr_0_1 , rr_0_2])]
```

The code segment above initializes a CTC instance, with 1 round and only 1 block and computes symbolic expressions for the internal states of the cipher. Then, we list the state after initial key addition.

```
sage: c.states[2]  
regs: [rr_0_0 + rr_0_1 , rr_0_1 + rr_0_2 , rr_0_0]  
rels: [(SBox(k0 + p0, k1 + p1, k2 + p2), [rr_0_0 , rr_0_1 , rr_0_2])]
```

This output corresponds to the internal state after applying the round function, which consists of the application of a substitution box and multiplying by a diffusion matrix. Note that the outputs of the function SBox are replaced by
variables $r_{r.0,0}$, $r_{r.0,1}$, $r_{r.0,2}$, and this relation is recorded in the $rels$ field of the state.

3. Algebraic Representation

SCrypt uses the symbolic expressions representing the states of a cipher to generate systems of polynomial equations which can be used in algebraic cryptanalysis. This process is controlled by parameters specified by the user. Hence, making it easy to generate and experiment with different algebraic representations of a cryptosystem.

Example. We use the MiniAES object created in the previous example to demonstrate equation generation capabilities of SCrypt.

In order to start the process of generating a system of polynomial equations, the user should create an $EqGen$ object, specifying the state which will be modelled algebraically, number of bits each symbolic variable represents, and a base field for the polynomial ring.

```
sage: import scrypt.eqgen
sage: eq = scrypt.eqgen.EqGen(m.states[-1], 4, base_field=GF(2))
```

This will allow the system to associate variables in the polynomial ring with the symbolic variables of the cipher state. Then, it is straightforward to calculate polynomials for simple operations in the symbolic ring.

Example. We use the $eq$ object created above to get polynomials for the symbolic expression $p_0 + k_0$.

```
sage: t = eq.sringel_to_polys(p0 + k0); t
[p_0_0 + k_0_0, p_0_1 + k_0_1, p_0_2 + k_0_2, p_0_3 + k_0_3]
```

The generated polynomials lie in a multivariate polynomial ring over $\mathbb{F}_2$.

```
sage: t[0].parent()
Multivariate Polynomial Ring in p_2_0, p_2_1, p_2_2, p_2_3, p_3_0, p_3_1, p_3_2, p_3_3, p_0_0, p_0_1, p_0_2, p_0_3, p_1_0, p_1_1, p_1_2, p_1_3, k_3_0, k_3_1, k_3_2, k_3_3, k_2_0, k_2_1, k_2_2, k_2_3, k_1_0, k_1_1, k_1_2, k_1_3, k_0_0, k_0_1, k_0_2, k_0_3, o_0_0, o_0_1, o_0_2, o_0_3, o_1_0, o_1_1, o_1_2, o_1_3, o_2_0, o_2_1, o_2_2, o_2_3, o_3_0, o_3_1, o_3_2, o_3_3 over Finite Field of size 2
```

To obtain algebraic expressions from symbolic ones which contain symbolic function applications, each symbolic function needs to provide a way to get a system of polynomial equations modeling the relations between its input and output variables adequately in a multivariate polynomial ring over a given field. The components provided in SCrypt, such as multiplication in a finite field and addition modulo a constant already have this capability. However, user provided functions,
such as various substitution boxes, still need to provide their algebraic representations. It is planned to include implementations of standard methods of obtaining such presentations such as interpolation or ones described in [2] and [8] in the future.

Example. The built-in field multiplication component is used in implementation of MiniAES in SCrypt. The object \texttt{m.FMul16} is initialized to denote multiplication in \(\mathbb{F}_{16} \cong \mathbb{F}_2[x]/(x^4 + x + 1)\).

\begin{verbatim}
sage: eq. string1.to_polys(m.FMul16(m.FElem2, p0)+k0)
[p0_3 + k0_0, p0_0 + p0_3 + k0_1, p0_1 + k0_2, p0_2 + k0_3]
\end{verbatim}

Since \texttt{m.FElem2}, which represents a root of \(x^4 + x + 1\), is a constant, the result of the multiplication is simple. We can see the polynomial expressions representing the multiplication operation by using symbolic variables for both arguments of \texttt{FMul16}.

\begin{verbatim}
sage: eq. string1.to_polys(m.FMul16(k0, p0))
[p0_0*k0_0 + p0_3*k0_1 + p0_2*k0_2 + p0_1*k0_3,
p0_1*k0_0 + p0_0*k0_1 + p0_3*k0_1 + p0_2*k0_1 + p0_3*k0_2 + p0_1*k0_3 + p0_2*k0_3,
p0_2*k0_0 + p0_1*k0_1 + p0_0*k0_2 + p0_3*k0_2 + p0_2*k0_3 + p0_3*k0_3,
p0_3*k0_0 + p0_2*k0_1 + p0_1*k0_2 + p0_0*k0_3 + p0_3*k0_3]
\end{verbatim}

Note that these polynomials are computed automatically by SCrypt.

4. Examples

We demonstrate the computational model and concepts employed by SCrypt via the implementation of the toy block cipher CTC [1, 6].

A round of CTC contains two operations, application of S-boxes and a linear diffusion layer. In order to model this block cipher in SCrypt, we construct the round function and the key generation function. Then, we call the \texttt{BlockCipher} constructor with these parameters as well as number of rounds and the block size.

Assuming the symbolic function \texttt{SBox} is defined before, indicating that it takes 3 inputs and has 3 outputs, it is easy to create an operation which applies this function to blocks in the current cipher state.

\begin{verbatim}
self.SubBytes = ParallelOp('S', SBox)
\end{verbatim}

Note that we don’t need to specify how many times \texttt{SBox} will be applied, or what the input/output sizes are. These parameters are computed at runtime automatically, allowing the user to experiment with different sizes, without changing the cipher description.

The diffusion layer consists of a simple linear operation, which we model as matrix multiplication. The following lines construct a square matrix, using the
variable `self.num_blocks` which is again available at runtime as the dimension parameter.

```python
mat = matrix(ZZ, self.num_blocks)
for j in range(1, self.num_blocks):
    mat[(j*1987 + 257) % self.num_blocks, 0] = 1
    mat[(j*1987 + 257) % self.num_blocks, (j + 137)%self.num_blocks] = 1
self.Diffusion = MatrixOp('Diffusion', mat)
```

The last line above uses the `MatrixOp` operation in SCrypt to create a symbolic operation which multiplies the current cipher state with the given matrix, named `Diffusion`.

In order to define the round function using the `SubBytes` and `ParallelOp` operation we defined, we create a chain of operations as follows. This makes the object `CTCRound` to take an input of `self.num_blocks` blocks, and apply the operations `SubBytes` and `Diffusion` in the given order.

```python
CTCRound = OpChain(self.num_blocks)
CTCRound.chain_op(self.SubBytes)
CTCRound.chain_op(self.Diffusion)
```

After these definitions, all we need to do to obtain the CTC block cipher is to call the initializer for the class `BlockCipher` with the functions we created earlier.

```python
BlockCipher.__init__ (self, CTCRound, None, self.num_blocks, self.num_blocks, 1, num_rounds)
```

Note that the chain of operations construct `OpChain` provided by SCrypt handles varying block sizes as well. In this model, defining the general structure of a Feistel cipher becomes as simple as the following lines:

```python
ApplyF = BinaryOp('ApplyF', 2, 0, 1, 0, func2=F)
HalfRound = OpChain(2, adjust_blocks=True)
HalfRound.chain_op(ApplyF)
HalfRound.chain_op(Swap)
```

We conclude with a listing of a sample session using the CTC implementation described above, to get a system of polynomials representing this cryptosystem.

```python
sage: import ctc
sage: c = ctc.CTC(2, 2)
sage: c.calc_states()
sage: import scrypt.eqgen
sage: eq = scrypt.eqgen.EqGen(c.states[-1], 1)
sage: eq.generate_system(GF(2))
sage: eq.pols
[k1_0\cdot k0_0 + k1_0\cdot p0_0 + k0_0\cdot p1_0 + p0_0\cdot p1_0 + k2_0 + k1_0 + k0_0 + r0_0\cdot 0_0 + p2_0 + p0_0 + p1_0 + 1.
```
5. Conclusion

We presented the software package SCrypt which separates the steps of describing the structure of a cryptosystem and algebraic representation of its components. This is achieved by using symbolic expressions to calculate internal states of the cryptosystem, and attaching algebraic structure to these expressions based on input from the user only when instructed to.

While SCrypt provides an adequate framework for defining different cryptosystems and experimenting with algebraic representations, there is still plenty of room for improvement.

SCrypt already includes implementations of the toy ciphers, SDES, Mini-AES, CTC and small scale variants of the AES, as well as compression functions of SHA-1 and SHA-2 family hash functions. We plan to add more examples and real world ciphers to this list. It is also planned to consider automating the choice of variable orderings based on the approach presented in [4]. Implementing algebraic approaches to test for applicability of classical cryptanalytic attacks such as differential and linear cryptanalysis is also planned.

References


Burçin Eröcal
Research Institute for Symbolic Computation
Johannes Kepler University
Altenbergerstrasse 69
A-4040 Linz, Austria
e-mail: burcin.erocal@risc.uni-linz.ac.at