Experimenting with Shuffle Block Cipher and SMT Solvers

Martin Stanek

Department of Computer Science
Comenius University
stanek@dcs.fmph.uniba.sk

Abstract

We experiment with the block cipher proposed by Hoang, Morris, and Rogaway [3], even though the cipher is insecure [8]. The cipher is based on swap-or-not shuffle, and we call it the Shuffle Block Cipher. We show how the cipher can be translated into SMT-LIB v2 format, suitable for automated solving by SMT solvers. We compare performance of various SMT solvers on the encryption and known plaintext attack problems.

Keywords: Shuffle, Cipher, Cryptanalysis, SMT solver.

1 Introduction

The swap-or-not shuffle [3] is a method of converting a pseudorandom function into a pseudorandom permutation. The shuffle can be used for small-domain and format-preserving encryptions [3, 5]. A construction of "standard" block cipher using this method was also proposed in [3]. We call this block cipher the Shuffle Block Cipher. Probably the most interesting feature of the cipher is its simplicity. The cipher iterates a large number of rounds, each consisting of single swap-or-not operation. However, the instantiation of the round functions proposed by the authors of the Shuffle Block Cipher in [3] is insecure, as pointed out by Vaudenay in CRYPTO 2012 Rump Session [8] (and independently by Gilbert). Nevertheless, we experiment with automated cryptanalysis of the Shuffle Block Cipher in this paper, using SMT solvers.

Our contribution can be summarized as follows:

- We show how the Shuffle Block Cipher can be translated into SMT-LIB v2 format, suitable for automated solving by SMT solvers.
- We use two problems – simple encryption of a plaintext and known plaintext attack – for comparison of four SMT solvers' performance (MathSAT5, Sonolar, Yices, and Z3). The best (performance-wise) SMT solver for these problems is Sonolar.
- We experiment with automated solving of KPA (known plaintext attack) problems with Sonolar. We try various combination of the parameters such as block length, round count, and the number of plaintext-ciphertext pairs. Most combinations are artificially small. Solving instances with larger values of the parameters with moderately large number of plaintext-ciphertext pairs seems unsolvable efficiently by the SMT solver.

We introduce the Shuffle Block Cipher and SMT solvers in Section 2. The translation of the cipher into SMT-LIB v2 format is presented in Section 3. We show the performance comparison of four SMT solvers on encryption and known plaintext attack problems in Section 4. Finally,

Section 5 describes the results of simple cryptanalysis for various combination of parameters. Certainly, cryptanalysis from [8] is the easiest way of attacking the Shuffle Block Cipher.

2 Preliminaries

2.1 The Shuffle Block Cipher

The Shuffle Block Cipher (further denoted as the SBC) is a block cipher with variable number of rounds and variable block length. We use the original notation from [3] in the description of the SBC. Let n be the block length and r be the number of rounds (both parameters are positive integers). The SBC uses a key KL that specifies $2r$ subkeys $K_1,\ldots,K_r,L_1,\ldots,L_r \in \{0,1\}^n$. We assume that these subkeys are randomly and uniformly chosen. Let $X \in \{0,1\}^n$ be an input block. Let $\oplus$ be a bitwise xor operator. An inner product of two n-bit vectors is denoted by $\odot$ symbol, i.e. $X \odot Y = X_1Y_1 \oplus \ldots \oplus X_nY_n$ for $X = (X_1,\ldots,X_n)$ and $Y = (Y_1,\ldots,Y_n)$. The SBC encryption is shown on Figure 1. The decryption takes the subkeys in reverse order.

$E_{KL}(X)$:
  for $i \leftarrow 1$ to r do
    $X' \leftarrow K_i \oplus X$
    $\hat{X} \leftarrow \max(X, X')$
    if $L_i \odot \hat{X} = 1$ then $X \leftarrow X'$
  return X

Figure 1: The SBC encryption algorithm

Remark. The inner product $L_i \odot \hat{X}$ was proposed in [3] as an instantiation of a random round function $F_i : \{0,1\}^n \to \{0,1\}$. In this instantiation the function is specified by vector L_i, i.e. $F_i(\hat{X}) = L_i \odot \hat{X}$.

The SBC uses a swap-or-not technique to construct pseudo-random permutation. Theoretical analysis shows that for general swap-or-not network, approximately $r = 6n$ rounds are necessary for a good bound on CCA-security. However, the authors state [3]:

> It is incorrect to think that the theoretical analysis suggests a value like $r = 6n$; for one thing, there is an enormous gap between computing a random round function $F_i(\hat{X})$ and an inner product $L_i \odot \hat{X}$.

The authors did not provide an explicit suggestion on r values. On the other hand, the SBC was proved to be insecure very soon – Vaudenay in CRYPTO 2012 Rump Session [8] (and independently Gilbert) pointed out that the round function is linear.

2.2 SMT Solvers

Satisfiability Modulo Theories (SMT) problems can be viewed as a generalization of SAT problems. Besides boolean variables, also non-binary variables, interpreted and uninterpreted functions and predicates are allowed. The types of variables and predicates depend on supported theories. There are various theories that can be supported by particular SMT solver, such as theory of bit-vectors, linear integer/real arithmetics, arrays, tuples etc. SMT solvers try to solve SMT problems – in

some cases by translating them to SAT problems and subsequently using a SAT solver, in other cases by employing specialized solvers for particular theory. SMT solvers are used in practice for software and hardware verification, type inference, static program analysis, test-case generation, scheduling and other applications [2]. Applications of SMT solvers for software security can be found in [7]. Even though there are some applications of SMT solvers for cryptanalysis, e.g. see [4], a direct use of SAT solvers is much more frequent.

The theory of bit-vectors is suitable for modeling the SBC. The theory consists of fixed-sized bit vector variables, functions for extracting individual bits from a vector, functions for addition, rotation, bitwise logical operators, comparison etc.

SMT-LIB version 2 is a standard format for specifying SMT problems [1]. Many SMT solvers more or less support SMT-LIB v2 format – it allows to compare their performance in our experiments. Therefore we use this format for modeling the SBC, instead of using the proprietary input languages or bindings/APIs of particular SMT solvers.

Test environment. All experiments were performed on 64-bit Ubuntu 14.04.1 with i7-2600 (3.40 GHz) processor and 4 GB memory. The following SMT solvers were used (64-bit versions): MathSAT5 5.2.12, Sonolar 2014-09-02, Yices 2.2.2, and Z3 4.3.2.

3 Modeling the Cipher

The encryption function of the SBC was modeled/encoded in SMT-LIB v2. Since the source code in SMT-LIB v2 format grows linearly with the number of rounds r and for each plaintext-ciphertext pair a separate encoding (with separate set of variables) must be produced, we generate the SMT-LIB v2 model with a help of a Python script. The SBC is very simple and therefore the translation into SMT-LIB v2 is straightforward. Let us go through all parts of the encoding. In order to keep the presentation compact we use very small values $n = 8$ and $r = 2$ in this description. The bit-vector theory in SMT-LIB v2 contains operations/predicates that come in handy: bvxor (bitwise xor), bvand (bitwise and), bvult (unsigned less than), extract (extract bits from a bitvector), bvshl (shift left).

3.1 Variables and Constants

We declare a variable for each subkey (as a bit-vector of length n):

```
(declare-fun k1 () (_ BitVec 8))
(declare-fun k2 () (_ BitVec 8))
(declare-fun l1 () (_ BitVec 8))
(declare-fun l2 () (_ BitVec 8))
```

Variables for encryption function are declared similarly. Variable x0 denotes a plaintext, and variable y denotes a ciphertext. Variables xp$i$, xh$i$, xi denote the value of internal variables X', $\hat{X}$ in round i, and variable X at the end of round i of the encryption function, respectively. The prefix a1 (and possibly a2, a3 etc.) is used to distinguish a specific plaintext-ciphertext pair.

```
(declare-fun a1x0 () (_ BitVec 8))
(declare-fun a1x1 () (_ BitVec 8))
(declare-fun a1xp1 () (_ BitVec 8))
(declare-fun a1xh1 () (_ BitVec 8))
(declare-fun a1x2 () (_ BitVec 8))
(declare-fun a1xp2 () (_ BitVec 8))
(declare-fun a1xh2 () (_ BitVec 8))
(declare-fun a1y () (_ BitVec 8))
```

Depending on a task we want to solve, the values of some variables are set. In case of single encryption, all subkeys are set and the plaintext a1x0 is set as well. For example (the constants in this example are written in hexadecimal):

```
(assert (= k1 #xee))
(assert (= k2 #x57))
(assert (= l1 #x67))
(assert (= l2 #xba))
(assert (= a1x0 #x25))
```

In case of known (or chosen) plaintext attacks we set just aix0 and aiy values for all known plaintext-ciphertext pairs.

3.2 Inner Product

We use two different methods for computing the inner product of two bit-vector. Both methods start with bitwise AND of input bit-vectors and then they compute a "bitxor" operation on the result, i.e. they XOR all its bits. The first method computes the bitxor by "fold-and-xor" approach. For $n = 8$: bitxor(x) = MSB$(F_1(F_2(F_4(x))))$ where $F_i(x) = x \oplus (x << i)$ and MSB returns the most significant bit of an argument. Using SMT-LIB v2 the bitxor operation looks like this:

```
(define-fun bitxor1 ((x (_ BitVec 8))) (_ BitVec 8)
  (bvxor (bvshl x #x01) x))
(define-fun bitxor2 ((x (_ BitVec 8))) (_ BitVec 8)
  (bvxor (bvshl x #x02) x))
(define-fun bitxor4 ((x (_ BitVec 8))) (_ BitVec 8)
  (bvxor (bvshl x #x04) x))
(define-fun bitxor ((x (_ BitVec 8))) (_ BitVec 1)
 ((_ extract 7 7) (bitxor1 (bitxor2 (bitxor4 x)))))
```

The second method simply extracts all bits and xors them together:

```
(define-fun bitxor ((x (_ BitVec 8))) (_ BitVec 1)
 (bvxor ((_ extract 7 7) x) (bvxor ((_ extract 6 6) x)
  (bvxor ((_ extract 5 5) x) (bvxor ((_ extract 4 4) x)
  (bvxor ((_ extract 3 3) x) (bvxor ((_ extract 2 2) x)
  (bvxor ((_ extract 1 1) x) ((_ extract 0 0) x)))))))))
```

A direct translation to SAT by SMT solvers might theoretically use more Boolean variables for the first method than for the second one. Later, we explore the impact of these two variants on SMT solvers performance.

3.3 Encryption Function

The translation of encryption function is straightforward – we unroll the loop. We do not use the linear representation of the round function from [8]. Another useful SMT-LIB v2 operation for this translation is ite (if-then-else). Example of encryption function for $r = 2$:

```
(assert (= a1xp1 (bvxor k1 a1x0)))
(assert (= a1xh1 (ite (bvult a1x0 a1xp1) a1xp1 a1x0)))
(assert (= a1x1 (ite (= (bitxor (bvand l1 a1xh1)) #b1) a1xp1 a1x0)))
(assert (= a1xp2 (bvxor k2 a1x1)))
(assert (= a1xh2 (ite (bvult a1x1 a1xp2) a1xp2 a1x1)))
(assert (= a1x2 (ite (= (bitxor (bvand l2 a1xh2)) #b1) a1xp2 a1x1)))
(assert (= a1y a1x2))
```

Even more compact representation of the encryption function is possible. Substituting some expressions, e.g. a$i$xh$j$ variable into aix$(j+1)$, or using helper functions can reduce the number of assert expressions or shorten them. However, such syntactical manipulations do not reduce the complexity of the problem for SMT solvers.

3.4 Preamble and Goals

We are interested in values of variables, not just knowing that the problem is satisfiable (we already know that), hence we want to produce a model for our input. Additionally, we use the bit-vector theory. These requirements are reflected in the preamble:

```
(set-option :produce-models true)
(set-logic QF_BV)
```

According to the problem we want to solve, we set the goal for encryption (value of the ciphertext):

```
(check-sat)
(get-value (a1y))
```

and for cryptanalysis we are interested in subkeys values:

```
(check-sat)
(get-value (k1 k2 l1 l2))
```

4 Performance Comparison of SMT Solvers

SMT solvers compete, similarly to SAT solvers, in SMT-COMP competition [6] using benchmarks in various logic divisions. We compare the performance of selected SMT solvers on problems related to the SBC. The goals of this comparison are twofold – simple tests allow to validate the correctness of the SBC model in SMT-LIB v2, and they allow to select the SMT solver with the best performance for further experiments.

Counting variables. Certainly, solving an encryption problem is much easier than finding keys in the known plaintext attack. In both cases the complexity of a problem can be measured (separately for each problem) by number of unknown n-bit variables. Simple calculation yields the following formulas (p denotes the number of plaintext-ciphertext pairs for KPA):

	number of n-bit variables
single encryption	$3r+1$
KPA	$2r+3rp$

4.1 Encryption

We use block length $n = 128$ and three different versions of the SBC (for $r \in \{128, 512, 1024\}$) in these tests. Each result presented in Table 1 is an average of 5 distinct instances of the encryption problem.

All SMT solvers were able to encrypt a plaintext in reasonable time, even for realistic values of n and r (the last column) that are supposedly secure for real-word application. Let us stress that this is not by any means an evaluation of the SBC performance itself. The results for each SMT solver scale expectedly with increasing number of rounds.

SMT solver	Average time [seconds] $r = 128$	$r = 512$	$r = 1024$
MathSAT5	0.04	0.13	0.25
Sonolar	0.05	0.20	0.42
Yices	0.04	0.14	0.29
Z3	0.38	1.83	3.88

Table 1: Performance of SMT solvers for single encryption

4.2 Known Plaintext Attack

The problem of finding keys for known plaintext-ciphertext pairs is substantially harder than simple encryption. It is unrealistic to expect a successful attack on the SBC with large number of rounds, large block length, and large number of plaintext-ciphertext pairs. For the purpose of SMT solver's comparison, we use small values $r \in \{5, 10, 15\}$ with the block length $n = 128$. The attacks were instantiated with 10 plaintext-ciphertext pairs, where the plaintexts were generated randomly. As noted in Section 3.2 there are two possible implementations of inner product operation – fold-and-xor and extract methods. We test both variants. The results are shown in Table 2 – again, each value is an average of solving 5 different problems. Let us note that we focus on a more detailed analysis in Section 5.

SMT solver	Average time [seconds] $r = 5$		$r = 10$		$r = 15$	
	fold	extract	fold	extract	fold	extract
MathSAT5	1.99	1.59	40.20	23.98	52.64	54.32
Sonolar	0.77	0.76	3.21	3.57	6.52	6.18
Yices	0.21	0.25	9.48	8.65	17.76	19.92
Z3	1.55	1.43	4.11	3.93	20.75	12.53

Table 2: Performance of SMT solvers for KPA

Clearly, Sonolar achieves the best performance. Therefore we use Sonolar for further analyses in Section 5. Let us note few observations from the tests:

- Different problems of the same size can have vastly different solving time for the same SMT solver. For example, the odd-looking average value 40.20 seconds for MathSAT5 is caused by an outlier 104.99 seconds (other values are smaller than the average). The least variations in solving times for given parameters are exhibited by Sonolar.
- The results for fold-and-xor and extract methods are counter-intuitive. The extract method should lead to smaller and simpler instances for underlying SAT solvers. It seems that extract method is superior only in case of Z3 solver. However, the results are inconclusive for other solvers.

5 Simple Cryptanalysis

Few experiments and thoughts on the SBC security are presented in this section. All results are obtained using Sonolar SMT solver, and the extract method for the inner product implementation. First, let us show a dramatic increase of solving time when KPA deals with more plaintext-ciphertext (or "PC" for short) pairs. Figure 2 shows situation for varying number of rounds, $n = 128$ and 10 or 20 PC pairs (each value is obtained as an average of 10 runs).

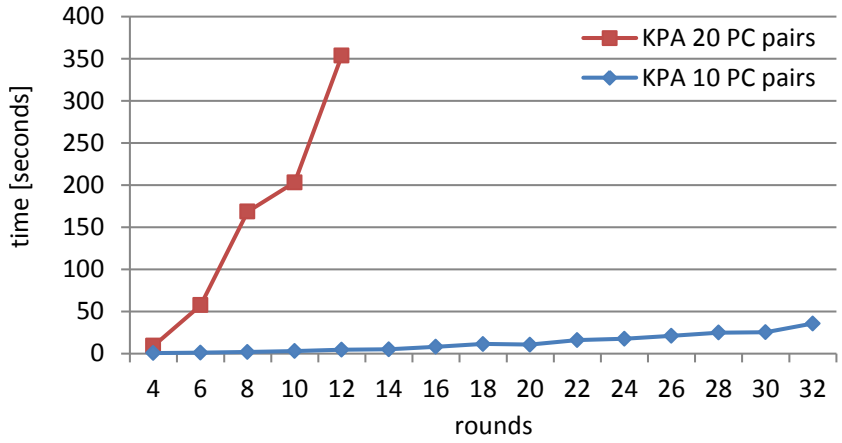

Figure 2: Comparing KPA with 10 and 20 PC pairs ($n = 128$)

Let us note that the problem is vastly under-defined for small number of PC pairs. The SBC has key length $2rn$ bits, assuming independent subkeys. A single PC pair reduces the entropy of a key by at most n bits. Moreover, a subkey L_i contributes to any encryption function by a single bit – the result of the inner product. There are many keys consistent with a reasonable set of PC pairs, i.e. the keys that transform each plaintext from the set into the corresponding ciphertext. The SMT solver simply finds some solution from the set of "consitent" keys.

Remark. We can also find an example of equivalent keys for some parameters, e.g. for $n = 8$ and $r = 2$ the keys $(K_1, K_2, L_1, L_2) = (0x6a, 0xa5, 0x62, 0xad)$ and $(K'_1, K'_2, L'_1, L'_2) = (0xcf, 0x6a, 0xad, 0x62)$, expressed as hexadecimal values, define the same permutation on $\{0,1\}^8$.

Our next experiment varies the block length n. We set the number of rounds $r = n$ and use 10 known PC pairs. The results in Figure 3 show a moderate increase of time needed to solve the KPA problem (the values are computed as an average of 10 instances). The situation changes dramatically (again) for 20 PC pairs: average time measured for $n = 4, 8, 12$ is 0.02 s, 0.60 s, and 234.35 s, respectively. The reason is that additional PC pairs substantially reduce the set of possible solutions for subkeys values, i.e. there is a small fraction of all subkeys that are also a valid solution for particular KPA problem. Hence the SMT solver (i.e. an underlying SAT solver) is forced to revise its guesses on unknown Boolean values more often.

In order to show the complexity of solving larger KPA problems, we choose $r = 6n$ for our last experiment. It is easy to find some solution when only a small number of PC pairs, e.g. 1 or 2, are given. However, further PC pairs increase the number of unknown variables and the complexity of the KPA problem.

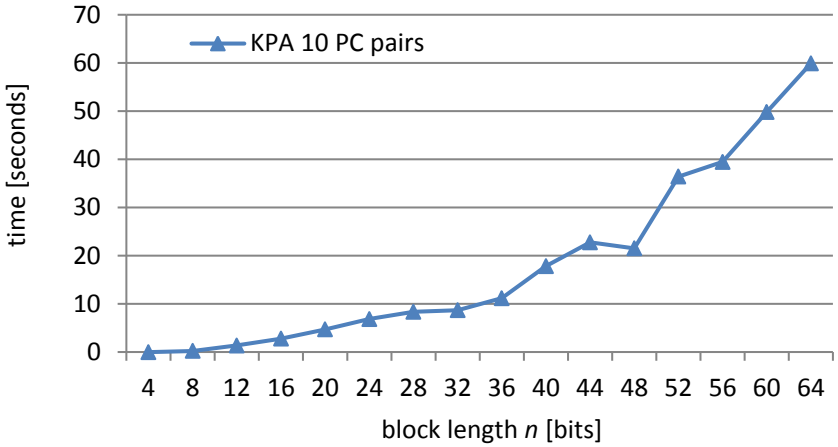

Figure 3: KPA for different block length with 10 PC pairs ($r = n$)

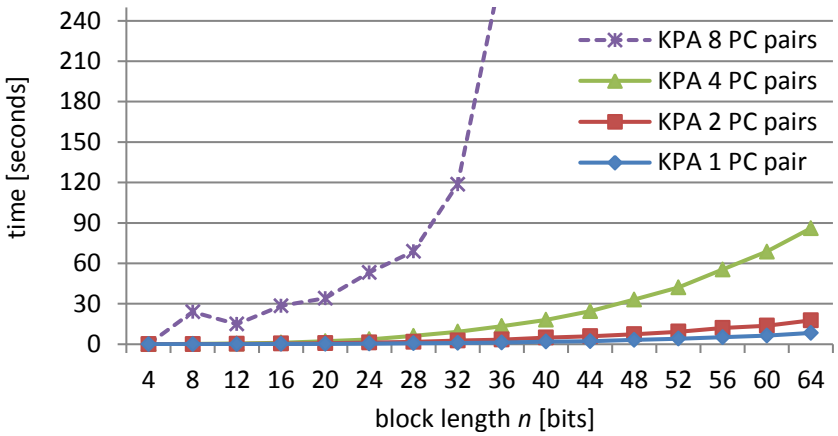

Figure 4: Comparing KPA for varying number of PC pairs ($r = 6n$)

6 Conclusion

The paper describes few experiments with the SBC. We modeled the SBC in SMT-LIB v2 and we used the SMT solver to solve small instances of the cipher in KPA scenarios. The results show that the instances with larger n and r parameters cannot be solved by this cryptanalytic approach in reasonable time. The cryptanalysis using linearity of the round function [8] is clearly superior to our experiments. This illustrates the gap between a "blind" cryptanalysis (solving without any insight into the problem) and cryptanalysis that uses an identified weakness of the cipher.

Acknowledgment

I would like to thank Viet Tung Hoang for pointing out the cryptanalysis of the SBC, presented by Serge Vaudenay in the CRYPTO 2012 Rump Session (and independently found by Henry Gilbert). The author also acknowledges support by VEGA 1/0259/13.

References

[1] Barrett C., Stump A., Tinelli, C.: The SMT-LIB Standard: Version 2.0, Department of Computer Science, The University of Iowa, 2010. Available at www.SMT-LIB.org

[2] De Moura L., Bjørner N.: Satisfiability Modulo Theories: Introduction and Applications, Communications of the ACM, Vol. 54, No. 9, ACM, 2011, pp. 69–77.

[3] Hoang V.T, Morris B., Rogaway P.: An Enciphering Scheme Based on a Card Shuffle, Advances in Cryptology – CRYPTO 2012, Lecture Notes in Computer Science Volume 7417, Springer, 2012, pp. 1–13.

[4] Jovanovic P., Neves S., Aumasson J.P.: Analysis of NORX, Cryptology ePrint Archive, Report 2014/317, 2014. http://eprint.iacr.org/2014/317

[5] Morris B., Rogaway P.: Sometimes-Recurse Shuffle, Advances in Cryptology – EUROCRYPT 2014, Lecture Notes in Computer Science Volume 8441, Springer, 2014, pp. 311–326.

[6] SMT-COMP 2014, Cok D., Deharbe D., Weber T. (Organizers), http://www.smtcomp.org/

[7] Vanegue J., Heelan S., Rolles R.: SMT Solvers in Software Security, Presented as part of the 6th USENIX Workshop on Offensive Technologies, WOOT'12, USENIX, 2012. https://www.usenix.org/system/files/conference/woot12/woot12-final26.pdf

[8] Vaudenay S.: The End of Encryption based on Card Shuffling, CRYPTO 2012 Rump Session, http://crypto.2012.rump.cr.yp.to/