MRHS SOLVER BASED ON LINEAR ALGEBRA AND EXHAUSTIVE SEARCH

HÅVARD RADDUM AND PAVOL ZAJAC

Abstract. We show how to build a binary matrix from the MRHS representation of a symmetric-key cipher. The matrix contains the cipher represented as an equation system and can be used to assess a cipher's resistance against algebraic attacks. We give an algorithm for solving the system and compute its complexity. The complexity is normally close to exhaustive search on the variables representing the user-selected key. Finally, we show that for some variants of LowMC, the joined MRHS matrix representation can be used to speed up regular encryption in addition to exhaustive key search.

1. Introduction

Encryption technology is being used in a large number of applications today, and many different encryption algorithms have been proposed for use in various environments. Security is always the most important consideration for a cryptographic primitive. This is achieved through cryptanalysis, where we try to find ways to break the security guarantees given by the designers. For symmetric ciphers this is usually equivalent to finding a secret key faster than doing exhaustive search on the secret key.

Several cryptanalytic techniques can be tried when assessing the strength of a cipher. In this paper we will focus on algebraic cryptanalysis, characterised by formulating the attacker's problem as solving a system of equations. Representing a cipher as an equation system can be done in several ways, and over different fields. In this paper we will only be concerned with the binary field $GF(2)$, and we will use the Multiple Right Hand Side (MRHS) [5] representation for constructing the equation systems.

Our contribution: It is easy to join all the individual MRHS equations into one matrix/vector multiplication with a (large) right-hand side set of potential solutions. We call this the joined system matrix. We show which linear operations we can do on the joined system matrix without changing its solution space, and without increasing the space complexity of the right-hand side set. This allows to bring the joined system matrix into a special form, allowing for fast execution of a simple brute-force solving algorithm. We describe a recursive algorithm using a guess/verify approach for solving a joined MRHS system and explain its complexity.

In the last part of the paper we test the algorithm on some well-known block ciphers and show that the analytic complexity estimates match the observed complexities very closely. One interesting observation we make is that for some of the proposed variants of LowMC [1] using very few S-boxes, we can actually do encryption via the MRHS solver faster than in the standard reference implementation. This can be explained with the fact that there is a very high number of linear operations compared to the number of non-linear operations in these versions. The joined MRHS representation merges all the dense linear layers in LowMC into one

Key words and phrases. Algebraic cryptanalysis, MRHS, LowMC.

This research was supported by project Cryptography brings security and freedom SK06-IV-01-001 funded by EEA Scholarship Programme Slovakia.

big matrix, and in total there are less xors to be done working on this matrix than executing the round-by-round linear layers in the standard specification.

2. Preliminaries

We denote the finite field with two elements as $\mathbb{F}$. All vectors over $\mathbb{F}$ are row vectors and are denoted by lower case letters. Sets of vectors are denoted by capital letters, and all matrices are denoted by boldface capital letters. The $p \times p$ identity matrix is denoted as $\mathbf{I}_p$, but if p is clear from the context we may just write $\mathbf{I}$.

Definition 2.1. [5] *A Multiple-Right-Hand-Sides (MRHS) equation over $\mathbb{F}$ is an expression of the form*

$$x\mathbf{M} \in S, \tag{1}$$

where $\mathbf{M}$ is an $(n \times l)$ matrix, and $S \subset \mathbb{F}^l$ is a set of l-bit vectors. We say that $x \in \mathbb{F}^n$ is a solution of the MRHS equation (1), if and only if $x\mathbf{M} \in S$.

A system of MRHS equations $\mathcal{M}$ is a set of m MRHS equations with the same dimension n, i.e.

$$\mathcal{M} = \big\{x\mathbf{M}_i \in S_i | 1 \leq i \leq m\big\}, \tag{2}$$

where each $\mathbf{M}_i$ is an $(n \times l_i)$-matrix and $S_i \subset \mathbb{F}^{l_i}$. The vector $x \in \mathbb{F}^n$ is a solution of the MRHS system $\mathcal{M}$, if it is a solution to all MRHS equations in $\mathcal{M}$, i.e. $x\mathbf{M}_i \in S_i$ for each $i = 1, 2, \ldots, m$. We denote the set of all solutions of a MRHS system $\mathcal{M}$ by $Sol(\mathcal{M})$.

Throughout the paper, n will always denote the number of variables in a MRHS equation system, and m will always be the number of equations in the system. The length of the vectors in S_i are l_i, but if all l_i in the system are the same we may just write l. By writing $S\mathbf{M}$ we mean the set $\{v\mathbf{M} | v \in S\}$ and with $w + S$ we mean the set $\{w + v | v \in S\}$.

Joined system matrix: Given a MRHS equation system $\mathcal{M} = \{x\mathbf{M}_i \in S_i\}$ we may concatenate all the $\mathbf{M}_i$'s columnwise since they all have the same number of rows. We denote the joined matrix by $\mathbf{M}$, and call it the *joined system matrix*:

$$\mathbf{M} = [\mathbf{M}_1 | \mathbf{M}_2 | \ldots | \mathbf{M}_m]$$

Similarly, we denote $S_1 \times S_2 \times \ldots \times S_m$ by S. The problem of solving a MRHS equation system can now be stated as finding some $x \in \mathbb{F}^n$ such that $x\mathbf{M} \in S$. A similar representation that was introduced in [8], but we use a different approach to solve the system.

Finally, the columns of $\mathbf{M}$ corresponding to $\mathbf{M}_i$ is called a *block*, and we sometimes speak of block i in $\mathbf{M}$.

3. Solving Algorithm

In this section we describe an algorithm for solving a MRHS system, and determine its complexity. We start with bringing the system into a special form.

3.1. Transforming MRHS System to Full Echelon Form.

We may perform linear transformations on the rows of $\mathbf{M}$ without changing the set S. Doing linear operations on the rows of $\mathbf{M}$ is essentially changing the variable basis, and can be captured in the following lemma.

Lemma 3.1. *Let x_0 be a solution to $x\mathbf{M} \in S$, let $\mathbf{U}$ be an invertible $n \times n$ matrix and let $\mathbf{A} = \mathbf{U}\mathbf{M}$. Then $y_0 = x_0\mathbf{U}^{-1}$ is a solution to $y\mathbf{A} \in S$.*

Proof: If $x_0\mathbf{M} = s \in S$, then $(x_0\mathbf{U}^{-1})(\mathbf{UM}) = y_0\mathbf{A} = s \in S$. □

We may also perform column operations on $\mathbf{M}$, but then we need to transform the set S in order to preserve the solution space.

Lemma 3.2. *Let x_0 be a solution to $x\mathbf{M} \in S$, let $r = \sum_{i=1}^{m} l_i$ and let $\mathbf{U}$ be a $r \times r$ invertible matrix. Then x_0 is a solution to $x\mathbf{MU} \in S\mathbf{U}$.*

Proof: If $x_0\mathbf{M} = s \in S$, then $x_0\mathbf{MU} = s\mathbf{U} \in S\mathbf{U}$. □

The problem with applying Lemma 3.2 in practice is that S can be very big. Each individual S_i are normally of small size but as $|S| = \prod_{i=1}^{m} |S_i|$, explicitly computing the set $S\mathbf{U}$ often has too high complexity to be done in practice.

However, by restricting $\mathbf{U}$ to a block diagonal matrix we can compute the set $S\mathbf{U}$ while keeping both time and memory complexity low.

Let $\mathbf{U}$ be a block diagonal matrix

$$\mathbf{U} = \begin{bmatrix} \mathbf{U}_1 & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{U}_2 & \dots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{U}_m \end{bmatrix} \tag{3}$$

where each $\mathbf{U}_i$ is a $(l_i \times l_i)$ invertible matrix. Then $S\mathbf{U} = S_1\mathbf{U}_1 \times S_2\mathbf{U}_2 \times \ldots \times S_m\mathbf{U}_m$. The memory requirement of storing $S\mathbf{U}$ is storing each individual $S_i\mathbf{U}_i$, which is the same as storing the original sets S_i. The time complexity for computing $S\mathbf{U}$ is doing $\sum_{i=1}^{m} |S_i|$ vector/matrix multiplications.

In the following we assume this restriction on $\mathbf{U}$ when doing column operations on $\mathbf{M}$. That is, we can do column operations within block i of $\mathbf{M}$ and transform the corresponding set S_i without altering the solution space.

Definition 3.1. *Given a MRHS system $\mathcal{M}$, we say that its joined system matrix $\mathbf{M}$ is in* full echelon form*, if each block in $\mathbf{M}$ has the following form*

$$\mathbf{M} = \left(\begin{array}{c|cc|c} \cdots & \mathbf{0} & \mathbf{T}_i & \cdots \\ \cdots & \mathbf{I}_{p_i} & \mathbf{0} & \cdots \\ \cdots & \mathbf{0} & \mathbf{0} & \cdots \end{array} \right),$$

with $0 \leq p_i \leq l_i$.

We can change any joined MRHS system $x\mathbf{M} \in S_1 \times S_2 \times \cdots S_m$ to a full echelon form as follows:

(1) Compute the matrix $\mathbf{E}$ that brings $\mathbf{M}$ into standard reduced row echelon form, where we also create 0's above every leading 1: $\mathbf{M}' = \mathbf{EM}$.
(2) Let the number of leading 1's in block i be p_i and let $\mathbf{U}$ initially be an $r \times r$ identity matrix. Permute the columns in block i such that all leading 1's are moved to the left, and make the same permutation on the same columns of $\mathbf{U}$. This creates an upper triangular sub-matrix at the place where $\mathbf{I}_{p_i}$ is in Definition 3.1. Next, add columns of the triangular sub-matrix, starting from the left-most one, to create zeroes to the right of the leading 1's on each row of block i. Make the same additions on the columns of $\mathbf{U}$. Then block i gets the form given in Definition 3.1, and we can define $\mathbf{U}$ as in (3) where each $\mathbf{U}_i$ corresponds to block i. Finally, $\mathbf{M}'' = \mathbf{M}'\mathbf{U}$ will then be in full echelon form.
(3) Compute new sets $S'_i = S_i\mathbf{U}_i$.

The system $x\mathbf{M}'' \in S'_1 \times S'_2 \times \cdots S'_m = S'$ is now in full echelon form. For every solution of this system we can compute the solution of the original system using Lemma 3.1. The transformation of the system to full echelon form is accomplished in polynomial time (in n) by doing linear algebra operations. This is done once

for the whole system as a pre-processing step, so in the following we will always assume that the system in question is in full echelon form.

3.2. **Algorithm Searching for Solution to MRHS System.** Here we present an algorithm for searching exhaustively for possible solutions to a MRHS system. Since $x \in \mathbb{F}^n$ we could expect that an exhaustive search algorithm would have complexity 2^n, but as we will see, the actual complexity for systems representing ciphers is a lot lower. This is because in these systems, once a (small) part of x has been guessed, the rest of x becomes uniquely determined and can be verified as correct or not. Hence the algorithm guesses on small parts of x, and keeps track of possible ways to extend a current guess.

Informal description of algorithm: The algorithm is accurately described in Alg. 1. The process can be briefly explained as follows.

We separate x into parts $x = (x_1, x_2, \ldots, x_m)$, where each x_i has length p_i (from Def. 3.1). We first fix x_1 to some value and compute $w_1 = (x_1, 0, \ldots, 0) \cdot \mathbf{M}$. We choose x_1 such that block 1 of w_1 is in S_1. Next, we choose x_2 and compute $w_2 = (x_1, x_2, 0, \ldots, 0) \cdot \mathbf{M}$ such that block 2 of w_2 is in S_2. Note that x_2 will not affect block 1 of w_2, due to $\mathbf{M}$ being in echelon form. So block 1 of w_2 is only determined by x_1 and will be unchanged. We continue this way: Assuming $x_1, \ldots, x_{i-1}$ has been fixed, we guess a value for x_i such that block i of $w_i = (x_1, \ldots, x_i, 0, \ldots, 0) \cdot \mathbf{M}$ is in S_i (block j of w_i remains in S_j for $j < i$).

At some point we run into cases where no possible choice of x_i is possible. If p_i is too small, it may be that no choice of x_i will produce a w_i whose block i is in S_i. Then the algorithm backtracks to the first point where we have an untried guess for x_j, and continues from there. Note also that p_i may even be 0, in which case x_i is empty and block i of w_{i-1} must already be in S_i in order for the algorithm to proceed. When the algorithm is able to complete x by selecting x_m such that block m of $x\mathbf{M}$ is in S_m we have found a solution to the system.

The reason for having $\mathbf{M}$ in full echelon form is for easy identification of possible values for x_i. The last $l_i - p_i$ bits of block i of w_i are independent of x_i, so these bits will be equal in w_{i-1} and w_i, regardless of choice of x_i. If we sort the vectors in S_i on the value of the last $l_i - p_i$ bits we can do a fast look-up to see which vectors in S_i that will be equal to w_i in the last $l_i - p_i$ bits. The first p_i bits of any such vector immediately gives the possible value for x_i, since x_i is multiplied with $\mathbf{I}_{p_i}$ in block i.

When p_i is small we can precompute the vectors $(0, \ldots, 0, x_i, 0, \ldots, 0) \cdot \mathbf{M} = z_i$ for all choices of x_i and store them in a table T_i. When the algorithm fixes a value for x_i we just look up the corresponding z_i from T_i and add it to w_{i-1} to produce w_i.

Parallelism: We can precompute a list L of partial solutions $(x_d|0)$ along with corresponding w_d's up to some depth d. Then we can distribute the search to $|L|$ parallel instances of the algorithm. This requires a separate memory for w_i's in each of the parallel tasks, but the parallel tasks can use the same set of precomputed T_i's (the tables are read-only). Furthermore, we can use internal parallelism to efficiently compute vector sums $w_i = w_{i-1} + z_{i,j}$.

In our implementation we store precomputed $z_{i,j}$'s and words w_i as sequences of 64-bit words. Thus, a single 64-bit XOR operation computes $64/l$ block sums. Even better speedup can be obtained on specialized hardware or with vector instructions (e.g., SSE).

3.3. **Algorithm complexity.** In each level of the recursion we compute one projection and do a table lookup. On each lookup we obtain a list $T_i[t_i]$ of vectors. The expected size of $T_i[t_i]$ is $|S_i| \cdot 2^{p_i - l_i}$. If the expected size is below 1, the T_i table

Algorithm 1 Solve MRHS system in full echelon form

Input: MRHS system $x\mathbf{M} \in S = S_1 \times S_2 \times \cdots \times S_m$ in full echelon form.
Output: Reduced set $V \subset \mathbb{F}^n$, such that $x\mathbf{M} \in S$ for each $x \in V$.

```
{PREPARATION PHASE}
for all i = 1, 2, ..., m do
    Initialize empty lookup table T_i.
    for all v_{i,j} ∈ S_i do
        x_{i,j} := proj_{1..p_i}(v_{i,j}), t_{i,j} := proj_{p_i+1..l_i}(v_{i,j}).
        z_{i,j} := (0|x_{i,j}|0) · M.
        Add (x_{i,j}, z_{i,j}) to the list in table entry T_i[t_{i,j}]
    end for
end for
i = 1, x_0 = 0, w_0 = 0, V = ∅

{RECURSION PHASE }
Input: i, partial solution (x_1, ..., x_{i-1}|0), vector w_{i-1}, V
t_i := proj_{i,p_i+1..l_i}(w_{i-1}).
for all (x_{i,j}, z_{i,j}) ∈ T_i[t_i] do
    if i = m then
        V := V ∪ {(x_1, ..., x_{m-1}|x_{m,j})}
    else
        w_i = w_{i-1} + z_{i,j}
        V := V ∪ Recursion(i + 1, (x_1, ..., x_{i-1}|x_{i,j}|0), w_i, V)
    end if
end for
return V
```

must be empty for some values of t_i, and we can expect the algorithm to backtrack at block i with probability at least $1 - \frac{|S_i|}{2^{l_i - p_i}}$. Thus, if we start the recursion N times on level i, we expect to continue $N \cdot |S_i| \cdot 2^{p_i - l_i}$ times to level $i+1$. Furthermore, before the recursion we need to add w_{i-1} with $z_{i,j}$, which requires $m - i + 1$ block-XORs (we know that the first part of $z_{i,j}$ is always 0 due to full echelon form of $\mathbf{M}$). These additions will typically dominate the running time of the algorithm.

If the vectors in the S_i's are chosen uniformly at random, the expected total number of recursion calls can be estimated as

$$N_{total} = \sum_{i=2}^{m} \prod_{j=1}^{i-1} |S_j| \cdot 2^{p_j - l_j}. \tag{4}$$

The total number of recursion calls corresponds to the number of accesses to the lookup tables.

The expected number of solutions can be estimated as

$$E(|Sol(\mathcal{M})|) = \prod_{j=1}^{m} |S_j| \cdot 2^{p_j - l_j}.$$

The total number of block XORs can be computed as

$$N_{XOR} = \sum_{i=2}^{m} (m - i + 1) \prod_{j=1}^{i-1} |S_j| \cdot 2^{p_j - l_j}.$$

We can slightly reduce the number of XORs by storing vectors $z_{i,j}$ which are equal to zero in a special format. This is especially useful in blocks where $p_i = 0$,

when the algorithm degenerates to just a look-up whether $T_i[t_{i,j}]$ is or is not an empty set.

We can estimate the chance of zero $z_{i,j}$ as 2^{-p_i} (one out of 2^{p_i} possible choices). Thus, the reduced number of XORs can be computed as

$$N_{XORed} = \sum_{i=2}^{m}(1-2^{-p_{i-1}})(m-i+1)\prod_{j=1}^{i-1}|S_j|\cdot 2^{p_j-l_j}.$$

On the bit level or the instruction level, the number of XORs must take into account the number of bits in each block, and the internal parallelism of XOR instructions. The number of single bit XORs (without taking into account zero $z_{i,j}$) can be estimated as

$$N_{XOR1} = \sum_{i=2}^{m}\sum_{b=i+1}^{m} l_b \prod_{j=1}^{i-1}|S_j|\cdot 2^{p_j-l_j}.$$

When using w-bit internal parallelism, we cannot directly divide N_{XOR1} by w, because on some levels we cannot use the whole w birs in the word. Instead, the number of expected w-bit XOR instructions can be expressed as

$$N_{XORw} = \sum_{i=2}^{m}\left(\left\lceil\frac{\sum_{b=i+1}^{m} l_b}{w}\right\rceil\prod_{j=1}^{i-1}|S_j|\cdot 2^{p_j-l_j}\right). \tag{5}$$

Similar estimates can be done for N_{XORed1} and N_{XOREdw} by multiplying the internal products by coefficient $(1-2^{-p_{i-1}})$ that corresponds to a probability of non-zero $z_{i,j}$:

$$N_{XOREdw} = \sum_{i=2}^{m}\left(\left\lceil\frac{\sum_{b=i+1}^{m} l_b}{w}\right\rceil\left(1-2^{-p_{i-1}}\right)\prod_{j=1}^{i-1}|S_j|\cdot 2^{p_j-l_j}\right). \tag{6}$$

Example: Suppose that $l_i = 3$, and $|S_i| = 4$ for each i (we can model AND-gates with random linear combinations of variables as inputs in this setting). Let us suppose that we have m variables and m MRHS equations (blocks). Now assume a simple case where $m = 3k$, and we can get a simple echelon form with $p_i = 3$ for $i = 1,\ldots,k$ and $p_i = 0$ for $k+1 \le i \le 3k$.

The expected number of solutions is

$$E(|Sol(\mathcal{M})|) = \left(\prod_{j=1}^{k} 2^2\cdot 1\right)\cdot\left(\prod_{j=k+1}^{3k} 2^2\cdot 2^{-3}\right) = 2^{2k}\cdot 2^{-2k} = 1.$$

The number of recursion calls is

$$N_{total} = \sum_{i=2}^{k+1} 2^{2(i-1)} + \sum_{i=k+2}^{3k} 2^{2k-(i-k-1)},$$

so $2^{2k+1} < N_{total} < 2^{2k+2}$. The total number of XORs will be between $k\cdot 2^{2k}$ and $2k\cdot 2^{2k}$. This is a factor $2^k/k$ lower than multiplication of all possible 2^{3k} x-vectors by matrix $\mathbf{M}$.

Another example: Suppose that $l_i = 3$, and $|S_i| = 4$ for each i again, m variables and m MRHS equations (blocks). Now assume a case, where a single leading 1 in each block, i.e. $p_i = 1$ for each i.

The expected number of solutions is as before

$$E(|Sol(\mathcal{M})|) = \left(\prod_{j=1}^{m} 2^2\cdot 2^{-2}\right) = 1.$$

The number of recursion calls is

$$N_{total} = \sum_{i=2}^{m} 1 = m - 1,$$

and the number of XORs is $(m^2 + m)/2$. This means that we can solve the system in quadratic time (the problem complexity is no longer exponential in system size).

Randomly generated systems will typically have almost all blocks without internal linear dependencies. Thus, they are most likely of type 1 (hard to solve), and systems of type 2 must be constructed artificially. On the other hand, systems produced from cryptanalytic problems have a lot of internal structure that comes from the algorithm implementation. We observe the effect of this structure for selected ciphers in Section 4.

4. Representing SL Ciphers as MRHS Systems

In the following we study several ciphers. We adopt the term *SL cipher* to a cipher that can be represented as a sequence of linear (or more precisely, affine) transformations and substitution layers realised by S-boxes.

4.1. **Modelling an SL Cipher as a MRHS System.** Let s, l_a, l_o,n_R, n_B, n_K denote the number of: S-boxes per round, input and output bits of S-boxes, rounds, input/output block size, and key size, respectively. We can construct an initial MRHS system representing an SL cipher with parameters $l_i = l_a + l_o$, $|S_i| = 2^{l_a}$, $m = s \cdot n_R$, and $n = 2n_B + n_K + ml_o$. The unknown $x \in GF(2)^n$ in the system consists of the n_B plaintext bits, n_K key bits, all S-box outputs and n_B ciphertext bits. For concreteness, let $x_1, \ldots, x_{n_B}$ be the plaintext bits, $x_{n_B+1}, \ldots, x_{n_B+n_K}$ be the bits of the user-selected key, $x_{n_B+n_K+1}, \ldots, x_{n-n_B}$ be the output bits of all S-boxes used in one encryption, and $x_{n-n_B+1}, \ldots, x_n$ be the ciphertext bits. If S-boxes are used in the key schedule they are considered to be used in the encryption.

As SL ciphers only have linear (affine if additions of constants occur) operations apart from S-boxes, all input bits to all S-boxes can be described as linear (affine) combinations of the variables we have defined. We create one MRHS equation for each S-box i as follows:

(7)
$$x \cdot \mathbf{M}_i \in \{(0 \oplus c || S(0 \oplus c)), (1 \oplus c || S(1 \oplus c)), \ldots, ((2^{l_a} - 1) \oplus c || S((2^{l_a} - 1) \oplus c))\} = S_i,$$

where c is the constant part of the input affine combinations and we use the natural mapping between integers and vectors over $GF(2)$.

The first l_a columns of $\mathbf{M}_i$ contain the coefficients of the linear combinations of the inputs to the S-box. The last l_o columns of $\mathbf{M}_i$ contain a single 1-bit each; $m_{j,t} = 1$ if x_j is the variable for output bit t, and $m_{j,t} = 0$ otherwise. In the end, the ciphertext bits can be described as linear combinations of the variables we have defined.

Constructing the joined system matrix of the MRHS system models the complete cipher. The model is flexible enough to cover all of the currently used ciphers that use S-boxes[1], and a linear or affine layer in-between layers of S-boxes.

The model is based on a single plaintext/ciphertext (P/C) pair. To model multiple encryption instances, new variables must be defined for plaintext, ciphertext and S-box output bits, except for S-boxes used in the key schedule. Variables for the user-selected key and key schedule S-boxes are re-used across different P/C pairs. If multiple P/C pairs are used, all MRHS equations may still be merged into one joined system matrix.

[1]An S-box can be any Boolean function $S : GF(2)^{l_a} \to GF(2)^{l_o}$, as long as l_a is not too large.

4.2. **Fixing Known Bits.** If the purpose of constructing the joined system matrix is to do algebraic cryptanalysis we assume we have a known plaintext/ciphertext pair. By fixing the first and last n_B variables in x to their correct values we get a reduced system. If the original joined system is $x\mathbf{M} \in S$, we can write it as

$$(p, x', c) \cdot \begin{bmatrix} \mathbf{M_p} \\ \mathbf{M'} \\ \mathbf{M_c} \end{bmatrix} \in S_1 \times S_2 \times \ldots \times S_m \tag{8}$$

where p and c are the known values of the P/C pair and x' are the remaining variables. Setting $p\mathbf{M_p} + c\mathbf{M_c} = w = (w_1, w_2, \ldots, w_m)$ we get a reduced MRHS system

$$x'\mathbf{M'} \in (w_1 + S_1) \times (w_2 + S_2) \times \ldots \times (w_m + S_m).$$

4.3. **Encryption as MRHS Solving.** In an algebraic attack, we fix the variables for the plaintext and ciphertext bits. The task for the cryptanalyst is then to solve the remaining system, using Alg. 1 or by other means, to find the values of the variables for the key bits.

However, it is also possible to do regular encryptions using the MRHS representation. In this case we fix the plaintext and the key variables. The task is then to "solve" the reduced system to find the values of the ciphertext variables. In a joined MRHS system where the initial equations were joined in the natural order (round by round) and both plaintext and key variables are fixed, Alg. 1 will not do any guessing but rather just do look-up's for the values of the intermediate variables before finding the ciphertext bits in the end.

For many ciphers there is nothing to gain from doing encryptions this way, but if the cipher contains a lot of linear operations, encryption can go faster using the MRHS representation. LowMC is a cipher that has a dense affine transformation in each round, and the MRHS representations packs all of this linearity more efficiently together such that less xors need to be done when encrypting via Alg. 1.

We show this for the LowMC version with one S-box per round, $l_a = l_o = 3$, 64-bit block and 164 rounds. We count the number of single-bit xors needed to be done in a straight-forward reference implementation and in solving one MRHS instance. The number of look-ups needed to be done is the same in both cases since each S-box in the cipher specification gives rise to one block in the joined system matrix of the MRHS representation. Since we are in encryption mode, we assume that all key material is fixed and precomputed in both cases.

The number of xors done in one encryption according to the LowMC specification can be computed as follows: The linear layer in each round needs approximately $n_B/2$ xors of n_B-bit words. The addition of round constant and round key accounts for 2 more n_B-bit xors. Over n_R rounds the number of single-bit xors is

$$n_R(n_B^2/2 + 2n_B).$$

Setting $n_B = 64$ and $n_R = 164$ gives 356 864 xors.

The number of columns in the joined MRHS matrix is ln_R, so each xor of one row in this matrix will count for ln_R single-bit xors. When the plaintext bits get fixed, we need to add approximately $n_B/2$ rows together to produce the initial vector w_0. Then Alg. 1 does n_R look-ups, each one adding exactly one vector of length ln_R to the current w_i. In total we get

$$ln_R(n_B/2 + n_R)$$

single-bit xors before the ciphertext can be found. With $n_R = 164, n_B = 64$ and $l = 6$ this number comes to 192 864. We see that the number of xors needed when encrypting via the MRHS representation is approximately 1.85 times lower than in the reference specification.

5. Experiments with Concrete Ciphers

In this section we report on experiments done with Alg. 1 on some ciphers. In the experiments we reduce the key space by always setting a certain amount of key bits to zero and get a reduced system. This is done to get practial running times so we can measure the observed time complexity of Alg. 1.

We have focused on 4 (families of) ciphers: DES [6], AES [3], Present [2], and version 2 of LowMC [1]. We have tried various experiments with key sizes between 18-24 bits. The results are very similar between the choice of key size, thus we only present results for 22-bit unknown key bits. This choice makes individual experiments reasonably fast, but not so fast that time measurement errors become significant.

These families all fit into our SL cipher framework, while providing enough variety in design choices (Feistel/SPN), S-box sizes, linear layer type (permutation, MDS, random), different key schedules. We only provide short notes on SL models of each family, as we suppose the reader is familiar with the design of these ciphers.

5.1. **Ciphers tested.** In our model we use standard DES parameters $n_B = 64$ and $n_K = 56$. The first part of the key is fixed to zero. We use the full set of 8 DES S-boxes in each round, with $l_a = 6$ and $l_o = 4$. There are 32 new variables introduced in each round, and the n_K user-selected key bits are used directly in the encryption. The total number of variables in the MRHS representation of DES before fixing any known data will then be $n = 2 \times 64 + 56 + 32n_R$.

Our model of AES has parameters $n_B = n_K = 128$, again with fixing part of the key bits to zero. The AES specification includes four S-boxes in the key schedule, thus we use 20 S-boxes in each round with $l_a = l_o = 8$. There are 160 new variables introduced in each round, 32 in the key schedule and 128 in the cipher block. The total number of variables in the system before fixing known data is $n = 2 \times 128 + 128 + 160 \times n_R$.

We have selected a version of Present with $n_B = 64$ and $n_K = 80$ (again zero-reduced). The key schedule uses one S-box so the total number of S-boxes per round is 17, with $l_a = l_o = 4$. There are then 68 new variables introduced in each round, so the total number of variables before inserting known data will be $n = 2 \times 64 + 80 + 68n_R$.

Finally, we focus on the LowMC cipher which allows a variable number of S-boxes, block and key size. Only $3s$ bits of the cipher block passes through S-boxes in each round, so we get $3s$ new variables in each round. All other parts of the cipher are linear, both in encryption and key schedule, and the total number of variables in the MRHS representation is $n = 2n_B + n_K + 3sn_R$.

We use a custom software implemented in SAGE [4] to generate instances of MRHS systems based on the SL model described above. The generator software produces an instance with n_k unknown key bits ($n_k \leq n_K$), with expected complexity of the (whole) exhaustive search 2^{n_k}.

The generated instance is given to a fast solver[2], which is a C implementation of Algorithm 1 from Section 3.2. The algorithm searches the full space (it does not stop after producing a solution). It reports the total number of recursive calls

[2]Please contact pavol.zajac@stuba.sk if you want to obtain the source codes of the current version of the solver.

(same as table lookups) c, the number of XORs x, and the running time t of the search.

The solver can also generate random instances of MRHS systems with specified parameters, and estimate the complexity N_{total} using equation (4), without actually solving the system. This is useful for larger instances. Similarly, we use equations (5) and (6) to estimate the number of $w = 64$-bit XOR instructions N_{XORw}, and N_{XORedw} (without, and with taking zero blocks into account, respectively).

5.2. **Brute force attacks with MRHS solver.** We started the experiments with estimating a CPUyear cost for exhaustive key search on DES with the MRHS solver, depending on the number of rounds. The results are summarized in Table 1. We compare the estimated exhaustive search time of our solver with the results obtained from OpenSSL `speed` command on the same PC. The results indicate that the running time of the MRHS solver are comparable to standard exhaustive search, with slight advantage for the four and five round versions.

TABLE 1. Running times for DES, plus estimate for exhaustive search

Rounds	22-bit key [s]	Full [CPUyear]	Ratio
4	0.06	31.73	0.12
5	0.32	176.10	0.68
6	0.82	447.08	1.73
7	1.41	770.16	2.98
8	2.32	1262.89	4.88
9	3.37	1834.37	7.09
10	4.58	2494.90	9.64
11	5.94	3235.56	12.50
12	7.38	4018.46	15.53
13	8.95	4870.76	18.82
14	10.60	5772.60	22.31
15	12.06	6563.20	25.36
16	13.68	7448.42	28.78
OpenSSL	0.48	258.76	1.0

In Table 2, we compare the results of 22-bit exhaustive key search for different cipher instances. We include the size of the system, the total running time of the solver, and the time of exhaustive search using a software implementation of the cipher. We have used DES and AES implemented in OpenSSL (1.0.2g) using the `speed` command. Present implementation was taken from [9], and LowMC implementation from [7].

We see that the MRHS solver is typically slower than the optimised cipher implementations. On the other hand, the complexity does not grow exponentially with the size of the system (as is expected for a random non-linear equation system). Depending on optimisations and implementation platform, we can expect that brute-force attack with MRHS solver is competitive with some implementations of ciphers. This is confirmed when comparing our implementation with the reference implementation of LowMC with a low number of S-boxes per round.

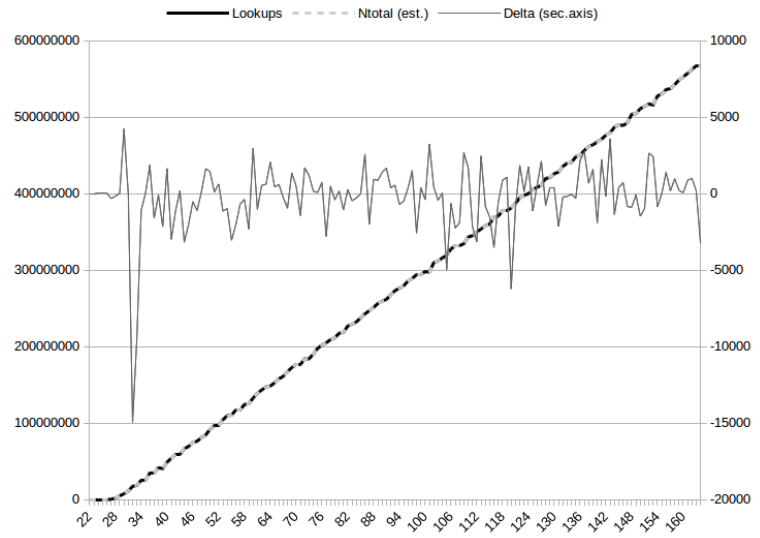

FIGURE 1. Comparison of the number of lookups in experiments with LowMC cipher to the estimated number of lookups N_{total}.

TABLE 2. Running times of the solver (MRHS) and reference implementations (ref. SW) for different ciphers

Cipher	s	n	m	l	$\lvert S_i \rvert$	MRHS [s]	ref. SW[s]
DES	8	470	128	10	64	13.6	0.48
Present80	17	2066	527	8	16	114.4	0.12
AES128	20	1494	200	16	256	57.9	0.58
LowMC64-1	1	450	164	6	8	12.9	45.82
LowMC64-2	2	450	164	6	8	12.4	23.71
LowMC128	31	1010	372	6	8	55.5	9.24
LowMC256	49	1530	588	6	8	108.8	25.71

5.3. **Expanded results for various versions of LowMC cipher.** In Fig. 1 and Fig. 2 we show results from exhaustive key search experiments with the LowMC cipher with a single S-box and variable number of rounds, from 22 to the recommended 164. The key size was set to 22 bits, so the running time of experiments is in the order of seconds.

We have measured the number of lookups and XORs in the implementation of Algorithm 1. The measured results are compared with the estimates obtained by equations (4), (5), and (6), respectively. The N_{total} estimate is very accurate for the real number of lookups. On the other hand, the number of XORs is typically between the estimates N_{XORedw}, and N_{XORw}. The staircase character corresponds to an internal parallelism in the algorithm implementation: each step corresponds to approximately 21 rounds, which adds 63 bits to the width of the system that fit into the 64-bit architecture used.

In Fig. 3, we compare the running time of the same experiment with the running time of the brute-force attack that uses the LowMC implementation from [7]. The brute force attack only uses the functions `cipher.setkey` and `cipher.encrypt` in a loop over all 22-bit keys. Both programs were compiled with the same compiler and level of optimisation and run on the same computer. Note that both solvers go through the whole range of keys/potential solutions, and do not stop if the key

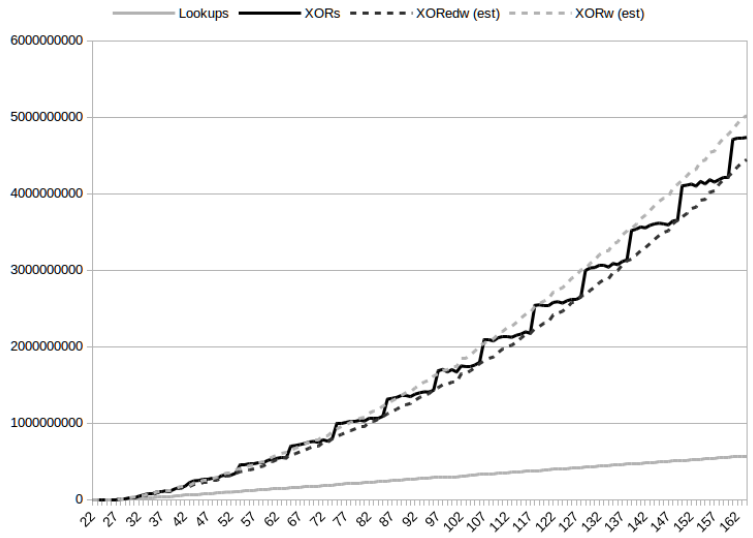

FIGURE 2. Comparison of the number of XORs in experiments with LowMC cipher to the estimated number of XORs by formulas (5) (upper bound) and (6) (lower bound).

is found sooner. For low number of rounds, algebraic representation gives a huge speedup in checking the key. This converges to about 3.5-times faster encryption via the MRHS solver for the full version of the cipher.

This speed-up can be explained by the different numbers of xors needed to be done in the reference implementation and in Alg. 1. In an exhaustive key search the plaintext and ciphertext are fixed, and only the key is changed for every encryption. The number of xors needed for checking one key in the reference implementation can be estimated as follows. To make one round key we must perform approximately $n_B n_K/2$ single-bit xors. Repeated over n_R rounds the complete key schedule costs $n_R n_B n_K/2$ xors. As we saw in Sec. 4.3 one encryption needs $n_R(n_B^2/2+2n_B)$ xors. In total it takes

$$n_R n_B n_K/2 + n_R(n_B^2/2 + 2n_B)$$

single-bit xors to check one key using the standard implementation. With $n_R = 164, n_B = 64$ and $n_K = 80$ this comes to 776 704 single-bit xors.

Fixing the key in the MRHS representation takes approximately $n_K/2$ xors of rows from the joined system matrix. Performing the encryption is the same as in Sec. 4.3, n_R xors of vectors of length ln_R. The total number of single-bit xors for checking one key in the MRHS model is then

$$ln_R(n_K/2 + n_R).$$

Setting $l = 6, n_R = 164$ and $n_K = 80$ gives a total of 200 736 single-bit xors. This is a factor 3.87 lower than in the reference specification, and explains the observed speed-up in exhaustive search using Alg. 1.

5.3.1. *Increasing the number of S-boxes.* In figures 4–6 we compare exhaustive search run-times across different versions of LowMC: with 8, 4, 2, and 1 S-boxes per round (denoted by s). In each version we set the maximum number of rounds as $164/s$, and plot the results for cipher versions with smaller number of Sboxes

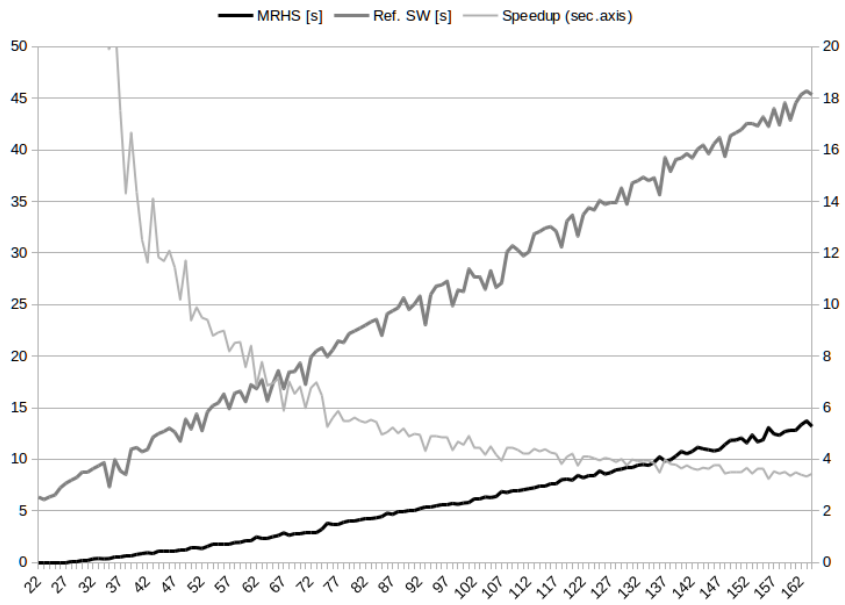

FIGURE 3. Comparison of the real time required to brute-force a 22-bit key with the MRHS solver and with the SW implementation of LowMC running in loop.

only in the range of rounds above the previously attained number of rounds. The behaviour of the solver is consistent across different versions of the cipher.

Note that the largest size of the MRHS system for each case is the same, as it is derived from the total number of S-boxes used in the encryption. This behaviour is most pronounced when comparing the running times with the reference software implementation of LowMC (Fig. 6): while the running time of the software implementation grows linearly in n_R, the running time of the solver depends both on the number of rounds and the number of S-boxes used in each round, and grows linearly in sn_R. Hence the MRHS solver's speed advantage quickly disappears when s increases.

6. Conclusions and Discussion

All symmetric ciphers can be modelled as a system of Boolean equations, represented as a fully joined MRHS matrix. We have devised an algorithm that solves the system in the MRHS matrix model, and estimated its complexity. Actual implementations show the estimates are very accurate. With other solvers, like F4, ordinary glueing/agreeing with MRHS systems, or SAT-solvers, it is difficult to predict actual running time from the initial system. The fully joined MRHS system helps on this situation, as solving time can be determined by only examining the structure of the matrix.

We also observe that for full-scale ciphers the complexity is only exponential in the number of variables representing the user-selected key, and not in the total number of variables. For ciphers with reduced rounds the joined MRHS matrix can reveal at which point we get lower than brute force solving complexity. For instance, 8 rounds of Simon32 has complexity 2^{62} while the key has 64 bits.

It is possible to guess the values of t linear combinations of variables such that the number of leading 1's in some blocks decreases by t. This will decrease some p_i's in (4) by a total of t, and hence we decrease running time with a factor 2^{-t}. An open

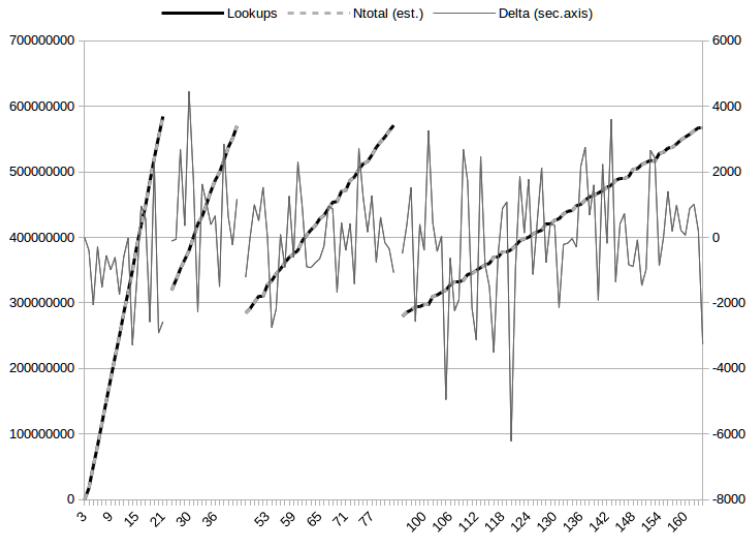

FIGURE 4. Comparison of the number of lookups in experiments with LowMC cipher to the estimated number of lookups N_{total}. Number of S-boxes from left to right: 8, 4, 2, 1.

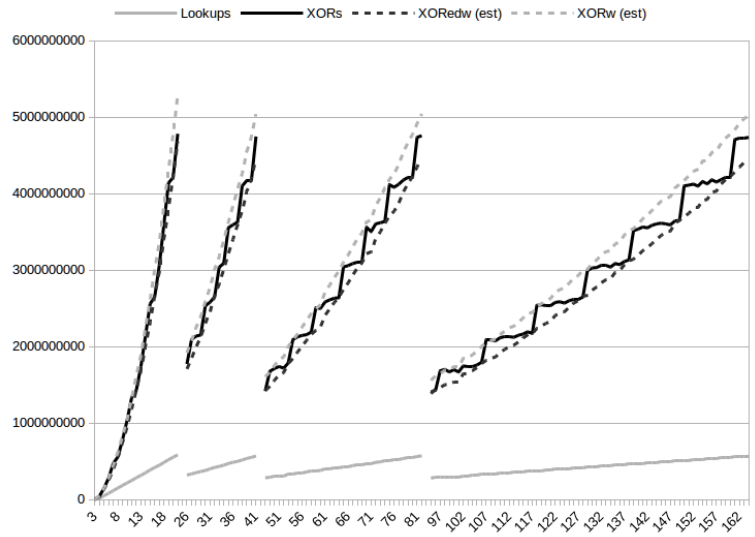

FIGURE 5. Comparison of the number of XORs in experiments with LowMC cipher to the estimated number of XORs by formulas (5) (upper bound) and (6) (lower bound). Number of S-boxes from left to right: 8, 4, 2, 1.

problem is to study if it is possible to gain *more* than a factor 2^{-t} when guessing some particular linear combinations. With the accurate complexity estimate given by the joined MRHS matrix it is possible to determine this beforehand, without running any experiments of actual solving.

The final interesting finding in this work is that versions of LowMC with very few S-boxes can be more efficiently implemented in the MRHS model. This comes from the fact that with few S-boxes the steps of the cipher are close to being successive

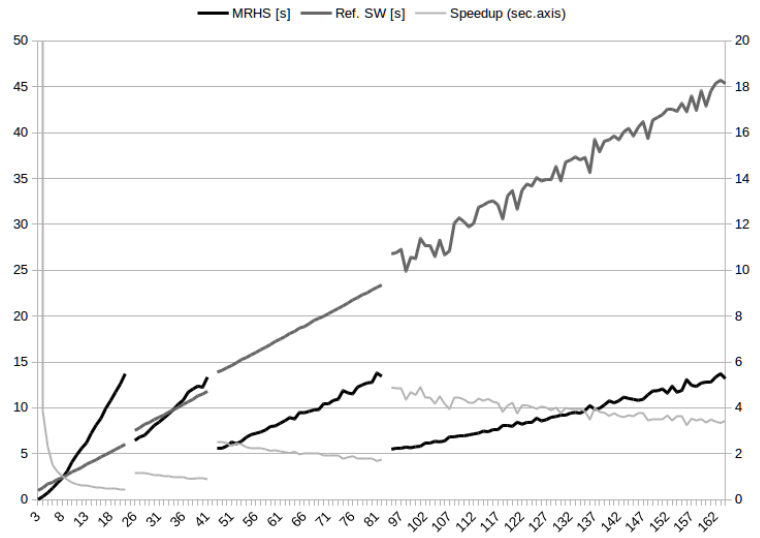

FIGURE 6. Comparison of the real time required to brute-force a 22-bit key with the MRHS solver and with the SW implementation of LowMC running in loop. Number of S-boxes from left to right: 8, 4, 2, 1.

linear operations. The MRHS implementation merges a lot of the linear-upon-linear parts of the operations, resulting in less xors needing to be done using the MRHS representation. It is then possible to implement encryption more efficiently, with a speed-up factor of 1.85 when doing the linear operations of the cipher.

In the exhaustive key search scenario the gain from using the MRHS solver is even bigger since the LowMC key schedule must be executed for every key tested. With one S-box per round the MRHS representation for exhaustive search is close to 4 times faster to use than the standard reference implementation. This may be interpreted as a valid attack since we can do a full search of the 80-bit key space in approximately the same time it takes to do 2^{78} standard LowMC encryptions. On the other hand, assuming that LowMC encryptions are *also* done in the MRHS model, this speed advantage disappears.

We hope that the community on analysing symmetric encryption algorithms finds the work in this paper useful, and that modelling a cipher using its joined MRHS matrix may serve as a tool in assessing ciphers' strength against algebraic cryptanalysis.

REFERENCES

[1] Martin Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for mpc and fhe. Cryptology ePrint Archive, Report 2016/687, 2016. `http://eprint.iacr.org/2016/687`.

[2] Andrey Bogdanov, Lars R Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew JB Robshaw, Yannick Seurin, and Charlotte Vikkelsoe. *PRESENT: An ultra-lightweight block cipher*. Springer, 2007.

[3] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[4] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 7.2)*, 2016. `http://www.sagemath.org`.

[5] H. Raddum and I. Semaev. Solving multiple right hand sides linear equations. *Design, Codes and Cryptography*, 49(1):147–160, 2008.

[6] Data Encryption Standard. Federal information processing standards publication (fips pub) 46-3, national bureau of standards. pages 46–3, 1999.
[7] Tyge Tiessen. An implementation of the lowmc block cipher family. `https://github.com/tyti/lowmc`, 2016.
[8] Pavol Zajac. A new method to solve MRHS equation systems and its connection to group factorization. *Journal of Mathematical Cryptology*, 7(4):367–381, 2013.
[9] Bo Zhu. An efficient software implementation of the block cipher present for 8-bit platforms. https://github.com/bozhu/PRESENT-C/, 2013.

Simula@UiB, Thormøhlensgate 55, N-5006 Bergen, Norway, haavardr@simula.no

Slovak University of Technology in Bratislava, Ilkovicova 3, 812 19 Bratislava, Slovakia, pavol.zajac@stuba.sk