Searching for Low Weight Codewords in Linear Binary Codes

Somitra Kumar Sanadhya* and Palash Sarkar

Applied Statistics Unit,
Indian Statistical Institute,
203, B.T. Road, Kolkata,
India 700108.
somitra_r@isical.ac.in, palash@isical.ac.in

Work in Progress

9^{th} April 2008

Abstract. In this work we revisit the known algorithms for searching for low weight codewords in linear binary codes. We propose some improvements on them and also propose a new efficient heuristic.

1 Introduction

The general problem of determining the least weight of a linear binary code is known to be NP-complete [1]. The motivation to study such a property in linear codes stems not only from the theoretical important and challenging open questions in coding theory but also due to its practical ramifications in cryptography.

The general problem being NP-complete, researchers have developed several algorithms to quickly find low weight codewords which are possibly close to the least weight of a given code. Some such algorithms were proposed in [4], [5] and [3]. We briefly explain these methods and then propose some improvements to them. We also propose a new method to search for low weight codewords. Some of the algorithms discussed in this work are used to find a low weight codeword for the linearized version of 23-step SHA-256 hash function. We hope to do detailed comparison of all the algorithms using some standard codes in coming days.

2 Overview of Existing Algorithms

Let G be the generator matrix and H be the parity check matrix of a code $\mathcal{C}$. Let the length of a codeword in $\mathcal{C}$ be n and the dimension of the code be k. Then the order of G is $(k \times n)$ and the order of H is $((n-k) \times n)$.

2.1 Generator Matrix Based Approaches

The following two algorithms use generator matrix of the code.

* This author is supported by the Ministry of Information Technology, Govt. of India.

Leon's Algorithm [4] This algorithm by Leon [4] starts with randomly permuting the columns of the generator matrix $G_{k\times n}$. A Gaussian operation is performed on G next to obtain an identity matrix I_k at beginning of the modified G. Let the matrix after the Gaussian be G', then $G' = (I_k, Z_{k\times(n-k)})$. The initial permutation of the columns which gives rise to such a decomposition of G' defines a k-element information set and a $(n-k)$-element redundant set for the code $\mathcal{C}$.

The algorithm uses a punctured code $\mathcal{C}'$ in which an L column selection from the redundant set is made. The rows are divided into two parts, each containing $k/2$ rows. Let the two parts be called set X and set Y. A linear combination of all p row combinations from X is computed punctured code and a list $\mathcal{L}_1$ restricted to the L columns is prepared. Similarly, another list $\mathcal{L}_2$ is prepared using the set Y. Matching pairs from the two lists are then found. Each matching pair gives a $2p$ row combination which produces a codeword vanishing on the L columns. Each such row combination when applied over the full code $\mathcal{C}$ will produce a codeword having weight $2p$ over the information set part, 0 over the chosen L columns and some weight over the rest of the redundant set part. If the aim is to find a codeword having weight w, then the contribution of the last part must be $w-2p$. If a row combination producing this weight exists, then the algorithm produces the codeword and stops, otherwise the process is repeated with different random initial permutations for the columns.

When the aim is to obtain a low weight codeword, then the least weight obtained in many iterations of the process above is produced. The parameters L and p are usually kept to be about 20 and 3 respectively for codes of length 500 or 1000. The main cost of the algorithm is in the Gaussian step.

Canteaut and Chabaud's Algorithm [3] Canteaut and Chabaud [3] modified Leon's algorithm such that the Gaussian step is done only once. They select one column from the redundant set and exchange it with a column from the information set. This process is repeatedly applied with the swap of randomly selected columns from the information set and the redundant set. After each swap, the weight on the L columns is computed similar to Leon's method. When this weight vanishes over the chosen L columns then the weight for the full code $\mathcal{C}$ is computed. With this modification, they show that the Gaussian step can be eliminated and that the algorithm still converges. They also provide detailed of analysis of rate of convergence of their algorithm.

2.2 Check Matrix Based Approach

To our knowledge, there is only one algorithm for finding low weight codewords using parity check matrix. This algorithm is described next.

Stern's Algorithm [5] This algorithm by Stern [5] predates the Canteaut-Chabaud algorithm described above. In this method, first the columns of the check matrix are permuted randomly. Then a Gaussian step is performed to obtain identity matrix I_{n-k} in the beginning of the check matrix. As in the cases described previously, the column permutation defines an information set of size $(n-k)$ and a redundant set of size k columns. The redundant set is divided into 2 parts, say X and Y. Each of these sets contain $n/2$ columns. A list $\mathcal{L}_1$ of linear combinations of p columns along L rows from X is constructed. Similarly, another list $\mathcal{L}_2$ of linear combinations of p columns along L rows from Y is constructed. Then matches from the two lists restricted to these L rows are found. Each match in the two lists gives a combination of L rows producing a codeword which has weight $2p$ along the redundant set. Each such codeword is examined along the information set and those codewords are selected which contribute weight $w-2p$ from this part, where w is the desired weight. This algorithm is run repeatedly with different initial column permutations.

To look for low weight codeword, one simply reports the lowest weight found using this approach without using a weight w in the algorithm a-priori. It is reported in [5] that this approach produces low weight codewords with significant probability for moderate sized codes.

3 Our Improvements to the Implementation of the Algorithms

We next describe our implementation level improvements to the algorithms described previously. These improvements are discussed with respect to the generator matrix based approach. We applied them for the check matrix based approach as well.

3.1 Word Oriented Approach

We use a 32-bit word oriented approach in the implementation of all the algorithms discussed. One principal cost of the algorithms is in computing the linear combination of rows. The step requiring linear combination of rows is implemented by a *for* loop which runs as many times as the length of the column. We represent 32 columns as one 32-bit word. With this word oriented approach, the linear combination of one row with another becomes 32 times faster because a word can be linearly combined with another word in one computer clock cycle.

In our implementation of the Canteaut-Chabaud algorithm, the swap of a column from the redundant set with a column from the information set is not done explicitly. Instead, only the part of the matrix restricted to the redundant set is modified so as to reflect the effect of the swap and the row operations which would be required to obtain identity matrix in the first part after the swap. We implement all these operation over 32-bit words.

3.2 Increasing the Number of Row Combinations

We experiment with obtaining a larger number of columns L from the redundant set such that the weight of the punctured code vanishes on them. As this number is increased, the probability for obtaining vanishing punctured code reduces. To be still able to obtain sufficient candidate row combinations of $2p$ rows to be tried on the full code $\mathcal{C}$, we need more rows to be linearly combined. Increasing the number of rows to combine makes sense even when the size of the punctured code is not increased. In this latter case, the quality of candidate row combinations may improve on the full code $\mathcal{C}$.

4 Our Improvements to the Techniques

We now discuss our improvements to the techniques of the algorithms described earlier.

4.1 Using Wagner's Generalized Birthday Attack

The algorithms discussed earlier use two lists, each of which is prepared by linear combinations of p rows of the generator (or check) matrix. The problem with increasing the number of rows to combine, for example p in the previous case, is that it is not practically feasible to attempt $\binom{k/2}{p}$ combinations when p is more than 3 or 4 for moderate values of k. To handle larger values of p, we incorporate a technique from [6]. The method in [6], which was earlier used in a special case in [2], is an extension of the birthday paradox in cryptography.

This time we create four lists of linear combinations of p rows on four disjoint sets of L columns from the redundant set. Using the tree based matching techniques of [6], we then find matching pairs restricted to the L columns from these four lists. Each matching pair gives us a combination of $4p$ rows which vanish on a punctured code of length L. Each of these matches is a good candidate for low weight codeword for the full code $\mathcal{C}$.

We can extend the approach outlined above and create 8, 16, $\ldots 2^t$ lists and then use a tree based method described in [6] to find matches in the lists. This will produce combinations of $2^t p$ rows, each of which vanishes on a punctured code of size L. This method can be useful when the length of the full code $\mathcal{C}$ is quite large. We investigated creation of up to 4 lists in the redundant set since the example codes we considered were not larger than length 512.

4.2 Closeness Parameterization of the Punctured Code

If the length of the punctured code is increased without increasing the number of row combinations, the probability of obtaining a codeword which vanishes on the punctured code is lowered. This means that some good candidate row combinations will not be considered while examining the full code. To handle this, we do not require the codewords to vanish completely on the punctured code, rather we look for a *low weight punctured code*. We implement this by a *closeness parameter*. A row combination which produces a codeword of weight less than *closeness parameter* in $\mathcal{C}'$ is examined as a candidate for low weight code for the full code $\mathcal{C}$.

The same technique can also be used in the multi-list approach discussed in Section 4.1. In that situation, the number of rows to combine have increased but we may not wish to lose on the candidate row combinations which produce a very low weight in the punctured code. We use a measure of *closeness of match* to keep such close matches in the candidate list for the full code.

5 A New Efficient Heuristic

The proposed heuristic works on the parity check matrix of the code. It starts with a random ordering of the columns of the check matrix $H_{(n-k)\times n}$. We attempt to obtain a weight $w = 2w_1$ using the proposed method.

We first partition the columns of the check matrix into two sets, say X and Y. The two sets contain $n/2$ columns each. We then create a list $\mathcal{L}_1$ by linearly combining w_1 columns from X. Each element of the list is a column vector of length $(n-k)$. Since the number of all w_1 combinations of $n/2$ columns is $\binom{n/2}{w_1}$, it may not be feasible to attempt all such combinations exhaustively. Therefore we propose to use a parameter l for the size of this list. We shortly describe a suitable value for this parameter. Similarly, we create a list $\mathcal{L}_2$ by linearly combining w_1 columns from Y. We propose to use the same size l for the list $\mathcal{L}_2$ as well.

If we can find a match in the two lists $\mathcal{L}_1$ and $\mathcal{L}_2$, then we have obtained $2w_1$ columns whose linear combination is zero and hence we have a codeword of weight $w = 2w_1$. In this case the heuristic is successful. If it is not, then we restart the search by randomizing the column ordering again. To obtain success, we need to ensure that the two lists have some intersection. Since there are a total of $2^{(n-k)}$ possible column vectors, birthday bound requires the size of the lists l to satisfy the bound $l \times l \geq 2^{(n-k)}$. Thus we need $l \geq 2^{(n-k)/2}$.

The above bound for l may still be large and it may be difficult to prepare lists this long. In such a case, we can extend the two list approach and prepare 4 or 8 or $\ldots 2^t$ lists, and look for a match among these lists by using Wagner's tree-based approach [6]. The advantage of our method is that it does not require any Gaussian to be performed. The cost of the algorithm is only in the preparation and matching of the lists, hence a single iteration of our algorithm is likely to be more efficient than a single iteration of other algorithms. We plan to do analysis of the rate of convergence of our algorithm later.

6 Results and Conclusions

We tested the algorithms of Leon, Stern and Canteaut-Chabaud on a code prepared by linearizing the 23-step SHA-256 hash function. In the linearization, all additions were replaced by XOR and the non-linear 3-input boolean functions f_{IF} and f_{MAJ} were approximated by their middle arguments. In the SHA-256 hash function, one block (512-bit) message is mapped to 8 registers of 32-bits each. Thus, its linearized version corresponds to a code having generator matrix of order 256×512. We tested the algorithms with improvements described in Sections 3 and 4. We found that the algorithm by Canteaut and Chabaud gave the best results. In a code of size 512, the method could produce a codeword of weight 79 which is detailed in hexadecimal in Table 1. The corresponding differential path is shown in Table 2. We are still implementing our new method described in Section 5.

Table 1. Least weight codeword of weight 79 for 23-step linearized version of SHA-256 found using Canteaut-Chabaud algorithm [3] with $p = 2$ and $L = 10$. The 16 words of 32-bit each are listed in hexadecimal.

0-3	c0080020 a0900121 40010212 0100088a
4-7	00100800 80020818 80808d02 40082000
8-11	42000682 92002000 409a0200 10200002
12-15	00590020 28025081 44100100 10520050

References

1. Elwyn R. Berlekamp, Robert J. McEliece, and Henk C. A. van Tilborg, *On the Inherent Intractability of Certain Coding Problems*, IEEE Transactions on Information Theory **24** (1978), 384–386.
2. Paul Camion and Jacques Patarin, *The Knapsack Hash Function proposed at Crypto'89 can be broken*, Advances in Cryptology - Eurocrypt '91, Workshop on the Theory and Application of of Cryptographic Techniques, Brighton, UK, April 8-11, 1991, Proceedings (Donald W. Davies, ed.), Lecture Notes in Computer Science, vol. 547, Springer, 1991, pp. 39–53.
3. Anne Canteaut and Florent Chabaud, *A New Algorithm for Finding Minimum-Weight Words in a Linear Code: Application to McEliece's Cryptosystem and to Narrow-Sense BCH Codes of Length 511*, IEEE Transactions on Information Theory **44** (1998), no. 1, 367–378.
4. Jeffrey S. Leon, *A probabilistic algorithm for computing minimum weights of large error-correcting codes*, IEEE Transactions on Information Theory **34** (1988), no. 5, 1354–1359.
5. Jacques Stern, *A Method for Finding Codewords of Small Weight*, Coding Theory and Applications (Gérard D. Cohen and Jacques Wolfmann, eds.), Lecture Notes in Computer Science, vol. 388, Springer, 1988, pp. 106–113.
6. David Wagner, *A Generalized Birthday Problem*, Advances in Cryptology - Crypto 2002, 22nd Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 2002, Proceedings (Moti Yung, ed.), Lecture Notes in Computer Science, vol. 2442, Springer, 2002, pp. 288–303.

Table 2. The 23-step SHA-256 differential path for the codeword given in Table 1. In this table ΔX stands for $X' \oplus X$.

Step i	δa_i	δb_i	δc_i	δd_i	δe_i	δf_i	δg_i	δh_i
IV	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
1	c0080020	00000000	00000000	00000000	c0080020	00000000	00000000	00000000
2	328cb309	c0080020	00000000	00000000	23883041	c0080020	00000000	00000000
3	6e95ef29	328cb309	c0080020	00000000	48bb7364	23883041	c0080020	00000000
4	c544699a	6e95ef29	328cb309	c0080020	829a704c	48bb7364	23883041	c0080020
5	7cb5d3f3	c544699a	6e95ef29	328cb309	3e0967aa	829a704c	48bb7364	23883041
6	2aa14c82	7cb5d3f3	c544699a	6e95ef29	4a90cab1	3e0967aa	829a704c	48bb7364
7	3a6a0d29	2aa14c82	7cb5d3f3	c544699a	43c13f73	4a90cab1	3e0967aa	829a704c
8	eebce7f7	3a6a0d29	2aa14c82	7cb5d3f3	8ebe361c	43c13f73	4a90cab1	3e0967aa
9	dc9de18b	eebce7f7	3a6a0d29	2aa14c82	adcdacf1	8ebe361c	43c13f73	4a90cab1
10	8a397f75	dc9de18b	eebce7f7	3a6a0d29	c2db67ff	adcdacf1	8ebe361c	43c13f73
11	63946125	8a397f75	dc9de18b	eebce7f7	f9bc55b9	c2db67ff	adcdacf1	8ebe361c
12	58c5a5d0	63946125	8a397f75	dc9de18b	3c0aac36	f9bc55b9	c2db67ff	adcdacf1
13	42f9c68c	58c5a5d0	63946125	8a397f75	d3d4a818	3c0aac36	f9bc55b9	c2db67ff
14	c7d14da2	42f9c68c	58c5a5d0	63946125	d6ebc061	d3d4a818	3c0aac36	f9bc55b9
15	4bd49e42	c7d14da2	42f9c68c	58c5a5d0	f36ddf16	d6ebc061	d3d4a818	3c0aac36
16	00000000	4bd49e42	c7d14da2	42f9c68c	ad8a9869	f36ddf16	d6ebc061	d3d4a818
17	00000000	00000000	4bd49e42	c7d14da2	092d58ce	ad8a9869	f36ddf16	d6ebc061
18	00000000	00000000	00000000	4bd49e42	c7d14da2	092d58ce	ad8a9869	f36ddf16
19	00000000	00000000	00000000	00000000	4bd49e42	c7d14da2	092d58ce	ad8a9869
20	00000000	00000000	00000000	00000000	00000000	4bd49e42	c7d14da2	092d58ce
21	00000000	00000000	00000000	00000000	00000000	00000000	4bd49e42	c7d14da2
22	00000000	00000000	00000000	00000000	00000000	00000000	00000000	4bd49e42
23	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000