Exploiting Coding Theory for Collision Attacks on SHA-1*

Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen

Institute for Applied Information Processing and Communications (IAIK)
Graz University of Technology, Austria
{Norbert.Pramstaller,Christian.Rechberger,Vincent.Rijmen}@iaik.tugraz.at

Abstract. In this article we show that coding theory can be exploited efficiently for the cryptanalysis of hash functions. We will mainly focus on SHA-1. We present different linear codes that are used to find low-weight differences that lead to a collision. We extend existing approaches and include recent results in the cryptanalysis of hash functions. With our approach we are able to find differences with very low weight. Based on the weight of these differences we conjecture the complexity for a collision attack on the full SHA-1.
Keywords: Linear code, low-weight vector, hash function, cryptanalysis, collision, SHA-1

1 Introduction

Hash functions are important cryptographic primitives. A hash function produces a hash value or message digest of fixed length for a given input message of arbitrary length. One of the required properties for a hash function is collision resistance. That means it should be practically infeasible to find two messages m and $m^* \neq m$ that produce the same hash value.

A lot of progress has been made during the last 10 years in the cryptanalysis of dedicated hash functions such as MD4, MD5, SHA-0, SHA-1 [1,5,6,12]. In 2004 and 2005, Wang *et al.* announced that they have broken the hash functions MD4, MD5, RIPEMD, HAVAL-128, SHA-0, and SHA-1 [14,16]. SHA-1, a widely used hash function in practice, has attracted the most attention and also in this article we will mainly focus on SHA-1.

Some of the attacks on SHA-1 exploit coding theory to find characteristics (propagation of an input difference through the compression function) that lead to a collision [10,12]. The basic idea is that the set of collision-producing differences can be described by a linear code. By applying probabilistic algorithms the attacker tries to find low-weight differences. The Hamming weight of the resulting low-weight differences directly maps to the complexity of the collision attack on SHA-1. Based on [10,12] we present several different linear codes that we use to search for low-weight differences. Our new approach is an extension of

* The work in this paper has been supported by the Austrian Science Fund (FWF), project P18138.

N.P. Smart (Ed.): Cryptography and Coding 2005, LNCS 3796, pp. 78–??, 2005.
http://www.springerlink.com/content/l0qp10m607864m72

the existing methods and includes some recent developments in the cryptanalysis of SHA-1. Furthermore, we present an algorithm that reduces the complexity of finding low-weight vectors for SHA-1 significantly compared to existing probabilistic algorithms. We are able to find very low-weight differences within minutes on an ordinary computer.

This article is structured as follows. In Section 2, we present the basic attack strategy and review recent results on the analysis of SHA-1. How we can construct linear codes to find low-weight collision-producing differences is shown in Section 3. Section 4 discusses probabilistic algorithms that can be used to search for low-weight differences. We also present an algorithm that leads to a remarkable decrease of the search complexity. The impact of the found low-weight differences on the complexity for a collision attack on SHA-1 is discussed in Section 5. In Section 6 we compare our low-weight difference with the vectors found by Wang *et al.* for the (academical) break of SHA-1. Finally, we draw conclusions in Section 7.

2 Finding collisions for SHA-1

In this section we shortly describe the hash function SHA-1. We present the basic attack strategy and review recent results in the analysis of SHA-1. For the remainder of this article we use the notation given in Table 1. Note that addition modulo 2 is denoted by '+' throughout the article.

Table 1. Used notation

notation	description
$A + B$	addition of A and B modulo 2 (XOR)
$A \vee B$	logical OR of two bit-strings A and B
M_t	input message word t (32-bits), index t starts with 0
W_t	expanded input message word t (32-bits), index t starts with 0
$A \ll n$	bit-rotation of A by n positions to the left
$A \gg n$	bit-rotation of A by n positions to the right
step	the SHA-1 compression function consists of 80 steps
round	the SHA-1 compression function consists of 4 rounds = 4×20 steps
A_j	bit value at position j
$A_{t,j}$	bit value at position j in step t
A'_j	bit difference at position j
$A'_{t,j}$	bit difference at position j in step t

2.1 Short description of SHA-1

The SHA family of hash functions is described in [11]. Briefly, the hash functions consist of two phases: a message expansion and a state update transformation. These phases are explained in more detail in the following. SHA-1 is currently

the most commonly used hash function. The predecessor of SHA-1 has the same state update but a simpler message expansion. Throughout the article we will always refer to SHA-1.

Message expansion. In SHA-1, the message expansion is defined as follows. The input message is split into 512-bit message blocks (after padding). A single message block is denoted by a row vector m. The message is also represented by 16 32-bit words, denoted by M_t, with $0 \leq t \leq 15$.

In the message expansion, this input is expanded linearly into 80 32-bit words W_t, also denoted as the 2560-bit expanded message row-vector w. The words W_t are defined as follows:

$$W_t = M_t, \qquad 0 \leq t \leq 15 \tag{1}$$
$$W_t = (W_{t-3} + W_{t-8} + W_{t-14} + W_{t-16}) \lll 1, \qquad 16 \leq t \leq 79\ . \tag{2}$$

Since the message expansion is linear, it can be described by a 512×2560 matrix $\mathbf{M}$ such that $w = m\mathbf{M}$. The message expansion starts with a copy of the message, cf. (1). Hence, there is a $512 \times 32(80-16)$ matrix $\mathbf{F}$ such that $\mathbf{M}$ can be written as:

$$\mathbf{M}_{512\times 2560} = [\mathbf{I}_{512}\ \mathbf{F}_{512\times 2048}]\ . \tag{3}$$

State update transformation. The state update transformation starts from a (fixed) initial value for 5 32-bit registers (referred to as iv) and updates them in 80 steps (0,...,79) by using the word W_t in step t. Figure 1 illustrates one step of the state update transformation. The function f depends on the step number: steps 0 to 19 (round 1) use the *IF-function* and steps 40 to 59 (round 3) use the *MAJ-function*:

$$f_{IF}(B,C,D) = BC + \overline{B}D \tag{4}$$
$$f_{MAJ}(B,C,D) = BC + BD + CD\ . \tag{5}$$

The remaining rounds 2 and 4, use a 3-input XOR referred to as f_{XOR}. A step constant K_t is added in every step. There are four different constants; one for each round. After the last application of the state update transformation, the initial register values are added to the final values (feed forward), and the result is either the input to the next iteration or the final hash value.

We linearize the state update by approximating f_{IF} and f_{MAJ} by a 3-input XOR. The linear state update can then be described by a 2560×160 matrix $\mathbf{S}$, a 160×160 matrix $\mathbf{T}$, and a vector k that produce the output vector o from the input message vector m:

$$o = m\mathbf{MS} + iv\mathbf{T} + k\ . \tag{6}$$

The (linear) transformation of the initial register value iv is described by the matrix $\mathbf{T}$. The constant k includes the step constants.

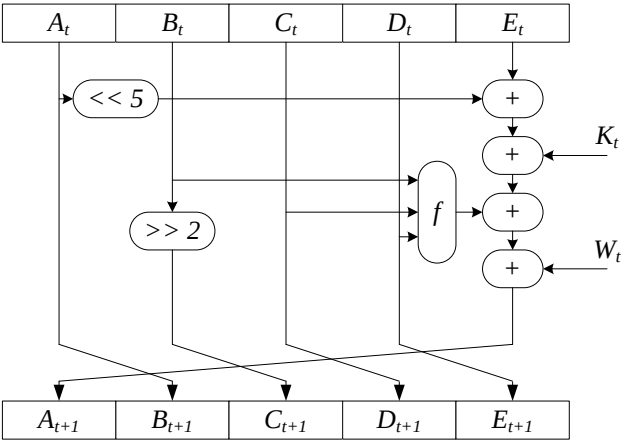

Fig. 1. One step of the linearized state update transformation of SHA-1

2.2 The basic attack strategy

Recent collision attacks on SHA-1 use the following strategy. First, a characteristic, *i.e.* the propagation of an input difference through the compression function of the hash function, is constructed. Second, messages are constructed, which follow the characteristic. This strategy is based on the attack on SHA-0 by Chabaud and Joux [5].

They observed that every collision for the linearized compression function of SHA (SHA-0, SHA-1) can be written as a linear combination of local collisions. These local collisions consist of a *perturbation* and several *corrections*. Rijmen and Oswald [12] described the first extension of this attack to SHA-1. Their method extends the Chabaud-Joux method and works with any characteristic that produces output difference zero.

Since a characteristic propagates in a deterministic way through a linear function, the characteristic is determined completely by the choice of the input difference. Hence, there are 2^{512} different characteristics. A fraction of 2^{-160} of these, results in a zero output difference (a collision). A difference corresponding to a characteristic is called a collision-producing difference.

Two messages m_1 and $m_2 = m_1 + \delta$ collide if

$$(m_1 + \delta)\mathbf{MS} - m_1\mathbf{MS} = 0 \Longleftrightarrow \delta\mathbf{MS} = 0, \tag{7}$$

where δ is the input difference. Therefore, we are interested in solutions of the following equation:

$$v\mathbf{S} = 0, \tag{8}$$

whereas $v = \delta\mathbf{M}$ represents a collision-producing difference. Among the set of 2^{352} solutions we are searching for a small subset where

– v has a low Hamming weight
– the probability for the characteristic to hold is maximal.

There is a strong correlation between these two requirements, which will be explained in Section 3. Using a suitable low-weight difference, the attack proceeds as follows:

– conditions for following the characteristic are derived
– some conditions are pre-fulfilled by setting certain bits in the message
– during the final search, the most naive approach to fulfill all remaining conditions is to preform random trials. The time-complexity of this final search is determined by the number of conditions which are not pre-fulfilled.

The problem of finding low-weight difference vectors is the main topic of this article. We present efficient algorithms to cover this search space in Section 4. Using the found low-weight difference, we describe a general way to derive conditions that need to hold in order to follow this difference in Section 5.

2.3 Overview of existing attacks on SHA-1

We now review recent advances in the analysis of SHA-1. The conclusions drawn in this section will be used in subsequent sections.

Using more than one message block. In multi-block collisions, we can also use characteristics that do not result in a zero output. For instance, for a two-block collision, all we require is that the output difference in both blocks is equal, because then, the final feed-forward will result in cancelation of the differences (with a certain probability). For an i-block collision, we get $512i - 160$ free bits ($512i - 320$ if we require that the perturbation pattern is a valid expanded message).

Easy conditions. For the second step of the attack, constructing a pair of messages that follows this characteristic, a number of conditions on message words and intermediate chaining variables need to be fulfilled. As already observed in [5], conditions on the first steps can be pre-fulfilled. Using the idea of *neutral bits*, this approach was extended to cover the first 20 steps of the compression function of SHA-0 [1]. Wang *et al.* and Klima do something similar for MD4 and MD5 to pre-fulfill conditions, which is there called message modification [8,15,17]. For this reason, whenever we refer to a weight of a characteristic (collision-producing difference), we omit the weight of the first 20 words, unless stated otherwise.

Exploiting non-linearity. The state update is a non-linear transformation, and this can be exploited during the construction of the characteristic. While for a linear transformation the characteristic is determined completely by the input

difference, in a non-linear transformation, one input difference can correspond to several characteristics.

Using a characteristic different from the one constructed from the linear approximation results in an overall increase of the number of equations. However, as explained before, the conditions in the first 15 steps are easy to fulfill. A good strategy is to look for a characteristic that has low weight and follows the linear approximation after the first 15 steps. This appears to be the strategy followed in [16]. A similar observation is made in [2,7]. We will use this strategy in Section 3.3 and Section 3.4.

3 From a set of collision-producing differences to a linear code

With the message expansion described by the matrix $\mathbf{M}_{512\times 2560} = [\mathbf{I}_{512} \times \mathbf{F}_{512\times 2048}]$ and the linearized state update described by the matrix $\mathbf{S}_{2560\times 160}$, the output (hash value) of one SHA-1 iteration is $o = m\mathbf{MS} + iv\mathbf{T} + k$ (cf. Section 2.1). Two messages m_1 and $m_1^* = m_1 + m_1'$ collide if:

$$o_1^* - o_1 = (m_1 + m_1')\mathbf{MS} + k - (m_1\mathbf{MS} + k) = m_1'\mathbf{MS} = 0\ . \tag{9}$$

Hence, the set of collision-producing differences is a linear code with check matrix $\mathbf{H}_{160\times 512} = (\mathbf{MS})^t$. The dimension of the code is $512-160 = 352$ and the length of the code is $n = 512$.

Observation 1 *The set of collision-producing differences is a linear code. Therefore, finding good low-weight characteristics corresponds to finding low-weight vectors in a linear code.*

Based on this observation we can now exploit well known and well studied methods of coding theory to search for low-weight differences. We are mainly interested in the existing probabilistic algorithms to search for low-weight vectors, since a low-weight difference corresponds to a low-weight codeword in a linear code. In the remainder of this section we present several linear codes representing the set of collision-producing differences for the linearized model of SHA-1 as described in Section 2. Note that if we talk about SHA-1 in this section, we always refer to the linearized model. For the different linear codes we also give the weights of the found differences. How the low-weight vectors are found is discussed in Section 4.

As described in Section 2, we are interested in finding low-weight differences that are valid expanded messages and collision producing. Later on, we apply the strategy discussed in Section 2.3, *i.e.* we do not require the difference to be collision-producing. With this approach we are able to find differences with lower weight. The found weights are summarized in Table 4.

3.1 Message expansion and state update—Code C_1

For our attack it is necessary to look at the expanded message words and therefore we define the following check matrix for the linear code C_1 with dimension $dim(C_1) = 352$ and length $n = 2560$:

$$\mathbf{H1}_{2208\times 2560} = \begin{bmatrix} \mathbf{S}^t{}_{160\times 2560} \\ \mathbf{F}^t_{2048\times 512}\ \mathbf{I}_{2048} \end{bmatrix} . \tag{10}$$

This check matrix is derived as following. First, we want to have a valid expanded message. Since $m\mathbf{M} = w = m_{1\times 512}[\mathbf{I}_{512}\mathbf{F}_{512\times 2048}]$ and $\mathbf{M}$ is a systematic generator matrix, we immediately get the check matrix $[\mathbf{F}^t_{2048\times 512}\mathbf{I}_{2048}]$. If a codeword w fulfills $w[\mathbf{F}^t_{2048\times 512}\mathbf{I}_{2048}]^t = 0$, w is a valid expanded message. Second, we require the codeword to be collision-producing. This condition is determined by the state update matrix $\mathbf{S}$. If $w\mathbf{S} = 0$ then w is collision-producing. Therefore, we have the check matrix $\mathbf{S}^t$. Combining these two check matrices leads to the check matrix $\mathbf{H1}$ in (10).

The resulting codewords of this check matrix are valid expanded messages and collision-producing differences. When applying a probabilistic algorithm to search for codewords in the code C_1 (see Section 4) we find a lowest weight of 436 for 80 steps. The same weight has also been found by Rijmen and Oswald in [12]. As already described in Section 2.2, we do not count the weight of the first 20 steps since we can pre-compute these messages such that the conditions are satisfied in the first 20 steps. The weights for different number of steps are listed in Table 2.

Table 2. Lowest weight found for code C_1

steps 0 . . . 79	steps 15 . . . 79*	steps 20 . . . 79
436	333	293

*weight also given in [12]

A thorough comparison of these results with the weights given by Matusiewicz and Pieprzyk in [10] is not possible. This is due to the fact that in [10] *perturbation* and *correction* patterns have to be valid expanded messages. Furthermore, Matusiewicz and Pieprzyk give only the weights for the *perturbation* patterns.

3.2 Message expansion only and multi-block messages—Code C_2

Instead of working with a single message block that leads to a collision, we can also work with multi-block messages that lead to a collision after i iterations (cf. Section 2.3). For instance take $i = 2$. After the first iteration we have an output difference $o'_1 \neq 0$ and after the second iteration we have a collision, *i.e.* $o'_2 = 0$.

The hash computation of two message is then given by

$$\begin{aligned} o_1 &= m_1\mathbf{MS} + iv\mathbf{T} + k \\ o_2 &= m_2\mathbf{MS} + o_1\mathbf{T} + k \\ &= m_2\mathbf{MS} + m_1\mathbf{MST} + \underbrace{iv\mathbf{T}^2 + k\mathbf{T} + k}_{constant} \ . \end{aligned}$$

Based on the same reasoning as for the check matrix **H1** in Section 3.1, we can construct a check matrix for two block messages as follows:

$$\mathbf{H2}_{4256\times5120} = \begin{bmatrix} \mathbf{ST}^t_{160\times2560} & \mathbf{S}^t_{160\times2560} \\ \mathbf{F}^t_{2048\times512}\mathbf{I}_{2048} & \mathbf{0}_{2048\times2560} \\ \mathbf{0}_{2048\times2560} & \mathbf{F}^t_{2048\times512}\mathbf{I}_{2048} \end{bmatrix} . \tag{11}$$

The same can be done for i message blocks that collide after i iterations. The output in iteration i is given by

$$o_i = \sum_{j=0}^{i-1} m_{i-j}\mathbf{MST}^j + \underbrace{iv\mathbf{T}^i + k\sum_{l=0}^{i-1}\mathbf{T}^l}_{constant} \ . \tag{12}$$

Searching for low-weight vectors for a two-block collision in C_2 with **H2** and a three-block collision with the check matrix **HM2** given in Appendix A, leads to the weights listed in Table 3.

Table 3. Weight for two and three message blocks

weight of collision-producing differences for steps 20-79				
two-block collision		three-block collision		
exp. message 1	exp. message 2	exp. message 1	exp. message 2	exp. message 3
152	198	107	130	144

As it can be seen in Table 3, using multi-block collisions results in a lower weight for each message block. The complexity for a collision attack is determined by the message block with the highest weight. Compared to the weight for a single-block collision in Table 2 (weight = 293), we achieve a remarkable improvement. However, as shown in Table 4, the weight of the chaining variables is very high. Why this weight is important and how we can reduce the weight of the chaining variables is presented in the following section.

3.3 Message expansion and state update—Code C_3

For deriving the conditions such that the difference vector propagates for the real SHA-1 in the same way as for the linearized model, we also have to count

the differences in the chaining variables (see Section 5). That means that for the previously derived collision-producing differences we still have to compute the weight in the chaining variables. It is clear that this leads to an increase of the total weight (see Table 4). Therefore, our new approach is to define a code that also counts in the chaining variables and to look for low-weight vectors in this larger code. This leads to lower weights for the total.

Furthermore, we now apply the strategy discussed in Section 2.3. In terms of our linear code, this means that we only require the codewords to be valid expanded messages and no longer to be collision-producing, *i.e.* they correspond to characteristics that produce zero ouput difference in the fully linearized compression function. This can be explained as follows. Our code considers only 60 out of 80 steps anyway. After 60 steps, we will have a non-zero difference. For a collision-producing difference, the 'ordinary' characteristic over the next 20 steps would bring this difference to zero. But in fact, for any difference with very large probability to hold (see Section 2.2), we will later be able to construct a special characteristic that maps the resulting difference to a zero difference in less than the remaining 20 steps. Hence, we can drop the requirement that the difference should be collision-producing. If we place the special steps at the beginning, then the number of conditions corresponding to the special steps can be ignored.

Searching for the codeword producing the lowest number of conditions in the last 60 steps, we will work backwards. Starting from a collision after step 79 (chaining variables $A_{80}, \ldots, E_{80}$), we will apply the inverse linearized state update transformation to compute the chaining variables for step 78,77,...,20. We obtain a generator matrix of the following form:

$$\mathbf{G3}_{512\times 11520} = [\mathbf{M}_{j\times n}\mathbf{A}_{j\times n}\mathbf{B}_{j\times n}\mathbf{C}_{j\times n}\mathbf{D}_{j\times n}\mathbf{E}_{j\times n}], \tag{13}$$

where $j = 512$ and $n = 1920$. The matrices $\mathbf{A}_{j\times n}, \ldots, \mathbf{E}_{j\times n}$ can easily be constructed by computing the state update transformation backwards starting from step 79 with $A'_{80} = B'_{80} = \cdots = E'_{80} = 0$ and ending at step 20. The matrix $\mathbf{M}_{j\times n}$ is defined in Section 2.1.

The matrix defined in (13) is a generator matrix for code C_3 with $dim(C_3) = 512$ and length $n = 11520$. The lowest weight we find for code C_3 is 297. Note, that this low-weight vector now also contains the weight of the chaining variables $A'_t, \ldots, E'_t$. The weight for the expanded message is only 127. Compared with the results of the previous sections (code C_1) we achieve a remarkable improvement by counting in the weight of the chaining variables and by only requiring that the codewords are valid expanded messages.

3.4 Message expansion, state update, and multi-block messages—Code C_4

As shown in Section 3.2, we are able to find differences with lower weight if we use multi-block messages. We will do the same for the code C_4. A multi-block collision with $i = 2$ is shown in Figure 2.

As it can be seen in Figure 2, if we have the same output difference for each iteration we have a collision after the second iteration due to the feed forward.

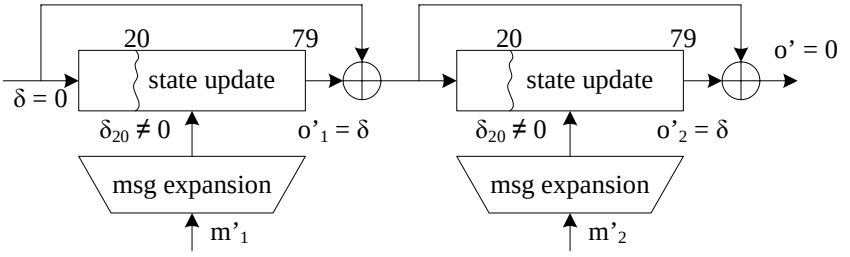

Fig. 2. Multi-block collision for SHA-1

We can construct a generator matrix as in Section 3.3 but we have to extend it such that we do not require a collision after the first iteration, *i.e.* we want an output difference of $o'_1 = o'_2 = \delta$. Therefore, we add 160 rows to the generator matrix in (13) that allow an output difference $o'_1 = o'_2 = \delta$. For the code C_4 we get a generator matrix

$$\mathbf{G4}_{672\times 11520} = \begin{bmatrix} \mathbf{M}_{j\times n} & \mathbf{A}_{j\times n} & \mathbf{B}_{j\times n} & \mathbf{C}_{j\times n} & \mathbf{D}_{j\times n} & \mathbf{E}_{j\times n} \\ \mathbf{0}_{l\times n} & \mathbf{A}'_{l\times n} & \mathbf{B}'_{l\times n} & \mathbf{C}'_{l\times n} & \mathbf{D}'_{l\times n} & \mathbf{E}'_{l\times n} \end{bmatrix}, \tag{14}$$

where $j = 512$, $l = 160$, and $n = 1920$. The matrix in (14) is a generator matrix for code C_4 with $dim(C_4) = 672$ and $n = 11520$. Searching for low-weight vectors in C_4 results in a smallest weight of 237. As we will show in Section 4, for this code it is infeasible to find codewords with weight 237 by using currently known algorithms (the same holds for code C_3). We found this difference vector by using an efficient way to reduce the search space as will be discussed in Section 4.2. Again, this weight includes also the weight of the chaining variables. For the message expansion only we have a weight of 108 (for one block we had 127). The difference vector is shown in Table 7, Appendix B.

3.5 Summary of found weights

To give an overview of the improvements achieved by constructing different codes we list the weights of the found codewords in Table 4.

Table 4. Summary of found weights

	Code C_1	**Code C_2**		**Code C_3**	**Code C_4**	
	single block	two-block		single-block	two-block	
		msg 1	msg 2		msg 1	msg 2
weight expanded message	293	152	198	127	108	108
weight state update	563	4730	4817	170	129	129
total weight	**856**	**4882**	**5015**	**297**	**237**	**237**

4 Finding low-weight vectors for linear codes representing the linearized SHA-1

In this section we describe different probabilistic algorithms that can be used to find low-weight vectors in linear codes. We describe the basic idea of these algorithms and present an algorithm that improves the low-weight vector search for SHA-1 significantly.

4.1 Probabilistic algorithms to find low-weight vectors

We will briefly discuss some probabilistic algorithms presented by Leon [9] and modified by Chabaud [4], Stern [13], and by Canteaut and Chabaud [3]. The basic approach of these algorithms is to take a (randomly permuted) subset of a code C and to search for low-weight vectors in this punctured code $C^\bullet$. A found low-weight codeword in the punctured code is a good candidate for a low-weight codeword in the initial code C.

A modified variant of Leon's algorithm [9] was presented by Chabaud [4]. It is applied to the generator matrix $\mathbf{G}_{k\times n}$ of a code C and defines the parameters p and s. The length of the punctured code $C^\bullet$ with generator matrix $\mathbf{Z} = \mathbf{Z}_{k\times(s-k)}$ is defined by s, where $s > dim(C^\bullet) = k$. For computing the codewords in $C^\bullet$ all linear combinations of at most p rows of $\mathbf{Z}$ are computed. The parameter p is usually 2 or 3. Values for the parameter s are $k+13, \ldots, k+20$ (see for instance [4]).

Stern's algorithm [13] is applied to the check matrix $\mathbf{H}_{(n-k)\times n}$. The parameters of the algorithm are l and p. The generator matrix $\mathbf{Z} = \mathbf{Z}_{l\times k}$ for the punctured code $C^\bullet$ is determined by k and l. The columns of $\mathbf{Z}$ are further split into two sets $\mathbf{Z}_1$ and $\mathbf{Z}_2$. Then the linear combinations of at most p columns are computed for both $\mathbf{Z}_1$ and $\mathbf{Z}_2$ and their weight is stored. Then searching for a collision of both weights allows to search for codewords of weight $2p$. Usually, the parameter p is 2 or 3 and l is at most 20 (see for instance [13]).

To compare these two algorithms we used the work-factor estimations to find an existing codeword with weight wt given by Chabaud [4]. For the comparison we used code C_4 (cf. Section 3.4) with $dim(C_4) = 672$, length $n = 11520$, and the weight $wt = 237$. The optimal parameters for Stern's algorithm are $p = 3$ and $l = 20$ for C_4. To find a codeword with $wt = 237$ in C_4 requires approximately 2^{50} elementary operations. Leon's algorithm, with parameters $p = 3$ and $s = dim(C_4) + 12$, requires approximately 2^{43} elementary operations.

Canteaut and Chabaud [3] have presented a modification of these algorithms. Instead of performing a Gaussian elimination after the random permutation in each iteration, Canteaut and Chabaud use a so-called 'Delta-Gauss' algorithm. More precisely, only two randomly selected columns are interchanged in each iteration, that is, only one step of a Gaussian elimination has to be performed. Even if this reduces the probability of finding a 'good' subset of the code, this approach leads to considerable improvements as they have shown for several codes in [3].

4.2 Improving low-weight search for SHA-1

During our research on the different codes we observed that the found low-weight vectors all have in common that the ones and zeroes occur in bands. More precisely, the ones in the expanded message words usually appear in the same position (see also Tables 6 and 7). This observation has also been reported by Rijmen and Oswald in [12]. This special property of the low-weight differences for SHA-1 can be used to improve the low-weight vector search as follows. By applying Algorithm 1 to the generator matrix we force certain bits in the codewords to zero. With this approach we are able to reduce the search space significantly. As already mentioned, the basic idea of the probabilistic algorithms described in the beginning of this section, is to use a randomly selected set of columns of the generator matrix $\mathbf{G}$ to construct the punctured code. This corresponds to a reduction of the search space. If we apply Algorithm 1 to $\mathbf{G}$, we actually do the same but we do not have any randomness in constructing the punctured code. Algorithm 1 shows the pseudo-code.

Algorithm 1. Forcing certain bits of the generator matrix to zero

Input: generator matrix $\mathbf{G}$ for code C, integer r defining the minimum $rank$ of $\mathbf{Z}$
Output: generator matrix $\mathbf{Z}$ for punctured code $C^\bullet$ with $rank(\mathbf{Z}) = r$

1: $\mathbf{Z} = \mathbf{G}$
2: **while** $rank(\mathbf{Z}) > r$ **do**
3: search in row x $(0 \leq x < rank(\mathbf{Z}))$ for a one in column y $(0 \leq y < length(\mathbf{Z}))$
4: add row x to all other rows that have a one in the same column
5: remove row x
6: **end while**
7: return $\mathbf{Z}$

Prior to applying the probabilistic search algorithms we apply Algorithm 1 to reduce the search space of the code. Since we force columns of the codewords to zero, we do not only reduce the dimension of the code but also the length. For the low-weight search we remove the zero-columns of $\mathbf{G}$. Computing the estimations for the complexities of this 'restricted code' shows that the expected number of operations decreases remarkably. For instance, applying Algorithm 1 to the generator matrix for code C_4 with $r = 50$ leads to the following values for the punctured code $C_4^\bullet$: $dim(C_4^\bullet) = 50$ and length $n = 2327$ (zero-columns removed). Stern's algorithm with optimal parameter $p = 2$ and $l = 4$ requires approx. 2^{37} elementary operations. For Leon's algorithm we get a work factor of approx. 2^{25} with $p = 3$ and $s = dim(C_4^\bullet) + 8$. With all the above-described algorithms we find the 237-weight difference within minutes on an ordinary computer by using Algorithm 1.

5 Low-weight vectors and their impact on the complexity of the attack

In this section we show how we can derive conditions for the low-weight differences found in Section 3. Based on the low-weight difference of code C_4, we will show some example conditions. The complexity for a collision attack depends on the number of conditions that have to be fulfilled. Since the number of conditions directly depends on the weight of the difference vector we see the correlation between weight and complexity: the lower the weight of the difference the lower the complexity for the attack.

The low-weight difference found for code C_4 leads to a collision after step 79 of the second compression function for the linearized SHA-1. Now, we want to define conditions such that the propagation of this difference is the same for the real SHA-1. In other words the conditions ensure that for this difference the real SHA-1 behaves like the linearized model.

As already mentioned in Section 2, the non-linear operations are f_{IF}, f_{MAJ}, and the addition modulo 2^{32}. Since we pre-compute message pairs such that all conditions in the first 20 steps are fulfilled, we only have to deal with f_{MAJ} and with the modular addition. For the addition we have to ensure that no carry occurs in the difference. For f_{MAJ}, we have to define conditions such that the differential behavior is the same as for f_{XOR}. Table 5 shows these conditions. For the sake of completeness also the conditions for f_{IF} are listed. Depending on the input difference we get conditions for the bit values of the inputs. For instance, if the input difference is $B'_jC'_jD'_j = 001$ then B_j and C_j have to be opposite, *i.e.* $B_j + C_j = 1$. The differential behavior of f_{MAJ} and f_{XOR} is the same if this condition is satisfied.

Table 5. Conditions that need to be fulfilled in order to have a differential behavior identical to that of an XOR

input difference $B'_jC'_jD'_j$	$f_{XOR}(B'_j, C'_j, D'_j)$	$f_{IF}(B'_j, C'_j, D'_j)$	$f_{MAJ}(B'_j, C'_j, D'_j)$
000	0	*always*	*always*
001	1	$B_j = 0$	$B_j + C_j = 1$
010	1	$B_j = 1$	$B_j + D_j = 1$
011	0	*never*	$C_j + D_j = 1$
100	1	$C_j + D_j = 1$	$C_j + D_j = 1$
101	0	$B_j + C_j + D_j = 0$	$B_j + D_j = 1$
110	0	$B_j + C_j + D_j = 0$	$B_j + C_j = 1$
111	1	$C_j + D_j = 0$	*always*

Now, we show an example how to derive conditions for f_{XOR} and f_{MAJ}. First, we take from Table 7 the difference corresponding to step $t = 28$ and bit position $j = 30$. We obtain the following:

$$A'_{t,j} = 0, B'_{t,j} = 0, C'_{t,j} = 1, D'_{t,j} = 0, E'_{t,j} = 0, A'_{t+1,j} = 0, W'_{t,j} = 1 \ .$$

For the following description we denote the output of f_{XOR} and f_{MAJ} by $F_{t,j}$.

Since $20 \leq t < 40$, the function f is f_{XOR}. Due to the input difference $C'_{t,j} = 1$ we always have $F'_{t,j} = 1$. Also $W'_{t,j} = 1$, and we have to ensure that there is no difference in the carry. This can be achieved by requiring that $F_{t,j}$ and $W_{t,j}$ have opposite values, *i.e.* $F_{t,j} + W_{t,j} = 1$. With $F_{t,j} = B_{t,j} + C_{t,j} + D_{t,j}$ we get $B_{t,j} + C_{t,j} + D_{t,j} + W_{t,j} = 1$. Since $B_t = A_{t-1}$, $C_t = A_{t-2} \ll 2$, $D_t = A_{t-3} \ll 2$, and $E_t = A_{t-4} \ll 2$, the condition for this example is:

$$A_{t-1,j} + A_{t-2,j+2} + A_{t-3,j+2} + W_{t,j} = 1 \ .$$

Second, we consider the difference for $t = 46$ and $j = 31$. This is the same difference as before but now f is f_{MAJ}, and therefore we have to ensure that f_{MAJ} behaves like f_{XOR}. For the input difference $B'_{t,j}C'_{t,j}D'_{t,j} = 010$, we first get the following condition (cf. Table 5): $B_{t,j} + D_{t,j} = 1$. If this condition is satisfied we have the same situation as for f_{XOR}, namely $F'_{t,j} = C'_{t,j}$. Different to the previous example we do not get any further condition because the difference occurs in bit-position 31. The difference for this example is:

$$A_{t-1,j} + A_{t-3,j+2} = 1 \ .$$

If we derive the equations (conditions) for the complete low-weight vector in Table 7 we get a set of 113 equations. The equations are either in A only or in A and W. We can rework some of the equations to get (linear) equations involving bits of the expanded message words W only. This equations can easily be solved since they can directly be mapped to conditions on the message words. After reworking the 113 equations, we get 31 in W only and 82 equations in A, and in A and W. The overall complexity of the attack is determined by the (nonlinear) equations involving bits of the chaining variables and/or expanded message words. This is due to the fact that after pre-fulfilling the conditions for the first 20 steps the remaining conditions are fulfilled using random trials. Hence, solving this 82 (nonlinear) equations takes at most 2^{82} steps.

6 Comparison with results of Wang *et al.*

In this section we compare the results of Wang *et al.* in [16] with the found low-weight difference given in Table 7. The difference in Table 7 is the lowest weight we found. The next higher weight we found (weight = 239) with the probabilistic search algorithms can also be constructed directly from the vector in Table 7. This is done by computing another iteration (see (2) and Figure 1) at the end and omitting the values of the first row such that we have again 60 steps. Since it is only a shifted version of the vector in Table 7 we can still use this vector for the comparison. The difference in Table 7, chaining variable A'_{t+1}, is the same disturbance vector as the one used by Wang *et al.* for near-collisions given in [16, Table 5] (italicized indices 20,...,79). To compare the two tables consider that Wang *et al.* index the steps from 1,...,80 (we from 0,...,79) but because Wang *et al.* use the shifted version the indices are the same except that the last pattern

(index 80 for Wang *et al.*) is missing in Table 7. Also the Hamming weight for round 2-4 given in [16, Table 6] for 80 steps is the same. In [16, Table 7] one can find the difference vectors and the according number of conditions. The number of conditions and the conjectured attack complexity we stated in the previous section is remarkable higher than the values from [16]. However, no details on the exact way to derive conditions is given in [16].

7 Conclusions

In this article we have shown how coding theory can be exploited efficiently for collision attacks on the hash function SHA-1. We gave an overview of existing attack strategies and presented a new approach that uses different linear codes for finding low-weight differences that lead to a collision. We also presented an algorithm that allows to find the low-weight differences very efficiently. Furthermore, we gave an outline on how we can derive conditions for the found low-weight difference. We have shown that the number of conditions and hence the complexity for a collision attack on SHA-1, directly depends on the Hamming weight of the low-weight differences found.

Currently we are still working on improving the condition generation phase to reduce the overall complexity of the collision attack on SHA-1. We will also extend our approach such that we can perform similar analyses of alternative hash functions such as the members of the SHA-2 family and RIPEMD-160.

Acknowledgements

We would like to thank Mario Lamberger for fruitful discussions and comments that improved the quality of this article.

References

1. Eli Biham and Rafi Chen. Near-Collisions of SHA-0. In *Proccedings of CRYPTO 2004*, volume 3152 of *LNCS*, pages 290–305. Springer, 2004.
2. Eli Biham, Rafi Chen, Antoine Joux, Patrick Carribault, Christophe Lemuet, and William Jalby. Collisions of SHA-0 and Reduced SHA-1. In *Proceedings of EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 36–57. Springer, 2005.
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(1):367–378, 1998.
4. Florent Chabaud. On the Security of Some Cryptosystems Based on Error-correcting Codes. In *Proceedings of EUROCRYPT 1994*, volume 950 of *LNCS*, pages 131–139. Springer, 1995.
5. Florent Chabaud and Antoine Joux. Differential Collisions in SHA-0. In *Proceedings of CRYPTO 1998*, volume 1462 of *LNCS*, pages 56–71. Springer, 1998.
6. Hans Dobbertin. Cryptanalysis of MD4. In *Proceedings of FSE 1996*, volume 1039 of *LNCS*, pages 53–69. Springer, 1996.

7. Antoine Joux, Patrick Carribault, William Jalby, and Christophe Lemuet. Full iterative differential collisions in SHA-0, 2004. Preprint.
8. Vlastimil Klima. Finding MD5 Collisions on a Notebook PC Using Multi-message Modifications, 2005. Preprint, available at `http://eprint.iacr.org/2005/102`.
9. Jeffrey S. Leon. A probabilistic algorithm for computing minimum weights of large error-correcting codes. *IEEE Transactions on Information Theory*, 34(5):1354–1359, 1988.
10. Krystian Matusiewicz and Josef Pieprzyk. Finding good differential patterns for attacks on SHA-1. In *Proccedings of WCC 2005*. Available online at `http://www.ics.mq.edu.au/~kmatus/FindingGD.pdf`.
11. National Institute of Standards and Technology (NIST). FIPS-180-2: Secure Hash Standard, August 2002. Available online at `http://www.itl.nist.gov/fipspubs/`.
12. Vincent Rijmen and Elisabeth Oswald. Update on SHA-1. In *Proceedings of CT-RSA 2005*, volume 3376 of *LNCS*, pages 58–71. Springer, 2005.
13. Jacques Stern. A method for finding codewords of small weight. In *Proccedings of Coding Theory and Applications 1988*, volume 388 of *LNCS*, pages 106–113. Springer, 1989.
14. Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Xiuyuan Yu. Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD, August 2004. Preprint, available at `http://eprint.iacr.org/2004/199`, presented at the Crypto 2004 rump session.
15. Xiaoyun Wang, Xuejia Lai, Dengguo Feng, Hui Chen, and Xiuyuan Yu. Cryptanalysis for Hash Functions MD4 and RIPEMD. In *Proceedings of EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 1–18. Springer, 2005.
16. Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding Collisions in the Full SHA-1. In *Proceedings of CRYPTO 2005*, volume 3621 of *LNCS*, pages 17–36. Springer,2005.
17. Xiaoyun Wang and Hongbo Yu. How to Break MD5 and Other Hash Functions. In *Proceedings of EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 19–35. Springer, 2005.

A Check matrix for 3-block collision

The hash output for three message blocks, is given by

$$o_3 = m_3\mathbf{MS} + m_2\mathbf{MST} + m_1\mathbf{MST}^2 + \underbrace{iv\mathbf{T}^3 + k\mathbf{T}^2 + k\mathbf{T} + k}_{constant} \ .$$

The set of collision-producing differences is a linear code with check matrix:

$$\mathbf{HM2}_{6304\times7680} = \begin{bmatrix} (\mathbf{ST}^2)^t_{160\times2560} & \mathbf{ST}^t_{160\times2560} & \mathbf{S}^t_{160\times2560} \\ \mathbf{F}^t_{2048\times512}\mathbf{I}_{2048} & \mathbf{0}_{2048\times2560} & \mathbf{0}_{2048\times2560} \\ \mathbf{0}_{2048\times2560} & \mathbf{F}^t_{2048\times512}\mathbf{I}_{2048} & \mathbf{0}_{2048\times2560} \\ \mathbf{0}_{2048\times2560} & \mathbf{0}_{2048\times2560} & \mathbf{F}^t_{2048\times512}\mathbf{I}_{2048} \end{bmatrix} . \quad (15)$$

B Found low-weight differences

Table 6. Lowest weight found for code C_2 — weight = 436

step	W_t	step	W_t
t=0	06E00000	t=40	1A780000
t=1	D9000000	t=41	F5000000
t=2	A2E00000	t=42	B7700000
t=3	82E00000	t=43	06800000
t=4	CD580000	t=44	78B00000
t=5	57500000	t=45	00000000
t=6	9B660000	t=46	6A900000
t=7	C0CE0000	t=47	60F00000
t=8	C0B20000	t=48	6C200000
t=9	D1F00000	t=49	E7100000
t=10	7D980000	t=50	8BC00000
t=11	C3BC0000	t=51	85D00000
t=12	3A500000	t=52	08000000
t=13	54C00000	t=53	80100000
t=14	BD840000	t=54	35000000
t=15	47BC0000	t=55	25900000
t=16	60E40000	t=56	82700000
t=17	6F280000	t=57	23200000
t=18	AB380000	t=58	C3200000
t=19	EDD00000	t=59	02400000
t=20	068C0000	t=60	B2000000
t=21	D0CC0000	t=61	47800000
t=22	17000000	t=62	63E00000
t=23	501C0000	t=63	20E00000
t=24	1A040000	t=64	44200000
t=25	D4C80000	t=65	84000000
t=26	99D80000	t=66	C0000000
t=27	C1500000	t=67	87400000
t=28	AB200000	t=68	16000000
t=29	B4D00000	t=69	44000000
t=30	16600000	t=70	A7A00000
t=31	47500000	t=71	50A00000
t=32	CA100000	t=72	82E00000
t=33	80A00000	t=73	C5800000
t=34	E6780000	t=74	23000000
t=35	6CB80000	t=75	80C00000
t=36	74180000	t=76	04C00000
t=37	44F00000	t=77	00C00000
t=38	EFB80000	t=78	01400000
t=39	8F380000	t=79	01000000

Table 7. Lowest weight found for code C_4 — weight = 237

step	W'_t	A'_{t+1}	B'_{t+1}	C'_{t+1}	D'_{t+1}	E'_{t+1}
t=20	80000040	00000000	00000002	00000000	a0000000	80000000
t=21	20000001	00000003	00000000	80000000	00000000	a0000000
t=22	20000060	00000000	00000003	00000000	80000000	00000000
t=23	80000001	00000002	00000000	c0000000	00000000	80000000
t=24	40000042	00000002	00000002	00000000	c0000000	00000000
t=25	c0000043	00000001	00000002	80000000	00000000	c0000000
t=26	40000022	00000000	00000001	80000000	80000000	00000000
t=27	00000003	00000002	00000000	40000000	80000000	80000000
t=28	40000042	00000002	00000002	00000000	40000000	80000000
t=29	c0000043	00000001	00000002	80000000	00000000	40000000
t=30	c0000022	00000000	00000001	80000000	80000000	00000000
t=31	00000001	00000000	00000000	40000000	80000000	80000000
t=32	40000002	00000002	00000000	00000000	40000000	80000000
t=33	c0000043	00000003	00000002	00000000	00000000	40000000
t=34	40000062	00000000	00000003	80000000	00000000	00000000
t=35	80000001	00000002	00000000	c0000000	80000000	00000000
t=36	40000042	00000002	00000002	00000000	c0000000	80000000
t=37	40000042	00000000	00000002	80000000	00000000	c0000000
t=38	40000002	00000000	00000000	80000000	80000000	00000000
t=39	00000002	00000002	00000000	00000000	80000000	80000000
t=40	00000040	00000000	00000002	00000000	00000000	80000000
t=41	80000002	00000000	00000000	80000000	00000000	00000000
t=42	80000000	00000000	00000000	00000000	80000000	00000000
t=43	80000002	00000002	00000000	00000000	00000000	80000000
t=44	80000040	00000000	00000002	00000000	00000000	00000000
t=45	00000000	00000002	00000000	80000000	00000000	00000000
t=46	80000040	00000000	00000002	00000000	80000000	00000000
t=47	80000000	00000002	00000000	80000000	00000000	80000000
t=48	00000040	00000000	00000002	00000000	80000000	00000000
t=49	80000000	00000002	00000000	80000000	00000000	80000000
t=50	00000040	00000000	00000002	00000000	80000000	00000000
t=51	80000002	00000000	00000000	80000000	00000000	80000000
t=52	00000000	00000000	00000000	00000000	80000000	00000000
t=53	80000000	00000000	00000000	00000000	00000000	80000000
t=54	80000000	00000000	00000000	00000000	00000000	00000000
t=55	00000000	00000000	00000000	00000000	00000000	00000000
t=56	00000000	00000000	00000000	00000000	00000000	00000000
t=57	00000000	00000000	00000000	00000000	00000000	00000000
t=58	00000000	00000000	00000000	00000000	00000000	00000000
t=59	00000000	00000000	00000000	00000000	00000000	00000000
t=60	00000000	00000000	00000000	00000000	00000000	00000000
t=61	00000000	00000000	00000000	00000000	00000000	00000000
t=62	00000000	00000000	00000000	00000000	00000000	00000000
t=63	00000000	00000000	00000000	00000000	00000000	00000000
t=64	00000000	00000000	00000000	00000000	00000000	00000000
t=65	00000004	00000004	00000000	00000000	00000000	00000000
t=66	00000080	00000000	00000004	00000000	00000000	00000000
t=67	00000004	00000000	00000000	00000001	00000000	00000000
t=68	00000009	00000008	00000000	00000000	00000001	00000000
t=69	00000101	00000000	00000008	00000000	00000000	00000001
t=70	00000009	00000000	00000000	00000002	00000000	00000000
t=71	00000012	00000010	00000000	00000000	00000002	00000000
t=72	00000202	00000000	00000010	00000000	00000000	00000002
t=73	0000001a	00000008	00000000	00000004	00000000	00000000
t=74	00000124	00000020	00000008	00000000	00000004	00000000
t=75	0000040c	00000000	00000020	00000002	00000000	00000004
t=76	00000026	00000000	00000000	00000008	00000002	00000000
t=77	0000004a	00000040	00000000	00000000	00000008	00000002
t=78	0000080a	00000000	00000040	00000000	00000000	00000008
t=79	00000060	00000028	00000000	00000010	00000000	00000000
weight	108	26	25	25	26	27