Using Random Error Correcting Codes in Near-Collision Attacks on Generic Hash-Functions

Inna Polak*, Adi Shamir**

Department of Computer Science and Applied Mathematics,
Weizmann Institute of Science
Rehovot 76100, Israel

Abstract. In this paper we consider the problem of finding a near-collision with Hamming distance bounded by r in a generic cryptographic hash function h whose outputs can be modeled as random n-bit strings. In 2011, Lamberger suggested a modified version of Pollard's rho method which computes a chain of values by alternately applying the hash function h and an error correcting code e to a random starting value x_0 until it cycles. This turns some (but not all) of the near-collisions in h into full collisions in $f = e \circ h$, which are easy to find. In 2012, Leurent improved Lamberger's memoryless algorithm by using any available amount of memory to store the endpoints of multiple chains of f values, and using Van Oorschot and Wiener's algorithm to find many full collisions in f, hoping that one of them will be an r-near-collision in h. This is currently the best known time/memory tradeoff algorithm for the problem.
The efficiency of both Lamberger's and Leurent's algorithms depend on the quality of their error correction code. Since they have to apply error correction to *any* bit string, they want to use perfect codes, but all the known constructions of such codes can correct only 1 or 3 errors. To deal with a larger number of errors, they recommend using a concatenation of many Hamming codes, each capable of correcting a single error in a particular subset of the bits, along with some projections. As we show in this paper, this is a suboptimal choice, which can be considerably improved by using randomly chosen linear codes instead of Hamming codes and storing a precomputed lookup table to make the error correction process efficient. We show both theoretically and experimentally that this is a better way to utilize the available memory, instead of devoting all the memory to the storage of chain endpoints. Compared to Leurent's algorithm, we demonstrate an improvement ratio which grows with the size of the problem. In particular, we experimentally verified an improvement ratio of about 3 in a small example with $n = 160$ and $r = 33$ which we implemented on a single PC, and mathematically predicted an improvement ratio of about 730 in a large example with $n = 1024$ and $r = 100$, using 2^{40} memory.

Keywords: hash function, near-collision, random-code, time-memory trade-off, generic attack

1 Introduction

A *hash function* h maps an arbitrarily long input into an n-bit digest. Cryptographically strong hash functions should be indistinguishable from random functions, and in particular it should be difficult to find collisions (defined as pairs (m_1, m_2) s.t. $m_1 \neq m_2$ and $h(m_1) = h(m_2)$) in fewer than $2^{n/2}$ evaluations of h.

In this paper we consider a weaker notion of collision called *r-near-collision*, in which up to r bits in $h(m_1)$ and $h(m_2)$ are allowed to be different. There are several reasons why we may want to study such near-collisions. First of all, in many applications such as the generation of cryptographic keys or MAC's, the standard output of a hash function is too long, and we use only a subset of its bits. In this case, a near-collision can become a real collision if the differing bits are thrown away. In addition, finding near-collisions is

* innapolak@gmail.com
** adi.shamir@weizmann.ac.il

often a useful first step when we try to find a multi-block collision in a hash function, as demonstrated in [20,12,18]). By studying the complexity of near-collision attacks on generic hash functions (which are modeled as random functions), we can get upper-bounds on the near-collision resistance of any concrete hash-function, but in some cases we can do much better (see [1,3,15]). Even when we cannot turn such a near-collision attack into a full collision attack, the mere existence of a better than expected near-collision attack may suffice to disqualify a new hash function proposal. Finally, the task of finding a near-collision can be used as a more flexible and accurate type of *proof-of-work* [5,13] than finding a full collision since there are more parameters that we can specify in defining the computational task.

Let us now introduce our notation. The finite field whose elements are $\{0,1\}^n$ is denoted by $\mathbb{F}_2^n$. We use the notation of d_H to describe the Hamming distance function, and $\|\|_H$ for the Hamming-weight. We call $x, y \in \mathbb{F}_2^n$ *R-close* vectors if $d_H(x,y) \leq R$. The ball of radius R around x in $\mathbb{F}_2^n$, defined as $\{y | d_H(y,x) \leq R\}$, is denoted by $B_R^n(x)$. Its volume is denoted by V_R^n, and is defined as

$$V_R^n = \sum_{i=0}^{R} \binom{n}{i}$$

Any r_1-near-collision is in particular also an r_2-near-collision for every $r_1 \leq r_2$ (when $r = 0$ it is simply a full-collision). Therefore, the difficulty of finding r-near-collision decreases as r increases. However, detecting such a collision as soon as it occurs becomes algorithmically harder as r increases. The probability of a random pair of points in $\mathbb{F}_2^n$ to be r-close is $q_r^n := V_r^n \cdot 2^{-n}$. By the birthday paradox, the first r-near-collision is thus expected to be seen after about $1/\sqrt{q_r^n} = 2^{n/2}/\sqrt{V_r^n}$ hash evaluations. These evaluations require less than $2^{n/2}$ time, but actually finding the unique near-collision among them requires more than $2^{n/2}$ time (since the property of being r-close for $r \geq 1$ is not transitive, there is no sorting order which will always place the nearly colliding values next to each other, see Appendix A). Note that if we continue to evaluate h on $O(2^{n/2})$ additional inputs, we expect to have one full-collision which is very easy to find in $O(2^{n/2})$ time (see [21]), and by definition it will also be an r-near-collision for any r. If we consider both the evaluation of h and the search step as unit time operations and try to minimize the total time complexity, our goal is to evaluate h on more points than absolutely necessary in order to make the search part faster, keeping each one of these complexities below the trivial bound of $2^{n/2}$.

When we consider the issue of memory complexity, there are many known algorithms ([6,2,16,17,14]) which use only constant or logarithmic amount of memory in order to find a full collision shortly after it is first created. Most of these algorithms are based on Pollard's rho method, which evaluates a chain of values of h, and uses the fact that any equality between two values on the chain implies an equality between their successors. Unfortunately, we cannot directly use this technique to find near-collisions, since the h-successors of two values which nearly collide can be arbitrarily far apart.

In 2011, Lamberger [8] suggested using a error correction code in order to turn some near-collisions into full collisions, and further studied such constructions in [7,9]. His proposed algorithm (described in greater detail in Section 2.1) uses a variant of Pollard's rho method which alternately applies the hash function h and the error correction operation e to some random initial value x_0 until it loops. He then hopes that the two colliding values after some e will be nearly colliding values after the previous h. In 2012, Leurent [10] extended Lamberger's algorithm by using Van Oorschot and Wiener's technique[19] to find many simultaneous collisions with an algorithm which can be parallelized. Leurent's algorithm is not memoryless, and suggests a time-memory tradeoff for the problem of finding near-

collisions. However, it used the same type of error correction codes that Lamberger used, along with some bit truncation (which can be viewed as a primitive type of error correction code whose code-words have zeroes in all the truncated positions).

Note that in standard error correction applications, we only have to correct bit strings which are in small balls surrounding each code-word, but the union of all these balls can be a tiny fraction of the whole space and thus we may be unable to change the vast majority of bit strings into code-words. In our application, we have to efficiently correct *any* bit string provided by h into a nearby codeword, and thus we want to use a *covering* code in which the union of all the sets of vectors which are corrected to each code-word exactly covers the space. A good code should minimize the following two types of errors in the algorithm: Two outputs of h may be very close to each other but will be missed by the algorithm if they are corrected by e into two different code-words, or they may be more than r apart but still mapped by e to the same code-word. The first type of error is very common since in a high dimensional space a random vector x is expected to be at maximal distance from its associated code-word, and thus even when we change x into a neighboring x' by a single bit flip, it is likely to move further away from the code-word and thus into the region surrounding a different code-word. The second type of error (which we call a "false alarm") is likely to occur when the error correction region around each code-word is a highly elongated ellipsoid rather than a sphere, which allows some pairs of vectors (e.g., at the opposite ends of the ellipsoid) to be mapped into the same code-word even though they are very far apart.

The ideal codes in our application are thus codes which partition the whole space into the disjoint union of spheres of the same radius. Such codes (called perfect codes) are extremely rare, and their known constructions can correct only one or three errors, which is too small for our application. To overcome this problem, Lamberger and Leurent proposed using a concatenation of several Hamming codes (which are perfect codes capable of correcting a single error), where each code is applied to a different set of bits. Unfortunately, this severely distorts the error correction regions surrounding each code-word. For example, if we divide $n = 240$ bits into 16 substrings and use a concatenation of 16 Hamming codes which can correct a single error among the 15 bits in each substring, then we can correct some combinations of 16 errors (if each error occurs in a different substring), but we cannot correct some combinations with just two errors (if they occur in the same substring). In addition, random patterns of 4 or more errors are likely to have such repetitions by the birthday paradox, and thus the average error correction capability of such concatenated codes is much smaller than the number of codes.

In this paper we propose to replace the Hamming codes by random linear error correcting codes denoted by $[n, k]$. Their code-words (which we hope to be uniformly distributed in the whole space) form a k-dimensional linear subset of vectors of length n over a finite field. The whole space can be partitioned into 2^k regions, where each region contains all the vectors which are closest to a particular code-word. A code has *covering radius* R if R is the smallest integer such that each region is contained in the R-ball around the code-word (and thus the union of these balls covers the entire space). A $[n, k]$ linear code with covering radius R is denoted by $[n, k]\, R$. In this case, when we find a collision in $f = e \circ h$, it is guaranteed to be a $2R$-near-collision in h by the triangle inequality.

Randomly selected linear codes do not have efficient error correction algorithms. We overcome this problem by devoting some of the available memory to a lookup table, which can be prepared in advance (but not for free - to have a fair comparison with previous algorithms we take this preprocessing time into account). We show that the performance of such codes differs from the theoretically best possible covering codes only by small constant factors, and is considerably better than the concatenation of Hamming codes proposed by

Lamberger and Leurent. In fact, the gap between the codes is already practically significant for small values of n, and grows in an unbounded way as we increase the parameters of the problem. In particular, we present experimental evidence that by using random codes we can improve Leurent's algorithm by a factor of at least 3 when trying to solve a simple problem such as finding a 33-near-collisions in the SHA-1 hash function, and present theoretical analysis which shows that finding 100-near-collisions in a hash function with $n = 1024$ can be improved by about three orders of magnitude for practical amounts of memory (see Table 1).

The paper is organized as follows. In Section 2 we describe previous algorithms for finding near-collisions. In Section 3 we analyze the properties of error correcting codes in the context of finding near-collisions. In Section 4 we consider random codes and linear random codes, and show how to construct them and how to implement their decoding function in constant time using a sufficiently large preprocessed lookup table. In Section 5 we show how previous memoryless algorithms for finding near-collisions can be improved by using random codes. In Section 6 we analyze the time-memory tradeoffs of the new algorithm and demonstrate that it improves Leurent's algorithm. We conclude in Section 7.

2 Previous Work

2.1 Lamberger's construction of a linear covering-code

Lamberger [8] tries to find codes which have an efficient error correction function e, have the desired covering radius R, and have a minimum number of code-words K. His algorithm then uses Pollard's Rho-method to find a collision in $e \circ h$ in $O(\sqrt{K})$ time. By the triangle inequality, the distance between two vectors decoded into the same code-word is at most $2R$, and thus he finds a $r = 2R$-near-collision in h.

Error correction codes which can correct any received message with errors in up to R coordinates are also covering-codes of radius R only if they are *perfect codes of radius* R [11]. Unfortunately, the only non-trivial known perfect codes in $\mathbb{F}_2^n$ are the $[23, 12]\,3$ Golay code and the $\left[2^i - 1, 2^i - i - 1\right] 1$ Hamming codes $\mathcal{H}_\mathrm{i}$ for $i \geq 1$. To efficiently handle more errors, Lamberger's approach is to concatenate several Hamming codes. Concatenation of m linear codes$[n_i, k_i]\,R_i$ for $1 \leq i \leq m$, results in $[n, k]\,R$ code where $n = \sum_{i=1}^{m} n_i$, $k = \sum_{i=1}^{m} k_i$ and $R = \sum_{i=1}^{m} R_i$.[1] The nearest-neighbor of a vector in $\mathbb{F}_2^n$ in such code is the nearest-neighbor in each one of the substrings separately. We denote such a concatenation of codes by the operator $\oplus$, and $t \times C$ will stand for $\bigoplus_{i=1}^{t} C$ (concatenation of the code C t times).

Hamming codes exist only for $n = 2^k - 1$. For arbitrary values of n, Lamberger suggests using an $[n, k]R$ code which is the concatenation of several Hamming codes of two consecutive sizes, along with the trivial projection code $[i, i]0$ in $\mathbb{F}_2^i$, i.e.,

$$\mathcal{H}_R^n = s \times \mathcal{H}_{l+1} \oplus (R - s) \times \mathcal{H}_\mathrm{l} \oplus \mathbb{F}_2^x \tag{2.1}$$

where $l := \left\lfloor \log_2\left(\frac{n}{R} + 1\right)\right\rfloor$, $s := \left\lfloor \frac{n - R\left(2^l - 1\right)}{2^l} \right\rfloor$ and $x := s \cdot \left(2^{l+1} - 1\right) + (R - s)\left(2^l - 1\right)$. The dimension of the code is:

$$k = n - R \cdot l - s \tag{2.2}$$

The size of the code is $K = 2^k$ and therefore his algorithm's complexity is $2^{k/2}$.

Lamberger proves that this method gives lower complexity than what can be achieved by projection alone (formally defined as the $[n, k]$ code $\mathcal{P}_k^n$ whose function $e = \pi_k^n$ sets certain $n - k$ coordinates to zero, which can also be viewed as the truncation of $n - k$ bits).

[1] The concatenation can be also presented as a direct sum of the codes

Similar analysis was carried out by Gordon [4], who compared the minimal-hamming-weight decoding functions of projection and random codes from the viewpoint of *locally sensitive hash-functions*, and proved that random codes are asymptotically better than projections in minimizing the distance between random points in the space and their corresponding code-words for given ratios of k/n when $n \to \infty$ [2]. Using $\pi^n_{n-r} \circ h$ in Pollard's algorithm, Lamberger finds an r-near-collision after about $2^{(n-r)/2}$ hash computations. It can be improved by truncating $2r+1$ bits, and using π^n_{n-2r-1}, for which a single trial of the Rho-method finds an r-near-collision with probability $1/2$. This gives an overall complexity of $2 \cdot 2^{(n-2r-1)/2} = 2^{(n+1)/2-r}$.

2.2 Time-memory trade-offs for near-collisions

Leurent in [10] provides a near-collision attack with a time-memory tradeoff. His main idea is to use the algorithm of Van Oorschot [19] for parallel collision search of many collisions when having some memory available.

Let $\pi^n_{n'}$ be the projection function that truncates $\tau = n - n'$, and let $\psi^{n'}_R$ be the decoding function of $\mathcal{H}^{n'}_R$ for a given R. Then the function used in Pollard's algorithm is $\psi^{n'}_R \circ \pi^n_{n'} \circ h$, and it finds as many full collisions in its domain as needed, until one of them happens to be an r-near-collision in the original hash function h. An equivalent representation of the code is:

$$\mathcal{Y}^n_{R,\tau} := \mathcal{H}^{n'}_R \oplus \mathcal{P}^\tau_0 \tag{2.3}$$

Let $p_{\tau,R}$ denote the probability that a detected near-collision in the algorithm is an r-near-collision, which can be calculated for given τ and R. Then $l_{\tau,R} := 1/p_{\tau,R}$ is the expected number of collisions that have to be considered until an r-near-collision in h is found. The dimensions of the codes $\mathcal{Y}^n_{R,\tau}$, which are the lengths of $\mathcal{H}^{n'}_R$, are denoted by $k_{\tau,R}$, and are given by Formula (2.2). The time-complexity of the algorithm is bounded by:

$$\left(\sqrt{\frac{\pi}{2}} + \frac{5\sqrt{l_{\tau,R}}}{\sqrt{M}}\right) \cdot \sqrt{l_{\tau,R}} \cdot 2^{k_{\tau,R}/2} \tag{2.4}$$

Leurent provides a script that calculates the complexity for every possible R, τ in the ranges $0 \le \tau \le n$ and $0 \le R \le 2r$, and returns the estimated optimal parameters which gives the lowest complexity bound.

3 Properties of Code-Systems for Finding Near-Collisions

We analyze algorithms for finding r-near-collisions ($r > 0$) using maps applied on the hash-values that increase the chance of colliding after the map for nearby hash values. We notice that error-correction codes are designed for different applications. Due to some analogous properties, we still use the term "decoding function" to describe the map, and the term "code" to describe the domain set of a map. However, the decoding function doesn't have to be the nearest-neighbor function. We will use the term *"code-system"* to describe the code together with its related decoding-function e. For convenience, we will use the same letter to describe the code-set and the code-system. The *radius of a code-system* in the generalized meaning is the maximum number of bit-flips made by the decoding function e.

The only near-collisions in h detectable by these algorithms are (m_1, m_2) such that both $h(m_1)$ and $h(m_1)$ are decoded to the same code-word ($e(h(m_1)) = e(h(m_2))$). This implies theoretical lower bounds for these methods.

[2] This is the special case with $p = 1/2$ of his claim, in which the distribution of the points in the n-dimensional space is uniform, and there are no assumptions about them being close to the code-words.

From now on, we will use the following notations:

- β_C will be the probability that a random pair x, y is decoded to the same code-word in C.
- $\rho_C(R)$ will be the *probability mass function* (PMF)[3] of the distance between pairs decoded to the same code-word in C.
- $\mathcal{R}_C(R)$ will be the *cumulative distribution function* (CDF) of ρ_C , which describes the probability for a pair decoded to the same code-word to be R-close.
- $\varphi_C(R)$ is the chance that a random R-close pair is decoded to the same code-word in C.

In this section we denote:

$A_r :=$ the event that x,y are r-close. When r is fixed, we will simply use A.

$B_C :=$ the event that a random pair of vectors x, y are decoded to the same code-word in C. When C is fixed, we will simply use B.

It is easy to verify that $\Pr[A] = q_r^n$ which is a constant for given n and r, and $\Pr[B_C] = \beta_C$.

The probability that a collision is detectable is $\Pr[A \wedge B]$, and by the birthday paradox, it requires $1/\sqrt{\Pr[A \wedge B]}$ trials, while the lower bound without the code is $1/\sqrt{\Pr[A]}$

By the conditional probability formula:

$$\Pr[A \wedge B] = \Pr[B|A] \cdot \Pr[A] = \Pr[A|B] \cdot \Pr[B] \tag{3.1}$$

Therefore, using the code increases the lower bound for hash computations by a factor of $1/\sqrt{\Pr[B|A]}$, which is exactly:

$$1/\sqrt{\varphi_C(r)} \tag{3.2}$$

The overall bound is:

$$\frac{1}{\sqrt{q_r^n} \cdot \sqrt{\varphi_C(r)}} \tag{3.3}$$

Hence, a higher $\varphi_C(r)$ is an indication of a potentially better code in terms of the number of hash calculations required.

We can also specifically look at an algorithm that ignores random collisions in the code-space until an r-near-collision is detected. The chance of a collision to be an r-near-collision is $\Pr[A|B]$ and therefore $1/\Pr[A|B]$ collisions are expected to be examined. As $\Pr[A|B] = \mathcal{R}_C(r)$, the time complexity of the algorithm is at least:

$$1/\mathcal{R}_C(r) \tag{3.4}$$

Hence, a higher $\mathcal{R}_C(r)$ is also an indication of a potentially better code in terms of the time complexity of the algorithm.

When we construct a code-system, there is a tradeoff between getting a higher φ_C and getting a higher $\mathcal{R}_C$. For example, the only code-system which satisfies $\varphi_C(r) = 1$ for $r \geq 1$ has a single code-word, and thus has the property $\Pr[A|B] = \Pr[A]$. So $1/\Pr[A] = 1/q_r^n$ computations are required, which is the square of the lower bound on hash-computations when the code is not used. However, code-systems which have equal β_C are comparable. By Equation (3.1):

$$\varphi_C(r) \cdot q_r^n = \mathcal{R}_C(r) \cdot \beta_C \tag{3.5}$$

[3] In statistics, a probability mass function f of a discrete random variable X is defined as: $f(k) = \Pr[X = k]$.

Thus, $\varphi_C(r)$ and $\mathcal{R}_C(r)$ are proportional to each other with ratio β_C/q_r^n. In particular, when the codes are of the same size and the pre-images of all the code-words are of uniform size, $\beta_C = 1/|C|$, and then:

$$\mathcal{R}_C(r) = (q_r^n \cdot |C|) \cdot \varphi_C(r) \tag{3.6}$$

Such special cases are $[n,k]$ linear codes, for which $|C| = 2^k$.

3.1 Concatenation of several code-systems

Lamberger's algorithm described in Section 2.1 uses covering codes which are a concatenation of error-correction codes. We can similarly combine several code-systems with codes $C_1, C_2, \ldots, C_t$ of lengths $n_1, n_2, \ldots, n_t$ by concatenating the codes into a code $C = \bigoplus_{i=1}^{t} C_i$ of length $n = \sum_{i=0}^{t} n_i$. The decoding function of the resulting system is the composition of all the decoding functions applied to their appropriate substrings. If the systems are of radii $R_1, R_2, \ldots, R_t$, the resultant system is of radius $R = \sum_{i=0}^{t} R_i$.

As the bits of random vectors are unrelated:

$$\beta_C = \Pr[B_C] = \Pr\left[\bigwedge_{i=1}^{t} B_{C_i}\right] = \prod_{i=1}^{t} \Pr[B_{C_i}] = \prod_{i=1}^{t} \beta_{C_i} \tag{3.7}$$

By the definition of the distribution ρ_C , it will be the convolution of all ρ_{C_i}:

$$\rho_C(R) = (\rho_{C_1} * \rho_{C_2} * \ldots * \rho_{C_t})[R] \tag{3.8}$$

The $\mathcal{R}_C(R)$ is the CDF of $\rho_C(R)$, and $\varphi_C(r)$ is given by Equation (3.5).

Let's denote:

$$V_C := \prod_{i=1}^{t} V_{R_i}^{n_i} \tag{3.9}$$

It describes the actual volume of a single code-word's pre-image if the concatenated codes are prefect, or an upper bound on the volume otherwise.

We have $\mathcal{R}_C(2R) = 1$ and we can calculate $\varphi_C(2R)$:

$$\beta_C = \prod_{i=1}^{t} \beta_{C_i} \leq \prod_{i=1}^{t} q_{R_i}^{n_i} = \prod_{i=1}^{t} \frac{V_{R_i}^{n_i}}{2^{n_i}} = \frac{\prod_{i=1}^{t} V_{R_i}^{n_i}}{2^n} = \frac{V_C}{2^n} \tag{3.10}$$

So:

$$\varphi_C(2R) = \frac{\beta_C}{q_{2R}^n} \leq \frac{V_C}{V_{2R}^n} \tag{3.11}$$

For perfect codes the inequalities becomes equalities. It is easy to see that V_C is strictly maximal when $t = 1$, and is V_R^n in that case. No other case reaches the bound of the perfect code of length n and radius R. The bound decreases when the vector is partitioned into more parts.

4 Random Codes

Suppose we could randomly sample a function $f : \{0,1\}^n \to \{0,1\}$ that returns 1 on $2^n/p$ of the inputs (i.e, it returns 1 with probability $p = 2^{-\mu}$ on a random point). We can use it as code-word indicator to define the code:

$$C_f = \{x \in \{0,1\}^n \,|\, f(x) = 1\}$$

Each element is decoded to its nearest-neighbor. To make it well defined in our metric, we have to break ties by defining a secondary order. For example, we can decide that among two vectors with the same Hamming distance to x, one is closer if the XOR between it and x has a larger numeric value in binary representation.

4.1 Limited radius version

We may force the above system into a system of some bounded radius R by changing the decoding function so that if there is no code-word in $B_R^n(x)$ then x is not corrected into any member of C_f, and remains unchanged. This applies to vectors in:

$$\widehat{C_f} := \overline{\bigcap_{c \in C_f} B_R^n(c)}$$

Let's denote the probability that x is decoded into C_f by α. Given the probability p, α is the complement of the probability of not succeeding in V_R^n trials:

$$\alpha = \alpha(n, R, p) = 1 - (1-p)^{V_R^n} \tag{4.1}$$

Notice that if $p = 1/V_R^n$ then:

$$\alpha = 1 - \left(1 - \frac{1}{V_R^n}\right)^{V_R^n} \approx 1 - \frac{1}{e}$$

Compared to a theoretical perfect code of radius R, β_C is smaller by α^2. However, due to the search order for code-words within $B_R^n(x)$, the function $\mathcal{R}(\cdot)$ increases and so does the bound in Equation (3.4). Therefore $\varphi(\cdot)$ decreases by less than α^2 and the lower bound in Equation (3.3) increases by at most $1/\alpha \approx 1.58$ which is a small constant.

4.2 Estimating the distribution functions

The size of the code is $|C| = 2^n \cdot p$ and the distribution of the code-words in the full space is expected to be close to uniform, so the pre-images of the code-words are of similar volumes and therefore:

$$\beta_C \approx 1/|C| = \frac{2^{-n}}{p} \tag{4.2}$$

So $\mathcal{R}_C$ and φ_C are related by Equation (3.6). By definition, $\mathcal{R}_C$ can be calculated as CDF of ρ_C, and we describe how to calculate the latter in Appendix B.

4.3 Linear random code

If f can be easily sampled or calculated, the decoding of a single x can be conducted by an average of $1/p$ evaluations of f. Even though the decoding does not require any hash-evaluations, this may be a high complexity operation for a small p. For a truly random function a full description of the decoding process, for example using a look-up table, would be impractical in terms of memory. However, as described below the situation is much better for linear codes.

A random $[n, n-\mu]$ code can be defined as the kernel of a randomly chosen matrix $A \in \{0,1\}^{\mu \times n}$ of maximal rank. We denote such a code by $\mathcal{C}_\mu^n$. The code-word indicator function $f(x)$ returns 1 when $Ax = 0$. In a neighborhood of a randomly chosen x it behaves similarly to a random function that returns 1 with probability $p = 2^{-\mu}$. As shown in Appendix C, a table of size $K = 2^\mu$ that describes the decoding operation in the vicinity of the zero code-word makes it possible to find the nearest neighbor in the entire space in constant time via simple shifts.

Even though we count the memory in units which are table entries, we would like to point out that due to the fact that we store only vectors of very low hamming-weight, the content of the table can be compressed and stored much more efficiently.

Read-Only-Memory vs. Random-Access-Memory Notice that the lookup table we construct does not depend on the hash-function, but only on the chosen linear code. For a constant matrix A which is randomly chosen in advance, the table can be constructed and hard-coded on a Read-Only-Memory (ROM), which is much cheaper than Random-Access-Memory (RAM), and thus a special-purpose computing machine may have more of it. Therefore, in scenarios which have almost no RAM, where we would normally use the Rho-method to find near-collisions, we may still use a large look-up table if we have enough ROM. Such a lookup table could also be stored on an external device or some common server which can be queried.

5 Rho-Method Algorithm using Random-Codes

In Section 2.1 we described Lamberger's algorithm for finding $2R$-near-collisions which is based on a single run of the Rho-method algorithm. For a general code-system C with the restriction of $\mathcal{R}_C = 1$, the complexity of this algorithm is:

$$1/\sqrt{\beta_C}$$

For a random-code with limited radius R, the complexity is larger by a factor of about $1/\alpha$ compared to a theoretical perfect code (see Section 4). The gap between the β of Lamberger's construction and a perfect code grows as R grows, due to the partition of the vector into more parts, as can be seen in Formula (3.10).

For example when looking for a 24-near-collision in $\mathbb{F}_2^{128}$, a random-code could improve Lamberger's algorithm by a significant factor of 69.3, ignoring the cost of the decoding. However, a linear-random-code of radius R requires a table of about V_R^n size, which is impractical for V_{12}^{128}.

Using a random-code-system $2\times\mathcal{C}_{26}^{64}$, when $\mathcal{C}_{26}^{64}$ has limited radius 6 and $\alpha \approx 0.7$, requires a lookup-table of practical size 2^{26}. We get $\beta_C = \left(\beta_{\mathcal{C}_{26}^{64}}\right)^2$ and $\beta_{\mathcal{C}_{26}^{64}} = \alpha^2 \cdot 2^{-64} \cdot V_6^{64} \approx 2^{-38.67}$. The total expected number of hash evaluations is $1/\sqrt{\beta_C} = 1/\beta_{\mathcal{C}_{26}^{64}} = 2^{38.67}$ (multiplied by a small constant that depends on the Rho-method's implementation, and can be ignored since it affects the two algorithms we compare similarly). Lamberger's algorithm uses a $[128, 87]12$ code and makes an expected number of $2^{43.5}$ hash evaluations. Therefore our code improves Lamberger's algorithm's time-complexity in this case by a factor of 28.4.

When we do not restrict the code to have $\mathcal{R}_C = 1$, the Rho-method has to be repeated $l_C := 1/\mathcal{R}_C$ times on average and the complexity increases to:

$$l_C/\sqrt{\beta_C}$$

Due to the large number of available parameters and the fact that we do not want to ignore constant factors, we estimated the optimal parameters for a random-code and for Lamberger's construction using a script which exhaustively searches over all their possible choices rather than via some asymptotic formula. For Lamberger's construction, we got that the optimal radius for finding a 24-near-collision is 28, and the number of expected hash-computations is about $2^{39.1}$, where $l_C \approx 8.6$. Using a linear random-code $2 \times \mathcal{C}_{34}^{128}$ our algorithm finds 24-near-collision after an expected number of $2^{35.3}$ hash computations, when $l_C \approx 39.1$. This is an improvement by a factor of 14.2.

6 Time-Memory Trade-off using Random-Codes

In Section 2.2 we described Leurent's algorithm [10] for finding near-collisions, using a table of size $M = 2^m$ to store the endpoints of chains in Van Oorschot and Wiener's algorithm.

Our goal is to improve the algorithm by using random-codes instead of the concatenation of Hamming-codes.

Although we can sometimes distinguish between the memory that is used for storing the chains and the memory that is used to store the look-up tables (as described in Section 4.3), in this section we consider the harder case in which we have only one type of memory of size M. Therefore, we are limited to use not more than M memory units *including* the look-up table of the random-codes. Considering the fact that we can compress the table (see the remark in Section 4.3) and the fact that when such codes are concatenated more than once we still store the table only once, we can assume for the sake of simplicity that we can use a random-code with up to m equations, and still consume only a small fraction of the available memory for its associated lookup table.

The code-systems we use are of the form:

$$\mathcal{Z}^n_{\mu,j,\tau} := j \times \mathcal{C}^{n'}_{\mu} \oplus \mathbb{F}_2^x \oplus \mathcal{P}^{\tau}_0 \tag{6.1}$$

where $n' = \lfloor (n-\tau)/j \rfloor$ and $x = (n-\tau) \mod j$.

When using the linear version of $\mathcal{C}^{n'}_{\mu}$, $\mathcal{Z}^n_{\mu,j,\tau}$ is a $[n, n-j\cdot\mu-\tau]$ linear code. The projection $\mathcal{P}^{\tau}_0$, for any value of τ, is a code-system that decodes all the vectors in $\mathbb{F}_2^{\tau}$ into the a single code-word and therefore has the properties: $\beta_{\mathcal{P}^{\tau}_0} = 1$, $\varphi_{\mathcal{P}^{\tau}_0} \equiv 1$ and $\mathcal{R}_{\mathcal{P}^{\tau}_0}$ distributes according to the binomial distribution $N\left(\tau, \frac{1}{2}\right)$. We calculate the distribution $\mathcal{R}_{\mathcal{Z}^n_{\mu,j,\tau}}(R)$ as a convolution between $N\left(\tau, \frac{1}{2}\right)$ and j times $\mathcal{R}_{\mathcal{C}^{n'}_{\mu}}(R)$. Then, the expected number of distinct collisions we have to find is:

$$l_{\mu,j,\tau} := 1/\mathcal{R}_{\mathcal{Z}^n_{\mu,j,\tau}}(r) \tag{6.2}$$

In order to optimize the algorithm, we choose τ and μ that minimize the following upper bound on the time complexity:

$$T(\mu,j,\tau) = \left(\sqrt{\frac{\pi}{2}} + \frac{5\sqrt{l_{\mu,j,\tau}}}{\sqrt{M}}\right) \cdot \sqrt{l_{\mu,j,\tau}} \cdot 2^{(n-j\cdot\tau-\mu)/2} \tag{6.3}$$

This can be done using brute-force computations of at most one value of $T(\mu,j,\tau)$ for every possible $\tau, j\cdot\mu \leq n$ (in fact, most of the parameters within the range can be easily ruled-out by simple estimations). We used a script to compute the exact values of the optimal parameters (see Appendix D).

We can generalize the formula for code-systems which are not necessarily linear [4]. Given a code-system C and the parameter r of the problem, if we calculate β_C and $\mathcal{R}_C(r)$, we can get the following complexity upper bound:

$$T = \left(\sqrt{\frac{\pi}{2}} + \frac{5}{\sqrt{M\cdot\mathcal{R}_C(r)}}\right) \cdot \sqrt{\frac{1}{\mathcal{R}_C(r)\cdot\beta_C}} \tag{6.4}$$

We would like to remind the reader that $\mathcal{R}_C(r)\cdot\beta_C = \varphi_C(r)\cdot q_r^n$ by Equation (3.5) which describes the probability of a random pair of vectors to be a detectable r-near-collision. This is another way to see that the significance of $\varphi_C(r)$ over $\mathcal{R}_C(r)$ grows when we have more memory. This form also emphasizes how Formula (6.3) should be adapted when limiting the radius.

In some application we may want to find a large number i of r-near-collisions. In this case we will have to find $i/\mathcal{R}_C(r)$ collisions and the formula becomes:

[4] Van Oorschot's analyzed his algorithm for hash-functions that induce random-graphs on their output domains. We assume that the special properties of the graph induced by$Dec\circ h$ do not affect much the performance of the algorithm. However, such effect may exist and could be further analyzed.

$$T = \left(\sqrt{\frac{\pi}{2}} + \frac{5 \cdot i}{\sqrt{M \cdot \mathcal{R}_C(r)}}\right) \cdot \sqrt{\frac{i}{\mathcal{R}_C(r) \cdot \beta_C}} \tag{6.5}$$

6.1 Complexity analysis

For relevant parameters, a single optimal random-code is better than 3 or more concatenated Hamming-Codes and this advantage grows when the number of concatenated codes increases, because it is particularly true for random-codes of similar dimension (which are comparable). However, the memory requirements for these random-codes also grows. The truncation-code $\mathcal{P}_0^\tau$ may have low $\mathcal{R}_{\mathcal{P}_0^\tau}$ for large τ, but at the same time it has the property $\varphi_{\mathcal{P}_0^\tau} \equiv 1$, which cannot be achieved by any other code-system. As was shown before, the importance of the φ distribution over $\mathcal{R}$ grows as M gets larger. Thus, generally speaking, the optimal τ is higher for larger M values, and the optimal code on the remaining part makes fewer bit-flips on random inputs. This can also be seen in [10, Table 1]. Thus, on one hand, when we have little memory we may not be able to construct the optimal random-code, and on the other hand, when we have a lot of memory the random-code does not improve much if at all.

Although the concatenation of random-codes is never optimal when we ignore memory considerations, the actual memory requirements for codes of smaller dimensions are much smaller. Moreover, a concatenation of a competitive random-code several times requires only a single common lookup-table, and by the relations described in Section 3.1, the improvement factors over Lamberger's construction of similar dimensions are roughly multiplied. Therefore, the advantage of random-codes grows with n, since we can truncate some of the bits and still have enough bits to partition into large-enough bit-ranges for which practical sized random-codes can be constructed. In Table 1 we show concrete sets of parameters for which our algorithm improves Leurent's algorithm by several orders of magnitude.

Table 1. Comparison of number of the hash-calculations using Leurent's algorithm that uses Lamberger's construction versus our algorithm that uses random-codes in two variants: with and without limiting the radius. The upper entry is based on experimental results. The lower 3 entries are predictions based on calculations made using a script. T values are in logarithmic scale.

		Leurent's algorithm [(2.3),(2.4)]	Our algorithm [(6.1),(6.3)]	Improvement ratio
n, r	m	$T\ (R, \tau)$	$T\ (\mu, j, \tau)$	
160, 33	16	above 35.06 (2, 106)[a]	33.46 (15, 1, 98) - limited radius $R = 3$	above 3
			34.91 (15, 1, 98)	minor
1024, 80	38	389.7 (14, 206)	381.3 (38, 6, 70)	354.5
1024, 100	40	363.6 (13, 269)	354.1 (40, 7, 79)	728.8
1024, 100	52	358.6 (12, 294)	348.1 (52, 7, 16)	1482

[a] In 10 executions of Leurent's algorithm we got one unusually bad result ($2^{38.4}$ hash-computations) and two mildly bad ones. It could be a result of unoptimized elements of the algorithm. Thus, we reduced the time-complexity of Leurent's algorithm by artificially terminating it after a certain number of steps, which is the number of computations as in the 2nd worst result, and took into consideration that our effort resulted in 9 successful experiments instead of 10. Van Oorschot suggests to restart the process after $10M = 10 \cdot 2^{16}$ collisions, but then it would not be optimal. However, we did not modify the experimental results of our algorithm in any way, and thus, the comparison in this table is actually biased in favor of Leurent's algorithm. In spite of this, our algorithm is about 3 times better in this small example in the limited radius version.

Although the predicted upper bounds for the case we tested experimentally were similar in Leurent's algorithm and in the two versions of our algorithm, in practice our code with

limited radius improved Leurent's algorithm for finding a 33-near-collision in SHA-1 hash function using 2^{16} memory by a factor of at least 3. This indicates that the improvement factors we obtain may be even higher than those we predict from our script. When we consider larger values of n, the improvement factors become much larger. For example, even our pessimistic estimates indicate that our algorithm is expected to improve Laurent's algorithm by a factor of about 730, which is almost three orders of magnitude, when looking for a 100-near-collision in a 1024-dimensional space using 2^{40} memory.

7 Conclusions and Further Work

In this paper we analyzed the two major statistical properties that make certain codes better for finding near-collisions, which are $\varphi(r)$ and $\mathcal{R}(r)$. We showed how to choose the optimal parameters of these random codes, described how to use lookup tables in order to decode an arbitrary vector into its related code-words in constant time, and discussed their advantages and disadvantages. We saw that random-codes have better properties than the concatenation of Hamming-codes of radius 1 for overall radii larger than 3, and that the gaps grow when the radius grows. We re-analyzed the time-memory trade-off of Leurent's algorithm after replacing the Hamming-codes with random-codes.

If we are allowed to use an unbounded amount of cheap ROM to store the fixed table used to decode vectors into their nearest code-words, we can achieve even larger improvements in many ranges of the parameters. For example, we can improve Lamberger's construction which uses the rho method by a factor of 69.3 in the settings described in Section 5. Without this assumption, i.e, when we had to use the available memory both for the table and for the endpoints, we still showed experimentally that for a small example there is a reduction of the number of hash evaluations by a factor of at least 3. The improvement ratio increases with n, and in Table 1 we showed concrete examples in which the improvement ratio is several orders of magnitude.

7.1 Further Work

We tried to analyze a *multi-code* variant of a random-code with limited radius R. Instead of leaving $1-\alpha$ fraction of the hash-values unchanged by the decoding function, we serially try a sequence of decoding-functions with various linear shifts of the code-words, until one of them succeeds. In other words, we used the same random matrix A to define a series of codes as the solutions of $Ax = v_i$. We chose $v_0 = 0$ and then each v_i was chosen from among the image-points whose pre-images cannot be decoded into the previous codes. This way, the distances between different code-components are at least R. This variation has negligible effect on $\mathcal{R}_C$ but increases the β_C and φ_C to about $\frac{\alpha}{2-\alpha}$ of the linear-version instead of α^2. However, in our practical experiments we did not get conclusive results about the effect on the time-complexity when using Van Oorschot's algorithm. This is possibly due to the effect of both variations on the statistical properties of the graph induced by $e \circ h$, which influences the probabilities of having new collisions overviewed along the run of the algorithm. This effect should be further analyzed.

We also suggest to consider other models in which the distance function is not the Hamming-distance. For example, we can consider a *weighted*-Hamming-distance in which bits have weights that correspond to the probability that they will be discarded when we extract a smaller number of random bits from the large output of the hash function.

References

1. Eli Biham and Rafi Chen. Near-collisions of sha-0. In *Advances in Cryptology–CRYPTO 2004*, pages 290–305. Springer, 2004.
2. Richard P Brent. An improved monte carlo factorization algorithm. *BIT Numerical Mathematics*, 20(2):176–184, 1980.
3. Florent Chabaud and Antoine Joux. Differential collisions in sha-0. In *Advances in Cryptology–CRYPTO'98*, pages 56–71. Springer, 1998.
4. Daniel M Gordon, Victor Miller, and Peter Ostapenko. Optimal hash functions for approximate closest pairs on the n-cube. *arXiv preprint arXiv:0806.3284*, 2008.
5. Markus Jakobsson and Ari Juels. Proofs of work and bread pudding protocols. In *Secure Information Networks*, pages 258–272. Springer, 1999.
6. DE Knuth. Seminumerical algorithm (arithmetic) the art of computer programming vol. 2, 1981.
7. Mario Lamberger, Florian Mendel, Vincent Rijmen, and Koen Simoens. Memoryless near-collisions via coding theory. *Designs, Codes and Cryptography*, 62(1):1–18, 2012.
8. Mario Lamberger and Vincent Rijmen. Optimal covering codes for finding near-collisions. In *Selected Areas in Cryptography*, pages 187–197. Springer, 2011.
9. Mario Lamberger and Elmar Teufl. Memoryless near-collisions, revisited. *Information Processing Letters*, 113(3):60–66, 2013.
10. Gaetan Leurent. Time-memory trade-offs for near-collisions. *IACR Cryptology ePrint Archive*, 2012:731, 2012.
11. Florence Jessie MacWilliams and Neil James Alexander Sloane. *The theory of error-correcting codes*, volume 16. Elsevier, 1977.
12. Florian Mendel and Martin Schläffer. On free-start collisions and collisions for tib3. In *Information Security*, pages 95–106. Springer, 2009.
13. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Consulted*, 1:2012, 2008.
14. Gabriel Nivasch. Cycle detection using a stack. *Information Processing Letters*, 90(3):135 – 140, 2004.
15. Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen. Exploiting coding theory for collision attacks on sha-1. In *Cryptography and Coding*, pages 78–95. Springer, 2005.
16. Jean-Jacques Quisquater and Jean-Paui Delescaille. How easy is collision search. new results and applications to des. In *Advances in Cryptology-Crypto'89 Proceedings*, pages 408–413. Springer, 1990.
17. Robert Sedgewick, Thomas G Szymanski, and Andrew C Yao. The complexity of finding cycles in periodic functions. *SIAM Journal on Computing*, 11(2):376–390, 1982.
18. Marc Stevens, Alexander Sotirov, Jacob Appelbaum, Arjen Lenstra, David Molnar, Dag Arne Osvik, and Benne De Weger. Short chosen-prefix collisions for md5 and the creation of a rogue ca certificate. In *Advances in Cryptology-CRYPTO 2009*, pages 55–69. Springer, 2009.
19. Paul C Van Oorschot and Michael J Wiener. Parallel collision search with cryptanalytic applications. *Journal of cryptology*, 12(1):1–28, 1999.
20. Xiaoyun Wang and Hongbo Yu. How to break md5 and other hash functions. In *Advances in Cryptology–EUROCRYPT 2005*, pages 19–35. Springer, 2005.
21. Gideon Yuval. How to swindle rabin. *Cryptologia*, 3(3):187–191, 1979.

A Generalization of Yuval's full-memory approach for near-collisions

Yuval's algorithm for finding a collision [21] computes and stores in a hash-table[5] distinct hash values until a collision between a new value and previous values is detected. It is based on the birthday paradox and has the time and memory complexity of $2^{n/2}$.

As mentioned in [8], its generalization for near-collisions requires checking the presence of all the values in $B_r^n(x)$ for each new hash value x. It reaches the lower bound in hash-computations but has time-complexity of $O\left(2^{n/2} \cdot \sqrt{V_r^n}\right)$ and requires $O\left(2^{n/2}/\sqrt{V_r^n}\right)$ memory. We can reduce the time-complexity by storing for every computed x all the pairs (y, m) such that $y \in B_R^n(x)$, for some $R \leq r/2$, and beforehand check for collisions with previous points by querying the points in $B_{r-R}^n(x)$. It is easy to see that the first r-collision is detected by the algorithm. The time complexity is reduced to $O\left(2^{n/2} \cdot V_{r-R}^n/\sqrt{V_r^n}\right)$, which is still above $O\left(2^{n/2}\right)$, but the memory demand is increased to $O\left(2^{n/2} \cdot V_R^n/\sqrt{V_r^n}\right)$.

[5] The hash-table is a data-structure in computer-science that implements insertions, searches and deletions of data in expected complexity of $O(1)$. It uses a generic hash-function for index-computations, but it should not be confused with cryptographic hash-functions.

B Calculation of ρ distribution of random-codes

First we calculate the distribution of the number of bits flipped by the decoding function on a random value x, that we denote by χ:

$$\chi(R) = \Pr[d_H(x, Dec(x)) = R] = \Pr[d_H(x, Dec(x)) \leq R] - \Pr[d_H(x, Dec(x)) \leq R-1] \approx \left(1 - e^{-pV_{R1}^n}\right) - \left(1 - e^{-pV_{R-1}^n}\right) = e^{-pV_{R-1}^n} - e^{-pV_R^n} \quad \text{(B.1)}$$

Then we estimate the distribution over the triples $(dist1, dist2, \#overlaps)$, when $dist1 = d_H(c, x_1)$, $dist2 = d_H(c, x_1)$ and $\#overlaps$ stands for the number of bits that are flipped by the Dec function for both x_1 and x_2.

$dist1$ and $dist2$ are independent and distributed according to χ. Therefore, the distribution of $(dist1, dist2, \#overlaps)$ is described by:

$$\Pr[(dist1, dist2, \#overlaps) = (b_1, b_2, s)] = \chi(b_1) \cdot \chi(b_2) \cdot \Pr[\#overlaps = s | dist1 = b_1 \wedge dist2 = b_2] \quad \text{(B.2)}$$

For any given pair, if the chances of every bit to be flipped on any message is the same, given $dist1$ and $dist2$, the number of overlaps between $dist1$ and $dist2$ bits distributes according to a hyper-geometric distribution. Combinatorially this is the number of "special items" that are picked when choosing $dist2$ distinct items from a pool of n items, of which $dist1$ are "special":

$$\Pr[\#overlaps = s | dist1 = b_1 \wedge dist2 = b_2] = \frac{\binom{b_1}{s} \cdot \binom{n-b_1}{b_2-s}}{\binom{n}{b_2}} \quad \text{(B.3)}$$

However, this estimate does not take into account that the secondary order we use works for our benefit. Generally speaking, the lower bits have larger probability to be flipped. For example, if not more than b bits are flipped in the majority of the vectors, the probability of finding a code-word within the first V_b^n trials is some $\alpha > 1/2$. If a certain vector is not decoded within these trials, there are V_b^{n-1} next trials to find a code-word that differs in $b+1$ bits, one of which is the first bit. Since b is much smaller than n, the ratio between V_b^n and V_b^{n-1} is close to 1. Therefore, in probability of almost α, if more than b bits are flipped, one of the flipped bits is going to be the first.

The actual distance between the hash values of pairs that correspond to a certain triple is $dist = dist1 + dist2 - 2 \cdot \#overlaps$.

$$\rho(R) := \sum_{b_1 + b_2 - 2s = R} \Pr[(dist1, dist2, \#overlaps) = (b_1, b_2, s)] \quad \text{(B.4)}$$

When we limit the radius to R, for every $b > R$ the value of $\chi(b)$ should be set to 0, and for every $b \leq R$ it should be divided by $\alpha = \sum_{i \leq R} \chi(i)$.

C Using a Lookup Table to Efficiently Decode Random Linear Codes

For a linear code defined in Section 4.3, the nearest-neighbor of a given x is $c = x + \Delta$ when Δ is the minimal vector such that $A(x \oplus \Delta) = 0$, or equivalently $Ax = A\Delta$. We will call Δ the *minimal pre-image* of $y = Ax$.

We construct a lookup-table that stores the minimal pre-image of y for any index $y \in \{0,1\}^\mu$. Then the decoding process for a given x takes a constant time: calculate the value $Ax \in \{0,1\}^\mu$, find its minimal pre-image from the table, and get the code-word $c = x \oplus \Delta$.

The table can be initialized efficiently by generating the smallest vectors in an increasing order starting from $\Delta = 0^n$, in a sort of a spiral (defined primarily by the Hamming distance and secondly by the secondary order). In each iteration we calculate $y = A\Delta$, and store the Δ at the y-th entry of the look-up table if it is empty. We stop when the entries of the table are filled. The size of the table is $K = 2^\mu$. By the coupon collector argument, the expected number of vectors being overviewed is $K \cdot \log K$, which is almost linear in K.

The distance between two vectors x_1 and x_2 that are encoded to the same code-word, by addition of Δ_1 and Δ_2 respectively, is:

$$d_H(x_1, x_2) = \|x_1 \oplus x_2\|_H = \|(c \oplus \Delta_1) \oplus (c \oplus \Delta_2)\|_H = \|\Delta_1 \oplus \Delta_2\|_H$$

Due to the linear independence of the rows of A, which we may assume, both Δ_1 and Δ_2 could be seen as randomly taken from the values in the table, independently of the code-word. Having the look-up table set, the distribution $\rho(r)$ is exactly the distribution of distances between pairs of entries in the table. It can be calculated in about $K^2/2$ steps or approximated experimentally by pair sampling.

C.1 Modification for a version with limited radius

In the version of the linear-random-code with radius R, we stop just before the weight of Δ reaches $R+1$ after V_R^n iterations. The remaining unset entries are set to zero, so a vector x for which Ax equals to the index of one of these entries will not be changed by the code (by xoring with zero).

D Script to estimate the time-complexity of the algorithm

The following script calculates the optimal parameters and estimates the complexity of our time-memory tradeoff algorithm.

It gets as input:

- n : The length of the domain
- max_dist : The maximal distance in the near-collision
- log_mem : $\log(M)$

It outputs the following parameters which describe the code defined in Equation (6.1):

- T : The complexity in logarithmic scale (see Equation (6.3))
- mu : μ. The rank of the matrix that defines the linear random code
- j : j. The number of times that the random-code is concatenated
- trunc : t. The number of truncated bits
- alarms : $l_{\mu,j,\tau}$ defined in Equation (6.2). The expected number of collisions detected in the code, until one is a [max_dist]-near-collision in h

Listing 1.1. A Matlab script that calculates the optimal parameters and estimates the complexity

```
function [T, mu, j, trunc, alarms] =
        near_collision_random_code(n, max_dist,log_mem)
    warning('OFF', 'MATLAB:nchoosek:LargeCoefficient');
    max_flips = ceil(min(max_dist/2+3, max_dist));
```

```
    mem = 2^log_mem;
    max_t = min(2*(max_dist+max(log_mem, 25)), n);
    t_vec = 0: max_t;
    code_l = n - t_vec;
    [t_temp, p_temp] = ndgrid(t_vec, 0:n);
    ff = (p_temp <= code_l(t_temp+1)) &
        (mod(p_temp, ceil(p_temp./log_mem)) == 0);
    t = t_temp(ff);
    p_pow = p_temp(ff);
    k = ceil(p_pow./log_mem);
    p_pow_k = p_pow./max(k, 1);
    n_k = floor((n-t)./max(k, 1));
    p_code = 2.^(-p_pow_k);
    n_k_min = min(n_k);
    code_lengths = n_k_min:n;
    code_length_inx  = n_k - n_k_min + 1;
    search_length = n - t - p_pow;
    bin_dist_vec = arrayfun(@(t_param)
        binomial_dist(t_param, max_dist),
        t_vec, 'UniformOutput', false);
    [i_param, j_param, r_param, mutual_overlap] =
         arrayfun(@(code_l_param) covering_dist_overlap(
         code_l_param , max_flips, max_dist),
         code_lengths, 'UniformOutput', false);
    balls_n_r_vec = arrayfun(@(i) balls_n_r(i, min(i, max_flips)),
         code_lengths, 'UniformOutput', false);
    inx = (1: numel(t))';
    chances = arrayfun(@(i) success_chances(bin_dist_vec(t(i)+1),
        k(i), p_code(i), i_param(code_length_inx(i)),
        j_param(code_length_inx(i)), r_param(code_length_inx(i)),
        mutual_overlap(code_length_inx(i)),
        balls_n_r_vec(code_length_inx(i)), max_dist),inx);
    col_needed = 1./chances;
    complexity= log2(sqrt(pi/2)+5*sqrt(col_needed./mem)) +
        search_length/2 + log2(col_needed)/2;
    [min_C,min_idx]=min(complexity);
    T = min_C;
    mu = p_pow_k(min_idx);
    j = k(min_idx);
    alarms = log2(col_needed(min_idx));
    trunc = t(min_idx);
    warning('ON', 'MATLAB:nchoosek:LargeCoefficient');
end

function [valid_chance] = success_chances(bin_dist_vec_cell, k,
        p_code, i_param_cell, j_param_cell, r_param_cell,
        mutual_overlap_cell, balls_n_r_vec_cell, max_dist)
    bin_dist_vec = cell2mat(bin_dist_vec_cell);
    i_param = cell2mat(i_param_cell);
```

```
    j_param = cell2mat(j_param_cell);
    r_param = cell2mat(r_param_cell);
    mutual_overlap = cell2mat(mutual_overlap_cell);
    balls_n_r_vec = cell2mat(balls_n_r_vec_cell);
    chances_cdf =  geocdf_my(balls_n_r_vec, p_code);
    flipped = zeros(numel(balls_n_r_vec), 1);
    flipped(1) = chances_cdf(1);
    flipped(2:end)= chances_cdf(2:end) - chances_cdf(1:(end-1));
    mutual_chances = mutual_overlap.*flipped(i_param+1)
        .*flipped(j_param+1);
    d_temp = accumarray(r_param+1, mutual_chances);
    d_code = d_temp(1: min(end, max_dist+1));
    d = bin_dist_vec;
    for i = 1: k
        d = conv(d_code, d(1: min(end, max_dist+1)));
    end
    valid_chance =  sum(d(1: min(end, max_dist+1)));
end

function [i_param, j_param, r_param, mutual_overlap] =
        covering_dist_overlap(n, max_flips, max_dist)
    max_f = min(n, max_flips);
    [i, j, overlaps] = ndgrid(0:max_f, 0:max_f, 0:max_f);
    ff = (i<=j) & (overlaps<=i) & (i+j-2*overlaps <= max_dist);
    i_param = i(ff);
    j_param = j(ff);
    overlaps_param = overlaps(ff);
    r_param = i_param + j_param -2*overlaps_param ;
    mutual_overlap = hygepdf(overlaps_param, n, i_param, j_param)
        .*(1+(i_param ~= j_param));
end

function [bin_dist] = binomial_dist(n, k)
   pd = makedist('Binomial','N',n);
   bin_dist = pdf(pd, 0: k);
end

function [ball_size] = balls_n_r(n, r)
   r_max =  min(n, r);
    circles = arrayfun(@(x) nchoosek(n,x), 0:r_max);
    ball_size = zeros(r+1, 1);
    ball_size(1) = 1;
    for ind = 2:r_max+1;
        ball_size(ind)= ball_size(ind-1)+circles(ind);
   end
    ball_size(r_max+2:end)= ball_size(r_max+1) ;
end
```

```
function y = geocdf_my(n,p)
    if p < 2^(-35)
        y = 1-exp(-n*p);
    else
        y = 1-(1-p).^n;
    end
end
```