Time-memory Trade-offs for Near-collisions

Gaëtan Leurent

UCL Crypto Group
Gaetan.Leurent@uclouvain.be

Abstract. In this work we consider generic algorithms to find near-collisions for a hash function. If we consider only hash computations, it is easy to compute a lower-bound for the complexity of near-collision algorithms, and to build a matching algorithm. However, this algorithm needs a lot of memory, and makes than $2^{n/2}$ memory accesses. Recently, several algorithms have been proposed without this memory requirement; they require more hash evaluations, but the attack is actually more practical. They can be divided in two main categories: some are based on truncation, and some are based on covering codes.
In this paper, we give a new insight to the generic complexity of a near-collision attack. First, we consider time-memory trade-offs for truncation-based algorithms. For a practical implementation, it seems reasonable to assume that *some* memory is available and we show that taking advantage of this memory can significantly reduce the complexity. Second, we show a new method combining truncation and covering codes. The new algorithm is always at least as good as the previous works, and often gives a significant improvement. We illustrate our results by giving a 10-near collision for MD5: our algorithm has a complexity of $2^{45.4}$ using 1TB of memory while the best previous algorithm required $2^{52.5}$ computations.

Keywords: Hash function, near-collision, generic attack, time-memory trade-off

1 Introduction

Hash functions are fundamental cryptographic primitives used in many constructions and protocols. A hash function takes a bitstring of arbitrary length as input, and outputs a digest, a small bitstring of fixed length n:

$$h : \{0,1\}^* \to \{0,1\}^n$$

When used in a cryptographic context, we expect a hash function to resist three major attacks:

Collision attack: Given h, find $x \neq x'$ s.t. $h(x) = h(x')$.
Second-preimage attack: Given h and x, find $x' \neq x$ s.t. $h(x) = h(x')$.
Preimage attack: Given h and $\overline{x}$, find x s.t. $h(x) = \overline{x}$.

Due to the birthday paradox, we have a generic collision attack with complexity $2^{n/2}$, while brute force preimage or second-preimage attacks have complexity 2^n: this defines the security requirements of a n-bit hash function.

More generally, we expect a hash function to behave *like a random function*. This requirement can not really be formalized but we expect that any property that can be shown on a given hash function should also be present on a random function.

In particular, we expect that it should be hard to find two messages resulting in a digest with a small difference. This property is called near-collision, and several attacks have been proposed in this setting recently [LT11,JF11,SWWD10,KL06,BC04].

It is relatively easy to give a lower bound on the complexity of near-collision attacks: one needs at least $2^{n/2}/\sqrt{\mathcal{B}_w(n)}$ hash function evaluation. However the only known way to reach this lower bound requires a lot of memory, and more than $2^{n/2}$ memory accesses. In order to bridge this gap, Lamberger *et al.* proposed a memory-less approach based on covering codes[LMRS12,LR10], with a complexity between $2^{n/2}$ and $2^{n/2}/\sqrt{\mathcal{B}_{w/2}(n)}$.

In this work, we revisit the problem of finding near-collision with an algorithm that can be efficiently implemented in practice. We start from the observation that the machines used to run this kind of large computation (clusters, GPUs, or dedicated hardware) usually have a decent amount of memory readily available, or it can be added at a reasonable cost. Therefore, we do not aim for a *memory-less* algorithm, we only aim for an algorithm with a practical amount of memory, and a practical number of memory accesses. Our result show that we can indeed reach a lower complexity than the memory-less algorithms based on covering codes.

We first review previous collision and near-collision algorithms in Section 2. We describe the main idea of our time-memory trade-off applied to truncation-based algorithms in Section 3, and we describe a more general algorithm in Section 4 that includes previous algorithms as special cases.

We use the following notations through this paper:

n	Hash function output size;
t	Truncated output size;
w	Maximum distance for near-collisions;
M	Memory size (number of chains stored);
$\mathcal{B}_w(n)$	Size of a Hamming ball of radius w.

2 Previous Works

Let us first discuss techniques to find full collisions (*i.e.* $w = 0$). This allows to explain the basic techniques which will be used later to find near-collision.

2.1 Finding full Collisions

The basic approach to find collisions or near-collisions in a generic manner is to evaluate the hash function a large number of times on random inputs, and to compute the Hamming distance for each pair of outputs. After i evaluations of the hash function, one can test $i(i-1)/2$ pairs, and this birthday effect allows to find collisions with only $O(2^{n/2})$ evaluations of the hash function. More precisely, the expected number of computation required is $\sqrt{\pi/2} \cdot 2^{n/2}$[vOW99, Appendix A]. When looking for full collisions, instead of comparing each new output to all the previous one, which require $\Omega(2^{2n})$ comparisons, one can create a list of all the outputs, and sort the list in time $O(n2^n)$, or use a hash table to reduce the number of comparisons to $O(2^n)$.

Memory-less algorithms. Even if we avoid the complexity of $O(2^{2n})$ comparisons, the memory complexity of this simple approach makes it impractical. Several works have shown that collisions can be found with little or no memory, with a small increase in the time complexity. The main idea was introduced by Pollard as the Rho algorithm for factorization [Pol75] and discrete logarithms [Pol78], and was later generalized to collision search. The hash function is first restricted from $\{0,1\}^* \to \{0,1\}^n$ to $\{0,1\}^n \to \{0,1\}^n$, so that it can be iterated. After some number of steps, a chain of iterations reaches a cycle, and the graph will have the shape of the greek letter ρ. On average, the cycle has length $O(2^{n/2})$ and is reached after $O(2^{n/2})$ steps. The point where the tail of the ρ meets with the cycle gives a collision in the hash function. It can be detected in time $O(2^{n/2})$ with little or no memory, using various cycle detection methods, such as Floyd's algorithm [Knu81], Brent's algorithm [Bre80], using distinguished points [QD89], or several other techniques [SSY82,Niv04] (these techniques mostly differ by the memory requirements, and the constant in the $O(\cdot)$: between 1 and 3).

In this work we focus on the distinguished point approach because it can be efficiently parallelized, and our focus is on problems with a relatively large complexity. The complexity of finding collisions using distinguished point is analyzed in detail by van Oorschot in [vOW99]. The main step of the algorithm is to compute chains of iterations, starting from a random point, and stopping when a *distinguished* point is reached, with an easily recognized feature, such as a number of leading zeroes. The algorithm uses a table to store M such chains (*i.e.* starting points and ending points) and when the same ending point is seen twice, this most likely correspond to a collision. To locate the collision, one has to run the computation again from the starting point.

The analysis of van Oorschot considers two different situations, depending on i, the number of collision one is looking for. An important parameter in the analysis is the proportion of distinguished point θ.

Finding a small number of collisions *i.e.* $i \ll M$.

If we have enough memory to store all the chains, we can expect to find i collisions after a workload of $\Theta(\sqrt{2^n i})$, since this covers $\Theta(2^n i)$ pairs of points. More precisely, the complexity given by van Oorschot[1] is $C_{small} = \sqrt{\pi/2} \cdot \sqrt{2^n i} + 2.5i/\theta$.

We choose the distinguishing property so that the memory will just be filled at the end, but we try to avoid overwriting chains, so we use $\theta = M/C_{small}$. This results in $C_{small} = \sqrt{\pi/2} \cdot \sqrt{2^n i}/(1 - 2.5i/M)$. If $i \ll M$, this becomes:

$$C_{small} = \sqrt{\pi/2} \cdot \sqrt{2^n i}.$$

There is a speedup factor of $\sqrt{i}$ compared to finding i collisions independently.

Finding a large number of collisions *i.e.* $i \gg M$.

In this case, the memory will have to be overwritten. The analysis of [vOW99] shows that when the memory is full, the complexity per collision is roughly $2^n\theta/M + 2/\theta$. This reaches a minimum of $\sqrt{8 \cdot 2^n/M}$ for $\theta = \sqrt{2M/2^n}$. More precisely, van

[1] In [vOW99], the complexity is given as $\sqrt{\pi/2} \cdot \sqrt{2^n i} + 2.5/\theta$, but this only holds if i is smaller than the number of processors used in the attack.

Oorschot performed experiments to determine the actual constants, and the resulting complexity is:

$$C_{large} = 5\sqrt{2^n/M} \cdot i,$$

when $\theta = 2.25\sqrt{M/2^n}$. There is a speedup factor of $\sqrt{M}/4$ compared to finding i collisions independently.

Global bound. More generally, we can express an upper bound on the complexity that works in both situations by summing the two expressions:

$$C \leq \left(\sqrt{\frac{\pi}{2}} + 5\sqrt{\frac{i}{M}}\right)\sqrt{2^n i}. \tag{1}$$

When $i \ll M$ or $i \gg M$, one term is negligible, and this expression is equivalent to C_{small} or C_{large}, respectively. Moreover, we verified experimentally that this is also an upper bound when $i \approx M$, and the bound is relatively tight. In all cases, there is a linear speedup when using several machines in parallel (see [vOW99] for full details).

2.2 Near-collisions

A w-near-collision is a pair of messages x, x' such that $\|h(x) \oplus h(x')\| \leq w$, where $\|\cdot\|$ is the Hamming weight. Let us first introduce some results regarding the Hamming distance.

Definition 1. *We denote the size of a Hamming ball of radius w by* $\mathcal{B}_w(n) = \#\{x \in \{0,1\}^n : \|x\| \leq w\}$.

Property 1. We have $\mathcal{B}_w(n) = \sum_{i=0}^{w} \binom{n}{i}$.

Property 2. The probability that a random pair x, x' results in a w-near-collisions is $\mathcal{B}_w(n)/2^n$.

Property 3. We have the following relation: $\mathcal{B}_w(n) = \mathcal{B}_w(n-1) + \mathcal{B}_{w-1}(n-1)$.

Lemma 1. *We have the following inequality:*

$$\mathcal{B}_{w-1}(x) \leq \binom{x}{w}\frac{w}{x-2w+1}$$

Proof. (following [Lug])

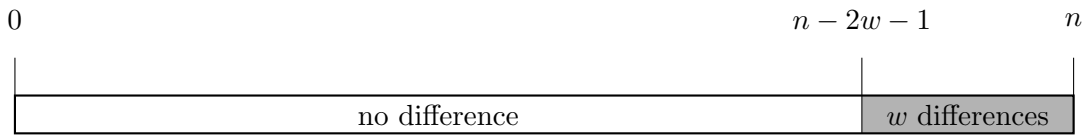
$$\begin{aligned}
\frac{\mathcal{B}_{w-1}(x)}{\binom{x}{w}} &= \frac{\binom{x}{w-1} + \binom{x}{w-2} + \binom{x}{w-3} + \cdots}{\binom{x}{w}} \\
&= \frac{w}{x-w+1} + \frac{w(w-1)}{(x-w+1)(x-w+2)} + \cdots \\
&\leq \frac{w}{x-w+1} + \left(\frac{w}{x-w+1}\right)^2 + \cdots \\
&\leq \frac{\frac{w}{x-w+1}}{1-\frac{w}{x-w+1}} = \frac{w}{x-2w+1} \qquad \text{using the sum of a geometric series} \quad \square
\end{aligned}$$

Memory-full algorithm. The obvious method to find near-collisions is to evaluate the hash function a large number of times on random inputs, and to compute the Hamming distance between each pair of outputs. After i evaluations of the hash function, one can test $i(i-1)/2$ pairs, and a pair gives a w-near-collision with probability $\mathcal{B}_w(n)/2^n$. The expected number of computation before finding a near-collision is $i = \sqrt{\pi/2 \cdot 2^n/\mathcal{B}_w(n)}$. This also gives a lower bound on the number of hash evaluations needed for any near-collision algorithm: we need at least $\sqrt{\pi/2 \cdot 2^n/\mathcal{B}_w(n)}$ evaluations in order to have a w-near-collision with a non-negligible probability.

However, this simple approach requires $i \cdot \mathcal{B}_w(n) = \Omega(\sqrt{2^n \cdot \mathcal{B}_w(n)})$ memory access to a table of size $i = \Omega(\sqrt{2^n/\mathcal{B}_w(n)})$. As opposed to a collision attack, we can not reduce this complexity using a sorting algorithm, a hash table, or chain of iterations; for any practical implementation, this will in fact be the bottleneck. This leads to the study of techniques to find near-collision without this huge memory complexity. Two categories of algorithms have proposed recently to solve this problem by reducing it to finding collision in a related function (which can done in a memory-less way).

Using collisions in a truncated hash function. A simple approach is to look for collisions in a truncated version of the hash function. In the simplest approach, the hash function is truncated to $t = n - w$ bits, and any collision in the truncated version will give a w-near-collision for the full hash function. More interestingly, if the hash function is truncated to $t = n - 2w - 1$ bits, a t-bit collision will give a w-near-collision of the full hash function with probability $1/2$ [LR10]. This gives a near-collision algorithm with expected complexity $\sqrt{\pi/2} \cdot 2^{(n-2w)/2}$ using a small amount of memory for the distinguished points.

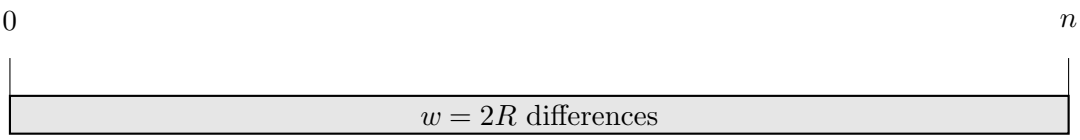
This can be represented graphically as:



More generally, one can truncate τ bits, find collisions in a $n - \tau$-bit function, and check the Hamming weight of the τ truncated bits. This will give a w-near-collision with probability $\mathcal{B}_w(\tau)/2^\tau$. The optimal value of τ can be found by evaluating the complexity for all choices of τ. This problem is discussed more formally in [LT12].

Using covering codes. A more efficient approach is to use covering codes, as proposed by Lamberger *et al.* [LR10,LMRS12]. The idea is to use a covering code with radius $w/2$, *i.e.* a set a codewords $\mathcal{C}$ such that for any point $x \in \{0,1\}^n$, there exists a codeword $c(x) \in \mathcal{C}$ with $\|x \oplus c(x)\| \leq w/2$. If the decoding function c is efficient, we can look for collisions in $c \circ h$. If $c(h(x)) = c(h(x'))$, then $h(x)$ and $h(x')$ are decoded to the same keyword c; we have $\|h(x') \oplus h(x)\| \leq \|h(x') \oplus c\| + \|h(x) \oplus c\| \leq w$, which gives a w-near-collision. With a code of dimension k, the attack has a complexity of $\sqrt{\pi/2} \cdot 2^{k/2}$.

This can represent graphically as:



Finding the optimal k and building a corresponding code is a hard problem. The sphere covering bound shows that $2^k \geq 2^n / \mathcal{B}_{w/2}(n)$, but there is a gap between the best known codes and this lower bound. This problem is discussed by Lamberger *et al.* in the context of near-collision attacks [LR10,LMRS12] using a concatenation of Hamming codes. With a given length n, and a covering radius of $R = w/2$, the optimal code following their construction has a dimension:

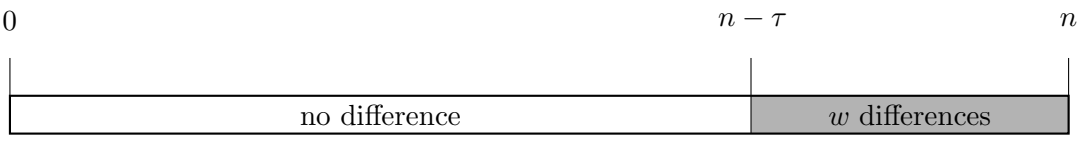
$$k = n - R \cdot \ell - r \tag{2}$$
$$\text{where} \quad \ell := \left\lfloor \log_2 \left(\frac{n}{R} + 1 \right) \right\rfloor \quad \text{and} \quad r := \left\lfloor \frac{n - R(2^\ell - 1)}{2^\ell} \right\rfloor .$$

This result is listed in Table 1 for some relevant values of the parameters, together with the lower bound implied by the sphere covering bound.

3 Time-memory Trade-off with Truncation

Our first algorithm is a simple generalization of the truncation based method described in Section 2.2. We observe that if we truncate τ bits with $\tau > 2w-1$, the probability that a collision in the truncated function is a near-collision of the full hash function decreases rapidly, and we need to find many collisions. In a truly memory-less approach, finding i such collisions require $\sqrt{\pi/2} \cdot \sqrt{2^{n-\tau}} \cdot i$ computations, and there is little to gain by truncating more than $2w-1$ bits. However, with some memory, this can be significantly reduced — by a factor $\sqrt{i}$ if $M \gg i$, or $\sqrt{M}/4$ if $M \ll i$, as detailed in Section 2.1.

In the following section, we explore this idea, and study the optimal value of τ and the complexity of the resulting attack, depending on how much memory is available. In a practical implementation of a near-collision attack, it seems reasonable to assume that *some* memory is available, and we show that this leads to significantly better attacks.



3.1 Complexity

Our algorithm is quite simple: we truncate τ bits of the hash function, and we look for collisions for the remaining $n - \tau$ bits. For each $n - \tau$-bit collision, we compute the Hamming distance in the truncated τ bits. We expect to find a w-near-collision after testing $i = 2^\tau / \mathcal{B}_w(\tau)$ collisions.

We observe that $i(\tau)$ is monotically increasing since $\mathcal{B}_w(\tau) = \mathcal{B}_w(\tau-1)+\mathcal{B}_{w-1}(\tau-1) < 2\mathcal{B}_w(\tau-1)$. With a small τ, we only need a small number of collision, but the collisions are harder to find because the number of non-truncated bit, $n-\tau$ is large. In order to find the best trade-off, we need an accurate evaluation of the complexity of the algorithm, depending on the value of τ and M. We use the analysis of van Oorschot [vOW99], as recalled in Section 2.1.

3.2 Finding Optimal Parameters

In order to find an algebraic characterization of the optimal τ, we follow the analysis of Section 2.1, and we consider two cases for the complexity, depending on the relationship between i and M.

Small number of truncated bits, small number of collisions *i.e.* $2^\tau/\mathcal{B}_w(\tau) \ll M$
The complexity is

$$C_{small} = \sqrt{\pi/2} \cdot \sqrt{2^{n-\tau} \cdot 2^\tau/\mathcal{B}_w(\tau)} = \sqrt{\pi/2} \cdot 2^{n/2}/\sqrt{\mathcal{B}_w(\tau)}.$$

This decreases when τ grows.

Large number of truncated bits, large number of collisions *i.e.* $2^\tau/\mathcal{B}_w(\tau) \gg M$
The complexity is

$$C_{large} = \frac{5\sqrt{2^{n-\tau}/M} \cdot 2^\tau}{\mathcal{B}_w(\tau)} = \frac{5 \cdot 2^{n/2+\tau/2}}{\mathcal{B}_w(\tau)\sqrt{M}}.$$

Let us study the variations of C_{large}:

$$\begin{aligned}
C_{large}(\tau-1) \le C_{large}(\tau) \quad & \frac{C_{large}(\tau-1)}{C_{large}(\tau)} \le 1 \\
& \frac{\mathcal{B}_w(\tau)}{\mathcal{B}_w(\tau-1)} \le \sqrt{2} \\
& \frac{\mathcal{B}_w(\tau-1)+\mathcal{B}_{w-1}(\tau-1)}{\mathcal{B}_w(\tau-1)} \le \sqrt{2} \\
& \frac{\mathcal{B}_{w-1}(\tau-1)}{\mathcal{B}_w(\tau-1)} \le \sqrt{2}-1 \\
& \frac{\mathcal{B}_w(\tau-1)}{\mathcal{B}_{w-1}(\tau-1)} \ge \sqrt{2}+1 \\
& \frac{\binom{\tau-1}{w}+\mathcal{B}_{w-1}(\tau-1)}{\mathcal{B}_{w-1}(\tau-1)} \ge \sqrt{2}+1 \\
& \frac{\binom{\tau-1}{w}}{\mathcal{B}_{w-1}(\tau-1)} \ge \sqrt{2}
\end{aligned}$$

Using Lemma 1, we have $\binom{\tau-1}{w}/\mathcal{B}_{w-1}(\tau-1) \ge \frac{\tau-2w}{w}$. When $\tau \ge (\sqrt{2}+2)w$, this gives $\binom{\tau-1}{w}/\mathcal{B}_{w-1}(\tau-1) \ge \sqrt{2}$, and $C(\tau)$ is increasing. Note that this formula only

makes sense when $M \ll 2^\tau/\mathcal{B}_w(\tau)$, *i.e.* for large values of τ, and the assumption that $\tau \geq (\sqrt{2}+2)w$ will be true in this domain for useful values of the parameters. In particular it is true as soon as $M > 2^{24}$ and $w < 48$.

For $M \approx 2^\tau/\mathcal{B}_w(\tau)$, the two expression are equal up to a small constant. Like in Section 2.1, the complexity is in fact continuous.

Optimal τ. When τ is small *i.e.* $2^\tau/\mathcal{B}_w(\tau) \ll M$, the complexity decreases with τ, but when τ is large, *i.e.* $2^\tau/\mathcal{B}_w(\tau) \gg M$, it increases with τ. This proves that the optimal choice of τ satisfies

$$2^\tau/\mathcal{B}_w(\tau) \approx M.$$

For this value of τ, the complexity is given by

$$C \approx 2^{n/2}/\sqrt{\mathcal{B}_w(\tau)}.$$

This is larger than the optimal complexity reached by the memory-full algorithm of $2^{n/2}/\sqrt{\mathcal{B}_w(n)}$, but for most parameters, it is better than the bound of $2^{n/2}/\sqrt{\mathcal{B}_{w/2}(n)}$ which limits covering-code based algorithms.

Optimal τ in practice. For given values of n and w, we can find a better estimation of the optimal τ. We use the upper bound of (1), which gives the following upper bound on the complexity:

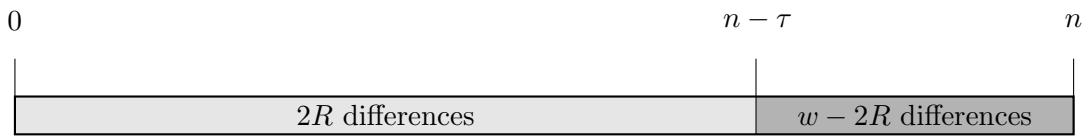
$$C \leq C_{small} + C_{\lg} = \left(\sqrt{\frac{\pi}{2}} + 5\sqrt{\frac{2^\tau/\mathcal{B}_w(\tau)}{M}}\right) \cdot \sqrt{\frac{2^n}{\mathcal{B}_w(\tau)}}.$$

To find a good trade-off, we evaluate this bound for all values of τ, and we use the τ that gives the lowest bound. Our experiments show that the upper bound is quite tight, and the τ found in this way is optimal or almost optimal.

4 Combining Truncation and Covering Codes

We can build a better algorithm by combining the truncation approach with the covering-code technique. When we truncate the hash function to $n-\tau$ bits, instead of looking for collisions in the truncated function, we can look for near-collisions using a covering code. More precisely, we use a covering code of radius R, to find $2R$-near-collisions in the truncated hash function. Then we check if one of the near-collisions have less than $w-2R$ active bits in the truncated part. This approach covers both the truncation based techniques (when $R=0$), and the previous covering-code based techniques (when $\tau=0$ and $R=w/2$). We is described by Algorithm 1.

This can be represent by the following diagram:



Algorithm 1 Find near-collisions

Input: n, w
Parameter: τ, R
Build the decoding function c of a covering code of length $n-\tau$, dimension k, radius R
repeat
Find a collision x, x' for $c \circ \mathrm{Trunc}_{n-\tau} \circ h$ ▷ $\|\mathsf{Trunc}_{n-\tau}(h(x)) \oplus \mathsf{Trunc}_{n-\tau}(h(x'))\| \leq 2R$
until $\|h(x) \oplus h(x')\| \leq w$

If we use a covering code of dimension k, length $n-\tau$, and radius R, we will have near-collisions with a distance of $2R$. Using the same ideas as in the previous section, we use a time-memory trade-off to find a large number of near-collisions; we can find i near-collisions for a cost of roughly $\sqrt{2^k i}$ if $i \ll M$ or $\sqrt{2^k/M} \cdot i$ if $i \gg M$. On average, we need $i = 2^\tau / \boldsymbol{\mathcal{B}}_{w-2R}(\tau)$ $2R$-near-collisions. Like in the previous section, we use the bound of (1) to evaluate the complexity of the attack:

$$C \leq \left(\sqrt{\frac{\pi}{2}} + 5\sqrt{\frac{2^\tau / \boldsymbol{\mathcal{B}}_{w-2R}(\tau)}{M}} \right) \sqrt{\frac{2^k \cdot 2^\tau}{\boldsymbol{\mathcal{B}}_{w-2R}(\tau)}}$$

It seems quite hard to give of close formula of the optimal choice of R and τ for a given n, w and M. In particular, we note that k is a function of R and τ. However, it is easy to find the optimal parameters by trying all the possibilities for R and τ, and evaluating the resulting complexity using (2) to compute the optimal k. We give the optimal parameters for several cases in Table 1.

Like in the previous section, we observe that the best parameters usually satisfy $2^\tau / \boldsymbol{\mathcal{B}}_{w_2}(\tau) \approx M$. Moreover, we note that for many parameters, the optimal choice gives $R = 0$ and we just have a truncation-based attack without any covering code. The covering codes allow to improve the complexity only for large values of n or small values of M.

4.1 Improved Analysis

In this analysis, we only consider near-collisions with less than $w - 2R$ active bits in the truncated part. However, the algorithm can find w-near-collisions with more active bits in the truncated part if the distance in the remaining part is strictly smaller than $2R$. In order to compute the probability that a collision in the covering code gives a w-near-collision for the full hash function, we use the distribution of the distance between two messages decoded to the same codeword, as given in [LMRS12, Section 3.6].

For a Hamming code $\boldsymbol{\mathcal{H}}_r$ of length $n = 2^r - 1$, the distribution is:

$$d(y, y') = \begin{cases} 0 & \text{with prob. } \frac{n+1}{(n+1)^2} \\ 1 & \text{with prob. } \frac{2n}{(n+1)^2} \\ 2 & \text{with prob. } \frac{n(n-1)}{(n+1)^2}. \end{cases}$$

The covering codes used in [LR10,LMRS12] are built as the direct sum of several Hamming codes, and we can compute the distribution of their distance as a convolution of the distribution for a Hamming code. For the truncation, the distribution is

$$d(y, y') = \begin{cases} 0 & \text{with prob. } 1/2^\tau \\ i & \text{with prob. } \binom{\tau}{i}/2^\tau \end{cases}$$

This allows to compute accurately the probability that a collision in the covering code is a near-collision for the hash function. The complexity will still be given by

$$C \leq \left(\sqrt{\frac{\pi}{2}} + 5\sqrt{\frac{i}{M}} \right) \sqrt{2^k \cdot i}$$

but we compute i from the distribution instead of using $i = 2^\tau / \mathcal{B}_{w-2R}(\tau)$. In addition, we can now consider a radius R larger than $w/2$, as suggested in [LMRS12]. In this case, most collisions in the covering code will have a distance larger than w, but the time-memory trade-off reduces the cost of finding many collisions.

4.2 Application.

Our final algorithm is quite broad, and the behavior will be very different depending on the parameters R and τ. We don't see how to analyze the optimal the choice of parameters, but for a given value of n, w and M, we can just evaluate the complexity for all values of the parameters R and τ and select the best ones.

We give the complexity of our algorithm for some values of n, w and M in Table 1, and we provide the code used to find the optimal parameters in Listing 1.1. We compare several possible trade-offs with previous approaches: the simple memory-full algorithm, the covering code algorithm of [LR10,LMRS12] and the corresponding lower bound, and the simple truncation of $2w - 1$ bits. With reasonable amounts of memory, our approach can lead to a significant improvement in the complexity.

We note that the number of memory access if relatively limited (in the order of M). The communication cost should not be a bottleneck for a practical implementation. Additionally, the memory does not need to be in a single machine, it can be distributed over the computing nodes. Like previous algorithms, our algorithm scales linearly when using more than one processor. Moreover, it should be noted that the memory-less algorithms do need some memory for an efficient parallelization.

We implemented this algorithm to verify that it behaves as expected, and we give a 10-near-collision for MD5 in Table 2.

Conclusion

In this work we present a new generic algorithm to find near-collision, that generalizes both the previous truncation-based algorithms, and the previous covering-code based algorithms. As opposed to previous work, we don't aim for a memory-less algorithm, but

Table 1. Comparision of various approaches. We omit a factor $\sqrt{\pi/2}$ so that birthday search is listed as $2^{n/2}$.

	M-Full[1]	Time-memory trade-off (τ, R)			Covering codes		$\tau = 2w - 1$
128 bits		2^{16} (1MB)	2^{26} (1GB)	2^{36} (1TB)	bound[2]	[LR10,LMRS12]	
$w = 2$	57.5	60.5 (1,1)	60.0 (25,0)	59.5 (35,0)	60.5	60.5	62.0
$w = 4$	52.3	57.6 (17,1)	56.5 (27,1)	55.6 (44,0)	57.5	58.0	60.0
$w = 6$	47.8	54.5 (19,2)	53.1 (35,1)	52.0 (46,1)	54.8	56.0	58.0
$w = 8$	43.8	51.6 (26,2)	49.8 (43,1)	48.5 (54,1)	52.3	54.0	56.0
$w = 10$	40.1	48.7 (33,2)	46.7 (50,1)	45.2 (62,1)	50.0	52.5	54.0
160 bits		2^{16} (1MB)	2^{26} (1GB)	2^{36} (1TB)			
$w = 2$	73.2	76.5 (5,1)	76.0 (17,1)	75.5 (35,0)	76.3	76.5	78.0
$w = 4$	67.7	73.2 (16,1)	72.2 (26,1)	71.6 (33,1)	73.2	74.0	76.0
$w = 6$	62.8	70.2 (24,1)	68.8 (33,1)	68.0 (46,1)	70.3	71.5	74.0
$w = 8$	58.5	67.3 (31,1)	65.7 (34,2)	64.5 (54,1)	67.7	69.5	72.0
$w = 10$	54.4	64.4 (33,2)	62.7 (45,2)	61.2 (62,1)	65.2	67.5	70.0
512 bits		2^{26} (1GB)	2^{36} (1TB)	2^{46} (1PB)			
$w = 2$	247.5	251.5 (2,2)	251.4 (26,1)	251.1 (36,1)	251.5	251.5	254.0
$w = 4$	240.3	247.7 (3,4)	247.2 (29,2)	246.7 (39,2)	247.5	248.0	252.0
$w = 6$	233.8	244.0 (27,2)	243.2 (38,2)	242.6 (49,2)	243.8	245.0	250.0
$w = 8$	227.7	240.5 (23,4)	239.6 (46,2)	238.7 (57,2)	240.3	242.0	248.0
$w = 10$	221.9	237.1 (30,4)	236.0 (42,4)	235.0 (65,2)	237.0	239.5	246.0

[1] Number of hash function evaluation needed. The actual complexity is dominated by memory accesses (more than $2^{n/2}$ accesses to a huge table).
[2] Lower bound for memory-less covering code approaches (sphere covering bound).

we study time-memory trade-offs. The algorithm has been implemented in practice, and we give actual complexity figures including the constants hidden in the analysis.

We show that with a practical amount of memory, this allows to select better parameters than previous works; in most cases we achieve a complexity lower than the sphere covering bound which limits the previous covering-code based algorithms. The main advantage comes from the parallel collision search algorithm of van Oorschot, which can find i collisions in time significantly less than $\sqrt{2^n} \cdot i$ when using some memory.

Table 2. 10-near collision for MD5. This was found after a 20 hour computation using 1TB of memory, and 152 cores.

x	x'	$x \oplus x'$
`b6 24 ac c6 40 94 08 84`	`0d 87 0f a4 00 4b 6c bf`	`bb a3 a3 62 40 df 64 3b`
`00 00 00 00 00 00 00 00`	`00 00 00 00 00 00 00 00`	`00 00 00 00 00 00 00 00`
MD5(x)	MD5(x')	MD5(x) $\oplus$ MD5(x')
`ac 6b 49 aa fe 42 4f f8`	`68 79 db a8 fe 52 4f f8`	`c4 12 92 02 00 10 00 00`
`8a c9 5d f6 ef 4f 7b 3d`	`8a c9 5d f6 ef 4f 7b 3d`	`00 00 00 00 00 00 00 00`

References

BC04. Eli Biham and Rafi Chen. Near-Collisions of SHA-0. In Matthew K. Franklin, editor, *CRYPTO*, volume 3152 of *Lecture Notes in Computer Science*, pages 290–305. Springer, 2004.

Bre80. R.P. Brent. An improved monte carlo factorization algorithm. *BIT Numerical Mathematics*, 20(2):176–184, 1980.

JF11. Jérémy Jean and Pierre-Alain Fouque. Practical Near-Collisions and Collisions on Round-Reduced ECHO-256 Compression Function. In Antoine Joux, editor, *FSE*, volume 6733 of *Lecture Notes in Computer Science*, pages 107–127. Springer, 2011.

KL06. John Kelsey and Stefan Lucks. Collisions and Near-Collisions for Reduced-Round Tiger. In Matthew J. B. Robshaw, editor, *FSE*, volume 4047 of *Lecture Notes in Computer Science*, pages 111–125. Springer, 2006.

Knu81. D.E. Knuth. Seminumerical algorithms, volume 2 of the art of computer programming, 1981.

LMRS12. Mario Lamberger, Florian Mendel, Vincent Rijmen, and Koen Simoens. Memoryless near-collisions via coding theory. *Des. Codes Cryptography*, 62(1):1–18, 2012.

LR10. Mario Lamberger and Vincent Rijmen. Optimal Covering Codes for Finding Near-Collisions. In Alex Biryukov, Guang Gong, and Douglas R. Stinson, editors, *Selected Areas in Cryptography*, volume 6544 of *Lecture Notes in Computer Science*, pages 187–197. Springer, 2010.

LT11. Gaëtan Leurent and Søren S. Thomsen. Practical Near-Collisions on the Compression Function of BMW. In Antoine Joux, editor, *FSE*, volume 6733 of *Lecture Notes in Computer Science*, pages 238–251. Springer, 2011.

LT12. Mario Lamberger and Elmar Teufl. Memoryless near-collisions, revisited. *CoRR*, abs/1209.4255, 2012.

Lug. Michael Lugo. Sum of "the first k" binomial coefficients for fixed n. MathOverflow. `http://mathoverflow.net/questions/17236` (version: 2010-03-05).

Niv04. G. Nivasch. Cycle detection using a stack. *Information Processing Letters*, 90(3):135–140, 2004.

Pol75. J.M. Pollard. A monte carlo method for factorization. *BIT Numerical Mathematics*, 15(3):331–334, 1975.

Pol78. J.M. Pollard. Monte carlo methods for index computation (mod p). *Mathematics of computation*, 32(143):918–924, 1978.

QD89. Jean-Jacques Quisquater and Jean-Paul Delescaille. How Easy is Collision Search. New Results and Applications to DES. In Gilles Brassard, editor, *CRYPTO*, volume 435 of *Lecture Notes in Computer Science*, pages 408–413. Springer, 1989.

SSY82. R. Sedgewick, T.G. Szymanski, and A.C. Yao. The complexity of finding cycles in periodic functions. *SIAM Journal on Computing*, 11:376, 1982.

SWWD10. Bozhan Su, Wenling Wu, Shuang Wu, and Le Dong. Near-Collisions on the Reduced-Round Compression Functions of Skein and BLAKE. In Swee-Huay Heng, Rebecca N. Wright, and Bok-Min Goi, editors, *CANS*, volume 6467 of *Lecture Notes in Computer Science*, pages 124–139. Springer, 2010.

vOW99. Paul C. van Oorschot and Michael J. Wiener. Parallel Collision Search with Cryptanalytic Applications. *J. Cryptology*, 12(1):1–28, 1999.

Listing 1.1. Sage code to compute the complexity of a generic near-collision attack

```
@CachedFunction
def covering_k(n,R):
    if R == 0:
        return n
    l = floor(log(n/R+1)/log(2))
    r = floor((n-R*(2^l-1))/2^l)
    return n - R*l - r

@CachedFunction
def covering_dist(n,R):
    if R == 0:
        return [1]
    l = floor(log(n/R+1)/log(2))
    r = floor((n-R*(2^l-1))/2^l)
    d = [1]
    m = 2^l
    for i in [1..R-r]:
        d = convolution(d, [m/m^2,(2*m-2)/m^2,((m-1)*(m-2))/m^2])
    m = 2^(l+1)
    for i in [1..r]:
        d = convolution(d, [m/m^2,(2*m-2)/m^2,((m-1)*(m-2))/m^2])
    return d

@CachedFunction
def binomial_dist(n):
    return [ binomial(n,i)/2^n for i in [0..n] ]

@CachedFunction
def prob_dist(n,t,R):
    return convolution(binomial_dist(t),covering_dist(n-t,R))

def near_complexity(n,mem,maxw):
    best = n
    for t in [0..n]:
        for R in [0..2*maxw]:
            K = prob_dist(n,t,R)
            p = sum(K[0:maxw+1])
            C = (sqrt(pi/2)+5*sqrt(1/p/2^mem)) * sqrt(2^covering_k(n-t,R)/p)
            C = float(log(C)/log(2))
            if C < best:
                best = C
                bestr = R
                bestt = t
    print "Compexity: %.1f (\\tau = %i, R = %i)" % (best, bestt, bestr)
    return best
```