Improved generic algorithms for 3-collisions

Antoine Joux[1] and Stefan Lucks[2]

[1] DGA and Université de Versailles Saint-Quentin-en-Yvelines
UVSQ PRISM, 45 avenue des États-Unis, F-78035, Versailles CEDEX, France
antoine.joux@m4x.org
[2] Bauhaus-Universität Weimar, 99423 Weimar, Germany
Stefan.Lucks@uni-weimar.de

Abstract. An r-collision for a function is a set of r distinct inputs with identical outputs. Actually finding r-collisions for a random map over a finite set of cardinality N requires at least about $N^{(r-1)/r}$ units of time on a sequential machine. For r=2, memoryless and well-parallelizable algorithms are known. The current paper describes memory-efficient and parallelizable algorithms for $r \geq 3$. The main results are: (1) A sequential algorithm for 3-collisions, roughly using memory N^{α} and time $N^{1-\alpha}$ for $\alpha \leq 1/3$. In particular, given $N^{1/3}$ units of storage, one can find 3-collisions in time $N^{2/3}$. (2) A parallelization of this algorithm using $N^{1/3}$ processors running in time $N^{1/3}$, where each single processor only needs a constant amount of memory. (3) A generalisation of this second approach to r-collisions for $r \geq 3$: given N^s parallel processors, with $s \leq (r-2)/r$, one can generate r-collisions roughly in time $N^{((r-1)/r)-s}$, using memory $N^{((r-2)/r)-s}$ on every processor.

Keywords: multicollision, random map, memory-efficient, parallel implementation, cryptanalysis

1 Introduction

The problem of finding collisions and multicollisions in random mappings is of significant interest for cryptography, and mainly for cryptanalysis. It is well known that finding an r-collision for a random map over a finite set of cardinality N requires [3] more than $N^{(r-1)/r}$ map evaluations.

Multicollisions for hash functions. If the map under consideration is a hash function, or has been derived from a hash function, many researchers consider faster multicollisions as a certificational hash function weakness. Accordingly, it was worrying for the research community to learn that multicollisions could be found much faster for a widely used class of hash functions: iterated hash functions [9]. For n-bit hash functions from this class, one can generate 2^k-collisions in time

[3] An r-collision is a set of r different inputs $x_1, \ldots, x_r$ which all generate the same output $\mathrm{map}(x_1) = \cdots = \mathrm{map}(x_r)$. For an r-collision, one needs to evaluate the map $(r!)^{1/r} \cdot N^{(r-1)/r}$ times [22]. For small r, we can approximate this by $O(N^{(r-1)/r})$.

$k \cdot 2^{n/2}$, rather than the expected time $2^{n(2^k-1)/2^k}$. The basic observation is straightforward: given a sequence of k consecutive 2-collisions, it is possible with iterated hash functions to consider the 2^k different messages obtained by taking all possible choices of message block for each collision and obtain 2^k times the same output. These iterated multicollisions have been generalized later, to more complex types of iterated hash functions, for example, see [6, 13, 7]. It was also remarked that these iterated multicollisions were a rediscovery and generalization of an older attack of Coppersmith [2].

In particular, this type of multicollisions allowed a surprising attack on hash cascades, i.e., hash functions H, which are the concatenation of two hash functions G_1 and G_2, i.e., $H(X) := (G_1(X), G_2(X))$. If, say, G_1 is an iterated hash function and vulnerable to the multicollision attack, and G_2 is any n-bit hash function, the adversary just needs to generate a $2^{n/2}$-multicollision for G_1. Thanks to the birthday paradox, among the $2^{n/2}$ messages colliding for G_1, one expects to find a pair of messages colliding for G_2 with constant probability. As a consequence, a collision for the $2n$-bit hash function H can be obtained with much less than 2^n hash evaluations.

Multicollisions for random maps. In contrast to [9], we consider generic attacks, and, accordingly, we model our functions as random maps. In that case, the number of $N^{(r-1)/r}$ is a lower bound on the sequential time required for finding a r-collision, and time-optimal algorithms are well-known. Furthermore, it is well-known how to find ordinary collisions (aka 2-collisions) with negligible memory (using Floyd, Brent or Nivasch [15] cycle finding algorithms), and also how to parallelize these algorithms using distinguished point methods [18–20, 23–27].

In general, the issue of memory-efficient and parallelizable r-collision algorithms appears to be an unsolved question. Authors usually assume $N^{(r-1)/r}$ units of memory (i.e., the maximum any algorithm can use in the given amount of time) and neglect parallelization entirely. For recent examples of the application of multicollisions to cryptography, see, e.g., the cryptanalysis of the SHA-3 candidates Aurora-512 [3, 21] and JH-512 [12, 29]. We stress that [3, 21, 12, 29] employ generic multicollisions as a part of their attacks, always assuming maximum memory and ignoring the issue of parallel implementations.

So the question is, *do authors need to be so pessimistic, or are there memory-efficient and parallelizable algorithms for r-collisions?* For small r, and mainly for $r = 3$, the current paper provides a clearly positive answer. As an application of our results, we will observe attacks on the SHA-3 candidate hash function Aurora-512. These attacks make heavy use of multicollisions on internal structures. Some attacks on other SHA-3 candidates don't benefit from our algorithms for different reasons. See section B of the appendix.

Notation. To avoid writing cumbersome logarithmic factors, we often express running times using the soft-Oh-notation. Namely, $\tilde{O}(g(n))$ is used as a shorthand for $O(g(n) \cdot \log(g(n))^k)$ for some fixed k.

2 Known algorithm for 3-collisions

While the number of values that needs to be computed before a 3-collision can be formed is often considered and analyzed, e.g. in [17, Appendix B] or [22], the known algorithmic method to find such a 3-collision is rarely considered in detail and is mostly folklore. In order to compare the new algorithms which we describe in sections 3 to 6 with existing algorithms, we thus give a precise description of the folklore algorithm, together with a larger variety of time/memory tradeoffs. Throughout this section, we fix two parameters α and β and consider 3-collisions for a function F defined on a set of cardinality N. The parameter α controls the amount of memory, limiting it to $\tilde{O}(N^\alpha)$. Similarly, β controls the running time, at $\tilde{O}(N^\beta)$. Of course, these parameters need to satisfy the relation $\alpha \leq \beta$.

We consider Algorithm 1. This algorithm is straightforward. First, it computes, stores and sorts N^α images of random points under F. For bookkeeping purposes, it also keeps track of the corresponding preimages. Second, it computes N^β additional images of random points and seek each in the precomputed table. Whenever a hit occurs, it is stored together with the initial preimage in the sorted table. The algorithm succeeds if one of the N^α original images is found twice more during the second phase and if the three corresponding preimages are distinct. In the formal description given as Algorithm 1, we added an optional step which packs colliding values generated during the first step into the same array element. If this optional step is omitted, then the early collisions are implicitly discarded. Indeed, in the second phase, we make sure that the search algorithm always returns the first position where a given value occurs among the known images $F(x)$. During the complexity analysis, we ignore the optional packing step since it runs in time N^α and can only improve the overall running time by making the algorithm stop earlier.

We now perform a rough heuristic analysis of Algorithm 1, where constants and logarithmic factors are ignored. On average, among the N^β images of the second phase, we expect that $N^{\alpha+\beta-1}$ values hit the sorted table of N^α elements. Due to the birthday paradox, after $N^{\alpha/2}$ hits, we expect a double hit to occur. At that point, the algorithm succeeds if the three known preimages corresponding to the double hit are distinct, which occurs with constant probability. For the algorithm to succeed, we need:

$$\alpha + \beta - 1 \geq \alpha/2,$$

as a consequence, to minimize the running time, we enforce the condition:

$$\alpha + 2\beta = 2. \tag{1}$$

For $\alpha = \beta$, we find $\alpha = \beta = 2/3$ and obtain the classical folklore result with time and memory $\tilde{O}(N^{2/3})$. Other tradeoffs are also possible. With constant memory, i.e. $\alpha = 0$, we find a running time $\tilde{O}(N)$. Another tradeoff with $\alpha = 1/2$ and $\beta = 3/4$ will be used as a point of comparison in section 3.

Algorithm 1 Folklore 3-collision finding algorithm

```
 Require: Oracle access to $F$ operating on $[0, N-1]$
 Require: Parameters: $\alpha \leq \beta$ satisfying condition 1
   Let $N_\alpha \longleftarrow \lceil N^\alpha \rfloor$
   Let $N_\beta \longleftarrow \lceil N^\beta \rfloor$
   Create arrays Img, $\mathrm{Pr}_1$ and $\mathrm{Pr}_2$ of $N_\alpha$ elements.

First step:
   for $i$ from 1 to $N_\alpha$ do
     Let $a \longleftarrow_R [0, N-1]$
     Let $\mathrm{Img}[i] \longleftarrow F(a)$
     Let $\mathrm{Pr}_1[i] \longleftarrow a$
     Let $\mathrm{Pr}_2[i] \longleftarrow \perp$
   end for
   Sort Img, applying the same permutation on elements of $\mathrm{Pr}_1$ and $\mathrm{Pr}_2$

Optional step (packing of existing collisions):
   Let $i \longleftarrow 1$
   while $i < N_\alpha$ do
     Let $j \longleftarrow i + 1$
     while $\mathrm{Img}[i] == \mathrm{Img}[j]$ do
       if $\mathrm{Pr}_1[i] \neq \mathrm{Pr}_1[j]$ then
         if $\mathrm{Pr}_2[i] == \perp$ then
           Let $\mathrm{Pr}_2[i] \longleftarrow \mathrm{Pr}_1[j]$
         else
           if $\mathrm{Pr}_2[i] \neq \mathrm{Pr}_1[j]$ then
             Output '3-Collision $(\mathrm{Pr}_1[i], \mathrm{Pr}_2[i], \mathrm{Pr}_1[j])$ under $F$' and Exit
           end if
         end if
       end if
       Let $j \longleftarrow j + 1$
     end while
     Let $i \longleftarrow j$
   end while

Second step:
   for $i$ from 1 to $N_\beta$ do
     Let $a \longleftarrow_R [0, N-1]$
     Let $b \longleftarrow F(a)$
     if $b$ is in Img (first occurrence in position $j$) then
       if $\mathrm{Pr}_1[j] \neq a$ then
         if $\mathrm{Pr}_2[j] == \perp$ then
           Let $\mathrm{Pr}_2[j] \longleftarrow a$
         else
           if $\mathrm{Pr}_2[j] \neq a$ then
             Output '3-Collision $(\mathrm{Pr}_1[j], \mathrm{Pr}_2[j], a)$ under $F$' and Exit
           end if
         end if
       end if
     end if
   end for
```

3 A new algorithm for 3-collisions

Now equipped with an analysis of Algorithm 1, we are ready to propose a new algorithm which offers different time-memory tradeoffs, which are better balanced for existing hardware. The basic idea is extremely simple: Instead of initializing an array with N^α images, we propose to initialize it with N^α collisions under F. To make this efficient in terms of memory use, each collision in the array is generated using a cycle finding algorithm on a (pseudo-)randomly permuted copy of F. Since each collision is found in time $N^{1/2}$ the total running time of this new first step is $N^{1/2+\alpha}$.

The second step is left unchanged, we simply create N^β images of random points until we hit one of the known collisions. Note that, thanks to the new first phase, it now suffices to land once on a known point to succeed. As a consequence, we can replace condition 1 by the weaker condition:

$$\alpha + \beta = 1. \tag{2}$$

Since the running time of the first step is $N^{1/2+\alpha}$, it would not make sense to have $\beta < 1/2 + \alpha$. Thus, we also enforce the condition $\alpha \leq 1/4$. Under this condition, the new algorithm runs in time $\tilde{O}(N^{1-\alpha})$ using $\tilde{O}(N^\alpha)$ bits of memory. In particular, we can find 3-collisions in time $\tilde{O}(N^{3/4})$ using $\tilde{O}(N^{1/4})$ bits of memory. This is a notable improvement over Algorithm 1 which requires $\tilde{O}(N^{1/2})$ bits of memory to achieve the same running time.

Note on the creation of the N^α initial collisions. One question that frequently arises when this algorithm is presented is: "Why is it necessary to randomize F with a pseudo-random permutation ?"

Behind this question is the idea that changing the starting point of the cycle finding algorithm should suffice to obtain random collisions. However, this is not true. Indeed, the analysis of random mapping (for example, see [4]) shows that on average a constant fraction of points belong to a so-called "giant tree". By definition, each starting point in the giant tree enters the main cycle in the same place. As a consequence, without randomization of F the corresponding collision would be generated over and over again and the 3-collision algorithm would not work.

4 Detailed complexity analysis of algorithms 1 and 2

In this section, we analyze in more details the complexity and success probability of algorithms 1 and 2, assuming that F is a random mapping. This detailed analysis particularly focuses on the following problematic issues which were initially neglected:

1. Among the N_α candidates stored in Img and its companion arrays, which fraction can non-trivially be completed into a 3-collision?

Algorithm 2 Improved 3-collision finding algorithm

```
Require: Oracle access to $F$ operating on $[0, N-1]$
Require: Family of pseudo-random permutation $\Pi_K$, indexed by $K$ in $\mathcal{K}$
Require: Parameters: $\alpha \leq \beta$ satisfying condition 2
  Let $N_\alpha \longleftarrow \lceil N^\alpha \rfloor$
  Let $N_\beta \longleftarrow \lceil N^\beta \rfloor$
  Create arrays Img, Pr$_1$ and Pr$_2$ of $N_\alpha$ elements.

First step:
  for $i$ from 1 to $N_\alpha$ do
    Let $K \longleftarrow_R \mathcal{K}$
    Use cycle finding algorithm on $F \circ \Pi_K$ to produce collision $F \circ \Pi_K(a) = F \circ \Pi_K(b)$
    Let Img[$i$] $\longleftarrow F \circ \Pi_K(a)$
    Let Pr$_1$[$i$] $\longleftarrow \Pi_K(a)$
    Let Pr$_2$[$i$] $\longleftarrow \Pi_K(b)$
  end for
  Sort Img, applying the same permutation on elements of Pr$_1$ and Pr$_2$

Optional step (packing of existing collisions):
  Let $i \longleftarrow 1$
  while $i < N_\alpha$ do
    Let $j \longleftarrow i + 1$
    while Img[$i$] == Img[$j$] do
      if Pr$_1$[$i$] $\neq$ Pr$_1$[$j$] then
        if Pr$_2$[$i$] $\neq$ Pr$_1$[$j$] then
          Output '3-Collision (Pr$_1$[$i$], Pr$_2$[$i$], Pr$_1$[$j$]) under $F$' and Exit
        end if
      end if
      Let $j \longleftarrow j + 1$
    end while
    Let $i \longleftarrow j$
  end while

Second step:
  for $i$ from 1 to $N_\beta$ do
    Let $a \longleftarrow_R [0, N-1]$
    Let $b \longleftarrow F(a)$
    if $b$ is in Img (first occurrence in position $j$) then
      if Pr$_1$[$j$] $\neq a$ then
        if Pr$_2$[$j$] == $\perp$ then
          Let Pr$_2$[$j$] $\longleftarrow a$
        else
          if Pr$_2$[$j$] $\neq a$ then
            Output '3-Collision (Pr$_1$[$j$], Pr$_2$[$j$], $a$) under $F$' and Exit
          end if
        end if
      end if
    end if
  end for
```

2. In the second step, when a value $F(a)$ hits the array Img, what is the probability of obtaining a real 3-collision and not simply replaying a known value of a?
3. Which logarithmic factors are hidden in the $\tilde{O}$ expression ?
4. In the first step of Algorithm 2, how can we make sure that we never encounter a bad configuration where the cycle finding algorithm runs for longer than $\tilde{O}(N^{1/2})$?

To answer the first question, remark that each candidate stored into Img is a random point that has at least one preimage for Algorithm 1 or at least two preimages for Algorithm 2. According to [4], we know that the expected fraction of points with exactly k distinct preimages is $e^{-1}/k!$. As a consequence, if we denote by P_k the fraction of points with at least k preimages, we find:

$$P_1 = \frac{e-1}{e}, \quad P_2 = \frac{e-2}{e} \quad \text{and} \quad P_3 = \frac{e-5/2}{e}.$$

The expected fraction of elements from Img which can be correctly completed into a 3-collision is $P_3/P_1 \approx 0.127$ for Algorithm 1 and $P_3/P_2 \approx 0.304$ for Algorithm 2. To compensate the loss, the easiest is to make the stored set larger by a factor of 8 in the first case and 3 in the second.

We now turn to the second question. Of course, at this point, the candidates that cannot be correctly completed need to be ignored. Among the original set of N_α candidates, we now focus on the subset of candidates that can correctly be computed and let N'_α denote the size of this subset. Since in the second phase we are sampling points uniformly at random, the *a posteriori* probability of having chosen one of the two already known preimages is at most $2/k$, where k is the number of distinct preimages for this point. Since $k \geq 3$, the *a posteriori* probability of choosing a new preimage is, at least, $1/3$. Similarly, for Algorithm 1, the *a posteriori* probability of choosing a preimage distinct from the single originally known one is at least $2/3$. To offset this loss of probability, N_β should be multiplied by a constant factor of 3.

The logarithmic factors involved in the third question are easy to find, they simply come from the sort and binary search steps. Note that when $N^\alpha \cdot \log(N^\alpha) < N^\beta$ the sort operation costs less than the second step and can be ignored. Moreover, as soon as $\alpha < \beta$, this bound is asymptotically achieved when N tends to infinity. However, the binary search appears within the second step and a real penalty is paid.

If we are willing to spend some extra memory – blowing up the memory by a constant factor –, this cost can be eliminated using hashing techniques. To cover the case of $N^\alpha \cdot \log(N^\alpha) = N^\beta$, we need a data structure with constant-time insert and lookup operations. One such data structure is “cuckoo hashing”, where lookup operations need worst-case constant time, and insert operations need expected constant time – as long as less than half of the memory slots are used [16].[4] However, for typical applications, the cost of the binary search ought

[4] Furthermore, delete operations only need worst-case constant time, and recent improvements even enable update operations in worst-case constant time [1].

to remain small, compared to the cost of evaluating the function F. Thus, in practice, we expect only a tiny benefit from using hash tables.

The simplest answer to the fourth question is to fix some upper bound on the allowed running time of each individual call to the collision through cycle finding algorithm. If the running time is exceeded, we abort and restart with a fresh permutation Π_K. With a time limit of the form $\lambda \sqrt{N}$ and a large enough value of λ, we make sure that each individual call to the cycle finding algorithm runs in time $O(N^{1/2})$ and the probability of success is a constant close to 1, say larger than 2/3.

5 A second algorithm with more tradeoff options

The algorithm presented in section 3 only works for memory up to $N^{1/4}$. This limitation is due to the way the collisions are generated during the first step of Algorithm 2. In order to extend the range of possible tradeoffs beyond that point, it suffices to find a replacement for this first step. Indeed, the second step clearly works with a larger value of α, as long as we keep the relation $\alpha + \beta = 1$. Of course, since no 3-collision is expected before we have performed $N^{2/3}$ evaluations of F, the best we can hope for is an algorithm with running time $N^{2/3}$. Such an algorithm may succeed if we can precompute a table containing $N^{1/3}$ ordinary collisions.

In this section, we consider the problem of generating $N^{1/3}$ collisions in time bounded by $\tilde{O}(N^{2/3})$ using at most $\tilde{O}(N^{1/3})$ bits of memory. Surprisingly, a simple method inspired from Hellman's time-memory tradeoff [5] is able to solve this problem. More generally, for $\alpha \leq 1/3$, this method allows us to compute N^{α} collisions in time less than $\tilde{O}(N^{1-\alpha})$ using at most $\tilde{O}(N^{\alpha})$ bits of memory. The idea is to first build N^{α} chains of length N^{γ}; each chain starts from a random point and is computed by repeatedly applying F up to the N^{γ}-th iteration. The end-point of each chain is stored together with its corresponding start-point. Once the chains have been build, we sort them by end-point values. Then, restarting from N^{α} new random points, we once again compute chains of length N^{γ}, the difference is that we now test after each evaluation of F whether the current value is one of the known end-points. In that case, we know that the chain we are currently computing has merged with one chain from the precomputation step. Such a merge usually corresponds to a collision, the only exception occurs when the start-point of the current chain already belongs to a precomputed chain (a "Robin Hood" using the terminology of [27]). Then, backtracking to the beginning of both chains, we can easily construct the corresponding collision. A pseudo-code description of this alternative first step is given as Algorithm 3.

Note that, instead of building two sets of chains, it is also possible to build a single set and look for previously known end-points. This alternative approach is a bit trickier to implement but uses fewer evaluations of F. However, the overall cost of the algorithm remains within the same order.

Clearly, since each of the two sets of chains we are constructing contain $N^{\alpha+\gamma}$ points, the expected number of collisions is $O(N^{2\alpha+2\gamma-1})$. Remembering that we

Algorithm 3 Alternative method for constructing N^{α} collisions

```
Require: Oracle access to $F$ operating on $[0, N-1]$
Require: Parameter: $\alpha \leq 1/3$
  Let $\gamma \longleftarrow (1-\alpha)/2$
  Let $N_\alpha \longleftarrow \lceil N^\alpha \rfloor$
  Let $N_\gamma \longleftarrow \lceil N^\gamma \rfloor$
  Create arrays Start and End of $N_\alpha$ elements.
  Create arrays Img, $\mathrm{Pr}_1$ and $\mathrm{Pr}_2$ of $N_\alpha$ elements.

Construction of first set:
  for $i$ from 1 to $N_\alpha$ do
    Let $a \longleftarrow_R [0, N-1]$
    Let Start[$i$] $\longleftarrow a$
    for $i$ from 1 to $N_\gamma$ do
      Let $a \longleftarrow F(a)$
    end for
    Let End[$i$] $\longleftarrow a$
  end for
  Sort End, applying the same permutation on elements of Start

Construction of second set and collisions:
  Let $t \longleftarrow 1$
  while $t < N_\alpha$ do
    Let $a \longleftarrow_R [0, N-1]$
    Let $b \longleftarrow a$
    for $j$ from 1 to $N_\gamma$ do
      Let $b \longleftarrow F(b)$
      if $b$ is in End (first occurrence in position $k$) then
        Let $a' \longleftarrow$ Start[$k$]
        for $l$ from 1 to $N_\gamma - j$ do
          Let $a' \longleftarrow F(a')$
        end for
        if $a \neq a'$ then
            {Checks that a genuine merge between chains exists}
          Let $b \longleftarrow F(a)$
          Let $b' \longleftarrow F(a')$
          while $b \neq b'$ do
            Let $a \longleftarrow b$
            Let $a' \longleftarrow b'$
            Let $b \longleftarrow F(a)$
            Let $b' \longleftarrow F(a')$
          end while
          Let Img[$t$] $\longleftarrow b$
          Let $\mathrm{Pr}_1[t] \longleftarrow a$
          Let $\mathrm{Pr}_2[t] \longleftarrow a'$
          Let $t \longleftarrow t+1$
        end if
        Exit Loop on $j$
      end if
    end for
  end while
  Return arrays Img, $\mathrm{Pr}_1$ and $\mathrm{Pr}_2$ containing $N_\alpha$ collisions.
```

wish to construct N^α collisions, we need to let $\gamma = (1-\alpha)/2$. The running time necessary to compute these collisions is $N^{\alpha+\gamma} = N^{(1+\alpha)/2}$. Note that, since $\alpha \leq 1/3$, we have $(1+\alpha)/2 \leq 1-\alpha$. As a consequence, the running time of the complete algorithm is dominated by the running time $N^\beta = N^{1-\alpha}$ of the second step.

6 Parallelizable 3-collision search

Since the computation involved during a search for 3-collisions is massive, it is essential to study the possibility of parallelizing such a search. For ordinary collisions, parallelization is studied in details in [27] using ideas introduced in [18–20, 23–26].

We first remark that the algorithms we have studied up to this point are badly suited to parallelization. Their main problem is that a large amount of memory needs to be replicated on every processor which is very impractical, especially when we want to use a large amount of low-end processors. We now propose an algorithm specifically suited to parallelization. For simplicity of exposition, we first assume that $N_p \approx N^{1/3}$ processors are available and aim at a running time $\tilde{O}(N^{1/3})$. Moreover, we would like each processor to use only a constant amount of memory. However, we assume that every processor can efficiently communicate with every other processor, as long as the amount of transmitted data remains small. It would be easy to adapt the approach to a network of small processors, with each processor connected to a central computer possessing $\tilde{O}(N^{1/3})$ bits of memory.

As for ordinary collisions, the key idea is to use *distinguished points*. By definition, a set of distinguished points is a set of points together with an efficient procedure for deciding membership. For example, the set of elements in $[0, M-1]$ can be used as a set of distinguished points since membership can be tested using a single comparison. Moreover, with this choice, the fraction of distinguished points among the whole set is simply M/N. Here, since we wish to have chains of average length $N^{1/3}$, we choose for M an integer near $N^{2/3}$.

The distinguished point algorithm works in two steps. During the first step, each processor starts from a random start-point s and iteratively applies F until a distinguished point d is encountered. It then transmits a triple (s, d, L), where L is the length of the path from s to d, to the processor whose number is $d \pmod{N_p}$. We abort any processor if it doesn't find a distinguished point within a reasonable amount of time, for example, following what [27] does for 2-collisions, we may abort after $20\,N/M$ steps. Once all the paths have been computed, we start the second step. Each processor looks at the triples it now holds. If a given value of d appears three or more times, the processor recomputes the corresponding chains, using the known length information to synchronize the chains. If three of the chains merge at a common position, a 3-collision is obtained.

Of course, even with less than $N^{1/3}$ processors, it is possible to do a partial parallelization. More precisely, given N^θ processors with $\theta \leq 1/3$, it is possible

Algorithm 4 Parallelizable 3-collisions using distinguished points

```
Require: Oracle access to $F$ operating on $[0, N-1]$
Require: Number of processors $N_p \leq N^{1/3}$
Require: Identity of current processor: $\mathrm{Id} \in [0, N_p - 1]$
  Let $M \longleftarrow \left\lceil N^{2/3} \right\rfloor$ {$M$ defines distinguished points}
  Let $L_{\mathrm{max}} = 20 \left\lceil N^{1/3} \right\rfloor$

Construction of triples:
  Let $s \longleftarrow_R [0, N-1]$; $a \longleftarrow s$; $L \longleftarrow 0$
  while $L < L_{\mathrm{max}}$ do
    Let $a \longleftarrow F(a)$; $L \longleftarrow L+1$
    if $a < M$ then
      Send triple $T \longleftarrow (s, a, L)$ to processor $a \pmod{N_p}$ and Exit Loop
    end if
  end while

Acquisition of triples:
  Store received triples $(s, d, L)$ in local arrays A, D, $\mathcal{L}$ numbered from 1 to $K$
  Sort D, applying the same permutation on elements of A and $\mathcal{L}$

Processing of triples:
  Let $i \longleftarrow 1$
  while $i \leq K$ do
    Let $j \longleftarrow i+1$
    while $j \leq K$ and $D[j] = D[i]$ do
      Let $j \longleftarrow j+1$
    end while
    if $j \geq i+3$ then
      Let $L \longleftarrow \max(\mathcal{L}[i], \cdots, \mathcal{L}[j-1])$
      for $\ell$ from $L$ downto 0 do
        for $k$ from $i$ to $j-1$ do
          if $\mathcal{L}[k] \geq \ell$ then
            Let D[$k$] $\longleftarrow$ A[$k$]; A[$k$] $\longleftarrow F($A[$k$]$))$
             {D[$k$] overwritten to keep previous value of A[$k$]}
          end if
        end for
        Check for 3 equal values in A[$i \cdots j-1$] with differing values of D
        If found, Output the 3-collision and Exit
      end for
    end if
    Let $i \longleftarrow j$
  end while
```

to find 3-collisions in time $\tilde{O}(N^{2/3-\theta})$. In that case, each processor needs a local memory of size $O(N^{1/3-\theta})$ to store all the triples it owns.

7 Extension to r-collisions, for $r > 3$

For r-collisions, recall that we need to evaluate F on approximately $r!^{1/r}\,N^{(r-1)/r}$ points before hoping for a collision. When considering that r is a fixed value, $r!^{1/r}$ is a constant and vanishes within the $\tilde{O}$ notation. With this new context, Algorithm 4 is quite easy to generalize. Here, the important parameter is to create shorter chains and compute more of them. The reason for shorter chains is that (as in Hellman's Algorithm [5]), we need to make sure that there are not too many collisions between one chain and all the others. Otherwise, the algorithm spends too much time recomputing the same evaluations of the random map, which is clearly a bad idea. To avoid this, we construct chains which are short enough to make sure that the average number of (initial[5]) collisions between an individual chain and all the other chains is a constant. Since the total number of elements in all the other chains is essentially $N^{(r-1)/r}$, the length of chains should remain below $N^{1/r}$.

To achieve maximal parallelization when searching for an r-collision, $N_p \approx N^{(r-2)/r}$ processors are required. The integer M that defines distinguished points should be near $N^{(r-1)/r}$. Each processor first builds a chain of average length $N^{1/r}$ (as before we abort after $20\,N/M$ steps), described by a triple (s, d, L). Each chain is sent to the processor whose number is $d \pmod{N_p}$. During the second step, any processor that holds a value of d that appears in r or more triples recomputes the corresponding chains. If r chains merge at the same position, a r-collision is obtained.

Given N^{θ} processors with $\theta \leq (r-2)/r$, it is possible to find r-collisions in time $\tilde{O}(N^{(r-1)/r-\theta})$. In that case, each processor needs a local memory of size $O(N^{(r-2)/r-\theta})$.

With a single processor, the required amount of memory is $O(N^{(r-2)/r})$. Thus, as r grows, the advantage of the single processor approach on the folklore algorithm (which requires $O(N^{(r-1)/r})$ memory) becomes smaller and smaller. As a consequence, for larger values of r, it is essential to rely on parallelization.

8 Conclusion

In this paper, we revisited the problem of constructing multicollisions on random mappings and showed that it can be done using less memory than required by the folklore algorithm. For 3-collisions, the sequential running remains at $\tilde{O}(N^{2/3})$ but the amount of memory can be reduced from $O(N^{2/3})$ to $O(N^{1/3})$. A remaining open problem is to determine whether this amount of memory can further be reduced.

[5] Of course, once a collision occurs, all the values that follow are colliding. However, we do not count these follow-up collisions.

Furthermore, finding 3-collisions can be very efficiently parallelized. Given $N^{1/3}$ parallel processors, each equipped with constant memory, the problem can be solved in time $\tilde{O}(N^{1/3})$. More generally for $r \geq 3$, we show how to generate r-collisions on N^θ processors, each with local memory $O(N^{(r-2)/r-\theta})$, in time $\tilde{O}(N^{(r-1)/r-\theta})$. It is interesting to note that the cost of the parallelizable approach in the full-cost model [28] decreases as θ grows.

References

1. Yuriy Arbitman, Moni Naor, and Gil Segev. De-amortized cuckoo hashing: Provable worst-case performance and experimental results. In *ICALP*, LNCS. Springer, 2009.
2. Don Coppersmith. Another birthday attack. In Hugh C. Williams, editor, *CRYPTO'85*, volume 218 of *LNCS*, pages 14–17, Santa Barbara, CA, USA, August 18–22, 1986. Springer-Verlag, Berlin, Germany.
3. Niels Ferguson and Stefan Lucks. Attacks on AURORA-512 and the double-mix Merkle-Damgård transform. Cryptology ePrint Archive, Report 2009/113, 2009.
4. Philippe Flajolet and Andrew M. Odlyzko. Random mapping statistics. In Jean-Jacques Quisquater and Joos Vandewalle, editors, *EUROCRYPT'89*, volume 434 of *LNCS*, pages 329–354, Houthalen, Belgium, April 10–13, 1990. Springer-Verlag, Berlin, Germany.
5. Martin E. Hellman. A cryptanalytic time-memory trade-off. *IEEE Transactions on Information Theory*, 26(4):401–406, 1980.
6. Jonathan J. Hoch and Adi Shamir. Breaking the ICE - finding multicollisions in iterated concatenated and expanded (ICE) hash functions. In Matthew J. B. Robshaw, editor, *FSE 2006*, volume 4047 of *LNCS*, pages 179–194, Graz, Austria, March 15–17, 2006. Springer-Verlag, Berlin, Germany.
7. Jonathan J. Hoch and Adi Shamir. On the strength of the concatenated hash combiner when all the hash functions are weak. In *ICALP (2)*, volume 5126 of *LNCS*, pages 616–630. Springer, 2008.
8. Tetsu Iwata, Kyoji Shibutani, Taizo Shirai, Shiho Moriai, and Toru Akishita. AURORA: a cryptographic hash algorithm family. Submission to NIST's SHA-3 competition, 2008.
9. Antoine Joux. Multicollisions in iterated hash functions. application to cascaded constructions. In Matthew Franklin, editor, *CRYPTO 2004*, volume 3152 of *LNCS*, pages 306–316, Santa Barbara, CA, USA, August 15–19, 2004. Springer-Verlag, Berlin, Germany.
10. Florian Mendel. Preimage attack on Blender. Available from `http://ehash.iaik.tugraz.at/ wiki/Blender`.
11. Florian Mendel, Christian Rechberger, and Martin Schläffer. Cryptanalysis of twister. In Proceedings of ACNS, Springer, To appear, Available from `http://ehash.iaik.tugraz.at/wiki/Twister`.
12. Florian Mendel and Sœren S. Thomsen. An observation on JH-512. Available from `http://ehash.iaik.tugraz.at/wiki/JH`.
13. Mridul Nandi and Douglas R. Stinson. Multicollision attacks on some generalized sequential hash functions. *IEEE Transactions on Information Theory*, 53(2):759–767, 2007.
14. Craig Newbold. Observations and attacks on the SHA-3 candidate Blender. Available from `http://ehash.iaik.tugraz.at/wiki/Blender`.

15. G. Nivasch. Cycle detection using a stack. *Information Processing Letter*, 90(3):135–140, 2004.
16. Rasmus Pagh and Flemming F. Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004.
17. Bart Preneel. *Analysis and Design of Cryptographic Hash Functions*. PhD thesis, KU Leuven, 1993.
18. Jean-Jacques Quisquater and Jean-Paul Delescaille. Other cycling tests for DES (abstract). In Carl Pomerance, editor, *CRYPTO'87*, volume 293 of *LNCS*, pages 255–256, Santa Barbara, CA, USA, August 16–20, 1988. Springer-Verlag, Berlin, Germany.
19. Jean-Jacques Quisquater and Jean-Paul Delescaille. How easy is collision search? application to DES (extended summary). In Jean-Jacques Quisquater and Joos Vandewalle, editors, *EUROCRYPT'89*, volume 434 of *LNCS*, pages 429–434, Houthalen, Belgium, April 10–13, 1990. Springer-Verlag, Berlin, Germany.
20. Jean-Jacques Quisquater and Jean-Paul Delescaille. How easy is collision search. New results and applications to DES. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 408–413, Santa Barbara, CA, USA, August 20–24, 1990. Springer-Verlag, Berlin, Germany.
21. Yu Sasaki. A collision attack on AURORA-512. Cryptology ePrint Archive, Report 2009/106, 2009.
22. Kazuhiro Suzuki, Dongvu Tonien, Kaoru Kurosawa, and Koji Toyota. Birthday paradox for multi-collisions. In Min Surp Rhee and Byoungcheon Lee, editors, *ICISC 06*, volume 4296 of *LNCS*, pages 29–40, Busan, Korea, November 30 – December 1, 2006. Springer-Verlag, Berlin, Germany.
23. Paul C. van Oorschot and Michael J. Wiener. A known plaintext attack on two-key triple encryption. In Ivan Damgård, editor, *EUROCRYPT'90*, volume 473 of *LNCS*, pages 318–325, Aarhus, Denmark, May 21–24, 1990. Springer-Verlag, Berlin, Germany.
24. Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with application to hash functions and discrete logarithms. In *ACM CCS 94*, pages 210–218, Fairfax, Virginia, USA, 1994. ACM Press.
25. Paul C. van Oorschot and Michael J. Wiener. Improving implementable meet-in-the-middle attacks by orders of magnitude. In Neal Koblitz, editor, *CRYPTO'96*, volume 1109 of *LNCS*, pages 229–236, Santa Barbara, CA, USA, August 18–22, 1996. Springer-Verlag, Berlin, Germany.
26. Paul C. van Oorschot and Michael J. Wiener. On Diffie-Hellman key agreement with short exponents. In Ueli M. Maurer, editor, *EUROCRYPT'96*, volume 1070 of *LNCS*, pages 332–343, Saragossa, Spain, May 12–16, 1996. Springer-Verlag, Berlin, Germany.
27. Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12(1):1–28, 1999.
28. Michael J. Wiener. The full cost of cryptanalytic attacks. *Journal of Cryptology*, 17(2):105–124, March 2004.
29. Hongjun Wu. The complexity of Mendel and Thomsen's preimage attack on JH-512. Available from `http://ehash.iaik.tugraz.at/wiki/JH`.

A Practical implementation

Since we only performed a heuristic analysis of our algorithms, in order to show that they are really effective, we decided to illustrate our 3-collision techniques with a practical example. For this purpose, we construct a random function by Xoring two copies of the DES algorithm (with two different keys). More precisely, we let:

$$F(x) = \mathrm{DES}_{K_1}(x) \oplus \mathrm{DES}_{K_2}(x),$$

where[6] $K_1 = (3322110077665544)_{16}$ and $K_2 = (3b2a19087f6e5d4c)_{16}$. Since x is on 64 bits, the time and memory requirements of the folklore algorithm are around 2^{43}. Where current computers are concerned, performing 2^{43} operations is easily feasible. However, storing 2^{43} values of x requires 2^{46} bytes, i.e. 64 Terabytes. As a consequence, finding 3-collisions on F with the basic parameters of the folklore algorithm is probably beyond feasibility. Using a different time-memory trade-off, restricting the storage to 2^{32} values would raise the time requirement to 2^{48} operations. This is within the range of currently accessible computations. However, since the algorithm is not parallelizable, it would require a high-end computer.

With the new algorithms presented in this paper, it becomes possible to compute triple collisions much more efficiently on the function F. For our implementation, we chose $M = 2^{44}$ to define the distinguished points, which yielded chains of expected length 2^{20}. The abort length was set at 8 times the expected length, rather than the factor 20 given in Algorithm 4. For computing the chains, we used a mix of 32 Intel Xeon processors at 2.8 GHz and 8 Nvidia CUDA cards (Tesla type). We collected a total of 35 447 322 chains and obtained 3 078 699 groups of three or more chains yielding the same distinguished endpoints. The largest group contained 36 chains, which shows that it would have been preferable to use slightly shorter chains. On processors only, this first phase would have taken about 94 CPU-days to run. On a single CUDA card, it would have taken 11.5 days.

For simplicity of implementation, the second phase of the algorithm was only performed on Intel processors and not on CUDA cards. It took less than 18 CPU-days to test all groups and it yielded the following triple-collisions:

$$\begin{aligned}
F(d332b9ba5e5a7d4e) &= F(51b8095db532afcc) = F(b084dc15dce042ab),\\
F(ca76ff906d6587cf) &= F(e1f7f59a5757d01b) = F(0285f58147e863c2),\\
F(c3783ef30c8bcc3d) &= F(65f14d412fd91173) = F(1042d827e5078000).
\end{aligned}$$

We would like to thank CEA/DAM[7] for kindly providing the necessary computing time on its Tesla servers.

[6] This keys might seem weird, but they should not have any special properties. In truth, we intended to choose $K_1 = (0011223344556677)_{16}$ and $K_2 = (08192a3b4c5d6e7f)_{16}$, i.e., $(8899aabbccddeeff)_{16}$ with high bits stripped. Unfortunately, the first-named author made a classical endianness mistake while implementing the algorithm.

[7] Commissariat à l'énergie atomique, Direction des applications militaires.

B Applications

B.1 Collisions for the Hash function AURORA-512

AURORA is a family of cryptographic hash functions submitted to the NIST SHA-3 hash function competition [8]. Like the other members of the AURORA family, AURORA-512 employs different internal compression functions, each mapping a 256-bit chaining value and a 512-bit message block to generate a new 256-bit chaining value. AURORA-512 is the high-end member of that family, maintaining an internal state of 512 bit. As required by the NIST, the authors of AURORA-512 explicitly claim "collision resistance of approximately 512/2 bits" for AURORA-512. In other words, collision attacks must not significantly improve over the generic birthday attack, which takes roughly the time of 2^{256} hash operations.

Internally, AURORA-512 works almost like the cascade of two iterated hash functions, except for one important extra operation:

$$\mathrm{MF} : \{0,1\}^n \times \{0,1\}^n \rightarrow \{0,1\}^n \times \{0,1\}^n.$$

See Algorithm 5 for a simplified description of AURORA-512.

Algorithm 5 AURORA-512: Hashing 8 message blocks.

```
Require: Input Chaining Values (Left, Right) $\in (\{0,1\}^{256})^2$
  for $i$ from 0 to 7 do
    Left $\longleftarrow$ Compress(Left, Message_Block($i$))
    Right $\longleftarrow$ Compress(Right, Message_Block($i$))
  end for
  (Left, Right) $\longleftarrow$ MF(Left, Right)
```

Every eighth iteration, MF is called to mix the two half-states. This seems to defend against the cascade-attack from [9]: Between two MF-operations, one can generate local collisions in each iteration in one of either the left string, or the right string. Thus, the adversary can get a local 2^8-collision. But to apply the attack from [9], one would rather need a 2^{128}-collision, so the attack fails.

Assume, for a moment, that the adversary has generated a 2^7-collision on Left in the first 7 iterations of the loop. For the right string, we have 2^7 different values $\mathrm{Right}_1, \mathrm{Right}_2, \ldots, \mathrm{Right}_{128}$. If two of them collide, a collision for AURORA-512 has been found. For a fixed Message_Block(7), the chance of a collision, i.e. of $j \neq k$ with

$$\begin{gathered}\mathrm{Compress}(\mathrm{Right}_j, \mathrm{Message\_Block}(7)) \\ = \\ \mathrm{Compress}(\mathrm{Right}_k, \mathrm{Message\_Block}(7))\end{gathered}$$

is about $2^7 \cdot (2^7 - 1) \cdot 2^{-1}/2^{256}$. By trying out $2^{256-(6+7)}$ different values for Message_Block(7), we expect to find a collision. Note that this means to make

2^7 calls to the function Compress. Hence, this attack takes the time of about $2^{256-(6+7)+7} = 2^{250}$ compression function calls, plus the time to generate the 2^7-collision at the beginning. This is essentially the memoryless variant of the attack from [3], except that the authors of [3] actually generate a 2^8-collision on Left, by exploiting the previous eight-tuple of message blocks. The attack is memoryless, since the adversary only needs to generate 2-collisions on Left, and the claimed time is 2^{249}.

In [3], Ferguson and Lucks further propose an attack which uses local r-collision, instead of local 2-collisions. A similar attack has been proposed independently [21]. Using eight local r-collisions allows to speed-up the attack to roughly $2^{256}/r^7$ compression function calls (plus the time to generate the required r-collisions). [3] suggest $r = 9$ (beyond that, computing the r-collisions becomes too costly) and claim time $2^{234.5}$, including the time to generate ten local 9-collisions. The price for the speed-up is utilizing a huge amount of memory, however.

Our memory-efficient 3-collision allows a different time-memory tradeoff. The time is $2^{256}/3^7 \approx 2^{245}$. Recall $N = 2^{256}$, and set $\alpha := 1/16$, $\beta := 15/16$ in Algorithm 2. In that case one local 3-collision requires time 2^{240}, which we neglect. The memory requirements are down to 2^{16}, i.e., almost negligible.

It is also possible to use more general r-collisions to further improve this attack. For example, we can use 4-collisions obtained using the algorithm of section 7. To simplify the comparison with previous attacks, we assume a single processor, i.e. set $\theta = 0$, however, with more processors, we would obtain an even better attack. With this choice, a 4-collision on 256-bits is obtained in time 2^{192} using a memory of size 2^{128}. The corresponding speedup is 4^7. Similarly, 8-collisions on 256 bits are obtained in time 2^{224} each, using 2^{192} units of memory. The speed-up is 8^7. Other trade-offs are possible.

The results on collision attacks for AURORA-512 can be summarised as follows:

r (arity)	time [compr. fn. calls]	memory	reference
9	$2^{234.5}$	$2^{229.6}$	[3]
8	2^{236}	2^{236}	[21]
2	2^{249}	—	[3]
3	2^{245}	2^{16}	(this paper)
4	2^{242}	2^{128}	(this paper)
8	2^{235}	2^{192}	(this paper)

B.2 Attacks on other Hash Functions

Several attacks on several other SHA-3 candidates make heavy use of multicollisions, and it appears a natural idea to plug in our algorithms for reducing the memory consumption of these attacks. We actually tried to do so, but only succeeded for Aurora-512. In the current section, we will explain why we failed for other obvious candidates.

Several attacks, such as the attacks on Blender [14, 10] and on Twister [11], employ multicollisions, but it turns out that these can actually be generated by Joux-style iterated 2-collisions, which is very memory-efficient – and also faster than our general multicollision algorithms, anyway.

An obvious candidate to employ our algorithms to improve given cryptanalytic attacks is a preimage attack on JH-512 [12]. Like Aurora, JH is a family of hash functions submitted to the SHA-3 competition. The high-end 512-bit variant is denoted as JH-512. Internally, JH-512 is a wide-pipe hash function with an internal state of 1024 bit, and it employs an invertible compression function. [12] propose a meet-in-the-middle attack which requires "$2^{510.3}$ compression function evaluations *and a similar amount of memory*" (our emphasis). The authors of [12] stress: "We do not claim that our attack breaks JH-512 (due to the high memory requirements)." The author of JH-512 provides a more detailed analysis of this attack, claiming "$2^{510.6}$ [units of] memory". A main phase of the attack is generating several 51-collisions on one half of the chaining values (i.e., on 512 bits). By applying our algorithms to this task, it is possible to reduce the memory required for this phase to $2^{(512/51)\cdot 49}$ units of memory.

But another phase of the attack from [12] is to apply the inverse of the compression function to generate 2^{509} internal target values. The attack successfully generates a message which hashes to a given preimage, if the first part of the message hashes to any of these 2^{509} target values. Finally, the overall amount of storage for the attack is dominated by storing these 2^{509} values, regardless of improving memory-efficiency of the multicollision phase.