Memory-Efficient Algorithms for Finding Needles in Haystacks

Itai Dinur[1(✉)], Orr Dunkelman[2], Nathan Keller[3], and Adi Shamir[4]

[1] Computer Science Department, Ben-Gurion University, Beersheba, Israel
dinuri@cs.bgu.ac.il
[2] Computer Science Department, University of Haifa, Haifa, Israel
[3] Department of Mathematics, Bar-Ilan University, Ramat Gan, Israel
[4] Computer Science Department, The Weizmann Institute, Rehovot, Israel

Abstract. One of the most common tasks in cryptography and cryptanalysis is to find some interesting event (a needle) in an exponentially large collection (haystack) of $N = 2^n$ possible events, or to demonstrate that no such event is likely to exist. In particular, we are interested in finding needles which are defined as events that happen with an unusually high probability of $p \gg 1/N$ in a haystack which is an almost uniform distribution on N possible events. When the search algorithm can only sample values from this distribution, the best known time/memory tradeoff for finding such an event requires $O(1/Mp^2)$ time given $O(M)$ memory.

In this paper we develop much faster needle searching algorithms in the common cryptographic setting in which the distribution is defined by applying some deterministic function f to random inputs. Such a distribution can be modelled by a random directed graph with N vertices in which almost all the vertices have $O(1)$ predecessors while the vertex we are looking for has an unusually large number of $O(pN)$ predecessors. When we are given only a constant amount of memory, we propose a new search methodology which we call **NestedRho**. As p increases, such random graphs undergo several subtle phase transitions, and thus the log-log dependence of the time complexity T on p becomes a piecewise linear curve which bends four times. Our new algorithm is faster than the $O(1/p^2)$ time complexity of the best previous algorithm in the full range of $1/N < p < 1$, and in particular it improves the previous time complexity by a significant factor of $\sqrt{N}$ for any p in the range $N^{-0.75} < p < N^{-0.5}$. When we are given more memory, we show how to combine the **NestedRho** technique with the parallel collision search technique in order to further reduce its time complexity. Finally, we show how to apply our new search technique to more complicated distributions

O. Dunkelman was supported in part by the Israeli Science Foundation through grant No. 827/12 and by the Commission of the European Communities through the Horizon 2020 program under project number 645622 PQCRYPTO.

N. Keller was supported by the Alon Fellowship.

A. Shamir—Part of the work was done while the fourth author was visiting the Institute of Theoretical Studies. He would like to thank ITS, Dr. Max Rössler, the Walter Haefner Foundation and the ETH Foundation.

M. Robshaw and J. Katz (Eds.): CRYPTO 2016, Part II, LNCS 9815, pp. 185–206, 2016.
DOI: 10.1007/978-3-662-53008-5_7

with multiple peaks when we want to find all the peaks whose probabilities are higher than p.

Keywords: Cryptanalysis · Needles in haystacks · Mode detection · Rho algorithms · Parallel collision search

1 Introduction

Almost everything we do in the construction and analysis of cryptographic schemes can be viewed as searching for needles in haystacks: identifying the correct key among all the possible keys, finding preimages in hash functions, looking for biases in the outputs of stream ciphers, determining the best differential and linear properties of a block cipher, hunting for smooth numbers in factoring algorithms, etc. As cryptanalysts, our goal is to find such needles with the most efficient algorithm, and as designers our goal is to make sure that such needles either do not exist or are too difficult to find.

Needles can be defined in many different ways, depending on what distinguishes them from all the other elements in the haystack. One common theme which characterizes many types of needles in cryptography is that they are probabilistic events which have the highest probability p among all the $N = 2^n$ possible events in the haystack. Such an element is called the *mode of the distribution*, and for the sake of simplicity we will first consider the case in which the distribution is almost flat: a single peak has a probability of $p \gg 1/N$ and all the other events have a probability of about $1/N$ (as depicted in Fig. 1). Later on we will consider the more general case of distributions in which there are several peaks of varying heights, and we want to find all of them.

Our goal in this paper is to analyze the complexity of this probabilistic needle finding problem, assuming that the haystack distribution is given as a black box. By abstracting away the details of the task and concentrating on its essence, we make our techniques applicable to a wide variety of situations. On the other hand, in this general form we can not use specific optimization tricks that can make the search for particular types of needles more efficient.[1]

We will be interested in optimizing both the time complexity and the memory complexity of the search algorithm. Since random-access memory is usually much more expensive than time, we will concentrate primarily on memory-efficient algorithms: We will start by analyzing the best possible time complexity of algorithms which can use only a constant amount of memory, and then study how the time complexity can be reduced by using some additional memory.

The paper is organized as follows: Sect. 2 formalizes our computational model. Section 3 describes the best previously known folklore algorithms for solving the problem. We then show how to use standard collision detection algorithms to identify the mode when its probability p is sufficiently large in Sect. 4. We follow

[1] We leave to future work specific applications of our techniques to the concrete problems mentioned at the beginning of this Section.

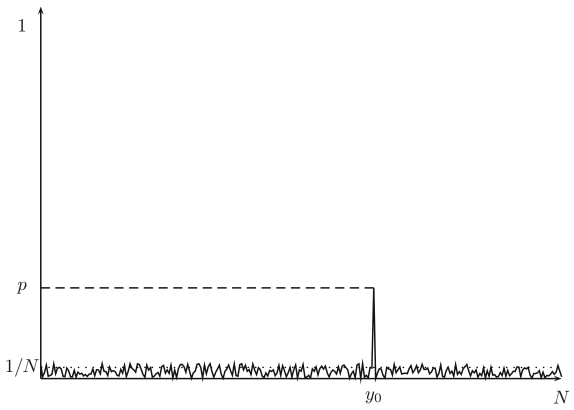


Fig. 1. An Example of the distributions that interest us

in Sect. 5 by introducing the new **2Rho** algorithm which uses a collision detection algorithm on the amplified mode probability obtained by running another collision detection algorithm on the original distribution. The algorithm is then extended to a general ***i*Rho** by using even deeper nesting of the collision detection algorithm in Sect. 6. We consider time-memory tradeoffs in Sect. 7, and discuss the adaptations needed when the probability distribution has multiple peaks and we want to find all of them in Sect. 8. Finally, Sect. 9 concludes the paper.

2 Problem Statement and Model Description

The simplest conceptual model for our problem is one in which the sampling black box has a button, and each time we press it we are charged a unit of time and we get a freshly chosen event from the distribution. We can thus test whether a particular y is the mode y_0 by counting how many times this y was sampled from the distribution in $O(1/p)$ trials. Notice that when we have a single available counter, we have to run this algorithm separately for each candidate y. The simplest possible algorithm sequentially tries all the N possible candidates, but we can use the given distribution in order to make a better choice of the next candidate to test. Since the correct candidate is suggested with an enhanced probability of $1/p$, the time complexity is reduced from N/p to $1/p^2$. When we have M available counters, we can get a linear speed up by testing M candidate values simultaneously with the same number of samples, provided that $1/p \geq M$. This trivial approach yields the best known algorithms for finding the mode of a flat distribution with a single peak.

However, closer inspection of the problem shows that in most of our cryptographic applications, the distribution we want to analyze is actually generated by some deterministic function f whose input is randomly chosen from a uniform probability distribution. For example, when we look for biases in the first n

output bits of a stream cipher, we choose a random key, apply to it the deterministic bit generator, and define the (possibly non-uniform) output distribution by saying that a particular bit string has a probability of i/N if it occurs as a prefix of the output string for i out of the N possible keys. Similarly, when we look for a high probability differential property, we choose random pairs of plaintexts with a certain input difference, and deterministically encrypt them under some fixed key. This process generates a distribution by counting how many times we get each output difference, and the mode of this distribution suggests the best differential on the block cipher which uses the selected input difference.

In such situations, we replace the button in the black box by an input which can accept N possible values. The box itself becomes deterministic, and we sample the distribution by providing to the box a randomly chosen input value. The main difference between the two models is that when we repeatedly press the button we get unrelated samples, but when we repeatedly provide the same input value we always get the same output value. As we show in this paper, this small difference leads to surprising new kinds of mode-finding algorithms which have much better complexities than the trivial algorithm outlined above.

The mapping from inputs to outputs defined by the function f can be viewed as a random directed bipartite graph such as the one presented in Fig. 2, in which one of the vertices has a large in-degree. For the sake of simplicity, we assume that the function f has the same number N of possible inputs and outputs[2], and then we can merge input and output vertices which have the same name to get the standard model of a random single-successor graph on N vertices. When we iterate the application of the function f in this graph, we follow a Rho-shaped path which starts with a tail and then gets into a cycle. The graph consists of a small number of disjoint cycles, and all the other vertices in the graph are hanging in the form of trees around these cycles.

As we increase the probability p from $1/N$ to 1, one of the vertices y_0 becomes increasingly popular as a target, and the graph changes its properties. For example, it is easy to show that when p crosses the threshold of $O(1/\sqrt{N})$, there is a sudden phase transition in which y_0 is expected to move from a tree into one of the cycles (where it becomes much easier to locate), and the expected length of its cycle starts to shrink (whereas earlier it was always the same). As we show later in the paper, more subtle phase changes happen when p crosses several earlier thresholds, and thus the log-log complexity of our mode-searching algorithm becomes a piecewise linear function that bends several times at those thresholds, as depicted by the solid line in Fig. 5. Compared to the dotted line which depicts the best previous $1/p^2$ complexity, we get a significant improvement in the whole range of possible p values.

[2] If there is some discrepancy, we can use the same truncation trick that Hellman used in his time/memory tradeoff to deal with cryptosystems in which the key and ciphertext sizes are different.

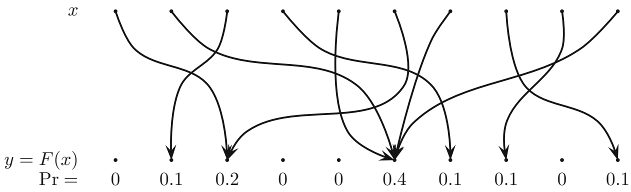


Fig. 2. A graph representation of a biased function f

2.1 Notations and Conventions

Notation 1. *The set* $\{1, 2, \dots, N\}$ *is denoted by* $[N]$*. Throughout the paper, all functions are from the set* $[N]$ *to itself. The* mode *of a function is the value in its range with the largest number of preimages.*

Problem Setup: The basic problem we study is the following. We are given a value $0 < p < 1$ and a function $f : [N] \to [N]$ which is generated by the following three-step process:

1. Choose $y_0 \in [N]$ uniformly at random.
2. Choose a subset $S \subset [N]$ uniformly at random amongst all subsets of size pN. Set $f(x) = y_0$ for all $x \in S$.
3. For each $x \notin S$, choose $f(x) \in [N] \setminus \{y_0\}$ uniformly at random.

By definition, the values of f on $[N] \setminus S$ can be simulated by a truly random oracle returning values in $[N] \setminus \{y_0\}$. Our initial goal is to detect y_0 with the fastest possible algorithm that uses only $O(1)$ memory cells. We can assume that the attacker knows p, since otherwise he can run a simple search algorithm with a geometrically decreasing sequence of probabilities (e.g., $1, 1/2, \dots, 1/2^i, \dots$) to find the highest value of p for which his attack succeeds (or stop when the attack becomes too expensive, which provides an upper bound on the probability of y_0, but does not identify it).

3 Trivial Memoryless Algorithms

In this section we formally present the simplest possible memoryless algorithms for detecting the mode for various values of p. They are based on sampling random points, and then checking whether they are indeed the required mode.

3.1 Memoryless Mode Verification Algorithm

We start the discussion by presenting a mode verification algorithm. The algorithm accepts a candidate y, and checks whether it is the mode y_0. The checking is done by choosing $O(1/p)$ random values, and verifying that sufficiently many

Algorithm 1. Mode Verification: Determining Whether a Given y is y_0

```
Initialize a counter ctr ← 0.
for i = 1 to c/p do
    Pick at random x ∈ [N], and compute y' = f(x).
    if y' = y_0 then
        Increment ctr.
    end if
end for
if ctr ≥ t then
    print y is y_0.
end if
```

of them are mapped to y under the function f. The algorithm is presented in Algorithm 1.

It is easy to see that Algorithm 1 makes c/p queries to $f(\cdot)$. Its success depends on the picked constants c and t. Assuming that indeed y is y_0 we expect that the number of times the chosen x leads to y is distributed according to a Poisson distribution with a mean value of c (otherwise, the distribution follows a Poisson distribution with a mean value of $c/Np \ll c$). Hence, for any desired success rate, one can easily choose c and the threshold t. For example, setting $c = 4$ and $t = 2$ offers a success rate of about 90.8 %.

3.2 Memoryless Sampling Algorithm

The sampling algorithm suggested in Algorithm 2 is based on picking at random a value x, computing a candidate $y = f(x)$ for the verification algorithm, and verifying whether y is indeed the correct y_0. It is easy to see that the algorithm is expected to probe $O(p^{-1})$ values of y, until y_0 is encountered, and that each verification takes $O(p^{-1})$, resulting in a running time of $O(p^{-2})$.

Algorithm 2. Finding y_0 by Sampling:

```
while y_0 was not found do
    Pick x ∈ [N] at random.
    Compute y = f(x).
    Call Algorithm 1 to check y.
end while
```

4 Using Rho-based Collision Detection Algorithms

We now present a different class of algorithms for detecting the mode, using collision detection algorithms combined with the trivial mode verification algorithm (Algorithm 1). These algorithms (such as Floyd's [7] or its variants [2,8]) start from some random point x, and iteratively apply f to it, i.e., produce the sequence $x, f(x), f^2(x) = f(f(x)), f^3(x), \ldots$, until a repetition is detected[3]. In the sequel, we call such algorithms "**Rho** algorithms". We denote the first repeated value in the sequence $x, f(x), f^2(x), \ldots$ by $f^{\mu}(x)$ and its second appearance by $f^{\mu+\lambda}(x)$, and call this common value the cycle's entry point.

Optimal Detection when $p \gg N^{-1/2}$. First, we show that when $p \gg N^{-1/2}$, the mode y_0 can be found in time $O(1/p)$. This complexity is clearly the best possible: if the number of queries to f is $o(1/p)$, then with overwhelming probability no preimage of y_0 is queried and so y_0 cannot be detected.

The idea is simple: we run a **Rho** algorithm, with an arbitrary random starting point x and an upper bound c/p on the length of the sequence for some small constant c. For such a length, we expect the mode y_0 to appear twice in the sequence with high probability, whereas due to the fact that $c/p < \sqrt{N}$ the collision found by **Rho** is not expected to be one of the other random values. We show the full analysis of the algorithm in Appendix A.

By using a memoryless **Rho** algorithm, we get a time complexity of $O(1/p)$ and a memory complexity of $O(1)$. As usual, the probability that y_0 is detected can be enhanced even further by repeating the algorithm with other starting points and checking each suggested point in time $O(1/p)$ using the trivial mode verification algorithm.

The RepeatedRho Algorithm: Detection in $O(p^{-3}N^{-1})$ for Arbitrary p. The above approach can be used for any value of p. However, when $p < 1/\sqrt{N}$, the probability that the output of **Rho** (i.e., the cycle's entry point) is indeed y_0 drops significantly. Specifically, we have the following lower bound, and one can easily show that the actual value is not significantly larger.

Proposition 1. *Assume that $p < 1/\sqrt{N}$ and thus **Rho** encounters $O(\sqrt{N})$ different output values until a collision is detected. Then the probability that **Rho** outputs y_0 is $\Omega(p^2 N)$.*

Proof. Since the probability of obtaining y_0 as the output is non-decreasing as a function of p, there is no loss of generality in assuming $p = cN^{-1/2}$ for a small c. In such a case, a lower bound on the probability of **Rho** producing y_0 is the probability that in the first $\sqrt{N}/2$ steps of the sequence $(x, f(x), f^2(x), \ldots)$, each value $y' \neq y_0$ appears at most once, while y_0 appears twice. Formally, let $L' = (x, f(x), \ldots, f^t(x))$, where $t = \min(\mu+\lambda, \sqrt{N}/2)$. Denote by $E_{y'}$ the event: "Each $y' \neq y_0$ appears at most once in L'", and by E_{y_0} the event: "y_0 appears twice in L'". Then

$$\Pr[\mathit{Output}(\mathbf{Rho}) = y_0] \geq \Pr[E_{y'} \wedge E_{y_0}] = \Pr[E_{y'}]\Pr[E_{y_0}|E_{y'}].$$

As we show in Appendix A, we have $\Pr[E_{y'}] \geq e^{-1/4} \approx 0.78$ and

$$\Pr[E_{y_0}|E_{y'}] = \Pr[X \geq 2 | X \sim \mathit{Poi}(|L'|p)] \geq \Pr[X \geq 2 | X \sim \mathit{Poi}(\sqrt{N}p/2)].$$

[3] Such a repetition must occur due to the fact that $f : [N] \to [N]$.

Finally, for any small λ we have

$$\Pr[X \geq 2 | X \sim Poi(\lambda)] = 1 - e^{-\lambda}(1+\lambda) \approx 1 - (1-\lambda)(1+\lambda) = \lambda^2,$$

and thus, combination of the above inequalities yields

$$\Pr[Output(\mathbf{Rho}) = y_0] \geq 0.78(\sqrt{N}p/2)^2 = 0.19p^2N,$$

as asserted. □

This yields the **RepeatedRho** algorithm – an $O(p^{-3}N^{-1})$ algorithm for detecting the mode: run the **Rho** algorithm $O(1/p^2N)$ times, and check each output point in $O(1/p)$ time using the mode verification algorithm. With a constant probability, y_0 is suggested by at least one of the **Rho** invocations and is thus verified. As $p^{-3}N^{-1} < p^{-2}$ for all $p > N^{-1}$, this algorithm outperforms the sampling algorithm (Algorithm 2) whose running time is $O(p^{-2})$ for all p. See Fig. 3 for comparison of the algorithms for different values of p.

The analysis above implicitly assumes that all the invocations of **Rho** are independent. However, this is clearly not the case if we apply **Rho** to the same function f, while changing only the starting point x in each invocation. Indeed, since $p < 1/\sqrt{N}$, y_0 is not expected to be on a cycle of f, and thus no matter how many times we run the **Rho** algorithm using the same f but different starting points, we will never encounter y_0 as a cycle entry point.

In order to make the invocations of **Rho** essentially independent, we introduce the notion of *flavors* of f, like the flavors used in Hellman's classical time-memory tradeoff attack [3]. More specifically, we define the v's flavor of f as the function $f_v(x) = f(x+v)$ where the addition is computed modulo N. The different flavors of f share some local properties (e.g., they preserve the number of preimages of each y, and thus y_0 remains the mode of the function), but have different global properties (e.g., when iterated, their graphs have a completely different partition into trees and cycles). In particular, it is common to consider the various flavors of f as unrelated functions, even though this is not formally justified. We define the output of the v's invocation of **Rho** as the entry point into the cycle defined by f_v when we start from point v, and run the **RepeatedRho** algorithm by calling **Rho** multiple times with different randomly chosen flavors.

5 The 2Rho Algorithm

In this section we introduce the **2Rho** algorithm, and show that running a **Rho** algorithm over the results of a **Rho** algorithm outperforms all the previously suggested algorithms.

The main idea behind the new algorithm is that a single application of **Rho** can be viewed as a *bootstrapping* step that amplifies the probability of y_0 to be sampled. Indeed, by Proposition 1, The probability that **Rho** with a randomly chosen flavor will output y_0 is $\Omega(p^2N)$, and as long as $p \gg N^{-1}$, this is significantly larger than the probability p that y_0 will be sampled by a single invocation

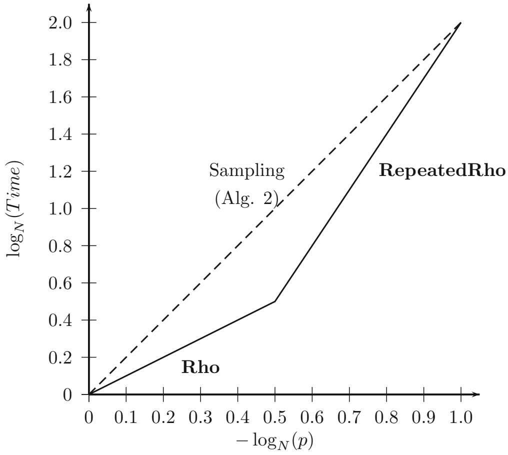
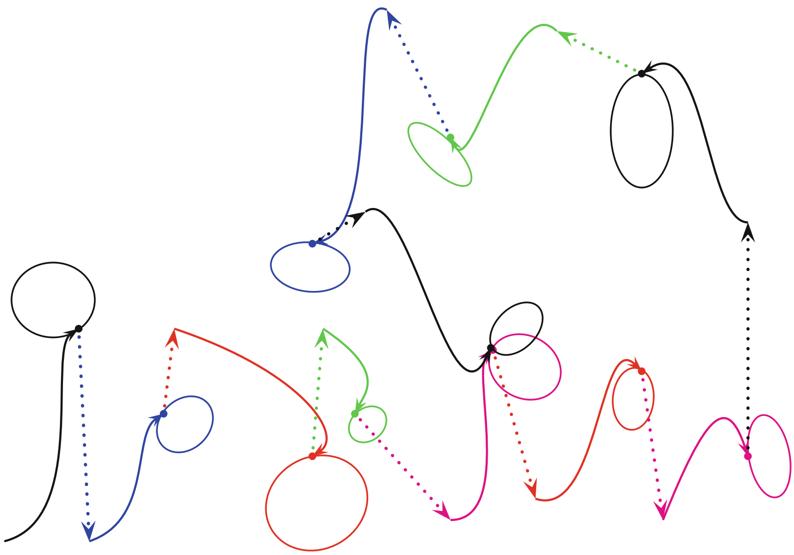


Fig. 3. Comparing the random sampling, **Rho**, and **RepeatedRho** algorithms

of f. Note that by symmetry the probabilities of all the other values of y to be returned by **Rho** with a random flavor remain uniformly low. We are thus facing exactly the same needle finding problem but with a magnified probability peak at exactly the same location y_0. In particular, if this new probability peak exceeds $N^{-0.5}$, we can find it by using a simple **Rho** algorithm. On the other hand, a single evaluation of **Rho** is now more time consuming than a single evaluation of f, and thus the bootstrapping yields a tradeoff between the total number of operations and the cost of each operation, so the parameters should be chosen properly in order to reduce the total complexity.

Our goal now is to formally define the new inner function $g(x)$ which will be used by the outer **Rho** algorithm. This g maps a flavor v to the cycle's entry point defined by running the **Rho** algorithm on the v's flavor of f (i.e., on f_v), starting with the initial value v. When we iterate g, we jump from a flavor to a cycle entry point, and then use the identity of the cycle's entry point to define the next flavor and starting point. This creates a large Rho structure over small Rho structures, as depicted in Fig. 4 in which the different colors indicate the different flavors of f we use in the algorithm. Each dotted line represents the first step we take when we switch to a new flavor, and is used only to visually separate the Rhos so that the figure will look more comprehensible. Note that the collision in the big cycle happens when we encounter the same cycle entry point a second time, but this does not imply that the two colliding small Rho's or their starting points are the same, since they typically use different flavors; it is only in the second and third times we meet the same cycle entry point that their corresponding Rho structures also becomes identical, and from then on we go through the same Rhos over and over.



Different colors represent different flavors of f.

Fig. 4. The **2Rho** algorithm (Color figure online)

We now turn our attention to a specific range of probabilities p for which the **2Rho** algorithm (that runs an outer **Rho** algorithm over an inner **Rho** algorithm) offers a significant gain over the previously described algorithms.

5.1 Analysis of 2Rho in the Range $N^{-3/4} \ll p \leq N^{-1/2}$

Assume that $N^{-3/4} \ll p \leq N^{-1/2}$, and construct the function g as described above. Defining $p' = \Pr[g(x) = y_0]$, we have shown that $p' = \Omega(p^2 N) \gg N^{-1/2}$, and thus the mode of g can be found optimally in $O(1/p')$ evaluations of g using the **2Rho** algorithm.

In order to find the total complexity of **2Rho**, we have to compute the complexity of each evaluation of g, i.e., of an evaluation of **Rho** algorithm.

Note that for $c > 1$, the probability that all values $x, f(x), f^2(x), \ldots, f^{c\sqrt{N}}(x)$ are different is at most

$$\frac{(N-1)(N-2)\cdot \ldots \cdot (N-c\sqrt{N})}{(N-1)^{c\sqrt{N}}} \leq \frac{(N-1)^{\sqrt{N}}(N-\sqrt{N}-1)^{(c-1)\sqrt{N}}}{(N-1)^{c\sqrt{N}}}$$
$$\leq \left(\frac{N-\sqrt{N}}{N}\right)^{(c-1)\sqrt{N}} \approx e^{-(c-1)}.$$

Hence, with an overwhelming probability, **Rho** finds a cycle in $O(\sqrt{N})$ operations. In order to avoid the rare cases where such algorithms take more time, we can slightly modify any **Rho** algorithm by stopping it after a predetermined number of f evaluations (e.g., $10\sqrt{N}$), in which case $g(x) = \mathbf{Rho}(f, x+1)$.[4] In any case, the expected time complexity of an evaluation of g is $O(\sqrt{N})$ evaluations of f.

Therefore, the time complexity of **2Rho** is $O(\frac{1}{p'} \cdot \sqrt{N}) = O(p^{-2}N^{-1/2})$ operations. This is significantly faster than the Sampling algorithm, and also significantly faster than **RepeatedRho**, since $p^{-2}N^{-1/2} < p^{-3}N^{-1}$ for all $p < N^{-1/2}$.

Just like the **Rho** algorithm, the nested **2Rho** algorithm can be repeated when $p < N^{-3/4}$, to yield an algorithm for any p. Indeed, repeating **2Rho** until the mode is found (and verified by the verification algorithm), takes $O(p'^{-3}N^{-1}) = O((p^2N)^{-3}N^{-1}) = O(p^{-6}N^{-4})$ evaluations of g, or $O(p^{-6}N^{-3.5})$ evaluations of f. Hence, **Repeated2Rho** is better than the **RepeatedRho** algorithm for $p > N^{-5/6}$ and is worse for $p < N^{-5/6}$.

Table 1 describes our experimental verification of the **2Rho** algorithm for different values of p in the range $N^{-0.79} \le p \le N^{-0.5}$. We used a relatively small $N = 2^{28}$ (which makes the transition at $p = N^{-0.75}$ more gradual than we expect it to be for larger N), and repeated each experiment 100 times with different random functions f.

Table 1. Success rate of **2Rho** for $N = 2^{28}$ over 100 experiments

$p = \Pr[y_0]$		Success
Value	$\log_N(p)$	Rate
2^{-14}	-0.5	100 %
2^{-15}	-0.54	100 %
2^{-16}	-0.57	100 %
2^{-17}	-0.61	97 %
2^{-18}	-0.64	91 %
2^{-19}	-0.68	71 %
2^{-20}	-0.71	32 %
2^{-21}	-0.75	8 %
2^{-22}	-0.79	0 %

6 Deeper Nesting of the Rho Algorithm

We now show how one can nest ***i*Rho** to obtain **(*i*+1)Rho**. We analyze the resulting complexities, and show that while for a small i, it yields better results,

[4] Of course, with a negligible probability, we may need to continue and define $g(x) = \mathbf{Rho}(f, x+2)$, and so forth.

as i becomes larger it loses to simpler algorithms. In particular, it is advantageous to nest the **NestedRho** algorithm up to four times, but not a fifth time.

The 3Rho Algorithm for $N^{-7/8} \ll p \leq N^{-3/4}$. Assume that $N^{-7/8} \ll p \leq N^{-3/4}$, and define a new function $h(x)$ which maps an input flavor x into the cycle's entry point defined by the **2Rho** algorithm. As in the analysis of **2Rho** above, we define $p'' = \Pr[h(x) = y_0]$, and can show that $p'' = \Omega(p'^2 N) = \Omega(p^4 N^2 N) \gg N^{-1/2}$. Hence, the mode of h can be found optimally in $O(1/p'')$ evaluations of h using **Rho**. Since each evaluation of h requires $O(\sqrt{N})$ evaluations of g, and since each evaluation of g requires $O(\sqrt{N})$ evaluations of f, the overall complexity of the algorithm is $O(p^{-4}N^{-3}N) = O(p^{-4}N^{-2})$ evaluations of f. We call this algorithm **3Rho**, as it essentially performs yet another nesting layer of **2Rho**.

Algorithm 3. $(i+1)$**Rho** Algorithm for the Function $f(\cdot)$ (Based on ***i*Rho**)

Input: a random input $x \in [N]$.
Set $z \leftarrow$ ***i*Rho**(f_x, x). ▷ Note that in the recursion, the flavors of f add up.
while Repeated value of z is not encountered **do**
    Set $z \leftarrow$ ***i*Rho**(f_z, z).
end while
Identify the repeated z value.[a]
return z.

---

[a]The identification can be done using Floyd's algorithm [7], or any of its variants.

The complexity of the **3Rho** algorithm is always better than that of **RepeatedRho** and is better than the $O(p^{-6}N^{-3.5})$ complexity of **Repeated2Rho** for all $p < N^{-3/4}$.

As in the case of **2Rho**, the **3Rho** algorithm can also be repeated as **Repeated3Rho** with mode verification to yield an algorithm for any p. The resulting complexity is $O(p''^{-3}N^{-1}) = O((p^4N^3)^{-3}N^{-1}) = O(p^{-12}N^{-10})$ evaluations of h, or $O(p^{-12}N^{-9})$ evaluations of f. This algorithm is better than the **RepeatedRho** for $p > N^{-8/9}$ and is worse for $p < N^{-8/9}$. However, it turns out that for $N^{-9/10} \ll p \leq N^{-7/8}$, we can do better by nesting **3Rho** yet another time.

The 4Rho Algorithm for $N^{-9/10} \ll p \leq N^{-7/8}$. Assume that $N^{-15/16} \ll p \leq N^{-7/8}$, and define a new mapping $\ell(x)$ which maps a flavor x into the cycle's entry point found by the **3Rho** algorithm. As in the above case of **3Rho**, we have $p''' = \Pr[\ell(x) = y_0] = \Omega(p''^2N) = \Omega(p^8N^6N) \gg N^{-1/2}$. Hence, the mode of ℓ can be found optimally in $O(1/p''')$ evaluations of ℓ using **Rho**.

Since each evaluation of ℓ requires $O(N^{1.5})$ evaluations of f, the overall complexity of the algorithm is $O(p^{-8}N^{-7}N^{1.5}) = O(p^{-8}N^{-5.5})$ evaluations of f. We call this algorithm **4Rho**, as it performs a four-layer nesting of **Rho**.

Unlike the previous algorithms, **4Rho** is not better than all previous algorithms in the whole range $N^{-15/16} \ll p \leq N^{-7/8}$. Indeed, as $p \rightarrow N^{-15/16}$, the

Table 2. Summary of the best complexities of algorithms for detecting the mode

Probability range	Complexity formula	Complexity range	Algorithm
$p \geq N^{-0.5}$	$T = p^{-1}$	$T \leq N^{0.5}$	**Rho**
$N^{-0.75} \leq p \leq N^{-0.5}$	$T = p^{-2}N^{-0.5}$	$N^{0.5} \leq T \leq N$	**2Rho**
$N^{-0.875} \leq p \leq N^{-0.75}$	$T = p^{-4}N^{-2}$	$N \leq T \leq N^{1.5}$	**3Rho**
$N^{-0.9} \leq p \leq N^{-0.875}$	$T = p^{-8}N^{-5.5}$	$N^{1.5} \leq T \leq N^{1.7}$	**4Rho**
$N^{-1} \leq p \leq N^{-0.9}$	$T = p^{-3}N^{-1}$	$N^{1.7} \leq T \leq N^{2}$	**RepeatedRho**

complexity of **4Rho** approaches N^2, which is higher than even the straightforward Sampling Algorithm. In particular, **4Rho** is faster than **RepeatedRho** only as long as $p > N^{-0.9}$, which explains why the complexity curve reduces its slope at the top right corner of Fig. 5.

We note that the natural extension to **5Rho** is clearly inferior for any p since the complexity of each step of the outer **Rho** requires N^2 steps, which is already higher than the overall complexity of the Sampling algorithm.

The complexities of the best algorithms we were able to achieve (as a function of p) are presented in a mathematical form in Table 2 and in graphical form in Fig. 5.

7 Time-Memory Tradeoffs

In this section, we revisit the basic problem of detecting the mode, but assume that we have $O(M)$ memory cells available. Our goal is to detect the mode as efficiently as possible, where the complexity is formulated as a function of the parameters N, p and M.

Before starting, we note that we only deal with the case of $p < N^{-1/2}$, as we already have an optimal[5] memoryless algorithm for the case of $p \geq N^{-1/2}$ (as shown in Sect. 4).

We begin by describing the basic parallel collision search algorithm of [10]. We then describe a sequence of algorithms that extend the ***i*Rho** memoryless algorithms using parallel collision search.

7.1 Parallel Collision Search

The parallel collision search (**PCS**) algorithm presented by van Oorschot and Wiener [10] is a memory-efficient algorithm for finding multiple collisions at low amortized cost per collision in a function f that maps $[N]$ to $[N]$. Since its introduction, the algorithm has been extensively used in cryptanalysis (e.g., in [4–6,9]). Given M memory cells, the algorithm builds a structure of

[5] Given additional memory and/or CPUs allows parallelizing **Rho** algorithms. At the same time, the total computational complexity (which is the focus of this paper) remains the same, or (in some cases) may become worse.

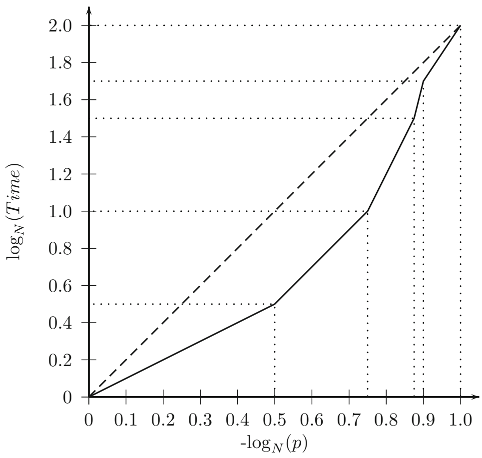


Fig. 5. Complexities of our best memoryless algorithms as a function of p

M chains which is similar to the one built in Hellman's time-memory tradeoff algorithm [3].

A chain in the structure starts at an arbitrary point x, and is evaluated by repeated applications of f (namely, $f^i(x) = f(f^{i-1}(x))$). The chains are terminated after about $\sqrt{N/M}$ evaluations of f, thus the structure contains a total of about $M \cdot \sqrt{N/M} = \sqrt{NM}$ points. Moreover, as $\sqrt{N/M} \cdot \sqrt{NM} = N$, according to the birthday paradox, each chain is expected to collide with another chain in the structure, and hence the chain structure contains $O(M)$ collisions. In order to find the $O(M)$ collisions efficiently, we define a set of *distinguished points* and terminate each chain once it reaches such a point. In our case, we define a set of $\sqrt{NM}$ distinguished points (e.g., the points whose $(\log_2(N) + \log_2(M))/2$ least significant bits are zero), and hence the expected chain size is $N/\sqrt{NM} = \sqrt{N/M}$ as required. The actual $O(M)$ collision points are recovered by sorting the M termination points of the chains (which are distinguished points), and restarting the chain computation for each colliding pair of chains. For sake of completeness we give in Appendix B the pseudo code for **PCS** (Algorithm 6). In total, the algorithm finds $O(M)$ collisions in $\sqrt{NM}$ time using $O(M)$ memory.

7.2 Mode Verification with Memory

The basic memoryless mode verification algorithm (Algorithm 1) can be extended to exploit memory, by checking multiple targets simultaneously. Namely, given M candidate y_i's, it is possible to check all of them at the same time for the cost of $O(1/p)$ queries to f, as suggested by Algorithm 4.

Using Algorithm 4, we can immediately improve the sampling algorithm (Algorithm 2). Instead of checking only one value at each call to the verification algorithm, we can now check M such values for the same complexity. Hence, Algorithm 5 picks each time M random values of y_i by random sampling, and calls Algorithm 4 to test which of them (if at all) is indeed y_0.

Algorithm 4. Mode Verification: Determining Whether y_0 is one of $y_1, y_2, \ldots y_M$

```
Initialize an array of counters ctr[i] ← 0 for 1 ≤ i ≤ M.
for j = 1 to c/p do
    Pick at random x ∈ [N], and compute y' = f(x).
    if y' = y_i for 1 ≤ i ≤ M then
        Increment ctr[i].
    end if
end for
for i = 1 to N do
    if ctr[i] ≥ t then
        print y_i is y_0.
    end if
end for
```

Algorithm 5. Finding y_0 by Sampling (with Memory):

```
while y_0 was not found do
    for i = 1 to M do
        Pick x_i ∈ [N] at random.
        Compute y_i = f(x_i).
    end for
    Call Algorithm 4 to check y_1, y_2, ..., y_M.
end while
```

The probability that a single call to Algorithm 4 (testing M images) succeeds is about Mp (assuming[6] $M \leq p^{-1}$), and therefore we expect $O(M^{-1}p^{-1})$ calls to Algorithm 4. Each such call takes $O(p^{-1})$ evaluations of f, and hence the total time complexity of the algorithm is $O(M^{-1}p^{-2})$. Note that for $M = 1$ this algorithm reduces to Algorithm 2.

7.3 Mode Detection with Parallel Collision Search

This algorithm runs **PCS** with M chains and checks the $O(M)$ collision points found by running Algorithm 4. This process is repeated until it finds y_0, where each repetition is performed with a different flavor of f.

Since the M chains cover about $\sqrt{NM}$ distinct points, the probability that two distinct preimages of the mode y_0 (which are not expected to be distinguished points) are covered by the structure is about $(\sqrt{NM} \cdot p)^2 = NM \cdot p^2$ (assuming $\sqrt{NM} \cdot p < 1$, i.e., $p < (NM)^{-0.5}$). In this case, the algorithm will successfully recover the mode y_0 using the mode verification algorithm. Therefore, the algorithm is expected to execute **PCS** (and mode verification) about $N^{-1}M^{-1} \cdot p^{-2}$ times, where each execution requires $O(p^{-1})$ time (assuming $p < (NM)^{-1/2}$, mode verification dominates **PCS** in terms of time complexity). In total, the time complexity of the algorithm is $O(M^{-1}N^{-1} \cdot p^{-3})$. Note that for $M = 1$ we obtain **RepeatedRho**.

[6] We note that when $M > p^{-1}$, it is sufficient to fill $O(p^{-1})$ memory cells.

The formula above is only valid for $p < (NM)^{-0.5}$ or $M < p^{-2}N^{-1}$. Otherwise, we can utilize only $M = p^{-2}N^{-1}$ memory and obtain the essentially optimal time complexity of $O(p^{-1})$.

7.4 Mode Detection with Parallel Collision Search Over 2Rho

We now assume that $M < p^{-2}N^{-1}$ (otherwise, we use the previous **PCS** algorithm to detect the mode with optimal complexity) and extend the **2Rho** algorithm using **PCS**. This is done by defining a chain structure, computed by iterating the function g (as defined in Sect. 5) whose execution is computed by iterating a particular flavor of f until a collision point is found. Each chain starts with an arbitrary input to g (which defines a flavor of f) and is terminated at a distinguished point of g. Namely, the distinguished points are defined on the outputs of g (which are the collision points in f). Once again, we use Algorithm 4 to test the $O(M)$ collisions of g.

As calculated in Sect. 5 the probability that the mode y_0 will be the collision point in a single run of **Rho** (an iteration of g) is $p' = p^2N$. Since the M chains of g cover about $\sqrt{NM}$ distinct collision points, the probability that two distinct preimages of the mode y_0 in g (which are not expected to be distinguished points) will be covered by the structure is about $(\sqrt{NM} \cdot p')^2 = NM \cdot p'^2 = NM \cdot p^4N^2 = M \cdot N^3p^4$ (assuming $\sqrt{NM} \cdot p' < 1$ or $p^2N \cdot (NM)^{1/2} < 1$, namely $p^2N^{3/2}M^{1/2} < 1$). As a result, we repeat the **PCS** algorithm (and the mode verification algorithm) $M^{-1} \cdot N^{-3}p^{-4}$ times (using distinct flavors of g). The **PCS** algorithm requires $(NM)^{1/2}$ invocations of g, each requiring $N^{1/2}$ time, namely, $N \cdot M^{1/2}$ time in total which dominates the complexity of the mode verification. Overall, the time complexity of the algorithm is $M^{-1} \cdot N^{-3}p^{-4} \cdot N \cdot M^{1/2} = M^{-1/2} \cdot N^{-2}p^{-4}$.

The formula above is only valid given that $p^2N^{3/2}M^{1/2} < 1$ or $M < p^{-4}N^{-3}$. Otherwise, we can utilize only $M = p^{-4}N^{-3}$ (assuming[7] $p^{-4}N^{-3} \geq 1$) memory and obtain time complexity of $M^{-1/2} \cdot N^{-2}p^{-4} = p^2N^{3/2} \cdot N^{-2}p^{-4} = p^{-2}N^{-1/2}$.

We now notice that it is possible to obtain more generic formulas that can be reused later. Essentially, the analysis of the algorithm depends on three parameters, as follows. The probability that the mode y_0 will be the collision point in a single run of **Rho** (an iteration of g) is $p' = p^2N$, which we denote as $p^{x_1}N^{x_2}$ for $x_1 = 2, x_2 = 1$ in our case. In addition, each invocation of g requires $N^{1/2}$ time, which we denote as N^{x_3} for $x_3 = 1/2$ in this case. Based on these parameters, we can redo the analysis above symbolically and obtain that the time complexity of the algorithm is $M^{-1/2} \cdot N^{-2x_2-1/2+x_3}p^{-2x_1}$.

This formula is only valid given that $M < p^{-2x_1}N^{-2x_2-1}$. Otherwise, we can utilize only $M = p^{-2x_1}N^{-2x_2-1}$ (assuming $p^{-2x_1}N^{-2x_2-1} \geq 1$) memory and obtain time complexity of $p^{-x_1}N^{-x_2+x_3}$.

[7] When $p^{-4}N^{-3} < 1$, the algorithm is not applicable in its current form.

7.5 Mode Detection with Parallel Collision Search over 3Rho

We continue to analyze the sequence of algorithms that extend **3Rho** using **PCS**. The idea is essentially the same as in the extension of **2Rho**, where the difference is the function over which **PCS** is performed.

Here, **PCS** is executed over the function h (as defined in Sect. 6) while calling Algorithm 4 to test the $O(M)$ collisions points of h.

As calculated in Sect. 6 the probability that the mode y_0 will be the collision point in a single run of g is $p'' = p^4N^3$, which we denote as $p^{x_1}N^{x_2}$ for $x_1 = 4, x_2 = 3$. In this case, each invocation of h requires N time, or N^{x_3} for $x_3 = 1$.

We now reuse the formulas obtained in Sect. 7.4 and consider our specific parameters $x_1 = 4, x_2 = 3, x_3 = 1$ for the case $M < p^{-2x_1}N^{-2x_2-1}$, or $M < p^{-8}N^{-7}$ assuming $p^{-8}N^{-7} \geq 1$ or $p \leq N^{-7/8}$. This gives time complexity of $M^{-1/2} \cdot N^{-2x_2-1/2+x_3}p^{-2x_1}$ or $M^{-1/2} \cdot N^{-5.5}p^{-8}$. Note that for $M = 1$ we obtain Algorithm **4Rho**.

For $M > p^{-8}N^{-7}$, we obtain time complexity of $p^{-x_1}N^{-x_2+x_3} = p^{-4}N^{-2}$. See Fig. 6 for comparison of different algorithms given $M = N^{1/4}$ memory.

Mode Detection with Parallel Collision Search over 2Rho. The extension of **PCS** over **4Rho** does not make sense since the function ℓ (defined in Sect. 6) used for **4Rho** is never iterated more than $N^{0.5}$ times in our algorithms. Hence, all its iterations can be covered by a single chain of **4Rho** and there is no benefit in using memory in this case.

7.6 Discussion

It is not intuitive to compare the algorithms described above, as their complexities are functions of both p and M. In order to get some intuition regarding their performance, we fix $M = N^{1/4}$ and summarize the complexity of the best algorithms for this case as a function of the single parameter p in Table 3. It is evident from the table that there is a range of p values for which we do not know how to efficiently exploit the memory. For example, consider $p = N^{-3/4}$, where our best algorithm is **PCS** over **2Rho**. However, it is actually a degenerate variant of **PCS** with $M = 1$ that coincides with the **2Rho** algorithm of Sect. 6.

8 Finding Multiple Peaks

We consider a generalization of our basic problem to the case that f is uniformly distributed except for k peaks. The peaks are denoted by $y_0, y_1, \ldots, y_{k-1}$, their associated probabilities are denoted by $p_0, p_1, \ldots, p_{k-1}$, and our goal is to find all of them.[8]

[8] In [1] a related problem is studied: Let f be a hash function. Assume that its range is smaller than its domain and that it is not balanced (i.e., not all outputs appear with the same probability). This work studies the effect of this irregularity on the complexity of the birthday collision search. In contrast, our work studies the algorithmic aspects of finding the collision (in a memory-efficient manner).

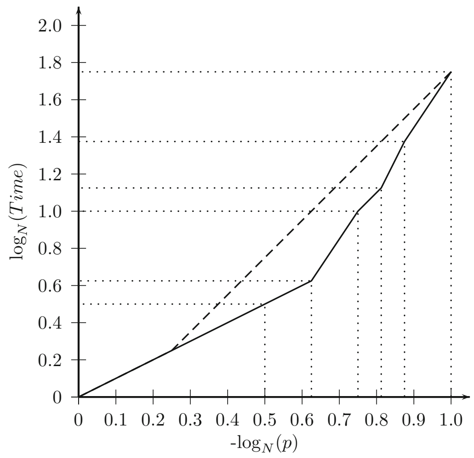


Fig. 6. Complexities of our best algorithms as a function of p given $M = N^{1/4}$ memory

Table 3. Summary of the best complexities of algorithms for detecting the mode with $M = N^{1/4}$

Probability range	Complexity formula	Complexity range	Algorithm
$p \geq N^{-0.5}$	$T = p^{-1}$	$T \leq N^{0.5}$	**Rho**
$N^{-5/8} \leq p \leq N^{-0.5}$	$T = p^{-1}$	$N^{0.5} \leq T \leq N^{5/8}$	**PCS**
$N^{-3/4} \leq p \leq N^{-5/8}$	$T = p^{-3}N^{-5/4}$	$N^{5/8} \leq T \leq N$	**PCS**
$N^{-13/16} \leq p \leq N^{-3/4}$	$T = p^{-2}N^{-1/2}$	$N \leq T \leq N^{9/8}$	**PCS** over **2Rho**
$N^{-7/8} \leq p \leq N^{-13/16}$	$T = p^{-4}N^{-17/8}$	$N^{9/8} \leq T \leq N^{11/8}$	**PCS** over **2Rho**
$N^{-1} < p \leq N^{-7/8}$	$T = p^{-3}N^{-5/4}$	$N^{11/8} \leq T \leq N^{7/4}$	**PCS**

The simplest case is one in which there are two peaks of equal height $p_0 = p_1$. By running the **NestedRho** algorithm several times with different flavors of f, we expect to find each one of y_0 and y_1 about half the time, and thus there is no need to modify anything.

The next case to consider is one in which there are only two peaks but $p_0 > p_1$. Due to the high power of p in our formulas, even moderate differences in the peak probabilities are amplified by the **NestedRho** algorithm to huge differences in the probability of finding the two peaks. For example, if p_0 is a thousand times bigger than p_1, and we run the algorithm multiple times, then we expect to find y_1 only in one in a million runs when we use **1Rho**, and only in one in a trillion runs when we use **2Rho**. Clearly, we have to reduce the attractiveness of y_0 before we have a realistic chance of noticing y_1.

The simplest way to neutralize the first peak we find (which is likely to be y_0), is to scatter its preimages so that they will point to different targets. Consider a modified function f' which is defined as f for any x for which $f(x) \neq y_0$, and as $f(x) + x$ for any x for which $f(x) = y_0$. In f', y_0 is no longer a peak, but y_1

remains at its original height. By applying **NestedRho** to f', we will find y_1 with high probability.

This can be easily generalized to a sequence of k peaks, provided that we have at least $O(k)$ memory to store all the peaks. Our algorithm is likely to discover them sequentially in decreasing order of probability, and we can decide to stop at any point when we run out of space or time.

The most general case is one in which we have a non-uniform distribution with no sharp peaks. In this case the output of the **NestedRho** algorithm has a preference to pick y values with higher probabilities, but may pick a lower probability y if there are many such values. In fact, the probability that our algorithm will pick a particular y is proportional to some power of its original probability, which depends on which nesting level we use (the detailed analysis is left for future work).

9 Conclusions and Open Problems

In this paper we introduced the generic problem of finding needles in haystacks, developed several novel techniques for its solution, and demonstrated the surprising complexity of its complexity function. Many problems remain open, such as:

1. Find non-trivial lower bounds on the time complexity of the problem.
2. Find better ways to exploit the available memory, beyond using **PCS**.
3. Extend the model to deal with other types of needles.
4. Find additional applications of the new **NestedRho** technique.

Acknowledgements. The authors thank Masha Gutman for her implementation of the experiments reported in Table 1.

A Detailed Complexity Analysis of the Rho Approach for $p > 1/\sqrt{N}$

Consider the sequence $x, f(x), f^2(x), \ldots$. If we limit the length of the sequence by $4/p$ "random" steps, then with high enough probability, we expect to encounter y_0 twice. On the other hand, the probability that a "random" value is encountered twice is low, since $4/p$ is significantly smaller than the "birthday bound" $\sqrt{N}$. Hence, y_0 is expected to be the first repeated point, and hence, the output.

Formally, let A be a "truncated" **Rho** algorithm:[9]

1. Choose $x \in [N]$ uniformly at random.
2. Run **Rho** algorithm that computes the chain $x, f(x), \ldots, f^{4/p}(x)$ (or shorter chain if a collision is found before).

[9] The reader may think of the algorithm as Floyd's one, but the same analysis holds for any "reasonable" memoryless detection algorithm.

(a) If a collision is detected, denote its value by y, and run the verification algorithm on y.
(b) If no collision is detected, output "FAIL".

Proposition 2. *Assume that Algorithm A is run in the case* $p \geq 16/\sqrt{N}$*. Then* $\Pr[Output(A) = y_0] \geq 0.69$*.*

Proof. Throughout the proof we consider the sequence $L = (x, f(x), \ldots, f^{\mu+\lambda}(x))$ of values encountered by the algorithm until the first repetition (inclusive) or until the process terminates (if a repetition was not encountered). By the definition of f, this sequence is distributed like an *independent* sampling of $\mu + \lambda$ elements of the distribution of $range(f)$. Note that if a meeting point is detected at step t, this implies that the sequence $x, f(x), \ldots, f^{2t}(x)$ contains a repetition, and thus, $|L| \leq 8/p \leq \sqrt{N}/2$.

First, we bound from above $\Pr[\exists y' \neq y_0 : Output(A) = y']$, i.e., the probability that some $y' \neq y_0$ appears twice in L. Consider all values non-equal to y_0 that appear in L. Since $|L| \leq \sqrt{N}/2$, the probability that they are mutually different is at least

$$\frac{(N-1)(N-2)\cdots(N-|L|)}{(N-1)^{|L|}} \geq \left(\frac{N-|L|}{N}\right)^{|L|} \geq \left(\frac{N-\sqrt{N}/2}{N}\right)^{\sqrt{N}/2} \approx e^{-1/4}.$$

Hence, $\Pr[\exists y' \neq y_0 : Output(A) = y'] \leq 1 - e^{-1/4} \approx 0.22$.

Second, we bound from above $\Pr[Output(A) = FAIL]$, i.e., the probability that neither y_0 nor any other value appears twice in L. Note that in such a case, $|L| = 4/p$ since no repetition is encountered. By the definition of f, for any k, $\Pr[f^k(x) = y_0] = p$. Hence, the number of occurrences of y_0 in L is distributed like a $Bin(|L|, p) = Bin(4/p, p)$ random variable, that can be approximated by a $Poi(|L|p) = Poi(4)$ random variable. Hence,

$$\Pr[Output(A) = FAIL] \leq \Pr[Poi(4) \leq 1] = e^{-4} + 4e^{-4} \approx 0.09.$$

Combining the two bounds, we obtain

$$\begin{aligned}\Pr[Output(A) = y_0] &= 1 - \Pr[\exists y' \neq y_0 : Output(A) = y'] - \Pr[Output(A) = FAIL] \\ &\geq 1 - 0.22 - 0.09 = 0.69,\end{aligned}$$

as asserted. $\square$

B The Parallel Collision Search Algorithm

Algorithm 6. Parallel Collision Search

```
Initialize an empty table of M entries.
for i = 1 to M do
    Pick at random a point x_i ∈ [N].
    Set tmp ← x_i, len ← 0.
    while f(tmp) is not a distinguished point do
        tmp ← f(tmp).
        Increment len.
    end while
    tmp ← f(tmp).
    Increment len.
    Store in the table the pair (tmp, x_i, len).
end for
for All collisions ((p_i, x_i, len_i), (p_j, x_j, len_j)) s.t. p_i = p_j do
    Set tmp_1 ← x_i, tmp_2 ← x_j.
    if len_1 > len_2 then
        for i = 1 to len_1 − len_2 do
            tmp_1 ← f(tmp_1)
        end for
    end if
    if len_2 > len_1 then
        for i = 1 to len_2 − len_1 do
            tmp_2 ← f(tmp_2)
        end for
    end if
    while f(tmp_1) ≠ f(tmp_2) do
        tmp_1 ← f(tmp_1), tmp_2 ← f(tmp_2)
    end while
    print tmp_1, tmp_2.
end for
```

References

1. Bellare, M., Kohno, T.: Hash function balance and its impact on birthday attacks. In: Cachin, C., Camenisch, J.L. (eds.) EUROCRYPT 2004. LNCS, vol. 3027, pp. 401–418. Springer, Heidelberg (2004)
2. Richard, R.P.: An improved monte carlo factorization algorithm. BIT Numer. Math. **20**(2), 176–184 (1980). doi:10.1007/BF01933190
3. Hellman, M.E.: A cryptanalytic time-memory trade-off. IEEE Trans. Inf. Theory **26**(4), 401–406 (1980)
4. Joux, A., Lucks, S.: Improved generic algorithms for 3-collisions. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 347–363. Springer, Heidelberg (2009)

5. Kelsey, J., Kohno, T.: Herding hash functions and the nostradamus attack. In: Vaudenay, S. (ed.) EUROCRYPT 2006. LNCS, vol. 4004, pp. 183–200. Springer, Heidelberg (2006)
6. Kelsey, J., Schneier, B.: Second preimages on n-bit hash functions for much less than 2^n work. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 474–490. Springer, Heidelberg (2005)
7. Knuth, D.E.: The Art of Computer Programming: Seminumerical Algorithms, vol. II. Addison-Wesley, Reading (1969)
8. Nivasch, G.: Cycle detection using a stack. Inf. Process. Lett. **90**(3), 135–140 (2004)
9. Stevens, M., Sotirov, A., Appelbaum, J., Lenstra, A., Molnar, D., Osvik, D.A., de Weger, B.: Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. In: Halevi, S. (ed.) CRYPTO 2009. LNCS, vol. 5677, pp. 55–69. Springer, Heidelberg (2009)
10. van Oorschot, P.C., Wiener, M.J.: Parallel collision search with cryptanalytic applications. J. Cryptol. **12**(1), 1–28 (1999)