Parallel Collision Search with Cryptanalytic Applications

Paul C. van Oorschot and *Michael J. Wiener*

Nortel, P.O. Box 3511 Station C, Ottawa, Ontario, K1Y 4H7, Canada

1996 September 23

Abstract. A simple new technique of parallelizing methods for solving search problems which seek collisions in pseudo-random walks is presented. This technique can be adapted to a wide range of cryptanalytic problems which can be reduced to finding collisions. General constructions are given showing how to adapt the technique to finding discrete logarithms in cyclic groups, finding meaningful collisions in hash functions, and performing meet-in-the-middle attacks such as a known-plaintext attack on double encryption. The new technique greatly extends the reach of practical attacks, providing the most cost-effective means known to date for defeating: the small subgroup used in certain schemes based on discrete logarithms such as Schnorr, DSA, and elliptic curve cryptosystems; hash functions such as MD5, RIPEMD, SHA-1, MDC-2, and MDC-4; and double encryption and three-key triple encryption. The practical significance of the technique is illustrated by giving the design for three $10 million custom machines which could be built with current technology: one finds elliptic curve logarithms in $GF(2^{155})$ thereby defeating a proposed elliptic curve cryptosystem in expected time 32 days, the second finds MD5 collisions in expected time 21 days, and the last recovers a double-DES key from 2 known plaintexts in expected time 4 years, which is four orders of magnitude faster than the conventional meet-in-the-middle attack on double-DES. Based on this attack, double-DES offers only 17 more bits of security than single-DES.

Key words. parallel collision search, cryptanalysis, discrete logarithm, hash collision, meet-in-the-middle attack, double encryption, elliptic curves.

1. Introduction

The power of parallelized attacks has been illustrated in work on integer factorization and cryptanalysis of DES. In the factoring of the RSA-129 challenge number and other factoring efforts (e.g. [26, 27]), the sieving process was distributed among a large number of workstations. Similar efforts have been undertaken on large parallel machines [14, 19]. In an exhaustive key search attack proposed for DES [44], a large number of inexpensive specialized processors were proposed to achieve a high degree of parallelism. In this paper, we provide a method for efficient parallelization of collision search techniques.[1]

[1] Preliminary versions of parts of this work have appeared in the proceedings of the Second ACM Conference on Computer and Communications Security [42] and in the proceedings of Crypto '96 [43].

Collision search is an important tool in cryptanalysis. A broad range of cryptanalytic problems such as computing discrete logarithms, finding hash function collisions, and meet-in-the-middle attacks can be reduced to the problem of finding two distinct inputs, a and b, to a function f such that $f(a) = f(b)$. The most efficient techniques for finding such collisions cannot be directly parallelized efficiently. The main contribution of this paper is a technique for efficient parallelization of collision search which reduces the attack time for many cryptographic schemes.

Of interest to collision search is the work of Amirazizi and Hellman showing that "time-memory trade-offs offer no asymptotic advantage over exhaustive search" [3], and that one must use multiple processors to take advantage of a large memory. For a fixed amount of resources, one can find the optimum time-memory-processor trade-off for mounting an attack. After optimizing both conventional techniques and the technique of parallelizing collision search presented here, one finds that parallel collision search has a much lower attack time for many cryptanalytic problems. Also of related interest to collision search are Hellman's time-memory trade-off for attacking block ciphers [22], the work on rigorous time-space trade-offs by Fiat and Naor [18], the DES cycling experiments of Kaliski, Rivest, and Sherman [23], the proof that DES is not a group [9], and the DES collisions found by Quisquater and Delescaille [35, 36].

The remainder of this paper is organized as follows. Section 2 reviews previous methods for collision search, and Section 3 motivates the need for efficient parallelization by showing that direct parallelization of previous methods is inefficient. Section 4 describes the new parallel collision search technique, and Section 5 applies it to three problems. Pollard's rho and lambda methods for computing discrete logarithms in cyclic groups are parallelized in Section 5.1. Even when only one processor is used, the new lambda method is 1.64 times faster than the original lambda method. Parallel collision search is applied to finding hash function collisions in Section 5.2, and to general meet-in-the-middle attacks in Section 5.3 which leads to the best attacks known on double encryption and three-key triple encryption. Section 6 gives machine designs and determines the run-time for attacking three different cryptographic schemes: an elliptic curve cryptosystem over $GF(2^{155})$ [1], the MD5 hash function [37], and double-encryption with DES [10]. Section 7 concludes the paper.

2. Previous Methods for Collision Search

The new technique for efficiently parallelizing collision search described in this paper is built upon Pollard's rho-method, which was originally applied to factoring [33] and discrete logarithms [34], but may be generalized to finding collisions in any function.

Pollard's rho-method for discrete logarithms is an improvement over the well-known "baby-step giant-step" algorithm, attributed to Shanks [25, pp. 9, 575]. Shanks' method allows one to compute discrete logarithms in a cyclic group G of order n in deterministic time $O(\sqrt{n})$ and space for $\sqrt{n}$ group elements. Pollard's rho-method also has time complexity $O(\sqrt{n})$ (heuristic time rather than deterministic), with only negligible space requirements; it is thus preferable. The time complexity of the rho-method for factoring is also heuristic, but there is progress towards a rigourous result by Bach [4]. For groups with additional structure (such as GF(p)), the powerful index calculus techniques are far superior, but do not apply to arbitrary cyclic groups. The rho-method is the best previous technique for proposed elliptic curve groups and for exploiting the small subgroup used in the Schnorr signature scheme [38], and DSA [11].

When searching for collisions among the outputs of some function f, the generalized rho-method is an improvement over the simple technique of selecting distinct inputs x_i for $i = 1, 2, \ldots$ and checking for a collision among the $f(x_i)$ values. Let n be the cardinality of the range of f. For the simple technique, the probability that no collision is found after selecting k inputs is

$$(1-1/n)(1-2/n)\ldots(1-(k-1)/n) \approx e^{-k^2/(2n)} \tag{1}$$

for large n and $k = O(\sqrt{n})$ [31]. The expected number of inputs that must be tried before a collision is found is $\sqrt{\pi n/2}$ (see Appendix A). Assuming that the $f(x_i)$ values are stored in a hash table so that new entries can be added in essentially constant time, this method finds a collision in $O(\sqrt{n})$ time and $O(\sqrt{n})$ memory. The large memory requirements can be eliminated using the rho-method. This method involves taking a pseudo-random walk through some finite set S. Conceptually, the shape of the path traced out resembles the letter rho, giving this method its name. Assume the function f has the same domain and range (i.e., f: $S \rightarrow S$). Select a starting value $x_0 \in S$, then produce the sequence $x_i = f(x_{i-1})$, for $i = 1, 2, \ldots$. Because S is finite, this sequence must eventually begin to cycle. The sequence will consist of a leader followed by an endlessly repeating cycle. If x_l is the last point on the leader before the cycle begins, then x_{l+1} is on the cycle. Let x_c be the point on the cycle that immediately precedes x_{l+1}. When $i = c$, we have a desired collision because $f(x_l) = f(x_c)$, but $x_l \neq x_c$. The run time analysis for the simple algorithm above also applies here. The expected number of steps taken on the pseudo-random walk before an element of S is repeated is $\sqrt{\pi n/2}$, where $n = |S|$. The advantage of this method is that the memory requirements are small if one uses a clever method of detecting a cycle.

A simple approach to detecting a collision with Pollard's rho method is to use Floyd's cycle-finding algorithm [24, Section 3.1, ex. 6], which has been optimized somewhat by Brent [7]. Start with two sequences, one applying f twice per step and the other applying f once per step, and compare the outputs of the sequences after each step. The two sequences will eventually reach the same point somewhere on the cycle.[1] However, this is roughly three times more work than is necessary. Sedgewick, Szymanski, and Yao showed that by saving a small number of the values from the sequence, one could step through the sequence just once and detect the repetition shortly

[1] At this point we have detected that a collision has occurred, but we have not found the point where the leader meets the cycle. As discussed later, finding this point is necessary in some cases (e.g., finding collisions in hash functions), but in other cases (e.g., Pollard's methods of factoring and computing discrete logarithms [33, 34]) it is sufficient that there are two distinct paths to the same point.

after it starts [40]. In finding DES collisions [35, 36], Quisquater and Delescaille took a different approach based on storing *distinguished points* [1], an idea noted earlier by Rivest (see [12, p.100]) to reduce the search time in Hellman's time-memory trade-off [22]. A distinguished point is one that has some easily checked property such as having a fixed number of leading zero bits. During the pseudo-random walk, points that satisfy the distinguishing property are stored. Repetition can be detected when a distinguished point is encountered a second time. The distinguishing property is selected so that enough distinguished points are encountered to limit the number of steps past the beginning of the repetition, but not so many that they cannot be stored easily.

3. Direct Parallelization of the Rho-Method

Pollard's rho method is inherently serial in nature; one must wait for a given invocation of the function f to complete before the next can begin. In discussing the rho-method for factorization, Brent considered running many processors in parallel each producing an independent sequence, and noted that "Unfortunately, parallel implementation of the "rho" method does not give linear speedup" [8, p. 29]. Analogous comments apply to the rho-method for computing logarithms and the generalized rho-method for collision search. Note that here each parallel processor is producing its own sequence of points independently of the others and each particular processor does not increase the probability of success of any other processor. With the classical occupancy distribution applicable to collision search, the probability of each new point succeeding in producing a collision increases with time as the number of computed points increases. Thus, none of the parallel processors will reach a success probability as high as is reached in the single-processor case (because the single processor runs for much longer), which leads to poor effectiveness of the parallel approach. If there are m processors, the probability that no collision has occurred for any processor after selecting k inputs out of a space of size n is $[(1-1/n)(1-2/n)\dots(1-(k-1)/n)]^m \approx e^{-k^2m/(2n)}$ (see equation (1)). This is (approximately) the same distribution that one gets with a single processor operating on a set with n/m elements. Thus the expected number of steps taken by each processor before a collision occurs is $\sqrt{\pi n/(2m)}$. Because the expected speedup is only a factor of $\sqrt{m}$, this is a very inefficient use of parallelization as it requires $\sqrt{m}$ times more processing cycles than the single processor (serial) version. This is the best previously reported use of parallelization for the rho-method.

In finding DES collisions [35], Quisquater and Delescaille were able to overcome a problem similar to the inefficient parallelization problem. To find pairs of DES keys which map a given plaintext to the same ciphertext, they used $f(x) = g(E_x(P))$, where P is the fixed plaintext, $E_x(\cdot)$ denotes DES encryption with key x, and g maps a 64-bit DES text to a 56-bit DES key. The loss

---

[1] Neither the cycle finding method of Sedgewick, Szymanski, and Yao nor distinguished point methods are applicable to the rho method of factoring. This is because we are cycling among elements modulo a prime, but we only have the representation of these elements modulo a multiple of the prime. Each element has multiple representations, which makes tests for equality non-trivial.

of information in the mapping g leads to "pseudo-collisions" when two keys encrypt to different ciphertexts, but g maps the ciphertexts to the same value. Only one in 2^8 collisions in f is a true DES collision rather than a pseudo-collision. They ran several sequences with different starting points within the same processor. All sequences contributed to the same list of distinguished points. The process continued until a true DES collision (rather than a pseudo-collision) was found. The time to get to the first collision is approximately $2^{56/2} = 2^{28}$, but the probability of getting a true (rather than a pseudo-) collision is 2^{-8}. If all data used to find a pseudo-collision is abandoned and one starts all over again, then the expected run time is $2^{28}/2^{-8} = 2^{36}$. However, by keeping previous data, the number of collisions found grows as the square of the time spent (because, the number of pairs of points grows as the square of the number of points produced). In this case, after about $2^{28}\sqrt{2^8} = 2^{32}$ steps, one would expect to have 2^8 collisions, one of which is expected to be a true DES collision. This eliminated the penalty caused by pseudo-collisions. However, when they parallelized this algorithm to m processors, they achieved a speed up of only a factor of $\sqrt{m}$ because the processors operated independently with different mappings g. In contrast, parallel collision search (Section 4) provides a speedup by a factor m for m processors.

4. Parallel Collision Search

In this section we describe the general parallel collision search algorithm. Two cases are considered: one where only a small number of random collisions are required (Section 4.1), and the second for problems where we require a large number of collisions (Section 4.2). These ideas may seem unmotivated at this point; some readers may choose to skip to Section 5 to see how these search algorithms are used to solve practical problems.

4.1 Finding a Small Number of Collisions

The goal in collision search is to create an appropriate function f and find two different inputs that produce the same output. This function f is chosen so that finding a collision serves some cryptanalytic purpose. To use the generalized rho-method, we require f to have the same domain and range (i.e., f: $S \rightarrow S$) and for f to be sufficiently complex that it behaves like a random mapping.[1] A random mapping is one which is selected with a uniform distribution across all possible mappings.

To perform a parallel collision search, each processor proceeds as follows. Select a starting point $x_0 \in S$ and produce the trail of points $x_i = f(x_{i-1})$, for $i = 1, 2, \ldots$ until a distinguished point x_d is reached based on some easily testable distinguishing property such as a fixed number of leading

---

[1] When we wish to find a collision for some function f': $D \rightarrow R$, $D \neq R$, we can define a function g: $R \rightarrow D$ and let $f = g \circ f'$. If $|D| \geq |R|$ then g can be made to be injective and a collision in f is also a collision in f'. If $|D| < |R|$ then g can be constructed so that the probability that a collision in f is also a collision in f' is $|D|/|R|$. If the behaviour of f is significantly different from a typical random function, then it may be possible to create a suitable function by composing f with some bijective mapping which has sufficiently random behaviour.

zero bits. Add the distinguished point to a single common list for all processors and start producing a new trail from a new starting point.[1] Depending on the application of collision search, other information must be stored with the distinguished point (e.g., one must store x_0 and d in order to quickly locate the points a and b such that $f(a) = f(b)$). A collision is detected when the same distinguished point appears twice in the central list. As illustrated in Figure 1, we have many processors taking pseudo-random walks through the set S producing many trails terminating at distinguished points. As soon as any trail touches another trail, the two will coincide from that point on (see trails 3 and 4 in Figure 1).[2] Trails 3 and 4 terminate at the same distinguished point $x_5 = x_4'$, and the collision in f is $f(x_2) = f(x_1')$. After the first collision is detected, if more collisions are required, one can continue producing trails and distinguished points until the desired number of collisions have been found.

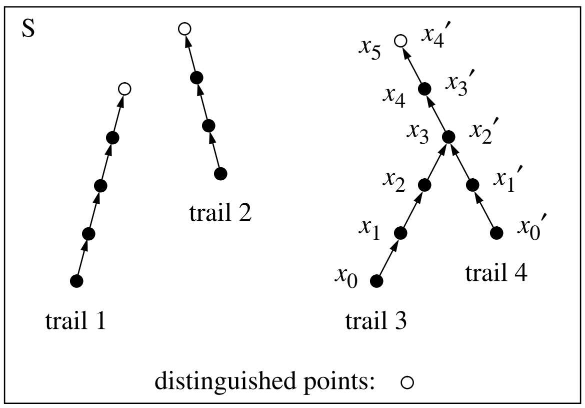

Figure 1. Parallelized Collision Search

Note that it is possible for one trail to collide with the starting point of another trail in which case we have a "Robin Hood" which does not yield a collision in f. Let θ be the proportion of points which satisfy the distinguishing property. The lengths of trails are geometrically distributed with mean $1/\theta$. In practice, θ is small enough that trails are long and Robin Hoods are rare. It is also possible for a trail to fall into a loop which contains no distinguished point. Left undetected, the processor involved would cease to contribute to the collision search. This problem can be handled by setting a maximum trail length of say $20/\theta$ and abandoning any trail which exceeds

[1] In the early description of parallel collision search [42], processors continued the trail from a distinguished point rather than starting at a new point. When several collisions are needed, the continuation method has the potential disadvantage of a processor's trail falling into a cycle and repeatedly detecting the same collisions.

[2] Although the collision depicted in Figure 1 resembles a lambda rather than a rho, it should not be confused with Pollard's lambda method for computing discrete logarithms. The rho method (which parallel collision search is based on) makes use of a pseudorandom function where all elements of a space are accessible on each iteration, whereas the lambda method requires an ordering of elements with each iteration taking a step whose distance is small relative to the total size of the space.

the maximum length. The proportion of trails whose length exceeds $20/\theta$ is $(1-\theta)^{20/\theta} \approx e^{-20}$. Each abandoned trail is about 20 times longer than the average, and so the proportion of work that is abandoned is approximately $20e^{-20} < 5\times10^{-8}$.

Run-Time Analysis for Finding a Small Number of Collisions. We now examine the expected run-time of the parallel collision search algorithm starting with the case where only a single collision is required. Of all the points on all the trails, if any two are the same, this will eventually lead to a duplicated distinguished point which will be detected. Let $n = |S|$, and let m be the number of processors producing trails. From Section 2, we expect to produce $\sqrt{\pi n/2}$ points before one trail touches another. The work required by each processor is $\sqrt{\pi n/2}/m$ steps. In some applications of collision search, not all collisions are useful. Let $q \le 1$ be the probability that a given collision is useful. The probability that no useful collision is found after k steps among all processors is

$$(1-q/n)(1-2q/n)\ldots(1-(k-1)q/n) \approx e^{-k^2q/(2n)}. \tag{2}$$

The work required by each processor to find a useful collision is $\sqrt{\pi n/(2q)}/m$ steps. This is only $1/\sqrt{q}$ times more than the case where all collisions are useful because the number of collisions grows as the square of the time spent. After a useful collision occurs, we expect the processor involved to have to produce an additional $1/\theta$ points before the trail strikes a distinguished point.[1] In some collision search applications, it is then necessary to locate the point on the two trails where the collision occurred (in the case of discrete logarithms (see Section 5.1) this is not necessary). To locate the collision efficiently, we need the starting points of the trails and their lengths. Begin by stepping the longer trail forward until its remaining length is the same as the other trail's length. Then step the trails forward in unison until they both hit the same point. Barring a Robin Hood, the points a and b preceding the common point are such that $f(a) = f(b)$, but $a \ne b$, as required. Stepping the longer trail must be done serially and then the two trails can be stepped forward with parallel processors. The expected time required for this process is $1.5/\theta$ iterations of f, which is the expected maximum of two random variables each with geometric distribution and mean $1/\theta$ (see Appendix B). In summary, the run-time to detect the first useful collision is

$$T = \begin{cases} \left(\sqrt{\pi n/(2q)}/m + 1/\theta\right) t & \text{if collisions merely need to be detected} \\ \left(\sqrt{\pi n/(2q)}/m + 2.5/\theta\right) t & \text{if collisions must be located} \end{cases} \tag{3}$$

where t is the time required for an iteration of f. Although it is expected that $1/q$ collisions will occur before a useful one is found, the terms $1/\theta$ and $2.5/\theta$ do not increase by a factor $1/q$

[1] There is an apparent paradox here because trails average $1/\theta$ in length, but the expected distance from a point of collision to the end of the trail is also $1/\theta$. Longer trails are more likely to be involved in a collision, which resolves the paradox.

because all other processors can go on working while a collision which has occurred is being detected and located (if necessary). The optimum value of θ is dependent on the relative costs of processors and memory, which is addressed in Section 6.

4.2 Finding a Large Number of Collisions

We now consider the case where many collisions are required in a function f: $S \rightarrow S$. In this case, of all the collisions that exist for the function f, only one is the "golden collision" which leads to a solution to the cryptanalytic problem at hand. An algorithm to find the golden collision is useful for performing meet-in-the-middle attacks faster than can be done with previous techniques (see Section 5.3). The algorithm proceeds in the same way as the case where only a small number of random collisions are required except that we continue collecting collisions until the golden collision is found. There must be some test available to determine if a given collision is the golden collision.

The algorithm begins with processors generating trails each starting from a different point $x_0 \in S$ and computing $x_i = f(x_{i-1})$, for $i = 1, 2, \ldots$ until a distinguished point x_d is reached after d steps, where the expected value of d is $1/\theta$, and $\theta < 1$ is the proportion of points which are distinguished. The triple (x_0, x_d, d) is stored in a memory. Let w be the number of triples that the memory can hold. To simplify access to the memory, we will assume that the memory address where a triple is stored is based on a fixed function of the distinguished point. For example, suppose that S = $\{0, 1, \ldots, n-1\}$, and the set of distinguished points is $\{0, 1, \ldots, \lceil n\theta \rceil - 1\}$. Then a triple (x_0, x_d, d) could be stored at address $\lfloor wx_d/(n\theta) \rfloor$ in the memory. If the memory element was already full, holding a different distinguished point, then the old data is simply overwritten. If the memory element was full, holding the same distinguished point, then we have a collision; in this case, the two triples are used to locate the collision, and the memory element is overwritten with the new triple. A collision is located as in Section 4.1 by first stepping along the longer trail until the remaining portion of the longer trail is the same length as the shorter trail, and then stepping along the trails in unison until they reach the same point. The points preceding the common point are distinct values which map to the same value.

The number of unordered pairs of distinct points in S is approximately $n^2/2$; for each pair, the probability is $1/n$ that the two points therein map to the same value through f. Therefore, the expected number of collisions that exist, considering all unordered pairs of distinct points, is about $n/2$. If all collisions were equally likely to occur, then one would expect to have to find $n/2$ collisions before the golden one was found. However, collisions are not equally likely to occur. For a given function f, the golden collision may have a very low probability of detection. It is necessary to have many versions of the function f all of which have a golden collision. The probability of detecting the golden collision will be different for each version of the function. Each version is tried for a fixed time period and then another is tried until the golden collision is found.

Run-Time Analysis for Finding a Large Number of Collisions. Let us begin with a simple, but flawed, run-time analysis. If the memory is full with w distinguished points, then the total number of points on the trails leading to those distinguished points is about w/θ. With each trail point generated with f in the space of size n, the probability of producing a point on one of the existing trails is $w/(n\theta)$. The required number of generated points per collision found is then $n\theta/w$. To locate a collision, each trail involved must be retraced from its start to the colliding point requiring a total of $2/\theta$ steps on average. Note that because so many collisions are generated, the process of not only generating, but also locating collisions should be parallelized. The total cost per collision detected is $n\theta/w + 2/\theta$ steps. This is a minimum of $\sqrt{8n/w}$ steps when $\theta = \sqrt{2w/n}$. The expected number of collisions generated before the golden collision is found is $n/2$, giving a total run-time of $(n/2)\sqrt{8n/w} = \sqrt{2n^3/w}$ function iterations.

There are a number of flaws in the analysis above. The memory for holding distinguished points is empty at the start of the algorithm, and thus is not full all of the time. Not all collisions are equally likely to occur. Not all distinguished points in the memory are equally likely to produce a collision. When a new distinguished point does not lead to a collision, it falls into one of the memory elements at random and reduces the expected number of empty memory elements. However, the number of empty memory elements stays the same when a collision occurs. Thus, collisions cause the memory to fill more slowly than one would expect in the standard occupancy distribution model. This fact combined with the fact that not all distinguished points are equally likely to produce a collision means that, over time, the distribution of distinguished points in the memory tends to become biased towards distinguished points with lower probability of producing a collision.

To examine the true performance of the algorithm, several simulations were run. Let $\theta = \alpha\sqrt{w/n}$, and let the number of distinguished points produced per version of the function be βw for some constants α and β. Each simulation was performed for a set of parameters (α, β, w, n) keeping track of the number of function iterations (i) and number of distinct collisions produced (c). On each simulation run, the results, i and c, were averaged across $\lceil 2^{16}/w \rceil$ versions of the function f. Although only one version of f was used for simulation runs where $w \geq 2^{16}$, multiple runs with the same parameters, but different versions of f, were found to give results which agreed to within less than 1%.

For the parameters used in a given simulation run, the expected number of versions of the function required to find the golden collision is $n/(2c)$, and the expected total run-time is $in/(2c)$ function iterations. Simulations were first run for $w = 2^{16}$, $n = 2^{32}$, and many different pairs of values for α and β. The values which minimized the total run-time were $\alpha = 2.25$ and $\beta = 10$. Simulations were then run with these values of α and β for various values of w and n (see Table 1). The run-time of each version of f is $O(\sqrt{nw})$, and the overall run-time of the algorithm to find the golden collision is $O(\sqrt{n^3/w})$. Each entry in Table 1 is the coefficient of $\sqrt{n^3/w}$ in the number of function iterations required to find the golden collision. The simulation results had very little variance across different values of n for each particular value of w, except that the run-times dropped slowly as w approached n. For values of w and n which would lead to $\theta > 1$ (an impossibility), the table entry has a "—"; empty entries correspond to simulations not run due to the excessive amount of computation required.

Table 1. Run-Time Results of Simulations of the Algorithm to Find the Golden Collision

n	w										
	2^0	2^2	2^4	2^6	2^8	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
2^{16}	9.13	4.18	2.84	2.52	2.40	2.33	2.25	—	—	—	—
2^{18}	9.09	4.17	2.87	2.52	2.43	2.39	2.33	2.23	—	—	—
2^{20}	9.09	4.14	2.85	2.52	2.44	2.43	2.39	2.33	2.23	—	—
2^{22}	9.04	4.20	2.87	2.56	2.47	2.43	2.42	2.39	2.34	2.24	—
2^{24}	9.07	4.15	2.85	2.54	2.47	2.43	2.42	2.41	2.38	2.34	2.25
2^{32}	9.04	4.22	2.87	2.53	2.48	2.45	2.44	2.45	2.45	2.47	2.50
2^{40}				2.61	2.48	2.44	2.43	2.43	2.46	2.44	2.44
2^{48}						2.45	2.43	2.44	2.44	2.44	2.44
2^{56}									2.43	2.43	

These simulations show that if $10w$ distinguished points are generated for each version of the function, $\theta = 2.25\sqrt{w/n}$, and $w \geq 2^{10}$, the expected run-time to find the golden collision can be slightly overestimated as

$$T = (2.5\sqrt{n^3/w}/m)t \tag{4}$$

where m is the number of processors, and t is the time required for a function iteration. Other results from the simulations are: for $2^{10} \leq w \leq n/2^{10}$, each function generates about $1.3w$ collisions, of which about $1.1w$ are distinct; 80% of the function iterations are devoted to generating distinguished points and 20% are devoted to locating collisions; the expected number of versions of the function required is $0.45n/w$; and the expected number of distinguished points written to memory is $4.5n$.

5. Cryptanalytic Applications of Parallel Collision Search

In this section, we apply parallel collision search to computing discrete logarithms in cyclic groups (Section 5.1), finding hash function collisions (Section 5.2), and meet-in-the-middle attacks (Section 5.3).

5.1 Application to Discrete Logarithms in Cyclic Groups

We now apply parallel collision search to the problem of finding discrete logarithms in a cyclic group G of order n with generator g. Given some element $y = g^x$ of G, we wish to find x. We first apply the parallelized rho-method of Section 4.1 and then show how to parallelize Pollard's lambda-method of computing discrete logarithms. Elliptic curve cryptosystems are an example of a system based on a discrete logarithm problem where collision search is the best known attack. Other examples are the Schnorr signature scheme [38] and DSA [11] when the subgroup of prime order q is attacked directly. For Schnorr and DSA, index calculus attack methods apply to the larger group GF(p), and the sizes of p and q determine which attack is superior.

The first step in finding a discrete logarithm is to take advantage of the factorization of n using a Pohlig-Hellman decomposition [32] (see also [28]). For each prime power p^k dividing n, find $x \bmod p^k$ as follows. Compute $g' = g^{n/p}$, a generator of a subgroup of order p. Consider $x \bmod p^k$ in radix p notation, $x \equiv \Sigma_{i=0}^{k-1} a_i p^i \pmod{p^k}$, where the a_i are unknown. For $i = 0, 1, \ldots, k{-}1$, find a_i as follows. Compute the known part of the exponent $z = \Sigma_{j=0}^{i-1} a_j p^j$, and compute $y' = (y/g^z)^{n/p^{i+1}}$ (note that $y' = (g')^{a_i}$). Recover a_i by computing the discrete logarithm of y' in the subgroup of order p. Finally, use the Chinese Remainder Theorem to recover x from the residues of x modulo each prime power dividing n. The running time will be dominated by the time required for the discrete logarithm in the subgroup of order p for the largest prime p dividing n. For the remainder of this section, we will consider the case where G is of prime order n=p.

In a parallel version of the rho-method for logarithms, we suggest the same iterating function used by Pollard [34]. Partition the set of group elements into three roughly equal size disjoint sets S_1, S_2, and S_3, based on some easily testable property. Define the iterating function:

$$x_{i+1} = \begin{cases} yx_i & \text{if } (x_i \in S_1) \\ x_i^2 & \text{if } (x_i \in S_2) \\ gx_i & \text{if } (x_i \in S_3) \end{cases}$$

Each processor performs the following steps independently. Choose random exponents $a_0, b_0 \in [0, \mathrm{p})$ and use the starting point $x_0 = g^{a_0} y^{b_0}$. A key point is that different processors use independent starting points of known relation to each other allowing collision information to be resolved into the recovery of logarithms. Compute the sequence defined above keeping track of the total exponents (modulo p) of g and y (note $x_i = g^{a_i} y^{b_i}$). When x_i is a distinguished point, contribute the triple (x_i, a_i, b_i) to a list common to all processors and begin again with a new starting point. Shortly after a collision occurs, the colliding processor will encounter a distinguished point and there will be a collision among the x_i values in the list. If the corresponding exponents in the two triples are a, b and c, d, then $g^a y^b = g^c y^d$ or $g^{a-c} = y^{d-b}$. Provided $b \not\equiv d \pmod p$, the desired logarithm can be computed as $\log_g y \equiv (a{-}c)\cdot(d{-}b)^{-1} \pmod p$; otherwise, the collision is not useful and the search must continue. Based on randomness assumptions, the probability that $b \equiv d \pmod{p}$ is very small if p is large enough to warrant using parallel collision detection.

We now refer to equation (3) to determine the expected run-time of the discrete logarithm algorithm. For this application of collision search, the probability that a collision is useful (q) is very close to 1, and collisions merely need to be detected, not located. The overall run-time is

$$T_\rho = (\sqrt{\pi p/2}/m + 1/\theta)t \tag{5}$$

The algorithm above is designed for the case where x, the logarithm being sought, could be any value less than the group order. However, in practical implementations of discrete logarithm-based systems, the exponent size is sometimes limited to a restricted range for faster exponentiation. The algorithm above will work in this case, but there is a faster approach. Begin by using a Pohlig-Hellman decomposition to find x modulo the smaller primes dividing the group order. Suppose that we are left with the problem of finding x given $y = g^x$ in a subgroup of prime order p, where $x \in [0, b)$ for some bound $b < p$. Pollard's lambda-method [34] (affectionately known as the method of catching kangaroos) is well-suited to this task. A single processor version of the lambda-method which does not use distinguished points proceeds as follows. Define an iterating function $x_{i+1} = x_i g^{a(x_i)}$, where a is a function taking values randomly from a set A. A possible choice for the set A is the powers of 2 (starting with 2^0) [1] up to some limit with the largest entry selected such that it determines a particular value for the mean of the set entries. Let the mean of the values in A be α (the optimum value of α is found below). If we think of a number line labelled with the powers of g, each iteration jumps the kangaroo forward by a distance of $a(x_i)$, and the distance jumped is determined solely by x_i. Start a tame kangaroo at $x_0 = g^b$ and let it make $\alpha\beta$ jumps (for some constant β optimized below) keeping track of the total distance travelled, and record the final resting spot and distance travelled. This kangaroo is tame because we know the logarithm of the points that it lands on. Now start a wild kangaroo at $x_0' = y$ and let it jump (with the same iterating function) keeping track of the distance travelled and checking whether it ever lands on the tame kangaroo's final resting spot. If the wild kangaroo ever lands on any of the same spots as the tame kangaroo, then it will follow the same path thereafter, and when it reaches the tame kangaroo we can find the desired logarithm from the total distances travelled by each kangaroo. If the method fails, set off another wild kangaroo with a starting point $x_0 = yg^z$ for some small known z.

We now determine the run-time of the algorithm and optimize the constants α and β. Note that we can pre-compute g^u for each $u \in A$ so that each kangaroo jump costs one group operation. After the wild kangaroo passes g^b, it takes about $\alpha\beta$ jumps before reaching the tame kangaroo with each jump having a probability of about $1/\alpha$ of landing on a spot that the tame kangaroo once landed on. The probability of success is about $1 - (1 - 1/\alpha)^{\alpha\beta} \approx 1 - e^{-\beta}$. The final resting

---

[1] Pollard speculated that powers of 2 would be acceptable [34], but feels that further investigation is needed.

spot for the tame kangaroo is at a distance of about $b + \alpha^2\beta$ from g^0. If the wild kangaroo travels this far (which takes about $b/\alpha + \alpha\beta$ jumps) we stop it because it must have passed the tame kangaroo without landing on it. Because the expected starting point for the wild kangaroo is $g^{b/2}$, when the algorithm succeeds, the expected number of steps is $b/(2\alpha) + \alpha\beta$. Overall, the tame kangaroo is sent once (taking $\alpha\beta$ jumps), and the wild kangaroo succeeds once and is expected to fail $1/(1-e^{-\beta}) - 1$ times. This gives a total run-time of $\alpha\beta - b/(2\alpha) + (\alpha\beta + b/\alpha)/(1-e^{-\beta})$ group operations. This is minimized when $\alpha = \sqrt{b\,(1 + e^{-\beta}) / (2\beta\,(2 - e^{-\beta}))}$. Using numerical techniques, we find that the run-time is minimized when $\beta \approx 1.39$ giving $\alpha \approx 0.51\sqrt{b}$ and a total run-time of approximately $3.28\sqrt{b}$ group operations.

The lambda-method suffers from the same parallelization problem as the rho-method: running m processors independently gives a speedup of only $\sqrt{m}$. However, the lambda method can be efficiently parallelized as follows. Again, let the mean of the elements of A be α (the optimum value of α is found below). Launch $m/2$ tame kangaroos with starting points $g^{(b/2)+iv}$, for $0 \le i < m$, where v is a small constant (avoid a power of 2 for v so that the probability of tame kangaroos following the same path is not abnormally high). At the same time, launch $m/2$ wild kangaroos with starting points yg^{b+iv}, for $0 \le i < m$. Whenever a kangaroo lands on a distinguished point, store it in a list common to all processors. With each distinguished point, store a flag indicating the kangaroo type (tame or wild) and the distance from g^0 (for tame kangaroos) or from y (for wild kangaroos). Suppose that a kangaroo reaches a distinguished point that is already in the list. If the kangaroos are of the same type, then move the trailing kangaroo forward by some small random distance so that it will not repeat the other kangaroo's path. If the distinguished point was produced by kangaroos of different type, then subtract the distances to get the desired logarithm and we are done.

We now determine the run-time of the parallelized lambda-method. Initially, the two groups of kangaroos are separated by some distance between 0 and $b/2$. The expected separation is $b/4$ and it takes about $b/(4\alpha)$ jumps for the trailing kangaroos to cover this distance. After this, the trailing kangaroos enter a region where the proportion of points landed on by leading kangaroos is $(m/2)/\alpha$. On each step, the probability that one of the $m/2$ trailing kangaroos lands on a spot previously occupied by a leading kangaroo is about $(m/2)^2/\alpha$, and the expected number of jumps for each kangaroo before this happens is $4\alpha/m^2$. Adding this to the time required to close the initial gap between the two groups of kangaroos, we get $b/(4\alpha) + 4\alpha/m^2$ jumps, which is a minimum of $2\sqrt{b}/m$ when $\alpha = (m/4)\sqrt{b}$. If the proportion of points satisfying the distinguishing property is θ, we expect it to take an additional $1/\theta$ jumps to reach the next distinguished point after a collision occurs. If the time required for a group operation is t, then the total run-time is

$$T_\lambda = (2\sqrt{b}/m + 1/\theta)t \tag{6}$$

In the analysis above, we assumed that the number of processors is even with the processors divided equally between tame and wild kangaroos. When there is only one processor, we can simulate two half-speed processors by alternating between jumps of one tame and one wild kangaroo. Thus using equation (6) with m=2 and $2t$ as the time to perform a group operation, we get $T_\lambda = (2\sqrt{b} + 2/\theta)t$ when there is only one processor. This agrees with (6) for m=1 except for the term involving θ which can be made small by choosing an appropriate distinguishing property. Compared to a run-time of $(3.28\sqrt{b})t$ for the previous version of the lambda-method which does not use distinguished points, the new method is 1.64 times faster.[1] It is also interesting to compare the run times of the parallelized lambda and rho-methods. For the case where the exponent is full size ($b = p$), the lambda-method is about 1.60 times slower; it becomes faster when $b < 0.39p$.

5.2 Application to Hash Functions

In this section we apply the parallel collision search technique to finding real collisions in hash functions. By "real" we mean not only collisions on the overall hash function (as opposed to simply on the compression function of a hash function), but such collisions which are meaningful in practice (e.g., where the input messages may be largely or entirely selected by an attacker). Such collisions were found by Dobbertin in MD4 [15]. These methods apply to any hash function including MD5 [37], RIPEMD [5] and its successors RIPEMD-128 and RIPEMD-160 [16], SHA-1 [39], and MDC-2 and MDC-4 [30]. We first review how hash functions are typically used in conjunction with digital signatures, and the classic attack of Yuval [45]. We then apply parallel collision search to extend this attack to allow parallelization and reduce memory requirements.

Hash functions are designed to take a message of arbitrary bitlength and map it to a fixed size output called a hash result. Let $H: M \to R$ be such a hash function. Typically, hash functions are constructed from a function $h: B \times R \to R$ which takes a fixed size block of message bits together with an intermediate hash result and produces a new intermediate hash result. A given message $m \in M$ is typically padded to a multiple of the block size and split into blocks $m_1, \ldots m_l \in B$. The padding often includes a field which indicates the number of bits in the original message. Beginning with some constant $r_0 \in R$, the sequence $r_i = h(m_i, r_{i-1})$ is computed for $i = 1, \ldots l$, and r_l is the hash result for message m.

Hash functions are commonly used in connection with digital signatures. Instead of signing a message directly, the message is first hashed and the hash result is signed. For cryptographic security, it must be computationally infeasible to find two messages that hash to the same value; otherwise, a digital signature could be copied from one message to the other.

---

[1] To be fair, Pollard's paper describing the lambda-method [34] discussed using a programmable calculator for performing discrete logarithms. The new lambda-method is not useful unless it is possible to store at least 4 or 5 distinguished points, and this would have been awkward on a programmable calculator.

Now suppose we have a message m that we would like the victim to digitally sign, but he is not willing to do so. A simple attack on the hash function can help to acquire the desired signature as follows [45]. Choose some other message m' that the victim is willing to sign. Find several ways to modify each of m and m' that do not alter their respective semantic meaning (e.g., adding extra spaces or making small changes in wording). The combinations of message modifications lead to many versions of a message, all of which have essentially the same meaning. Then hash the different versions of m and m' until we find two versions that give the same hash result. The victim will sign the version of m', and we can copy the signature to m. This attack requires $O(\sqrt{n})$ time and space, where $n = |R|$.

The memory requirements for a hash function attack can be eliminated using collision search techniques. Let $m \in M$, and let $g_m\colon R \rightarrow M$ be an injective function which takes a hash result (and a fixed message m) as input and produces a perturbation of m with the same semantic meaning to the signer. For example, the bit positions in a hash result could correspond to sentences in the message m with each hash result bit deciding whether to place one or two spaces after the period. Partition the set R into two roughly equal size subsets S_1 and S_2 based on some easily testable property of a hash result. Then define a function $f\colon R \rightarrow R$ as follows, with m and m' as described above as implicit constants.

$$f(r) = \begin{cases} H(g_m(r)) & \text{if } (r \in S_1) \\ H(g_{m'}(r)) & \text{if } (r \in S_2) \end{cases}$$

Using the parallel collision search technique from section 4.1, find pairs of hash results a and b such that $f(a) = f(b)$, but $a \neq b$. A collision is not useful unless a and b are in different subsets S_i, S_j of R, which will occur with probability $1/2$. Suppose $a \in S_1$ and $b \in S_2$; then $H(g_m(a)) = H(g_{m'}(b))$ (i.e., we have versions of m and m' which give the same hash result).

With this approach, each iteration of f requires applying the hash function to an entire message. To reduce the amount of computation required, we choose mappings g_m and $g_{m'}$ to affect as few message blocks as possible. For g_m and $g_{m'}$ to be injective, we must be able to code the bits of a hash result into the affected message blocks. One example of a way to do this would be to find four non-printable characters and code two bits per byte. Another example is to code data within the shading of a company logo in a word processor file. For the most popular hash functions, message blocks are longer than hash results (i.e., $|B| > |R|$), and in the following, we will assume that the bits of a hash result can be coded into a single message block. The message m consists of the blocks $m_1, \ldots m_l$. Message m' consists of the blocks $m'_1, \ldots m'_j, m_{j+1}, \ldots m_l$. Note that m and m' must have the same length so that the length information coded in their final blocks will be the same. The effect of g_m and $g_{m'}$ is coded into blocks m_j and m'_j. If the intermediate hash results are the same for m and m' after block number j, then the final hash results will be the same as well. The function f now just needs to use one iteration of the hash function. Let r_{j-1} and r'_{j-1} be the intermediate hash results for m and m' after j–1 blocks. Replace g_m and $g_{m'}$ with $g\colon R \rightarrow B$ which maps a hash result to a message block; this assumes we are free to manipulate the message blocks m_j and m'_j in their entirety. Then use the following function f for collision search.

$$f(r) = \begin{cases} h(g(r), r_{j-1}) & \text{if } r \text{ is even} \\ h(g(r), r'_{j-1}) & \text{if } r \text{ is odd} \end{cases}$$

Referring to equation (3), collisions must be detected and located, and the probability that a collision is useful is $q = 1/2$, giving a run-time for finding a real hash collision of

$$T_{\text{hash}} = (\sqrt{\pi n}/m + 2.5/\theta)t \tag{7}$$

where $n = |\mathrm{R}|$, m is the number of processors, and t is the time required for a hash function iteration.

5.3 Application to Meet-in-the-middle Attacks

We now apply parallel collision search to meet-in-the-middle attacks. We begin by examining the run-time of the standard approach to performing such attacks and then show that parallel collision search can greatly improve run-time. Meet-in-the-middle attacks on double-DES and triple-DES are used to illustrate the techniques; see van Oorschot and Wiener [43] for additional applications including discrete logarithm attacks for the special case of bounded-weight exponents, and attacking a scheme for server-aided RSA computations. Other work on attacking multiple encryption includes the meet-in-the-middle attack on double-DES described by Diffie and Hellman [13] and generalized by Even and Goldreich [17], and attacks on two-key triple-encryption [29, 41].

In a general meet-in-the-middle attack, we have two functions, f_1: $D_1 \to R$ and f_2: $D_2 \to R$, and we wish to find two particular inputs $a \in D_1$ and $b \in D_2$, such that $f_1(a) = f_2(b)$. Let $n_1 = |D_1|$ and $n_2 = |D_2|$. Without loss of generality, assume $n_1 \le n_2$. If there are many pairs of inputs which satisfy this condition, there must be some further test to determine which pair is correct. For the case of double-DES encryption we assume that we have a plaintext-ciphertext pair (P, C) such that $C = E_{k_2}(E_{k_1}(P))$, where $E_x(\cdot)$ denotes DES encryption with key x, and k_1 and k_2 are the unknown keys. If we let $f_1(x) = E_x(P)$ and $f_2(x) = E_x^{-1}(C)$, then $f_1(k_1) = f_2(k_2)$. Note that P and C are constants; the inputs sought are k_1 and k_2. Many pairs of keys will match the particular plaintext-ciphertext pair. By testing each pair of keys on a second plaintext-ciphertext pair, with high probability only the correct pair of keys will remain.

A simple approach to performing the meet-in-the-middle attack proceeds as follows. Compute $f_1(x)$ for all $x \in D_1$ and store the $(f_1(x), x)$ pairs in a table (using standard hashing on the $f_1(x)$ values to allow lookup in constant time). For each $y \in D_2$, compute $f_2(y)$ and look it up in the table. If there is a match, then the candidate pair of inputs x and y must be tested to see if they are the correct inputs (a and b). This method requires, on average, $n_1 + n_2/2$ function evaluations and memory for n_1 pairs. For double-DES, this is $(3/2)2^{56}$ function evaluations and 2^{56} stored pairs. Obviously, this is not a practical amount of memory. Suppose that available memory can hold only w pairs. The attack can be modified as follows [17]. Divide the space D_1 into subsets of size w. For each subset, compute and store the $(f_1(x), x)$ pairs, and then for each $y \in D_2$, compute $f_2(y)$ and look it up. The expected run-time for this memory-limited version of the attack is $(1/2)(n_1/w)(w + n_2) \approx n_1 n_2/(2w)$ function evaluations.

To apply the method of finding a golden collision in Section 4.2 to meet-in-the-middle attacks, we must construct a function f for collision search. Because f must have the same domain and range, we split D_2 into subsets of size n_1 and perform a collision search for each subset. To simplify the following description, assume that n_1 divides n_2. Mappings from the integers to D_1 and D_2 are needed; let $h_1\colon I \to D_1$ and $h_2\colon I{\times}J \to D_2$ be bijective mappings, where $I = \{0, 1, \ldots n_1{-}1\}$ and $J = \{0, 1, \ldots (n_2/n_1){-}1\}$. Let $f_1'(x) = f_1(h_1(x))$ and $f_2'(x) = f_2(h_2(x, j))$, where $j \in J$ is a fixed value defining a subset of D_2. Treating j as a constant, both f_1' and f_2' are mappings from I to R. Let $g\colon R \to I{\times}\{1,2\}$ be a surjective mapping that takes an element of R to a pair which consists of an element of I and a bit to select either f_1' or f_2'. The mapping g should distribute the elements of R fairly uniformly across $I{\times}\{1,2\}$. We can now define $f\colon S \to S$, where $S = I{\times}\{1,2\}$, as $f(x,i) = \mathrm{g}(f_i'(x))$.

To show that f has a collision related to the desired inputs a and b, begin by letting $h_1^{-1}(a) = z$ and $h_2^{-1}(b) = (u,v)$. When $j = v$, $f_1(a) = f_2(b)$ implies that $g(f_1(h_1(z))) = g(f_2(h_2(u, j)))$, which means that $f(z,1) = f(u,2)$. Therefore, when $j = v$, one of the collisions of f leads to the desired inputs a and b. To use the algorithm for finding a golden collision, several versions of f are required. This can be achieved by choosing different versions of the mapping g. To find a and b, repeat the following for each version of f. For each $j \in J$, use the algorithm in Section 4.2 to search for the golden collision. Using equation (4), the expected run-time, for $\theta = 2.25\sqrt{w/|S|}$, is $|J|(2.5\sqrt{|S|^3/w}/m)t$, where w is the number of memory elements, m is the number of processors, and t is the time required for a function iteration. Because $|J| = n_2/n_1$ and $|S| = 2n_1$, the expected run-time of a meet-in-the-middle attack based on this parallel collision search technique is

$$T_{\mathrm{m}} = (7n_2\sqrt{n_1/w}/m)t \tag{8}$$

Compared to the standard approach whose run-time is $n_1 n_2 t/(2wm)$, the meet-in-the-middle attack based on parallel collision search is $0.07\sqrt{n_1/w}$ times faster. Another point in comparing the two approaches is that the standard approach requires an access to the memory common to all processors after every function evaluation, but the collision search approach only accesses memory every $1/\theta = 0.63\sqrt{n_1/w}$ function evaluations. Overall, the new method requires $n/(18w)$ times fewer memory accesses than the standard approach. In a highly parallel attack, the standard approach is limited by memory access time.

Applying collision search to attacking double-DES, $n_1 = n_2 = 2^{56}$ so that there is no need for splitting up D_2, h_1 and h_2 are the identity mappings, $f_1'(x) = E_x(P)$, and $f_2'(x) = E_x^{-1}(C)$. The mapping g takes the first 56 bits of the encryption or decryption result as the next key, and uses the next 8 bits of the result to derive a bit to choose between f_1' and f_2'. Because these 8 bits can take on 256 different values, an unbiased bit can be derived from them in $\binom{256}{128}$ ways, which is more than enough versions of f. Using equation (8), the expected attack time is

$$T_{2DES} = 7 \cdot 2^{84} t / (m\sqrt{w}) \tag{9}$$

Applying collision search to attacking triple-DES with three independent keys, if we use the point after the first encryption as the middle point, we have $n_1 = 2^{56}$ and $n_2 = 2^{112}$. Using equation (8), the expected attack time is

$$T_{3DES} = 7 \cdot 2^{140} t / (m\sqrt{w}) \tag{10}$$

The meet-in-the-middle attack based on collision search relies on having $|R| \geq 2|D_1|$. However, this would not be the case for attacking double encryption with an algorithm whose key size is greater than the block size. In this case, one could use more than one plaintext-ciphertext pair and define f_1 to encrypt multiple concatenated known plaintexts and concatenate the results. Define f_2 similarly. This increases the effective block size.

For the case where $|R| = |D_1|$, which would occur for double encryption when the key size and block size are equal, one could re-define the mapping g to compute the element of $\{1,2\}$ from the element of I that it outputs. This reduces $|S|$ by a factor of 2 saving a factor of $2\sqrt{2}$ in run-time (see equation (4) with $n = |S|$), but costs a factor of 4 because there is only one chance in 4 that there are two elements of R that map through g to $(a,1)$ and $(b,2)$. Overall, the run-time is $\sqrt{2}$ times greater, which is better than using two plaintext-ciphertext pairs which costs a factor of 2 because each function iteration would require two encryptions or decryptions.

6. Machine Designs

In practice, the true measure of the effectiveness of a cryptanalytic attack is the time required to complete the attack given some limitation of resources. For casual or “unfunded” attackers, a common measure of resources is the number of PCs and workstations which can be harnessed for the task. For well-funded attackers, a better measure is simply a dollar amount. For most attacks, a custom approach is far more effective than using available cycles on computers connected to the internet. Notable exceptions to this general rule are factoring and discrete logarithms in GF(p) using index calculus techniques, where general-purpose computers are fairly well suited to the tasks.

In this section, we consider the case of a well-funded attacker with \$10 million[1] available for a custom machine. Given this resource limitation, we estimate the time required to complete three different cryptanalytic problems using parallel collision search: computing elliptic curve logarithms over GF(2^{155}), finding collisions in the MD5 hash function, and recovering double-DES keys. To do this, it is necessary to estimate costs of custom processors, memory, and other components. Given these cost estimates, one can optimize θ (the proportion of distinguished points) and w (the number of memory elements) to minimize the attack time.

6.1 Elliptic Curve Logarithms Over GF(2^{155})

We now apply parallel collision search to a discrete-logarithm-based cryptosystem of Agnew et al. [1] which uses elliptic curve cryptosystems over GF(2^{155}). The security of such systems is apparently bounded by the difficulty of finding discrete logarithms over the group of points on the elliptic curve. Curves are used for which the best known discrete logarithm attack is Pollard's rho-method, and to make such attacks infeasible, they recommend curves where the order of the elliptic curve group contains a prime factor with at least 36 decimal digits (corresponding to $p \approx 10^{36}$ as discussed in Section 5.1). They analyze the attack effort as follows. Each step of the rho-method requires a number of arithmetic operations over the elliptic curve (specifically, for the implementation cited, 3 elliptic curve additions for Floyd's cycle-finding requiring a total of 39 field multiplications, each taking 155 clock cycles at 40 MHz), and if 1000 devices are used in parallel to compute a logarithm, they note the computation would still require 1500 years. (Although no method was known by which the rho-method could be parallelized, the existence of a parallelized version with perfect linear speedup was implicit in this reasoning.) It was previously believed [1, p.809]: "Provided that the square root attacks are the best attacks on the elliptic logarithm problem, we see that elliptic curves over F_{2^m} with m about 130 provide very secure systems". However, the analysis below indicates that the lower bound of 10^{36} for the size of the largest prime factor of the order of the group is too small to provide what we would understand as *very secure* systems.

Let u be the cost of each custom processor (including overhead for housing the custom chips in a machine), v be the cost of a memory element (including overhead for housing the memory), and t be the time for a function evaluation. The elliptic curve system over GF(2^{155}) can be implemented in less than 1 mm^2 of silicon in 1.5 µm technology and can perform an addition in 13×155 clock cycles at 40 MHz [1, 2]. This gives $t = 13 \times 155 / (40 \times 10^6)$ seconds. About 75 of these cells plus input/output and logic to detect distinguished points could be put on a \$20 chip. Based on a DES key search design [44], the overhead of building a machine with many of these chips would be about \$7 per chip. Each chip plus its overhead costs \$27 and contains 75 processors giving a cost of $u = \$0.36$ per processor. Each memory element must hold a triple consisting of a distinguished point x_i and two integers a_i and b_i such that $x_i = a_i g + b_i y$, where g is

---

[1] All estimates are given in U.S. dollars.

the generator of the elliptic curve subgroup of order p, and y is the elliptic curve point whose logarithm is sought. Because $p \approx 10^{36}$, a_i and b_i require 15 bytes each. A point on an elliptic curve over $GF(2^{155})$ can be represented in 155 bits for one coordinate plus one bit for the other coordinate. The distinguished point x_i will have 32 leading zero bits (based on the distinguishing property derived below) which do not need to be stored. Thus x_i requires $155+1-32 = 124$ bits or 16 bytes. Overall, a triple requires 46 bytes. Assuming that triples are stored in a standard hash table and the memory is limited to 99% full (to limit the number of probes necessary to find an empty element), each triple consumes $46/(0.99)$ bytes of memory. Assuming that memory costs \$50 per Mbyte[1], we have $v = \$0.0022$.

Given a budget limitation of $B = \$10$ million, the constraint on m (number of processors) and w (number of memory elements) is

$$mu + wv \; = \; B \tag{11}$$

To build a machine, we must fix the amount of memory. Then in the case that the desired logarithm is not found before the memory fills up, continue by simply overwriting new distinguished points on top of old ones. After the memory fills up, there are a total of about w/θ points on the trails leading to the distinguished points in the memory. This affects the run-time as shown in Appendix A. Combining the expected run-time of equation (5) with the effect of a memory constraint given by equation (18) with $z = w/\theta$ and $n=p$, we get a run-time of

$$\left(\left(\sum_{k=0}^{w/\theta - 1} e^{-k^2/(2p)} + (p\theta/w)\, e^{-w^2/(2p\theta^2)} \right) (1/m) + 1/\theta \right) t .$$

Using numerical techniques to minimize this expression subject to the constraint of equation (11), we find that the run-time is a minimum of 32 days when $\theta = (0.93)2^{-32}$, $m = 2.5\times10^7$, and $w = 3.8\times10^8$. This is a total of 330000 processor chips and 16 Gbytes of memory. This is not an impractical amount of memory; the existing general purpose parallel machine of [19] has 37 Gbytes of memory. Note also that θ is close to $(234/256)2^{-32}$, which can be achieved by defining a point as distinguished if it has 32 leading zero bits and the next 8 bits have a binary value less than 234. The overall rate at which the processors produce triples is $m\theta/t = 107$ per second, which can be accommodated quite easily by the large memory. The triples generated would be passed up from the processors through a hierarchy of controllers to a single processor which accesses the memory. Because the system could easily handle 1000 times as many triples per second, very little buffering is required to handle bursts in the arrival of triples. The number of triples lost due to bursts would be insignificant. Note that the only cost of losing a triple is wasting the time required to generate it; there is no further impact on the rest of the algorithm.

---

[1] This memory cost estimate is quite conservative. As of Sept. 1996, \$25 per Mbyte is a better estimate.

This analysis makes use of the basic hardware described by Agnew et al. [1], and does not take into account possible optimizations from pipelining the elliptic curve implementation or from using currently available silicon technology; thus an attacker could do even better. Note that the computation of the discrete logarithm in the large subgroup of order approximately 10^{36} will by far dominate other computational costs in the entire Pohlig-Hellman decomposition. This follows because the order of the elliptic curve group over $GF(2^m)$ is $2^m + O(2^{m/2})$, and $2^{155}/10^{36} < 10^{11}$, and so the remaining subgroups are necessarily relatively small for $m = 155$.

6.2 MD5 Collisions

We now apply the hash function collision techniques of Section 5.2 to the MD5 hash function. The size of the MD5 hash result space is $n = 2^{128}$. A 1995 internal study of an MD5 silicon implementation by Nortel (Bell-Northern Research) indicates that a collision search chip for one iteration of MD5 with 64 levels of pipelining (recall that MD5 involves 4 rounds of 16 computations per 512-bit block) could be built with a total area of (310 mils)^2 and would run at 50 MHz if designed in a 0.5 μm CMOS process. Due to the pipelining, each chip essentially consists of 64 independent processors, with each processor requiring $t = 64/(50\text{ MHz}) = 1.28\ \mu s$ to perform an iteration of f. The chips would also contain logic to detect distinguished points to minimize the rate of input and output to keep costs down. Such a chip would cost about \$15 in high volume. Based on a DES key search design [44], the overhead of building a machine with many of these chips would be about \$7 per chip. This includes the cost of a hierarchy of controllers and communications path to a central memory of distinguished points. Each chip (plus overhead) costs \$22 and contains 64 processors giving a cost of $u = \$0.34$ per processor. The only cost that has not yet been accounted for is the memory for the distinguished points.

Each memory element must hold a triple consisting of a starting point on a trail, the distinguished point at the end of the trail, and the number of steps taken. The starting point can be represented with 5 bytes because fewer than 2^{40} trails will be produced (as shown in the analysis below), and the starting points can consist of 88 ones (which need not be stored) followed by a 40-bit count. The number of steps can be represented with 5 bytes because trails longer than $20/\theta$ are abandoned (see the optimum value of θ derived below). The distinguished point can be represented in 12 bytes because a hash value is 16 bytes long and, as shown below, distinguished points have at least 4 leading zero bytes which need not be stored. Overall, a triple requires 22 bytes. Assuming that triples are stored in a standard hash table and the memory is limited to 99% full (to limit the number of probes necessary to find an empty element), each triple consumes $22/(0.99)$ bytes of memory. Assuming that memory costs \$50 per Mbyte, we have $v = \$0.0011$.

Given a budget limitation of $B = \$10$ million, the constraint on m (number of processors) and w (number of memory elements) is given by equation (11). To build a machine, we must fix the amount of memory, and it is possible that an MD5 collision will not be found before the memory fills up. In this case, we continue by simply overwriting new distinguished points on top of old ones. After the memory fills up, there are a total of about w/θ points on the trails leading to the distinguished points in the memory. This affects the run-time as shown in Appendix A. Combining the expected run-time of equation (7) with the effect of a memory constraint given by equation (18) with $z = w/\theta$, and taking into account the fact that the probability that a collision is useful is $1/2$, we get a run-time of

$$\left(\left(\sum_{k=0}^{w/\theta-1} e^{-k^2/(4n)} + (2n\theta/w)\, e^{-w^2/(4n\theta^2)}\right)(1/m) + 2.5/\theta\right)t\,.$$

Using numerical techniques to minimize this expression subject to the constraint of equation (11), we find that the run-time is a minimum of 21 days when $\theta = (0.66)2^{-35}$, $m = 2.7\times10^7$, and $w = 8.6\times10^8$. This is a total of 420000 processor chips and 18 Gbytes of memory. The expected total number of trails produced among the processors is $1.2(2^{29})$, which is much less than the 2^{40} allowed for above. The overall rate at which the processors produce distinguished points is $m\theta/t = 405$ per second, which can be accommodated quite easily by the large memory. Because the system could easily handle many times more triples per second, very little buffering is required to handle bursts in the arrival of triples, and the number of triples lost due to bursts would be insignificant.

The run-time of this attack is proportional to the square root of the hash result space. Thus, if this attack were applied to SHA-1 whose hash results are 160 bits long (compared to 128 bits for MD5), the attack would take 2^{16} times longer.

6.3 Known-Plaintext Attack on Double-DES

In this section, the meet-in-the-middle attack method of Section 5.3 is applied to a known-plaintext attack on double-DES. The run-time of a meet-in-the-middle attack is given by equation (8) subject to the constraint of equation (11). Solving (11) for m (the number of processors) and substituting into (8), we get

$$T_{\mathrm{m}} = 7n_2\sqrt{n_1}\,ut/(\sqrt{w}\,(B - vw)) \qquad (12)$$

This is a minimum when $w = B/(3v)$ (one-third of the budget is spent on memory) giving

$$T_{\mathrm{m}} = 18n_2\sqrt{n_1 v}\,ut/B^{3/2} \qquad (13)$$

The most interesting feature of this run-time expression is that the attack time decreases as the money spent to the power of 1.5 (e.g., spending 4 times more money makes the attack 8 times faster). It makes sense that increasing the budget gives better than linear speed-up because the increase in number of processors gives linear speed-up, and we get additional speed-up in detecting collisions as a result of having more distinguished points in a larger memory.

To estimate the run-time for attacking double-DES, estimates are required for each of the constants in equation (13). The budget is B = \$10 million, and the size of the DES key space is $n_1 = n_2 = 2^{56}$. Based on a DES key search design [44], a chip implementing DES encryption with 16 pipeline stages can be made to run a 50 MHz for \$10.50 per chip with an additional \$7 per chip in overhead to house the chips in a machine with a hierarchy of controllers. To attack double-DES, the chip must support DES decryption as well (by supporting a reversed key schedule) and must detect and report distinguished points. This brings the chip cost plus overhead to about \$20. Due to the pipelining, each chip consists of 16 processors giving a cost of u = \$1.25 per processor, and an encryption (or decryption) time of $t = 16/(50 \text{ MHz}) = 0.32\ \mu s$. With 2/3 of the budget, we can buy $m = 5.3\times10^6$ processors or 330000 chips of which 80% are devoted to generating distinguished points, and 20% are devoted to locating collisions.

The analysis below shows that the overall rate at which the memory is accessed is too high for there to be a single interface to the memory. For this reason, the memory is divided into 16 Mbyte segments (organized in 32-bit words) each with its own controller to read and write triples which consist of a starting point on a trail, the distinguished point at the end of the trail, and the number of steps taken. By dividing the memory into segments, we introduce a new problem of how to direct each triple to the correct memory segment. To illustrate how this can be done, consider a small example where there are only 4 memory segments. Each segment is responsible for values which begin with a particular 2-bit pattern. As shown in Figure 2, the processors pass values to a routing stage which then passes the values to one of two other routing chips based on one of the bits in the value. The next routing stage passes the value to one of two memory controllers based on another bit of the value. Each routing chip contains a buffer (memory) to deal with peaks in the supply of values.

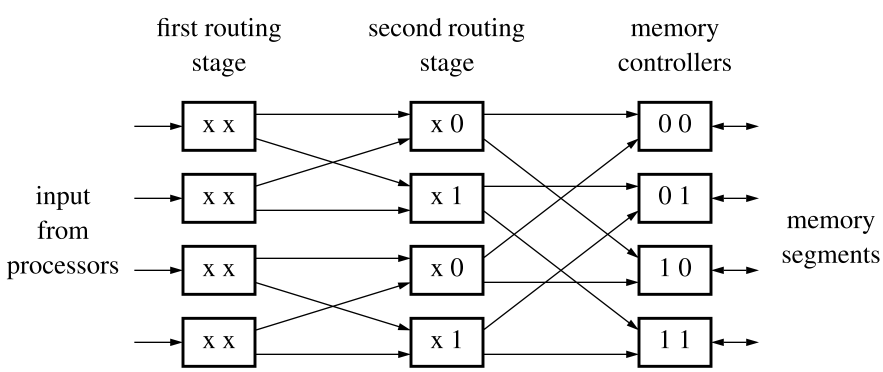

Figure 2. Example of Routing Values to Memory Segments

In general, k routing stages are required for 2^k memory segments. For the problem at hand, 12 routing stages suffice (as shown below). Each routing chip contains some memory to deal with an uneven supply of triples. It is no great loss if all routing chips between a processor and a memory segment have their buffers fill up once in a while and a small proportion of triples are lost.[1] Each chip in the first routing stage collects triples from several processors. Each subsequent stage uses a different bit of the distinguished point to direct the triple toward the correct memory segment. Allowing \$10 per routing chip, \$20 for a memory controller, and \$100 to house 12 routing chips and a memory controller in a machine, the overall cost of 16 Mbyte memory segment is \$800 (at \$50 per Mbyte) + \$240 = \$1040. One third of the budget can buy 50 Gbytes or 3200 such memory segments (which is less than 2^{12} so that 12 stages of routing chips to direct triples to memory suffice). Because a triple requires 12 bytes (as shown next), this memory can hold $w = 4.5 \times 10^9$ triples. With $n = 2^{57}$, we have $\theta = 2.25\sqrt{w/n} = 0.81(2^{-11})$. The starting point of a triple can be represented with 5 bytes because $10w$ trails are produced per version of the iterating function and $w < 2^{40}/10$. The 57-bit starting points can consist of 17 ones followed by a 40-bit count. The number of steps taken can be represented with 2 bytes because trails longer than $20/\theta$ ($< 2^{16}$) are abandoned. The distinguished point can be represented with 5 bytes because each point consists of 57 bits of which 11 are leading zeros due to the distinguishing property and another 11 are implied by the memory segment that the triple is stored in, leaving 35 bits. One of the remaining 5 bits could be used to indicate whether a memory element contains a triple or not. Overall, a triple requires 12 bytes (three 32-bit words). With a cost of \$1040 per 16 Mbytes, the cost of a memory element is $v = \$0.00074$.

For each distinguished point sent to a memory segment, the memory controller reads a triple from a memory element and then writes the new triple into the memory element. In Section 4.2, it was shown that of $10w$ trails generated, $1.3w$ collisions are produced. This means that 13% of the time it is necessary to read all three 32-bit words from memory and 87% of the time only one word must be read (for a negligible fraction of the time, two words are read if most of a stored distinguished point matches the new one, but the second read determines that there is no match). It is always necessary to write all three words of the new triple. This gives a mean of 4.26 memory accesses for each triple. If the distinguished point is already in the memory, there is a collision and the two triples are passed to the processors which locate collisions. The overall rate at which distinguished points are generated is $(80\%) m\theta/t = 5.3 \times 10^9$ per second. These are spread across 3200 memory segments and each requires 4.26 memory accesses (on average) giving an average of 140 ns between memory accesses in each segment. This is more time than is required to access modern memory, and is thus realistic. The buffering in the 12 routing stages between processors and memory will smooth out peak demands on a memory segment so that few triples are lost.

---

[1] The design requirements are much more relaxed than for a highly parallel general-purpose super-computer. For the problem at hand, data flows in only one direction (from processors which generate trails to memory or from memory to processors which locate collisions) and some data can be lost without affecting the correctness of the algorithm. For general-purpose machines, data flows in both directions so that transit time is important (not just throughput), and no data can be lost.

Substituting the constant values into equation (13), we get a total run-time of 4 years. This is about 2^{17} times as long as the 21 minutes required to attack single-DES with a \$10 million machine [44]. In Section 5.3, it was determined that meet-in-the-middle attacks based on parallel collision search are $0.07\sqrt{n_1 / w} = 280$ times faster than a conventional approach. This does not take into account the fact that a basic step of the new algorithm is a 20 ns pipeline stage compared to a basic step of four 32-bit memory accesses (about 320 ns) for the conventional approach, giving another factor of 16. There is also the fact that there is greater cost in the conventional approach due to having to split up the memory into smaller segments to handle the memory access rate. Overall, the new approach performs a known-plaintext attack on double-DES about four orders of magnitude faster than the conventional approach. This is based on an attacker having a budget of \$10 million. For a lower budget, the advantage is greater, and vice-versa (see [43, Table 1]).

7. Conclusion

Obvious methods of parallelizing collision search algorithms do not give linear speedup; when m processors are used, collision search is only $\sqrt{m}$ times faster than when one processor is used. The new method for parallelizing Pollard's rho-method presented herein, based on the use of distinguished points, gives linear speedup.

Parallel collision search applies to a wide range of cryptanalytic problems. One such problem is performing discrete logarithms in cyclic groups. For some groups where there is no known subexponential attack, such as certain elliptic curve groups, parallel collision search gives the most efficient known method of performing discrete logarithms. This discrete logarithm technique can also be used for direct attacks on the small subgroup used in the Schnorr signature scheme and DSA. Pollard's lambda-method for discrete logarithms (which is useful when the logarithm is known to lie in a restricted interval) can also be parallelized with the use of distinguished points. Even when only one processor is used, the new lambda method is 1.64 times faster than the previously best variation.

Parallel collision search also gives the best known method of finding collisions between meaningful messages for many hash functions including MD5, RIPEMD-160, SHA-1, MDC-2, and MDC-4. The only details of the specific hash function which enter into such an analysis are the bitsize of the hash result, and the speed and cost of a hardware implementation possible with current technology. The natural conclusion from the analysis is that 128-bit hash results are, depending on the application, too small to provide adequate security for the future, and perhaps marginal even for today. When greater security is required, SHA-1 or RIPEMD-160 [16] may be used.

A general construction for performing meet-in-the-middle attacks using parallel collision search was given. This gives a large speed improvement over the standard approach to meet-in-the-middle attacks, and gives the best attacks known on double encryption and three-key triple encryption.

To illustrate the use of parallel collision search for practical cryptanalytic problems, designs were given for three \$10 million custom machines which could be built with current technology: one finds elliptic curve logarithms in GF(2^{155}) thereby defeating a proposed elliptic curve cryptosystem in expected time 32 days; the second finds MD5 collisions in expected time 21 days; and the last recovers a double-DES key from two known plaintexts in expected time 4 years, which is more than 10000 times faster than the conventional meet-in-the-middle attack on double-DES. Based on the new attack, double-DES offers about 17 bits more security than single-DES. Because the new attack has faster than linear speedup as machine size increases, the advantage of double-DES over single-DES shrinks as the budget for the attack increases or as technology becomes faster and cheaper.

Acknowledgments

We would like to thank John Pollard for correcting a note about [4] and for clarifying a fundamental difference between his two rho-methods. We would also like to thank Bart Preneel for his helpful suggestions, Burt Kaliski for discussions regarding multiple encryption, Kevin McCurley for alerting us to reference [8], and Don Coppersmith for an observation about the algorithm for finding a large number of collisions which simplified its description and made it about $\sqrt{2}$ times faster.

Appendix A: Expected Number of Steps to a Collision

This appendix sketches a derivation of the expected number of elements of a set of size n that must be selected at random with replacement before any element is selected twice. The case where the number of elements stored and available for collision is limited is considered as well.

Lemma 1. Let X be the random variable for the number of elements selected before duplication. Then $\mathrm{E}(X) \approx \sqrt{\pi n/2}$.

Proof.
$$\Pr(X > k) = (1-1/n)(1-2/n)\ldots(1-(k-1)/n) \approx e^{-k^2/(2n)} \tag{14}$$

for large n and $k = O(\sqrt{n})$ [31].

$$\mathrm{E}(X) = \sum_{k=1}^{\infty} k \cdot \Pr(X = k) = \sum_{k=1}^{\infty} k \cdot (\Pr(X > k-1) - \Pr(X > k)) = \sum_{k=0}^{\infty} \Pr(X > k) \tag{15}$$

Therefore, the expected number of elements chosen before duplication is

$$E(X) \approx \sum_{k=0}^{\infty} e^{-k^2/(2n)} \approx \int_0^{\infty} e^{-x^2/(2n)}\,dx = \sqrt{\pi n/2} \tag{16}$$

The error in approximating the sum with the integral is at most 1 because the function is monotonically decreasing and never exceeds 1. The integral is a standard definite integral. This result is also given by Flajolet and Odlyzko [20] (see also Knuth [24, p.8, ex.12]). ❑

For an implementation where one seeks a duplicated element, all previous elements must be stored. But, the actual amount of memory available for a real attack will be limited. Let z be the number of elements which can be stored. After the memory fills, only collisions with the z stored elements can be detected, not collisions with elements which have been overwritten in the memory.

Lemma 2. Let Y be the random variable for the number of elements selected before duplication is detected when the memory size is z. Then

$$E(Y) = \sum_{k=0}^{z-1} e^{-k^2/(2n)} + (n/z)e^{-z^2/(2n)} .$$

Proof. This distribution is the same as that of X up to $k = z$, after which the probability of collision is z/n on each step.

$$\Pr(Y > k) \approx \begin{cases} e^{-k^2/(2n)} & \text{if } (k < z) \\ (1 - z/n)^{k-z} e^{-z^2/(2n)} & \text{otherwise} \end{cases} \tag{17}$$

$$E(Y) = \sum_{k=0}^{z-1} e^{-k^2/(2n)} + (n/z)e^{-z^2/(2n)} \tag{18}$$

This completes the proof. ❑

Appendix B: Expected Maximum of Two Geometric Distributions

This appendix proves a lemma about the expected maximum of two geometric distributions. Let X_1 and X_2 be two independent random variables with geometric distribution and mean $1/p$ for some probability p. Then $\Pr(X_i = k) = p(1-p)^{k-1}$ and $\Pr(X_i < k) = 1 - (1-p)^{k-1}$ for $k \geq 1$ and $i = 1,2$ (for an introduction to geometric probability distributions, see Blake [6] for example).

Lemma 3. Let Y be the random variable equal to the maximum of X_1 and X_2. Then $E(Y) \approx 1.5/p$ for p small.

Proof.
$$\begin{aligned} \Pr(Y = k) &= \Pr(X_1 < k)\cdot\Pr(X_2 = k) + \Pr(X_2 < k)\cdot\Pr(X_1 = k) + \Pr(X_1 = k)\cdot\Pr(X_2 = k) \\ &= 2p(1-p)^{k-1} - (2p - p^2)(1-p)^{2(k-1)} \end{aligned}$$

$$\mathrm{E}(Y) = \sum_{k=1}^{\infty} k \cdot \left(2p(1-p)^{k-1} - (2p-p^2)(1-p)^{2(k-1)}\right)$$

Making use of the identity $\sum_{k=1}^{\infty} ka^{k-1} = 1/(1-a)^2$ for $0 < a < 1$,

$$\mathrm{E}(Y) = 2/p - 1/(2p - p^2) \approx 1.5/p \quad \text{for } p \text{ small.} \qquad \square$$

References

[1] G.B. Agnew, R.C. Mullin, and S.A. Vanstone, "An implementation of elliptic curve cryptosystems over $F_{2^{155}}$", *IEEE J. Selected Areas in Communications*, vol. 11, no. 5 (June 1993), pp. 804-813.

[2] G.B. Agnew, R.C. Mullin, and S.A. Vanstone, "On the Development of a Fast Elliptic Curve Cryptosystem", *Lecture Notes in Computer Science 658: Advances in Cryptology - Eurocrypt '92 Proceedings*, Springer-Verlag, pp. 482-487.

[3] H.R. Amirazizi and M.E. Hellman, "Time-Memory-Processor Tradeoffs", *IEEE-IT*, vol.34, no. 3 (1988), pp. 505-512.

[4] E. Bach, "Toward a Theory of Pollard's Rho Method", *Information and Computation*, vol. 90 (1991), pp. 139-155.

[5] A. Bosselaers and B. Preneel, editors, "Integrity Primitives for Secure Information Systems: Final Report of RACE Integrity Primitives Evaluation (RIPE-RACE 1040)", *Lecture Notes in Computer Science 1007*, Springer-Verlag, 1995.

[6] I.F. Blake, "An Introduction to Applied Probability", Wiley, New York, 1979.

[7] R.P. Brent, "An improved Monte Carlo factorization algorithm", *BIT* 20 (1980), pp. 176-184.

[8] R.P. Brent, "Parallel algorithms for integer factorization", London Mathematical Society Lecture Note Series vol. 154, *Number Theory and Cryptography*, J.H. Loxton (ed.), pp. 26-37, Cambridge University Press, 1990.

[9] K.W. Campbell and M.J. Wiener, "DES is not a Group", *Lecture Notes in Computer Science 740: Advances in Cryptology - Crypto'92 Proceedings*, Springer-Verlag, pp. 512-520.

[10] "Data Encryption Standard", National Bureau of Standards (U.S.), Federal Information Processing Standards Publication (FIPS PUB) 46, National Technical Information Service, Springfield VA, 1977.

[11] "Digital Signature Standard", NIST, U.S. Department of Commerce, Federal Information Processing Standards Publication (FIPS PUB) 186, May 1994.

[12] D.E. Denning, *Cryptography and Data Security*, Addison-Wesley, 1982.

[13] W. Diffie and M. Hellman, "Exhaustive cryptanalysis of the NBS Data Encryption Standard", *Computer* vol.10 no.6 (June 1977) pp. 74-84.

[14] B. Dixon and A.K. Lenstra,."Factoring Integers Using SIMD Sieves", *Lecture Notes in Computer Science 765: Advances in Cryptology - Eurocrypt '93*, Springer-Verlag, pp. 28-39.

[15] H. Dobbertin, "Cryptanalysis of MD4", *Lecture Notes in Computer Science 1039, Fast Software Encryption, Third Annual Workshop*, Springer, 1996, pp. 53-69.

[16] H. Dobbertin, A. Bosselaers, and B. Preneel, "RIPEMD-160: A Strengthened Version of RIPEMD", *Lecture Notes in Computer Science 1039, Fast Software Encryption, Third Annual Workshop*, Springer, 1996, pp. 71-82.

[17] S. Even and O. Goldreich, "On the Power of Cascade Ciphers", *ACM Transactions on Computer Systems*, vol. 3, no. 2, May 1985.

[18] A. Fiat and M. Naor, "Rigorous Time/Space Tradeoffs for Inverting Functions", manuscript. Preliminary version in *Proc. 23rd ACM STOC* 1991, pp. 534-541.

[19] R. Golliver, A.K. Lenstra, and K.S. McCurley, "Lattice Sieving and Trial Division", presented at the Algorithmic Number Theory Symposium (ANTS'94), Cornell University, May 1994.

[20] P. Flajolet and A.M. Odlyzko, "Random Mapping Statistics", *Lecture Notes in Computer Science 434: Advances in Cryptology - Eurocrypt '89 Proceedings*, Springer-Verlag, pp. 329-354.

[21] R. Heiman, "A note on discrete logarithms with special structure". *Lecture Notes in Computer Science 658: Advances in Cryptology - Eurocrypt '92*, Springer-Verlag, pp. 454-457.

[22] M.E. Hellman, "A cryptanalytic time-memory trade-off", *IEEE-IT*, vol.6 (1980), pp. 401-406.

[23] B.S. Kaliski Jr., R.L. Rivest, and A.T. Sherman, "Is the Data Encryption Standard a Group? (Results of Cycling Experiments on DES)", *J. Cryptology*, vol. 1, no. 1 (1988), pp. 3-36.

[24] D.E. Knuth, *The Art of Computer Programming, vol. 2: Seminumerical Algorithms*, 2nd edition, Addison-Wesley, 1981.

[25] D.E. Knuth, *The Art of Computer Programming, vol. 3: Sorting and Searching*, Addison-Wesley, 1973.

[26] A.K. Lenstra, H.W. Lenstra, M.S. Manasse, and J.M. Pollard, "The Factorization of the ninth Fermat Number", *Math. Comp.* vol. 61 (1993), pp. 319-349.

[27] A.K. Lenstra and M.S. Manasse, "Factoring by electronic mail", *Lecture Notes in Computer Science 434: Advances in Cryptology - Eurocrypt '89 Proceedings*, Springer-Verlag, pp. 355-371.

[28] K.S. McCurley, "The discrete logarithm problem", pp. 49-74 in *Cryptology and Computational Number Theory*, Proc. Symp. Applied Math., vol. 42 (1990), American Math. Society.

[29] R. Merkle and M. Hellman, "On the Security of Multiple Encryption", *Communications of the ACM*, vol. 24, no. 7, July 1981, pp. 465-467. See also *Communications of the ACM*, vol. 24, no. 11, Nov. 1981, p. 776.

[30] C.H. Meyer and M. Schilling, "Secure Program Load with Modification Detection Code", *Proc. of 6th Worldwide Congress on Computer and Communications Security and Protection (SECURICOM '88)*, Paris, France, March 1988, pp. 111-130.

[31] K. Nishimura and M. Sibuya, "Probability to meet in the middle", *J. Cryptology*, vol. 2, no. 1 (1990), pp. 13-22.

[32] S.C. Pohlig and M.E. Hellman, "An improved algorithm for computing discrete logarithms over GF(p) and its cryptographic significance", *IEEE-IT*, vol. 24 (1978), pp. 106-110.

[33] J.M. Pollard, "A Monte Carlo method for factorization", *BIT*, vol. 15 (1975), pp. 331-334.

[34] J.M. Pollard, "Monte Carlo Methods for Index Computation (mod p)", *Math.Comp.* vol. 32, no. 143, July 1978, pp. 918-924.

[35] J.-J. Quisquater and J.-P. Delescaille, "How easy is collision search? Application to DES", *Lecture Notes in Computer Science 434: Advances in Cryptology - Eurocrypt'89 Proceedings*, Springer-Verlag, pp. 429-434.

[36] J.-J. Quisquater and J.-P. Delescaille, "How easy is collision search. New results and applications to DES", *Lecture Notes in Computer Science 435: Advances in Cryptology - Crypto'89 Proceedings*, Springer-Verlag, pp. 408-413.

[37] R. Rivest, "The MD5 Message-Digest Algorithm", Internet RFC 1321, April 1992.

[38] C.P. Schnorr, "Efficient Signature Generation by Smart Cards", *J. Cryptology*, vol. 4, no. 3 (1991), pp. 161-174.

[39] "Secure Hash Standard", NIST, U.S. Department of Commerce, Federal Information Processing Standards Publication (FIPS PUB) 180-1, April 1995.

[40] R. Sedgewick, T.G. Szymanski, and A.C. Yao, "The complexity of finding cycles in periodic functions", *Siam J. Computing*, vol. 11, no. 2, 1982, pp. 376-390.

[41] P.C. van Oorschot and M.J. Wiener, "A Known-Plaintext Attack on Two-Key Triple Encryption", *Lecture Notes in Computer Science 473: Advances in Cryptology - Eurocrypt'90 Proceedings*, Springer-Verlag, pp. 318-325.

[42] P.C. van Oorschot and M.J. Wiener, "Parallel Collision Search with Application to Hash Functions and Discrete Logarithms", *2nd ACM Conference on Computer and Communications Security*, Fairfax, Virginia, November 1994, pp. 210-218.

[43] P.C. van Oorschot and M.J. Wiener, “Improving Implementable Meet-in-the-Middle Attacks by Orders of Magnitude”, *Lecture Notes in Computer Science 1109: Advances in Cryptology - Crypto’96 Proceedings*, Springer, pp. 229-236.

[44] M.J. Wiener, “Efficient DES Key Search”, Bell-Northern Research. Filed for general accessibility as TR-244 (May 1994), School of Computer Science, Carleton University, Ottawa, Canada. Presented at the Rump Session of Crypto ’93.

[45] G. Yuval, “How to Swindle Rabin”, *Cryptologia*, vol. 3, no. 3, July 1979, pp. 187-189.