Parallel Collision Search with Application to Hash Functions and Discrete Logarithms

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

Bell-Northern Research, P.O. Box 3511 Station C, Ottawa, Ontario, K1Y 4H7, Canada

1994 August 17

Abstract

Current techniques for collision search with feasible memory requirements involve pseudo-random walks through some space where one must wait for the result of the current step before the next step can begin. These techniques are serial in nature, and direct parallelization is inefficient. We present a simple new method of parallelizing collision searches that greatly extends the reach of practical attacks. The new method is illustrated with applications to hash functions and discrete logarithms in cyclic groups. In the case of hash functions, we begin with two messages; the first is a message that we want our target to digitally sign, and the second is a message that the target is willing to sign. Using collision search adapted for hashing collisions, one can find slightly altered versions of these messages such that the two new messages give the same hash result. As a particular example, a \$10 million custom machine for applying parallel collision search to the MD5 hash function could complete an attack with an expected run time of 24 days. This machine would be specific to MD5, but could be used for any pair of messages. For discrete logarithms in cyclic groups, ideas from Pollard's rho and lambda methods for index computation are combined to allow efficient parallel implementation using the new method. As a concrete example, we consider an elliptic curve cryptosystem over $GF(2^{155})$ with the order of the curve having largest prime factor of approximate size 10^{36}. A \$10 million machine custom built for this finite field could compute a discrete logarithm with an expected run time of 36 days.

1. Introduction

The power of parallelized attacks has been illustrated in recent work on integer factorization and cryptanalysis of the Data Encryption Standard (DES) [7]. In the factoring of the RSA-129 challenge number and other factoring efforts (e.g. [14, 15]), the sieving process was distributed among a large number of workstations. Similar efforts have been undertaken on large parallel machines [8, 9]. In a recent exhaustive key search attack proposed for DES [24], a large number of inexpensive specialized processors were proposed to achieve a high degree of parallelism. The importance of producing memoryless versions of attacks is well recognized (e.g. [6, 21]), but even a memoryless attack is of little practical use unless it can be efficiently parallelized. In this paper, we provide a method for efficient parallelization of collision search techniques.

The remainder of this paper is organized as follows. Section 2 reviews general collision search techniques and presents a new parallel method. This new method allows, for the first time, efficient use of parallelization in collision search and greatly extends the reach of practical attacks. Section 3 illustrates the application of parallel collision search by adapting it to finding hash function collisions; we choose as an illustrative example the well-known hash function MD5 [22] and propose a high-level design for a machine to find MD5 collisions in hardware. For \$10 million,[1] one could expect to find a collision in 24 days. This was not possible by previous techniques whereby the efficiency of parallelization was $O(\sqrt{m})$, rather than $O(m)$ for m processors. Section 4 discusses using parallel collision search to find discrete logarithms in cyclic groups. As an illustrative example we consider an elliptic curve cryptosystem over $GF(2^{155})$ with the order of the curve having largest prime factor of size 10^{36}, and again give a

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

high-level hardware design to attack this system. A \$10 million custom built machine could complete a logarithm with an expected time of 36 days. No efficient algorithm has previously appeared in the literature for parallelizing the computation of discrete logarithms in cyclic groups; again, the most efficient previously documented algorithms offered $O(\sqrt{m})$ speedup for m processors, rather than linear speedup. Section 5 contains concluding remarks.

2. Parallel Collision Search

In this section we first review known methods for collision search, and then describe how to efficiently parallelize this task. The goal in collision search is to take a given 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. We make the reasonable assumption that f is sufficiently complex that it behaves like a random mapping.

An obvious method for finding a collision is to select distinct inputs x_i for $i = 1, 2, \ldots$ and check for a collision in the $f(x_i)$ values. Let n be the cardinality of the range of f. The probability that no collision is found after selecting k inputs is $(1-1/n)(1-2/n)\ldots(1-(k-1)/n) \approx e^{-k^2/(2n)}$ for large n and $k = O(\sqrt{n})$ [17]. The expected number of inputs that must be tried before a collision is found is $\sqrt{\pi n/2}$ [10]. Assuming that the $f(x_i)$ values are stored in a hash table so that new entries can be added in 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 Pollard's rho method [19, 20]. 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: $S \rightarrow S$).[1] 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}. Then 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 [12, ex 3.1-6]. 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.[2] 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 after it starts [23]. Quisquater and Delescaille took a different approach based on storing *distinguished points* [21]. A distinguished point is one that has some easily checked property such as having a 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.

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 start. One way to parallelize this algorithm is to start each processor with a different value x_0 and wait until one of them finds a collision. However, the classical occupancy distribution applicable to collision search leads to poor effectiveness of this approach. If there are m processors, the probability that no collision has occurred for any processor after selecting k inputs is $[(1-1/n)(1-2/n)\ldots(1-(k-1)/n)]^m \approx e^{-k^2 m/(2n)}$. 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

[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 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 $|D|/|R|$.

[2] 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 [19, 20]) it is sufficient that there are two distinct paths to the same point.

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 inefficient use of parallelization can be overcome by modifying the method of distinguished points as follows. We start all processors at their own value x_0 and as they encounter distinguished points, these points are contributed to a single common list for all processors. 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. As soon as any trail touches another trail, the two trails will coincide from that point on and the two processors involved will produce the same list of distinguished points thereafter (see processors 3 and 4 in Figure 1). In general, such a collision is due to coincident paths rather than traversing a cycle as in Pollard's rho method.[1] If additional collisions are desired, processor 3 can be restarted at a new value x_0 and one can wait for some other pair of trails to collide. Because a collision can be detected between any two points produced by any of the m processors, the run time analysis is similar to the single processor case, except that the points are produced m times faster. The expected number of steps taken by each processor is $\sqrt{\pi n/2}/m$.

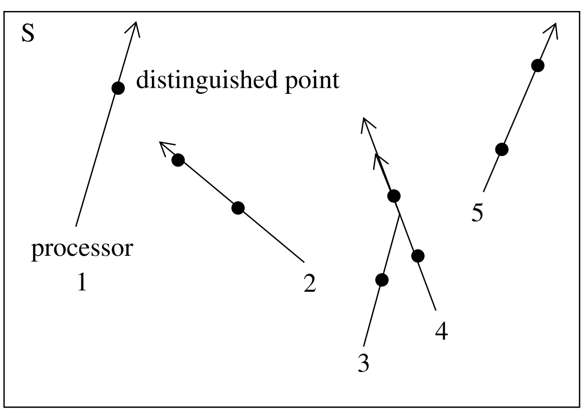

Figure 1: Parallelized Collision Search

Quisquater and Delescaille used an idea similar to this parallel collision search in finding DES collisions [21]. They had to contend with "pseudo-collisions" caused by a mapping g which could not be injective because the size of the DES key space is 2^{56}, which is 2^8 times smaller than its message space. They ran several trails with different starting points within the same processor. All trails 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 leading to 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 suffered by standard techniques due to 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, our new technique described above provides a speedup by a factor m for m processors.

3. Finding (Real) Collisions in Hash Functions

In this section we apply the parallel collision search technique to finding real collisions in hash functions. We first review how hash functions are typically used in conjunction with digital signatures, and the classic attack of Yuval. We then apply parallel collision search to extend this attack allowing parallelization and reducing memory requirements, and consider as an example the impact on the MD5 hash function.

Hash functions are designed to take a message consisting of an arbitrary number of binary bits and map it to a fixed size output called a hash result. Let $H: M \rightarrow R$ be such a hash function. Typically, hash functions are constructed from a function $h: B \times R \rightarrow R$ 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

[1] In fact, the method of distinguished points conceptually resembles Pollard's lambda method (discussed later), where two sequences of points are computed with the hope that at some (union) point they coincide, whereafter the sequences are identical.

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 moved from one message to the other.

Now suppose we have a message m that we would like our adversary 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 [25]. Choose some other message m' that the adversary 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 adversary will sign the version of m', and we can move 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: $R \to M$ 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, each bit of a hash result may correspond to one possible modification of the message m. 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: $R \to R$ as follows, with m and m' as described above as implicit constants.

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

Using the parallel collision search technique from section 2, find pairs of hash results a and b such that $f(a) = f(b)$, but $a \neq b$. There is about a 50% chance that random hash results a and b are in different subsets of R. The collision search is not interrupted until a collision is verified to have a and b in different subsets S_i, S_j of R. Without loss of generality, 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). The probability that we fail to find the desired type of collision after a total of k iterations of f among all processors is approximately[1] $(1-1/2n)(1-2/2n)\dots(1-(k-1)/2n) \approx e^{-k^2/(4n)}$. The expected total number of iterations required among all processors is $\sqrt{\pi n}$, or $\sqrt{\pi n}/m$ iterations per processor for m processors (recall that n is the cardinality of the hash function space).

Let us now apply these techniques to the MD5 hash function [22]. With MD5, the hash result is 128 bits, and messages are divided into 512-bit blocks. To reduce the amount of computation required, we will assume that the effect of the mappings g_m and $g_{m'}$ can be confined to a single block within the messages. We justify this as follows. For g_m and $g_{m'}$ to be injective, we must be able to code 128 bits of information into a 512-bit message block. One way to do this would be to find four non-printable characters and code two bits per byte. Another way is to hide data within a 512-bit segment of a word processor file in such a way that the appearance of the file is not altered.

The message m consists of the blocks $m_1, \dots m_l$. Message m' consists of the blocks $m'_1, \dots m'_j, m_{j+1}, \dots m_l$. 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: $R \to B$ which maps a hash result to a 512-bit message block; this assumes we are free to manipulate the 512-bit blocks m_j and m'_j in their entirety. Then use the following function f for collision search.

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

An internal study [4] 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 would be computing 64 independent sequences with different starting points. Each sequence would have evaluations of f performed at 1/64 of 50 MHz. Overall, there would be 50 million evaluations of f per second. The chips would also

[1] This expression is not quite precise (even for perfectly random functions f) as it does not account for the fact that "undesirable" collisions result in two identical points being in the pool of generated points, and the second of these does not increase the chance of future collisions. However, as the expected number of undesirable collisions prior to a desirable collision is 1 (as follows from the 50% chance noted earlier), this imprecision is negligible.

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 recent DES key search design [24], 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. The only cost that has not yet been accounted for is the memory for the distinguished points.

For a \$10 million budget, we suggest an appropriate distinguishing property for hash results is having 36 leading zero bits. We expect to perform $2^{64}\sqrt{\pi}$ iterations of f before a collision occurs. Only about 2^{-36} of these will give distinguished points. For storage, we need 22 bytes per distinguished point; this includes 12 bytes for the hash value (leading zero bits are not stored), four bytes to identify the sequence of points, and six bytes for the number of steps taken in the sequence (allowing for 2^{48} steps which is more than is required). The number of steps taken in the sequence will be used later (see below) to locate the collision. Allowing for a factor of two in memory size in case the search takes longer than expected and a factor of 1.5 for overhead due to using a hash table for the list of distinguished points, we need about 30 Gbytes of memory. (Note that this is not impractical; the existing general purpose parallel machine of [9] has 37 Gbytes of memory.) Assuming that the cost of memory and the boards to house it is \$75/Mbyte, the memory cost is \$2.25 million. At \$22 per chip (including overhead), the remaining \$7.75 million can buy about 350 000 chips.

The total search time will consist of three components: the times for the collision to occur, be detected, and then located. A collision is expected to occur after a total of $2^{64}\sqrt{\pi}$ iterations of f divided among all sequences. After the desired collision occurs, all sequences will continue until the colliding sequence encounters a distinguished point to allow the collision to be detected. The distribution of this number of steps is geometric with mean 2^{36}. Finally, we must locate the collision as follows. Find the previous distinguished points produced by the two colliding sequences in the central list (a relatively fast operation even if performed by linear search). Using the count of the number of steps taken that is stored with each distinguished point, step one of the previous distinguished points forward until the counts of the previous points differ by the same amount as the counts of the colliding distinguished points. Then the two previous points are stepped forward until they produce equal outputs. (We assume that we have one or two special chips which are designed to produce two independent sequences and compare their outputs.) On the step before equality occurs, we have located the collision values. The run time of locating the collision is the maximum of two geometric distributions, and has an expected run time of $(1.5)2^{36}$ steps. Note that the main collision search is not halted until a useful collision is found. The total expected run time for 350 000 chips with 64 levels of pipelining is [1]

$$\frac{64}{50\text{ MHz}}\left(\frac{2^{64}\sqrt{\pi}}{(350000)\ (64)} + 2^{36}\,(1+1.5)\right) \cong 24\text{ days}$$

Thus, a \$10 million machine would take 24 days to find two useful messages that MD5 hashes to the same value.

4. Application to Discrete Logarithms in Cyclic Groups

In this section we apply the parallel collision search technique to discrete logarithms in cyclic groups. We first review known methods for computing discrete logarithms in cyclic groups and then discuss the use of parallel collision search on this problem.

Known methods for computing logarithms in cyclic groups. Recall that the powerful index-calculus techniques, which can be applied to groups with additional structure, do not apply to arbitrary cyclic groups. For the latter, the well-known “baby-step giant-step” algorithm, attributed to Shanks [13, pp. 9, 575], allows one to compute discrete logarithms in a cyclic group G of order n in deterministic time $O(\sqrt{n}\log n)$ and space for $\sqrt{n}$ group elements. If one uses hashing instead of sorting at a particular stage (described below), the running time falls to $O(\sqrt{n})$ steps; here, one step is a group operation. If p is the largest prime divisor of n, Shanks’ algorithm can be generalized to run in deterministic time and space $O(\sqrt{p})$ [20]; moreover, space can be traded off against time [18], which may be of interest if one anticipates computing a large number of logarithms.

Pollard’s lesser-known rho-method for discrete logarithms [20], based on the same theory as his rho-method for factoring [19], also has time complexity $O(\sqrt{p})$ (originally given as heuristic time rather than deterministic, but

[1] Summing the expected values of the three components of the run time ignores the possibility that a second collision which occurs later than the first will actually be detected and located earlier. The actual expected run time will be slightly less than the calculated value.

rigorously proven by Bach [3]), with only negligible space requirements; it is thus preferable. This method requires knowledge of the (exact) order of the group, whereas Shanks' method requires only an upper bound. Pollard's rarely discussed lambda-method for computing discrete logarithms can be used when the logarithm sought is known to lie in a specified interval [20]; this algorithm may terminate (with controllably small probability) without yielding the answer. More specifically, it yields the log x of a group element in time $O(\sqrt{w})$ and space for $O(\log w)$ group elements, provided one knows that $b < x < b+w$ with integers b and w known. Shanks' method, similarly modified to apply to a restricted interval of width w, can guarantee success in similar running time, but requires space for $\sqrt{w}$ group elements; again time could be traded off for space. Pollard's rho-method does not seem adaptable to restricted intervals (thus motivating the lambda-method). Other refinements are possible if it is known that the logarithm being sought belongs to a group with special structure (e.g. [11]).

Direct parallelization of known methods for logarithms in cyclic groups. Regarding parallelization of discrete log algorithms for cyclic groups, little work has been done. We now briefly review the known results, and obvious parallelizations of previous methods. In discussing the rho-method for factorization, Brent notes that "Unfortunately, parallel implementation of the "rho" method does not give linear speedup" [5, p. 29]. He continues

> "A plausible use of parallelism is to try several different pseudorandom sequences (generated by different polynomials f). If we have m processors and use m different sequences in parallel, the probability that the first k values in each sequence are distinct mod p is approximately $\exp(-k^2m/2p)$, so the speedup is $O(\sqrt{m})$ … ."

Analogous comments apply to the rho-method for computing logarithms. Note that here each parallel processor is working independently of the others (rather than interactively), and does not increase the probability of success of any other processor. This is the best previously reported use of parallelization for computing logarithms in cyclic groups.

A naive parallelization of Pollard's lambda-method among m processors is as follows. If the logarithm is known to lie in an interval of width w, assign each of m processors to search disjoint subintervals of width w/m. The running time for each processor is then proportional to $\sqrt{w/m}$, yielding speedup by a factor of $\sqrt{m}$. Recall the objective of m-fold parallelization is a speedup by a factor linear in m, ideally by m itself; thus the above parallelization is inefficient.

We now consider a naive parallelization of Shanks' method. The basic method for the cyclic group $GF(p)^*$ with generator g finds the logarithm of a group element $y \equiv g^x \pmod p$ as follows. Set $t = \lceil \sqrt{p} \rceil$. The process consists of two phases, with Phase 1 a one-time computation and Phase 2 carried out for each logarithm. Phase 1: compute g^{ti} for $0 \le i < t$; order these elements (either by sorting or hashing). Phase 2: compute yg^j for $0 \le j < t$; check for a common element with the first list by either sorting or conventional hashing. (In the latter case, each Phase 2 element is not actually inserted into the hash table, but an attempt is made to do so to check whether it is already present.) A common element is guaranteed, yielding $x \equiv ti - j \pmod{p-1}$. Phase 2 can be subdivided among m processors by subdividing the range of j into m subintervals of width t/m. Each parallel processor will complete Phase 2 in at most t/m steps, for a Phase 2 speedup factor of m; one of the processors will find the desired collision. A similar speedup can be achieved for Phase 1. However, one would still require memory for $t = \lceil \sqrt{p} \rceil$ group elements and a parallel method of writing to this memory in Phase 1 and reading from it in Phase 2 without slowing down the processors. As before, time and space can be traded off, but due to the above issues, this method remains impractical for large groups.

A new parallelized discrete log algorithm for cyclic groups. We now discuss how to adapt the parallel collision search techniques of Section 2 to the problem at hand. For a cyclic group of order n, suppose one wants to find the discrete logarithm of a group element with respect to some generator. If n has a known factorization into prime powers, this problem can be reduced to one of finding c discrete logarithms in a subgroup of size p for each prime power p^c dividing n (e.g., see [18] or [16]). For the reduced problem, we have $y = g^x$, where g generates a subgroup of order p, y is known, and we wish to find x. When p is small, one can simply compute all of the subgroup elements and compare them to y, or use the single processor version of Pollard's rho method. For large primes, one can use the parallelized rho method as now described.

In a parallel version of the rho method for logarithms, we suggest the same iterating function used by Pollard [20]. Partition the set of subgroup 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 follows the following steps independently. Choose random exponents $a_0, b_0 \in [0, p)$ and use the starting point $x_0 = g^{a_0}y^{b_0}$. [1] Compute the sequence defined above keeping track of the exponents of g and y at each step ($x_i = g^{a_i}y^{b_i}$). When x_i is a distinguished point, contribute the triple (x_i, a_i, b_i) to a list common to all processors. The expected number of steps for each of m processors before a collision occurs is $\sqrt{\pi p/2}/m$. Shortly after the 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 $g^s = y^t$, where $s \equiv a-c \pmod p$ and $t \equiv d-b \pmod p$. Provided $t \not\equiv 0 \pmod p$, the desired logarithm can be computed as $\log_g y \equiv s \cdot t^{-1} \pmod p$; otherwise, the collision is not useful and the search must continue. Based on randomness assumptions, the probability that $t \equiv 0 \pmod p$ is very small if p is large enough to warrant using parallel collision detection.

Impact on elliptic curve cryptosystems over GF(2^{155}). We now discuss the practical utility of this new method by choosing, as an illustrative example, its application 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 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 above). 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.

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]. About 75 of these cells plus input/output and logic to detect distinguished points could be put on a \$20 chip. We now summarize the results of an analysis similar to that for MD5 in Section 3. With a \$10 million budget, we suggest a distinguishing property that distinguishes one in 2^{33} group elements. We expect to perform $10^{18}\sqrt{\pi/2}$ group operations before a collision occurs. For storage, we need 46 bytes per distinguished point; this includes 16 bytes for the point on the curve and 15 bytes for each exponent (a_i and b_i).[2] Allowing for a factor of two in memory size in case the search takes longer than expected and a factor of 1.5 for overhead due to using a hash table for the list of distinguished points, we need about 20 Gbytes of memory. Assuming \$75/Mbyte, the memory cost is \$1.5 million, and the remaining budget can buy a machine with about 315 000 chips (at \$27 each, including an estimated \$7/chip overhead costs) or a total of 23.6 million processor cells. The expected time to complete a discrete logarithm is

$$\frac{13\,(155)}{40\text{ MHz}}\left(\frac{10^{18}\sqrt{\pi/2}}{23.6\times 10^{6}\text{ processors}} + 2^{33}\right) \cong 36\text{ days}$$

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. Note that the computation of the discrete logarithm in the large subgroup of size approximately 10^{36} will by far dominate other computational costs in the entire Pohlig-Hellman decomposition. This follows because the

[1] A key point is that different processors use independent starting points, but after such points are chosen, they are of known relation to one another. This allows collision information to be resolved into the recovery of logarithms.

[2] A point on an elliptic curve can be represented uniquely with one coordinate and one other bit. With a distinguishing property of 33 zero bits, a point can be represented in 155+1−33 = 123 bits or 16 bytes. The exponents are computed modulo p, the size of the subgroup, and for $p < 10^{36}$, the exponents can be represented in 15 bytes each.

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 $m = 155$.

5. Concluding Remarks

The parallelization of collision search techniques presented herein greatly extends the reach of practical attacks. Regarding collisions for hash functions, we have singled out MD5 due to its popularity, but the new technique can be applied to efficiently parallelize the search for collisions between meaningful messages for any hash function. Indeed, after fixing a budget, the only details of the specific hash function which enter into an analysis such as ours are the size of the hash result, and the speed and cost of a hardware implementation possible with current technology. The natural conclusion from our analysis is that 128-bit hash results are, depending on the application, too small to provide adequate security for the future, and perhaps even for today. For the case of discrete logarithms in cyclic groups, while we chose to apply our new techniques to a specific cryptosystem which has actually been built, the technique again has general application. For the specific elliptic curve cryptosystem over GF(2^{155}) which we have considered, we note that the security could be improved by increasing the recommended 10^{36} lower bound on the smallest prime factor of the order of the elliptic curve group. For comparison to the 24 days and 36 days, respectively, required to attack the above-mentioned systems using our new techniques, exhaustive DES key search with the same budget of $10 million would take about 21 minutes [24]. When comparing these times, it is worth noting that the MD5 collision search is based on 0.5 µm technology, the elliptic curve logarithm implementation is based on 1.5 µm, and DES key search is based on 0.8 µm. Due to the nature of the estimates, our analysis for MD5 collision search and the above elliptic curve logarithm problem should be taken, for all intents and purposes, to mean that these problems are of roughly equivalent difficulty. The important point is that both are within the reach of a motivated opponent, and in general, the comfort-level for "adequate" security should be adjusted appropriately.

The parallel collision search technique presented is a general tool with many other potential applications; indeed, it would appear to be of use whenever one wishes to find a collision in pseudo-random walks through a large space. One such application may be in one version of the elliptic curve factoring algorithm whose second phase involves such a random walk [5, p. 32]; another, of course, is to parallelized Pollard rho factoring [19], although better general factoring methods are available.

Acknowledgments

We would like to thank Bart Preneel for his helpful suggestions and Kevin McCurley for making us aware of reference [5].

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

[4] Bell-Northern Research, Internal study of MD5 Silicon Implementation, 1994.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

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

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

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

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

[21] 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.

[22] R. Rivest, “The MD5 Message-Digest Algorithm”, Internet RFC 1321, April 1992.

[23] 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.

[24] M.J. Wiener, “Efficient DES Key Search”, TR-244 (May 1994), School of Computer Science, Carleton University, Ottawa, Canada. (Presented at the Rump Session of Crypto ’93.)

[25] G. Yuval, “How to Swindle Rabin”, *Cryptologia*, vol. 3 (3) (July 1979), pp. 187-189.