J. Cryptology (2004) 17: 105–124
DOI: 10.1007/s00145-003-0213-5

Journal of
CRYPTOLOGY
© 2003 International Association for Cryptologic Research

The Full Cost of Cryptanalytic Attacks

Michael J. Wiener
20 Hennepin Street, Nepean,
Ontario, Canada K2J 3Z4
michael.wiener@sympatico.ca

Communicated by Arjen Lenstra

Received 5 June 2002 and revised 12 August 2002
Online publication 10 July 2003

Abstract. An open question about the asymptotic cost of connecting many processors to a large memory using three dimensions for wiring is answered, and this result is used to find the full cost of several cryptanalytic attacks. In many cases this full cost is higher than the accepted complexity of a given algorithm based on the number of processor steps. The full costs of several cryptanalytic attacks are determined, including Shanks' method for computing discrete logarithms in cyclic groups of prime order n, which requires $n^{1/2+o(1)}$ processor steps, but, when all factors are taken into account, has full cost $n^{2/3+o(1)}$. Other attacks analyzed are factoring with the number field sieve, generic attacks on block ciphers, attacks on double and triple encryption, and finding hash collisions. In many cases parallel collision search gives a significant asymptotic advantage over well-known generic attacks.

Key words. Cryptanalysis, Discrete logarithm, Factoring, Number field sieve, Parallel collision search, Meet-in-the-middle attack, Double encryption, Triple encryption, Hash collision.

1. Introduction

The resources required to carry out an algorithm are often described in terms of the number of steps and amount of memory required. For example, some algorithms with input n may require $\Theta(n)$ steps and $\Theta(n)$ memory elements.[1] Does this make the algorithms' overall cost $\Theta(n)$, $\Theta(n^2)$, or something else? The purpose of this paper is to show how requirements for the number of steps and memory can be combined in a meaningful way to give an overall assessment of the cost of an algorithm.

Current practice in stating the cost of an algorithm is very processor-centric; we count the total number of operations performed by all processors. An exception is the work

[1] We use the usual asymptotic notation $O(\cdot)$ for an upper bound, $\Omega(\cdot)$ for a lower bound, $\Theta(\cdot)$ for a tight bound, and $y(n) = o(x(n))$ means that $y(n)/x(n)$ goes to zero as n goes to infinity.

of Amirazizi and Hellman on time–memory–processor trade-offs [1], which is closely related to the topic of this paper.

To examine this focus on processors, we take a superficial look at the components that make up a processor: memory elements, logic gates, and lengths of wire, among other components. Each clock cycle these components perform their functions: memory elements maintain or change state, logic gates perform boolean operations, and wires carry current. Viewed at this level, there is little difference among processors, memory chips, and communications devices; all contain memory elements, logic gates, and wires. All of these components perform a "step" every clock cycle regardless of whether they are considered part of a processor, memory device, or communications device. All of these steps count in measuring the cost of an algorithm performed on a particular collection of hardware. The reader need not become concerned at this point that we will engage in tedious attempts to count all components and their steps. We are concerned with asymptotics; some constant amount of computation may require billions of component steps, but we will be satisfied to call its cost $\Theta(1)$.

To illustrate what it means to count all costs, consider the example of an algorithm run on a processor in $\Theta(n)$ steps using $\Theta(n)$ memory. There are $\Theta(n)$ components that each take $\Theta(n)$ steps, and it may seem harsh to say that the full cost is $\Theta(n^2)$, but that is the real cost if you hook up one lonely PC to a large memory. This cost comes from a combination of the algorithm and the way it is implemented. If the algorithm is parallelized across $\Theta(n^{1/2})$ processors (all accessing the one large memory in parallel) so that the time is reduced to $\Theta(n^{1/2})$, but there are still $\Theta(n)$ components (dominated by the memory), then the full cost is reduced to $\Theta(n^{3/2})$. One may be tempted to say that only a small fraction of the memory is actually doing anything at any one time, and that the full cost should really be $\Theta(n)$. Unless the majority of the memory that is idle could be used for something else, which seems unlikely, this is not a useful view. If the entire memory is unavailable for other purposes for the full duration of the algorithm execution, then its full cost should be charged during each unit of time of the computation.

The full cost of an algorithm run on a collection of hardware is the number of components multiplied by the duration of their use. This is called the throughput cost by Lenstra et al. [9], and is also used by Bernstein [2]. To say something useful about an algorithm itself rather than the combination of the algorithm and the hardware that implements it, we seek the implementation of the algorithm that minimizes full cost. This often involves choosing the optimal degree of parallelism. Another factor that can affect this full cost is the number of instances of the algorithm run at once, called simultaneity by Amirazizi and Hellman [1] who used the full cost approach.

In defense of the processor-centric method of measuring algorithm cost, it sometimes turns out that improved attacks are found that reduce memory requirements without changing the number of processor steps. An example is the improvement in computing discrete logarithms in groups of prime order n from the method attributed to D. Shanks by Knuth [7, p. 591], which requires storage for $n^{1/2}$ group elements, to Pollard's rho method [15], which requires that only a small constant number of group elements be stored (see Section 4). By counting only processor steps in attacks when choosing key sizes, the cryptographer is being conservative. However, this should not be turned around to say that Shanks' method and the rho method have the same full cost because they do not.

Throughout this paper the tight bound $\Theta(\cdot)$ is used frequently instead of the more familiar upper bound $O(\cdot)$. This should not be taken to mean that a particular problem cannot be solved with less cost than the asserted tight bound. What it means is that the particular approach taken to solving the problem has the asserted cost. For example, multiplying two n-bit numbers is known to require $O(n \log n)$ steps, but most implementations use a technique that takes $\Theta(n^2)$ steps.

The rest of this paper is organized as follows. In Section 2 we answer an open question about the cost of connecting many processors to a large memory, and objections to this cost analysis are discussed in Section 3. The cost analysis is then used to assess the full costs of several types of cryptanalytic attacks: discrete logarithm computation (Section 4), factoring (Section 5), attacking block ciphers (Section 6), double encryption (Section 7), and triple encryption (Section 8), and finding hash collisions (Section 9). For the attacks on encryption and multiple encryption, only the basic electronic codebook mode is considered. Handschuh and Preneel deal with attacks on multiple encryption with various modes of operation [5].

2. Full Cost of Connecting Many Processors to a Large Memory

For cryptanalytic attacks that require a large memory, we often require the use of parallel processors to minimize the full cost of the attack. This leads to the need to connect many processors efficiently to the same large memory. We are concerned with the case where the processors repeatedly access random locations in the large memory in parallel rather than the case where each processor is operating within its own small section of the memory. To illustrate what we mean by "parallel," consider the case where 1000 small processors access a large memory broken into 1000 blocks. If the processors generate a memory access every 100 ns, then the memory must support 1000 memory accesses every 100 ns. For most attacks, a processor need not be blocked while waiting for a read or write operation to complete; many access requests can be dispatched with the results returning sometime later, possibly out of order.

We consider first the case of n processors accessing n blocks of memory at a high rate, and then deal with the more general case where the number of processors and number of memory blocks are not equal, and the required memory access rate may be lower.

2.1. *Connecting n Processors to n Blocks of Memory*

Figure 1 shows one way to connect eight processors to eight blocks of memory using switching elements to take two streams of requests and sort them based on one bit of the memory address. Only the components for sending the requests are shown. A similar set of components is required to send data back to the processors. The requests include three address bits indicating which memory block is being accessed. The initial stream of requests from each processor contains all memory addresses (shown as XXX). The first column of switching elements sorts these requests (into addresses 0XX and 1XX) based on the most significant memory block address bit. The second column sorts based on the second bit, and the last column sorts based on the bottom bit.

This architecture can be extended to $n = 2^k$ processors and memory blocks using $\Theta(n \log n)$ switching elements. This would seem to be the dominant cost, but what

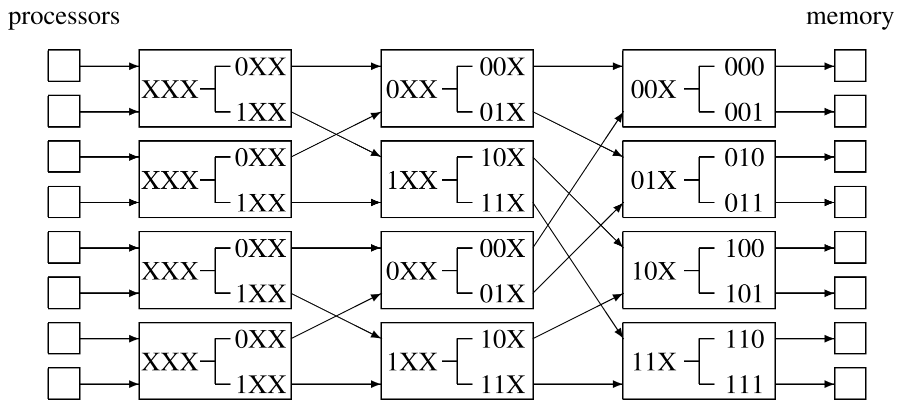

Fig. 1. Switching network connecting processors to memory.

about the wires? Amirazizi and Hellman treated the cost of the wires as no more than a logarithmic factor times n, but left it as an open problem requiring further study [1]. The linear arrangement of the processors and memory elements in Fig. 1 is not optimal for minimizing wire length, but it is instructive to analyze this case first. To count wire length, treat the columns of components as being right beside each other so that only vertical distance is considered. Then the wires connected to the processors and the wires connected to the memory have length 0. The total length of wire connecting the first two columns of switching elements is 4, and the total length of wire connecting the last two columns of switching elements is 8, for a total of 12. For 16 processors and memory blocks, the total wire length would be $8 + 16 + 32 = 56$. In general for 2^k processors and memory blocks, the total wire length is $2^{k-1}(2^{k-1} - 1) = \Theta(n^2)$. More than half of this total wire length is between the last two stages where there are n wires of average length $n/2$. By packing the memory elements in two dimensions, the average distance covered by the wires between the last two stages can be reduced to $\Theta(n^{1/2})$ for a total wiring cost of $\Theta(n^{3/2})$. The total volume of wire is proportional to its length, and therefore the volume of wire is also $\Theta(n^{3/2})$, so that the wires have to be packed in three dimensions to limit the longest wires to $\Theta(n^{1/2})$ in length. We cannot pack wires any tighter than filling three dimensions, and thus the cost of wiring cannot be less than $\Theta(n^{3/2})$ for this method of connecting processors to memory. The cost of wiring in this network dominates all other costs including switching elements, processors, and memory. The idea that wires dominate costs may be hard to believe at first, but calling them "wires" is a little misleading. Very high speed communications over even a short distance such as a meter requires more than just a simple wire.

For optimal performance, the switching elements in Fig. 1 cannot wait for one memory access request to get to the next switching element before sending another. The wires must be pipelined using repeaters or latches so that many memory access requests can be in transit at one time.

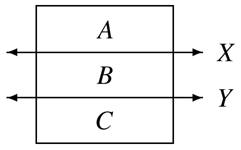

Fig. 2. Processors, memory, and switching network arranged in a cube.

Is it possible to reduce wiring costs below $\Theta(n^{3/2})$ with some different method of connecting processors to memory? The surprising answer is that it is not possible. Suppose that the total length of wire needed is $\Theta(n^{1+z})$ for some $z \geq 0$, and that the processors, memory, and switching network are packed into a cube (see Fig. 2). Because the volume of wire is $\Theta(n^{1+z})$, the cube must have edges of length $\Theta(n^{(1+z)/3})$. Assume that the memory blocks and processors are distributed uniformly throughout the cube so that the cube can be divided into three disjoint regions A, B, and C containing roughly equal numbers of processors and memory blocks by parallel planes X and Y that are a distance $\Theta(n^{(1+z)/3})$ apart. (These assumptions will be removed in the proof of the more general result in Section 2.2.) About one-third of the memory access requests from region A are destined for region C. Thus there are $\Theta(n)$ requests that must be carried a distance $\Theta(n^{(1+z)/3})$. It may be possible to merge multiple streams of requests onto a single wire (perhaps with fiber optics), but only a constant amount of this is possible. It is necessary to run $\Theta(n)$ wires over the distance $\Theta(n^{(1+z)/3})$ for a total cost of $\Theta(n^{(4+z)/3})$. We assumed initially that the wire cost was $\Theta(n^{1+z})$. Solving $(4 + z)/3 = 1 + z$ yields $z = \frac{1}{2}$. Thus, the wiring cost cannot be reduced below $\Theta(n^{3/2})$, which was achieved by the architecture in Fig. 1.

Closely related to this result is Rosenberg's work where he showed that for several types of circuits of dimension n, the total area required in a two-dimensional realization including wiring is $n^{2+o(1)}$, and the total volume required in a three-dimensional realization is $n^{3/2+o(1)}$ [16].

To reduce wiring costs below the $\Theta(n^{3/2})$ bound requires that more than a constant amount of information be carried through a constant volume, which seems not to be possible with current wired and wireless technologies. One idea for avoiding this limitation is to have the columns of switching elements communicate by firing photons directly to the next column [19]. This results in a more than constant number of photons occupying each unit of space. However, if some form of optic cable is required or repeaters are required, then $\Theta(n^{3/2})$ components are needed. Thus, it seems likely that this idea can only give a constant speed-up (but perhaps the constant is large).

To test whether the results so far seem reasonable, we move to the more familiar context of interconnected computers. In this setting the processors are the computers' central processing units, and the large memory is the collective memories in all the computers. Each computer generates a stream of memory access requests to and from the memories of the other computers (and occasionally its own). Each computer also satisfies external requests by either writing data into its own memory or reading data from its memory and sending the result back to the requesting computer. These computers may be connected by a local area network or perhaps the internet. Does it make sense that the cost of connecting the computers exceeds the cost of the computers themselves? Imagine that all internet-connected computers are each generating millions of memory access requests per second. This is a much more demanding scenario than what is handled by the current internet. Generally, data rates into browsers are in the millions of bytes per day range rather than millions of bytes per second. Also, browsers tend to collect data from a small number of servers per day, rather than accessing millions of other computers per second. The internet would have to be many orders of magnitude more expensive to handle this more demanding task. The alternative is to use the internet as it is and have the problem being solved take several orders of magnitude longer to complete.

2.2. *General Case*

To this point we have considered the case where each processor generates memory requests at a data rate of $\Theta(1)$ bits per unit time. However, some cryptanalytic attacks require a lower memory access rate. For example, if each processor generates a request of size $\log n$ every n units of time, we say that the memory access rate per processor is $(\log n)/n$. Also, the number of processors and number of memory blocks may not be equal. A result about the total number of components required in the general case is given in Theorem 1, whose proof makes use of the following lemma.

Lemma 1. *When n non-overlapping objects of unit area are in a plane, there exist two parallel lines, top and bottom, at least a distance $(n/5)^{1/2}$ apart such that the total object area (excluding any gaps) above the top line is at least $n/5$, and the total object area below the bottom line is at least $n/5$.*

Proof. Choose any two parallel lines Y and Z that divide the total object area so that three-fifths of the area is between the lines and the two regions outside the lines each have one-fifth of the object area. Some of the objects may be split by a line so that their areas are split across two regions. Let Y' and Z' be parallel lines such that they are perpendicular to Y and Z and they divide the region between Y and Z into three sub-regions with each sub-region holding one-fifth of the total object area. The rectangle defined by the four lines must have area at least $n/5$ because it contains object area $n/5$. One side of this rectangle has to be at least $(n/5)^{1/2}$ long. Therefore, one of the pairs of lines must be at least $(n/5)^{1/2}$ apart. Both pairs of lines are such that the two regions outside the lines contain at least one-fifth of the object area, which completes the proof. □

The following theorem gives the total component cost for the general case.

Theorem 1. *The total number of components required to allow each of p processors uniformly random access to m memory elements at a memory access rate r, including the components in the processors, memory, switching elements, and wires is $\Theta(p + m + (pr)^{3/2})$.*

Proof. Proving this result requires finding an architecture that meets the asserted number of components to establish the upper bound, and showing that it is not possible to use fewer components to establish the lower bound. Group the processors into pr groups of $1/r$ processors each. Each group has a master processor that "speaks" for the group, and all memory access requests are passed serially from one processor in the group to the next until they are aggregated into a stream of $\Theta(1)$ requests per unit time at the master processor. Divide the memory into pr pieces of size $m/(pr)$ each. Use a switching network as shown in Fig. 1 to connect the processor groups to the memory pieces. This requires $\Theta(pr\log(pr))$ switching elements. Let c be the total number of components, and let w be the total length of wire. The volume occupied by the components is then $\Theta(c)$, and the average length of the wires between the last two columns of switching elements is $\Theta(c^{1/3})$. There are pr of these wires for a total wire cost of $w = \Theta(prc^{1/3})$. Because the total number of components c is at least as large as the number of processors p, we have $w = \Omega(p^{4/3}r)$. This is more than the cost of the switching elements $\Theta(pr\log(pr))$ because memory cannot be accessed at a faster than constant rate $r = O(1)$, and $\log p = o(p^{1/3})$. Therefore, we can ignore the cost of the switching elements. The total number of components is then $c = \Theta(p + m + w) = \Theta(p + m + prc^{1/3})$. This simplifies to $c = \Theta(p + m + (pr)^{3/2})$. Note that this does not mean that the wiring cost is $\Theta((pr)^{3/2})$. In fact, $w = \Theta(pr(p^{1/3}+m^{1/3})+(pr)^{3/2})$, but the two extra terms cannot be larger than both $p+m$ and $(pr)^{3/2}$ and thus are not needed in the expression for c. By finding an architecture that meets the bound claimed in this theorem, we have established that it is an upper bound.

To establish the lower bound, we will show that the wiring cost is $w = \Omega((pr)^{3/2})$ without making assumptions about the location of processors and memory. Choose a plane X such that half of the processors are on each side (see Fig. 3). Any processor or memory element intersecting plane X is considered to be on both sides because the internal wiring of the processor or memory element extends to both sides of the plane. Plane X divides the memory into two disjoint subsets with one subset having at least half of the memory. Without loss of generality, assume that at least $p/2$ processors are left of plane X, and $m/2$ memory elements are on the right. Let the units of r be such

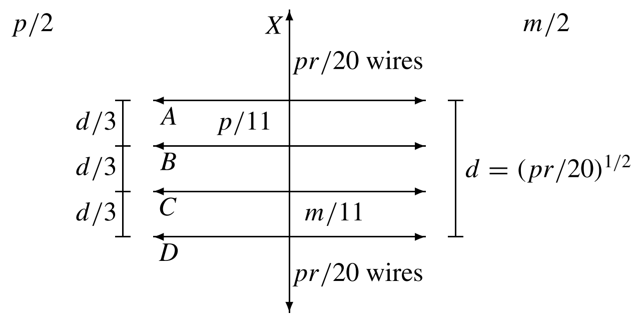

Fig. 3. Planes dividing processors and memory for proof of Theorem 1.

that the memory accesses by at most $1/r$ processors can be carried on a single wire, or equivalently for parallel access with 64 data lines and 64 address lines, the memory accesses for at most $128/r$ processors can be carried on 128 wires. Half of the memory accesses from the $p/2$ processors on the left are destined for the $m/2$ memory elements on the right requiring $pr/4$ wires to cross plane X. If the wire cross sections have unit area, then by Lemma 1 there exist two parallel lines on plane X at least a distance $d = (pr/20)^{1/2}$ apart such that $pr/20$ of the wires cross X above the top line and $pr/20$ of the wires cross below the bottom line. In Fig. 3 these lines are perpendicular to the page and are the intersection of planes A and D with plane X. Let B and C be planes parallel to A and D that divide the space between A and D into thirds (with B between A and C). The $pr/20$ wires above plane A carry memory accesses from at least $p/10$ of the processors to the $m/2$ memory elements on the other side of X. The number of these processors that are more than $d/3 = \Theta((pr)^{1/2})$ away from plane A must be $o(p)$ or else the total wire length connecting these processors to plane X would be $\Theta((pr)^{3/2})$. Therefore, $p/11$ of these processors must be above plane B. By similar reasoning, $m/11$ of the memory elements must be below plane C. One-eleventh of the memory accesses from the $p/11$ processors above plane B are destined for the $m/11$ memory elements below plane C requiring $pr/121$ wires to cross planes B and C, which are a distance $d/3 = \Theta((pr)^{1/2})$ apart. Thus there must be a subset of the wires requiring total length $\Theta((pr)^{3/2})$ so that the overall wiring cost is $\Omega((pr)^{3/2})$. This establishes the lower bound and proves that the number of components required is $c = \Theta(p + m + (pr)^{3/2})$. $\square$

We now relate the full cost of an algorithm to the traditional count of the number of processor operations in the following corollary.

Corollary 1. *For an algorithm where p processors access a memory of size m at rate r, and the total number of processor operations is T, the full cost of the algorithm is* $F = \Theta((T/p)(p + m + (pr)^{3/2}))$.

Proof. By Theorem 1, the number of components required for this algorithm is $c = \Theta(p + m + (pr)^{3/2})$. Multiplying c by the time spent by each processor, T/p, gives the full cost of the algorithm: $F = \Theta((T/p)(p + m + (pr)^{3/2}))$. $\square$

If we rewrite the full cost from Corollary 1 as $F = \Theta(T(1 + m/p + p^{1/2}r^{3/2}))$, we see that $F = \Omega(T)$, and that $F = \Theta(T)$ if and only if $p = \Omega(m)$ and $r = O(p^{-1/3})$. Therefore, the full cost of an algorithm is never less than the traditional count of processor steps, and we have equality when the amount of memory per processor is $O(1)$ and the memory access rate is sufficiently low. This memory access rate restriction applies only to the memory common to all processors; the processors can access their own private memories of size $O(1)$ at high rate without increasing full cost.

The case where the memory access rate r is high will occur several times in analyzing attacks making it convenient to use the following corollary.

Corollary 2. *For an algorithm where the rate r at which p processors access a memory of size m is high*, $1/r = m^{o(1)}$, *the memory size is independent of the number of proces-*

sors, and the total number of processor operations is T, the full cost of the algorithm is a minimum of $F = \Theta(Trm^{1/3})$ when $p = \Theta(m^{2/3}/r)$.

Proof. By Theorem 1, the number of components required is $c = \Theta(p+m+(pr)^{3/2})$. Because r is high, the first term, p, is not significant. Multiplying c by the time spent by each processor, T/p, gives the full cost of the algorithm: $\Theta((T/p)(m + (pr)^{3/2}))$. This full cost is a minimum of $F = \Theta(Trm^{1/3})$ when $p = \Theta(m^{2/3}/r)$. □

Thus, when the memory access rate is high, the full cost is proportional to the total number of processor steps multiplied by the cube root of the memory size, and this full cost is achieved when the number of processors is proportional to the two-thirds power of memory size. This does not mean that when the memory size is 1 Gbyte (10^9 bytes), there have to be a million processors. A detailed analysis of some attack that does not ignore constant factors may show that the optimal number of processors is $p = (m/10^9)^{2/3}$.

3. Objections

Here we address two possible objections to the analysis of wiring costs in Section 2. The first objection is an argument that wiring costs are higher than indicated in Theorem 1, and the second objection is an argument that for practical purposes, wiring costs are actually lower.

The first objection is that in silicon, we are essentially confined to two-dimensional designs, and that the cube in Fig. 2 is unrealistic. If circuits are confined to having $\Theta(1)$ thickness, then the total component cost in Theorem 1 goes up to $\Theta(p + m + (pr)^2)$ due to higher wire costs. This line of argument makes sense for a design that is confined to a single chip, and even when multiple chips are placed on a board there is a limit to the number of layers of interconnect available. However, as multiple boards are placed in racks, the design becomes quite three-dimensional. Connections between the first and last boards in a rack or between racks do not have to run the full lengths of all intervening boards. For cryptanalytic problems whose size is in the current range of interest, it is definitely necessary to use multiple boards to complete the attack in a reasonable time, and therefore it is reasonable to treat the complete circuits as three-dimensional.

The second objection is that wires are quite thin and cheap compared with processors and memory blocks, and there is no danger that we will design circuits so large that the wires will fill up space to the point where the rest of the components have to be spread out to make room for wires. Unfortunately, we cannot connect processors to memories with long cheap wires without degrading performance. Just the extra time for conducting electricity through a meter-long wire is significant, and there are other long wire effects that limit speed as well. Memory access requests would need to be pipelined with latches at some fixed spacing to avoid speed reductions. These wires plus pipelining components take up space on boards and add significantly to costs. As the number of processors and memory blocks gets into the thousands, making all the high-speed connections becomes costly. To address the objection that wires are thin, Vitányi treated wires as having no volume in showing that exponential computations cannot be completed in polynomial time with parallelism [22]. In our case if wires are assumed to have zero thickness but non-zero cost per unit length, the wiring costs in Section 2.1 drop from $\Theta(n^{3/2})$ to $\Theta(n^{4/3})$, the number of components in Theorem 1 drops to $\Theta(p + m + pr(p+m)^{1/3})$, the full cost in Corollary 1 drops to $\Theta((T/p)(p+m+pr(p+m)^{1/3}))$, and Corollary 2 is unaffected. The zero-volume wire assumption only leads to a reduced full cost for designs where the wiring cost is dominant, as was the case in Section 2.1. However, for the optimized cryptanalytic designs described in the following sections, the wiring costs never dominate, and therefore the conclusions would be unaffected if wires were treated as having no volume.

4. Discrete Logarithm Problem

Many cryptographic schemes are based on the difficulty of solving the discrete logarithm problem: given a generator g of a cyclic group G of prime order n and an element g^x in G, find x. Shoup showed that a generic discrete logarithm algorithm, one that does not exploit any special properties of the encodings of group elements, must perform $\Omega(n^{1/2})$ group operations [17]. He also said that “one cannot substantially improve upon the Pohlig–Hellman algorithm using generic algorithms.” This is true with respect to the number of group operations performed. However, when the full cost of the attack is considered, it is possible to improve upon the Pohlig–Hellman algorithm [14] (which uses Shanks’ method). Pollard’s rho method [15] is not deterministic, but it eliminates the large memory needed by Shanks’ method, reducing the full cost to $\Theta(n^{1/2})$ times the cost of performing a group operation. This makes Shoup’s bound tight with respect to both number of group operations and full cost.

In the following subsections we examine Shanks’ method and the rho method in more detail to determine their full costs for both the case where a single logarithm is sought and the more general case where many logarithms within the same group are sought.

4.1. *Shanks’ Method*

Shanks’ method for computing a discrete logarithm (also called the baby-step giant-step algorithm [11, p. 104]) proceeds as follows. Pre-computation: choose a positive integer a, compute g^a in $\Theta(\log a)$ group operations, and compute and store (i, g^{ia}) for $1 \leq i \leq \lceil n/a \rceil$ in a hash table with g^{ia} as the index. Logarithm computation: take the group element $y = g^x$ with unknown index x and compute and look up yg^j in the table for $j = 0, 1, \ldots$ until one of the yg^j is found in the table. We now have $yg^j = g^{ia}$ for some i and j, which gives $x = ia - j$. If $\log a = O(n/a)$, then the pre-computation requires $\Theta(\log a + n/a) = \Theta(n/a)$ group operations and table writes, and the logarithm computation stage requires at most $a - 1$ group operations and a table reads. The computed table can be reused for multiple logarithm computations in the same group G.

Let us assume that the storage space required to represent a group element is $\Theta(\log n)$, the time t required to perform a group operation on a processor is such that $t = \Omega(\log n)$ and $t = n^{o(1)}$, and the time required for the hash computation for table lookups is $O(t)$. If s logarithms are to be computed, then the total time spent across all p processors is $T = \Theta((sa + n/a)t)$. The memory required is $m = \Theta((n/a)\log n)$, and the memory access rate is $r = \Theta((\log n)/t)$. By Corollary 2, the full cost of the algorithm is $F = \Theta(Trm^{1/3}) = \Theta((sa + n/a)(n/a)^{1/3}(\log n)^{4/3})$ when $p = \Theta(m^{2/3}/r)$. This cost is a minimum of $F = \Theta(s^{2/3}n^{2/3}(\log n)^{4/3})$ when $a = \Theta((n/s)^{1/2})$ and $p = \Theta((sn/\log n)^{1/3}t)$. Because a cannot be less than 1, the cost grows linearly with s for $s > n$. The full cost per discrete logarithm solution is $F/s = \Theta(n^{2/3}(\log n)^{4/3}/s^{1/3})$. It is interesting to note that the algorithm cost does not depend upon the time t required to perform a group operation, but that the number of processors does depend on t. This makes sense because the cost of the processors is insignificant compared with the rest of the components required, and if the group operation time increases, we can compensate by increasing the number of processors without affecting the overall algorithm cost.

Two cases of interest are $s = 1$ and $s = n$. If only a single logarithm is to be found, then the algorithm cost is $\Theta(n^{2/3}(\log n)^{4/3})$. If more than n logarithms are to be found, then the algorithm becomes a simple table look up where the table contains all group elements and their corresponding logarithms, and there are $p = \Theta(n^{2/3}t/(\log n)^{1/3})$ processors accessing the table simultaneously. The cost per logarithm in this case is $\Theta(n^{1/3}(\log n)^{4/3})$. For an attacker who uses Shanks' method for computing discrete logarithms, the cost of the first logarithm is $n^{2/3+o(1)}$, and thereafter the cost per logarithm drops to a minimum of $n^{1/3+o(1)}$.

4.2. *Pollard's Rho Method*

Pollard's rho method for computing discrete logarithms requires $\Theta(n^{1/2})$ group operations and only needs enough memory for $\Theta(1)$ group elements. This is a considerable reduction in memory requirements compared with Shanks' method which ultimately leads to a much lower full cost of the algorithm. If a group operation takes time $t = n^{o(1)}$ on a processor, and the space required to represent a group element is $\Theta(\log n)$, then the number of components is $\Theta(\log n)$, and the processor time is $\Theta(n^{1/2}t)$, for a full cost of $\Theta(n^{1/2}t \log n)$.

To compute multiple logarithms efficiently requires parallel processing. Although Pollard's rho method cannot be directly parallelized efficiently, a related method called parallel collision search [21] can be parallelized efficiently. When parallel collision search is applied to a single discrete logarithm, all of the processors work on the one problem. However, each processor could be working to find a different discrete logarithm as long as the iterating function used by all processors does not depend on the group element whose logarithm is sought. This requirement is satisfied by an iterating function $f\colon G \to G$ suggested by Teske [18], where G is partitioned into about 20 disjoint sets T_i, each set is assigned a fixed randomly chosen group element g^{x_i} with known logarithm x_i, and $f(y) = yg^{x_i}$ if $y \in T_i$.

For parallel collision search to find s logarithms, $T = \Theta((ns)^{1/2}t)$ processor steps are required across $p = \Theta(s)$ processors. The memory must be large enough for each processor to contribute $\Theta(1)$ group elements. The memory size is then $m = \Theta(s \log n)$. With each processor writing $\Theta(1)$ group elements to the memory during the time that it performs $T/p = \Theta((n/s)^{1/2}t)$ processor steps, the writing rate is $r = \Theta((s/n)^{1/2}(\log n)/t)$. By Corollary 1, the full cost is $F = \Theta((T/p)(p + m + (pr)^{3/2}))$. Substituting for T, p, m, and r in this equation, it can be shown that the full cost per solution is $F/s = \Theta((t \log n)((n/s)^{1/2} + s^{3/4}(\log n)^{1/2}/(n^{1/4}t^{3/2})))$. This cost per solution is a minimum of $\Theta(n^{1/5}t^{2/5}(\log n)^{6/5})$ when $s = \Theta(n^{3/5}t^{6/5}/(\log n)^{2/5})$.

For an attacker who uses parallel collision search for computing discrete logarithms, the full cost of the first logarithm is $n^{1/2+o(1)}$, and thereafter the cost per logarithm drops to a minimum of $n^{1/5+o(1)}$. This is a considerable improvement over Shanks' method.

5. Factoring

Factoring an integer n using the number field sieve (NFS) [8] involves a relation collection step and a matrix step [3], [13]. The costs of these two steps are traded off against each other in selecting NFS smoothness bounds [9]. Bernstein [2] observed that in the standard analysis of NFS, this trade-off is based on the traditional measure of processor operations, but that the full cost is actually higher. Define $L(\alpha) = e^{(\alpha+o(1))(\log n)^{1/3}(\log\log n)^{2/3}}$, and let $c = (92 + 26\sqrt{13})^{1/3}/3 = 1.90188\ldots$. Then with the standard trade-off, both steps of NFS require $L(c)$ processor steps, but the matrix step requires $L(c/2)$ memory. With no parallelization and no adjustment of the smoothness bounds, the full cost of NFS is $L(3c/2)$. Bernstein designed a mesh sorting circuit for the matrix step and adjusted the smoothness bounds to lower the full cost of factoring to $L((\frac{5}{3})^{4/3}) = L(1.97605\ldots)$ [2]. This same asymptotic cost is achieved by the two-dimensional mesh routing network of Lenstra et al. [9].

The cost of the matrix step is dominated by the cost of multiplying a sparse matrix A by a vector v, both over GF(2), and each with dimension D that is determined by the smoothness bounds. Approximately D such multiplications are required. Other operations, including inner products of vectors, are required in the matrix step, but none have higher cost than the matrix-vector multiplication. Each column of A has $D^{o(1)}$ non-zero entries. Because v is over GF(2), multiplying A by v amounts to summing the columns of A corresponding to non-zero elements of v. The columns of A are stored as a list of row indices of the non-zero elements so that $D^{1+o(1)}$ storage is required. A single processor version of the multiply proceeds as follows. Initialize the product vector elements to zero. For each column of A whose corresponding bit in v is non-zero, and for each row index r in the column, toggle bit r of the product vector. Repeating this D times requires $D^{2+o(1)}$ time. The memory required is $D^{1+o(1)}$ making the full cost $D^{3+o(1)}$. Bernstein's circuit reduced this full cost to $D^{5/2+o(1)}$ [2].

A further improvement is possible using a design based on the circuit in Fig. 1. Use $D^{2/3+o(1)}$ processors, each responsible for contributing $D^{1/3+o(1)}$ of the columns of A to each product. It helps to think of each processor owning the part of the memory corresponding to its part of A and its parts of v and the product vector. Each processor sends row numbers through the switching elements, and the row numbers corresponding to its part of the product vector return. These returning row numbers are used by each processor to form its part of the product vector. This reduces the multiply time to $D^{1/3+o(1)}$ using $D^{1+o(1)}$ components (memory and wiring). The full cost of all D multiples is down to $D^{7/3+o(1)}$. This asymptotic cost can also be achieved using a three-dimensional version of Bernstein's circuit or a three-dimensional mesh routing network [9].

A technicality here is that the design in Fig. 1 assumed that the memory accesses are uniformly distributed across all memory addresses. This is not the case here because the rows of A do not all have the same density. We can compensate for this by changing the addressing decisions in the switching elements. Suppose that one-quarter of the non-zero elements of A are in the first R_1 rows, the next one-quarter are between rows R_1 and R_2, the next one-quarter are between rows R_2 and R_3, and the final one-quarter are between rows R_3 and D. Then the first set of switching elements splits the streams of row numbers into those $\leq R_2$ and those $> R_2$. Half of the second set of switching elements use R_1 as a threshold, and the other half use R_3 as a threshold. One technicality remains. This change in addressing decisions only works if all rows have fewer than $D^{1/3}$ non-zero entries. Beyond this row density, some of the switching elements from the last stage would still be swamped because too many accesses would be directed to a single memory block. This can be handled by observing that if a switching element sees two identical row numbers, it can simply throw them away because they cancel each other. The switching elements already receive the row numbers in pairs and can eliminate a pair if they are the same. If a fraction f of the capacity of a switching element's inputs consist of a single row number r, then after elimination of (r, r) pairs, r is expected to make up $2f(1 - f)$ of the capacity of one of the switching element's outputs. The output fraction is a maximum of $\frac{1}{2}$ when $f = \frac{1}{2}$. Thus, one row number never appears often enough to swamp a switching element. The expected amount of row number elimination should be taken into account when choosing the switching threshold for each switching element.

We have shown that the cost of the matrix step can be reduced to $D^{7/3+o(1)}$. Using the analysis of Lenstra et al. for when the general matrix exponent is 2ε [9], we have $\varepsilon = \frac{7}{6}$ for this design, and the full cost of factoring is reduced to $L((\frac{49}{18})^{2/3}) = L(1.94961\ldots)$. Comparing this with $L(1.90188\ldots)$ for the standard analysis of NFS based on processor steps, the difference is a factor of $L(0.04773\ldots)$, which is less than a factor of 5 when $n = 2^{1024}$ (ignoring the $o(1)$).

One possible interpretation of this result is that past factoring efforts have had a matrix step with high full cost, and maybe we can do much better with this new approach. This may be true asymptotically, but not for factoring numbers whose size is in the current range of interest. In past factoring efforts, the matrix step has been cheaper than the relation collection step, and even if the matrix step were free, it would not be possible to speed up the relation collection step much [9]. The results here on factoring should not affect the size of numbers currently used for cryptographic purposes.

6. Encryption

Given a block cipher whose encryption and decryption functions are $E_k(\cdot)$ and $E_k^{-1}(\cdot)$, respectively, and a plaintext–ciphertext pair (P, C) where $C = E_\kappa(P)$ for a particular key κ chosen at random from a set of n keys, the cryptanalyst wishes to find κ. It is assumed that either P is large enough that (P, C) uniquely determines the key (which may mean that the size of P is actually a multiple of the natural block size of the cipher), or that additional plaintext–ciphertext pairs are known to the cryptanalyst to determine κ uniquely.

It is not unreasonable to assume that an attacker can obtain a known plaintext. Most forms of electronic communications contain standard header fields that change little if at all from one communication to the next. In many cases the known plaintext is the same value for all communications of a particular type over a long period of time.

One way to find the key is by exhaustive search: for each of the n keys, decrypt C and see if the result is P. Exhaustive search is still possible when P is not known if a "plausible" P can be recognized due to some known redundancy. If the time required to encrypt or decrypt is t, and keys and texts are all of size $\Theta(\log n)$, then the full cost of this attack is $\Theta(nt \log n)$.

For a chosen-plaintext attack, where the cryptanalyst gets to choose P, it is possible to reduce the attack time by using an enormous memory. The attack proceeds as follows. Choose a fixed plaintext P. Store $(k, E_k(P))$ for all n keys k in a hash table with $E_k(P)$ as the index. The memory required is $m = \Theta(n \log n)$. Look up C in the table to get κ. Using this very large table does not make sense unless it is used to find at least $s = \Theta(n)$ keys. Suppose that there are s keys to be recovered, and the time required to get the chosen plaintext P encrypted is $t = n^{o(1)}$. The memory access rate for each processor is $r = \Theta((\log n)/t)$, and the total number of processor steps is $T = \Theta(nt)$. By Corollary 2, the full cost of the s attacks is a minimum of $F = \Theta(Trm^{1/3}) = \Theta((n \log n)^{4/3})$ when $p = \Theta(m^{2/3}/r) = \Theta(n^{2/3}t/(\log n)^{1/3})$ processors are used. The cost per solution is $\Theta(n^{1/3}(\log n)^{4/3})$.

An interesting attack that provides some middle ground between exhaustive search and table look up is Hellman's time–memory trade-off [6], which proceeds as follows. Pre-computation: choose a positive integer a and let $b = \lceil n/a^2 \rceil$. Choose a constant plaintext P and a function h that maps ciphertexts to keys so that $f(k) = h(E_k(P))$ defines a mapping of the key space onto itself. For $i = 1, \dots, b$, choose a starting key $x_{i,0}$ at random and iterate f on $x_{i,0}$ a times ($x_{i,j+1} = f(x_{i,j})$ for $j = 0, \dots, a-1$) to produce an ending key $x_{i,a}$, and store $(x_{i,0}, x_{i,a})$ in a hash table with $x_{i,a}$ as the index. Key recovery phase: use the ciphertext C corresponding to chosen plaintext P to compute $y_0 = h(C)$ and look up y_0, $y_1 = f(y_0)$, $y_2 = f(y_1), \dots, y_{a-1} = f(y_{a-2})$ in the hash table to see if any one of them is equal to one of the ending keys. Suppose that $y_d = x_{i,a}$. Then there is a good chance that $x_{i,a-d-1} = \kappa$; otherwise we have a false alarm. Iterate f on $x_{i,0}$ (stored in the table with $x_{i,a}$) to get the candidate key $x_{i,a-d-1}$ and check it on some other plaintext–ciphertext pair.

The probability that the procedure above will succeed is $\Theta(1/a)$. Therefore, this procedure must be repeated for $\Theta(a)$ rounds (with a different version of f in each round created by changing the mapping h) before κ will be found. The intent of the original algorithm is that all rounds of pre-computation should be done ahead of time and have the same cost as an exhaustive search, and the key recovery phase should be faster (per key) than exhaustive search. During pre-computation, memory size is $m = \Theta(ab \log n) = \Theta((n/a) \log n)$, and the memory access rate is $r = \Theta((\log n)/(at))$. The full cost of pre-computation is the time spent by each processor, $\Theta(nt/p)$, multiplied by the number of components (by Theorem 1) $c = \Theta(p + m + (pr)^{3/2})$. Due to the iterative nature of the algorithm, the maximum parallelism possible is $p = \Theta(n/a)$. The full cost is the cost of exhaustive search, $\Theta(nt \log n)$, when $a = \Omega(n^{1/4}(\log n)^{1/4}/t^{3/4})$. If a is smaller than this lower bound, then the wiring cost becomes dominant, and the pre-computation cost exceeds the cost of exhaustive search.

The key recovery phase requires the same amount of memory as the pre-computation, $m = \Theta((n/a) \log n)$, but the memory access rate increases to $r = \Theta((\log n)/t)$, and the total number of processor steps is $T = \Theta(a^2 t)$. By Corollary 2, the full cost per recovered key is $\Theta(Trm^{1/3}) = \Theta(n^{1/3}(\log n)^{4/3}a^{5/3})$, when $p = \Theta(m^{2/3}/r) =$

$\Theta(n^{2/3}t/((\log n)^{1/3}a^{2/3}))$. (When $a = 1$, this is the same cost as the table look up method. This is because the time–memory trade-off essentially becomes the same as the table look up method when $a = 1$.) However, there is a catch here. We cannot parallelize the recovery of a particular key across more than $\Theta(a)$ processors. Therefore, we have to work on $p/a = \Theta(n^{2/3}t/((\log n)^{1/3}a^{5/3}))$ problems simultaneously to achieve the required level of parallelism. The full cost of finding this number of keys turns out to be $\Theta(nt \log n)$, the same as the cost of exhaustive search. This means that even if we perform the pre-computation in advance, we cannot recover a key with less cost than exhaustive search, but we can find many keys with a total cost the same as exhaustive search. Keep in mind that this is a statement concerning asymptotics, and that there may be an advantage of a constant factor over exhaustive search due to the relative costs of processors and memory. When a is equal to the lower bound from the analysis of the pre-computation phase, $a = \Theta(n^{1/4}(\log n)^{1/4}/t^{3/4})$, the total number of keys that can be found for the cost of exhaustive search is $\Theta(n^{1/4}t^{9/4}/(\log n)^{3/4})$. It is interesting to note that fewer processors are used in the key recovery stage than in the pre-computation stage ($n^{3/4+o(1)}$ versus $n^{1/2+o(1)}$). It actually makes sense to leave the extra processors idle during key recovery. The higher memory access rate during recovery means that fewer processors can access the memory without driving up the wiring costs.

The time–memory trade-off can be improved using the idea of distinguished points. Suppose that some fraction of keys are considered distinguished (perhaps those with a certain number of leading zero bits). Then instead of iterating f on the starting keys exactly a times, we iterate until a distinguished key is reached, with the distinguishing property chosen so that about n/a of the keys are distinguished. The main difference this makes is in the key recovery phase where it is only necessary to consult the memory when one of the y_d is distinguished, thereby lowering the memory access rate by a factor of a to $r = \Theta((\log n)/(at))$. The total memory required can be reduced as well by interleaving the $\Theta(a)$ rounds of pre-computation with the rounds of key recovery. This allows each round of pre-computation to overwrite the results from the previous round, reducing memory requirements to $m = \Theta((n/a^2)\log n)$. The total number of processor steps across p processors for the rounds of pre-computation and recovery of s keys is $T = \Theta((n + sa^2)t)$. By Corollary 1, the full cost of this algorithm is $\Theta((t/p)(n + sa^2)(p + m + (pr)^{3/2}))$. After substituting for m and r in this expression, it can be shown that to limit the full cost to that of exhaustive search, $a = \Theta((n/s)^{1/2})$, and $p = \Theta(s)$, which makes the full cost $\Theta((nt/s)(s \log n + (s^{3/2}(\log n)/(n^{1/2}t))^{3/2}))$. The maximum number of keys that can be found for a total cost of $\Theta(nt \log n)$ is $s = \Theta(n^{3/5}t^{6/5}/(\log n)^{2/5})$.

For a full cost equal to the cost of a single exhaustive search ($n^{1+o(1)}$), Hellman's time–memory trade-off can find $n^{1/4+o(1)}$ keys, and a modified version can find $n^{3/5+o(1)}$ keys. The cost per recovered key can be reduced to $n^{1/3+o(1)}$ when using the table look up method to find $n^{1+o(1)}$ keys simultaneously.

7. Double Encryption

Double encryption consists of encrypting each plaintext block twice with two independent keys: $C = E_{k_2}(E_{k_1}(P))$. Because a meet-in-the-middle attack is possible [4], double encryption is widely believed to offer little advantage over regular single encryption. Here we do an asymptotic analysis of different attack approaches.

The size of the key space for double encryption is n^2, and we can use the techniques for recovering many keys at once with a chosen-plaintext attack from Section 6 replacing n with n^2. However, we focus here on a known-plaintext attack (with just enough plaintext to determine the keys uniquely) on one instance of double encryption. The best generic attack on one instance of single encryption seems to be exhaustive search. Attacking double encryption by exhaustive search is possible, but better approaches are a simple meet-in-the-middle attack and a version based on parallel collision search [21, Section 5.3].

A simple meet-in-the-middle attack on double encryption is based on the observation that $E_{k_1}(P) = E_{k_2}^{-1}(C)$ and proceeds as follows. For each possible key k_1, compute $E_{k_1}(P)$ and store $(k_1, E_{k_1}(P))$ in a hash table indexed by $E_{k_1}(P)$. For each possible k_2, compute $E_{k_2}^{-1}(C)$ and look it up in the table. Whenever, $E_{k_2}^{-1}(C)$ is in the table, we have a candidate key pair (k_1, k_2) that can be tested on other plaintext–ciphertext pairs. The memory size required is $m = \Theta(n \log n)$, and if encryption and decryption take time $t = n^{o(1)}$, then the memory access rate is $r = \Theta((\log n)/t)$, and the total number of processor steps is $T = \Theta(nt)$. By Corollary 2, the full cost of a meet-in-the-middle attack on double encryption is $F = \Theta(Trm^{1/3}) = \Theta((n \log n)^{4/3})$.

For parallel collision search applied to meet-in-the-middle attacks, if there are w memory locations (each of size $\Theta(\log n)$ so that $m = \Theta(w \log n)$), the total number of processor steps across p processors is $T = \Theta(n^{3/2}t/w^{1/2})$ [21], and a memory access of size $\Theta(\log n)$ is made every $(n/w)^{1/2}$ encryptions for a memory access rate of $r = \Theta(w^{1/2}(\log n)/(n^{1/2}t))$. By Corollary 1, the full cost of the algorithm is $F = \Theta((n^{3/2}t/(w^{1/2}p))(p + m + (pr)^{3/2}))$. This cost is a minimum of $\Theta(n^{6/5}t^{2/5}(\log n)^{4/5})$ when $p = \Theta(w \log n)$ and $w = \Theta(n^{3/5}t^{6/5}/(\log n)^{8/5})$.

In summary, the full cost of a known-plaintext attack on double encryption using a simple meet-in-the-middle attack is $n^{4/3+o(1)}$, and this can be reduced to $n^{6/5+o(1)}$ using parallel collision search. Therefore, double encryption does offer some security advantage over single encryption (based on known attacks).

8. Triple Encryption

Not surprisingly, triple encryption consists of encrypting each plaintext block three times, although the middle encryption is actually a decryption operation in most implementations. There are two main variants: three-key triple encryption has three independent keys, $C = E_{k_3}(E_{k_2}^{-1}(E_{k_1}(P)))$, and two-key triple encryption has only two independent keys with the key for the first encryption reused for the last encryption, $C = E_{k_1}(E_{k_2}^{-1}(E_{k_1}(P)))$.

Three-key triple encryption can be attacked as follows. For each possible value for key k_1, perform a double encryption attack using $E_{k_1}(P)$ as plaintext and C as ciphertext. For a key space of cardinality n, this increases the attack cost by a factor of n over the cost of attacking double encryption. Using the results from Section 7, the full cost of a known-plaintext attack on three-key triple encryption using a simple meet-in-the-middle attack is $n^{7/3+o(1)}$, and this can be reduced to $n^{11/5+o(1)}$ using parallel collision search.

Lucks gives an interesting attack that reduces the total number of processor steps by a constant factor at the cost of requiring more known plaintexts and more memory [10]. However, the larger memory causes this attack to have higher full cost than using parallel collision search.

The more interesting case is two-key triple encryption. Merkle and Hellman [12] describe an attack that requires n chosen plaintexts that proceeds as follows. Choose some fixed text M. For each possible key k, compute and store $(k, E_k^{-1}(M))$ in a hash table indexed by $E_k^{-1}(M)$. For each possible key k_1, compute $E_{k_1}^{-1}(M)$ and get it encrypted by the system under attack. Take the resulting ciphertext C, and compute $E_{k_1}^{-1}(C)$. If k_1 is the key being used by the system under attack for the first and last encryptions of the triple encryption, then those outer encryptions are stripped off, and we have $E_{k_1}^{-1}(C) = E_{k_2}^{-1}(M)$. Look up $E_{k_1}^{-1}(C)$ in the table to get a candidate value for k_2. Test k_1 and k_2 on other plaintext–ciphertext pairs. The memory size required is $m = \Theta(n \log n)$, and if encryption and decryption take time $t = n^{o(1)}$, then the memory access rate is $r = \Theta((\log n)/t)$, and the total time spent by all processors is $T = \Theta(nt)$. By Corollary 2, the full cost of this attack on two-key triple encryption is $F = \Theta(Trm^{1/3}) = \Theta((n \log n)^{4/3})$, the same cost as the simple meet-in-the-middle attack on double encryption if we ignore the need for an enormous number of chosen plaintexts.

A known-plaintext variant of the Merkle–Hellman attack requires fewer texts at the cost of more computation [20]. Suppose that there are w known plaintexts. Choose a constant M and seek a plaintext P such that $E_{k_1}(P) = M$ as follows. Store the plaintext–ciphertext pairs in a first hash table indexed on the plaintext values. For each possible key i, decrypt M and look up the resulting plaintext in this first table. If it is there, then decrypt the corresponding ciphertext with key i and put the text and i in a second table indexed on the text value. After the second table is constructed, then for each possible key j, decrypt M and look up the resulting text in the second table. For each appearance of the text in the second table, we have a candidate key pair (i, j) to be tested on other plaintext–ciphertext pairs. If this procedure does not succeed, then discard the second table and repeat with another randomly chosen M. The second table is large if the size of the message space u is less than n, and if u is small enough, the second table contains repeated values that cause the run-time to be dominated by the cost of checking candidate key pairs generated by the algorithm. To simplify the analysis of this attack, we assume that the size of the message space u is at least as large as the size of the key space ($u \geq n$). This attack requires memory size $m = \Theta(w \log n)$, a memory access rate of $r = \Theta((\log n)/t)$, and the total time spent by all processors is $T = \Theta(unt/w)$. By Corollary 2, the full cost of this attack on two-key triple encryption is $F = \Theta(Trm^{1/3}) = \Theta(nu(\log n)^{4/3}/w^{2/3})$, which is never less than the cost of the Merkle–Hellman attack. The two attacks have equal full cost (and are essentially the same attack) if $u = n$ and $w = u$.

9. Hash Collisions

Given a hash function $H(\cdot)$ whose output space has cardinality n, the cryptanalyst wishes to find a collision, which is two inputs x_1 and x_2 such that $x_1 \neq x_2$ and $H(x_1) = H(x_2)$. A simple approach is to repeat the following. Choose an input x that has not been tried yet, and add the pair $(x, H(x))$ to a hash table indexed by the $H(x)$ value. (There is an unfortunate name clash here: a hash function is a cryptographic operation, and a hash table is a type of list that allows fast access without having to sort list entries.) If the hash output $H(x)$ is already in the table, then we have a collision. The expected number of iterations before a collision is found is $\Theta(n^{1/2})$. If the time required to hash an input is $t = n^{o(1)}$, then this attack requires a memory size of $m = \Theta(n^{1/2} \log n)$, the memory access rate is $r = \Theta((\log n)/t)$, and the total number of processor steps is $T = \Theta(n^{1/2}t)$. By Corollary 2, the full cost of this attack is $F = \Theta(Trm^{1/3}) = \Theta(n^{2/3}(\log n)^{4/3})$ when the number of processors used is $p = \Theta(m^{2/3}/r) = \Theta(n^{1/3}t/(\log n)^{1/3})$.

The simple attack's large memory requirement can be eliminated using Pollard's rho method [15] adapted for hash collisions or the parallelized version [21]. These methods reduce the full cost of finding a hash collision to $\Theta(n^{1/2}t \log n)$ possibly using only a single processor.

10. Conclusion

We can now answer the question posed in the Introduction: if an algorithm with input n requires $\Theta(n)$ processor steps and $\Theta(n)$ memory elements, is its full cost $\Theta(n)$, $\Theta(n^2)$, or something in between? The answer depends on the number of processors used and the rate at which those processors access memory. If the algorithm cannot be parallelized, then its full cost is $\Theta(n^2)$. If it can be parallelized to an arbitrary degree, then its full cost is between $\Theta(n)$ and $\Theta(n^{4/3})$ depending on the memory access rate.

In general, if an algorithm takes T processor steps spread across p processors, with the processors accessing a common memory of size m at a rate r, the algorithm's full cost is $F = \Theta(T + Tm/p + Tp^{1/2}r^{3/2})$. The first term is processor costs, the second is memory costs, and the last is due to the cost of connecting the processors to the memory. For the full cost to match the traditional measure of algorithm cost, i.e., $F = \Theta(T)$, we must have $p = \Omega(m)$, and the memory access rate must be low, $r = O(p^{-1/3})$. This does not mean that in a real attack there must be as many processors as bits of memory; the number of processors may be $p = m/10^9$, for example.

If the full cost exceeds the number of processor steps for a particular approach to an attack, this can be viewed as a failure to parallelize the attack properly. For many attacks, it is not known whether such "proper" parallelization is possible.

Table 1 summarizes the costs of various cryptanalytic attacks using the best available implementation to minimize cost by both the standard measure of processor steps and the full cost measure.

In all cases in Table 1, the optimal design point did not have wire costs exceeding both processor and memory costs. This means that the table entries would be the same if wires were assumed to have no volume.

Acknowledgments

I thank Eran Tromer, Arnold Rosenberg, Arjen Lenstra, Bart Preneel, and the anonymous referees for helpful comments on drafts of this paper.

Table 1. Attack costs in processor steps and full cost.

Cryptanalytic problem	Attack method	Processor steps	Full cost
One discrete logarithm	Shanks	$n^{1/2+o(1)}$	$n^{2/3+o(1)}$
(prime group order n)	Parallel collision search	$n^{1/2+o(1)}$	$n^{1/2+o(1)}$
nth Discrete logarithm	Shanks	$n^{o(1)}$	$n^{1/3+o(1)}$
(prime group order n)	Parallel collision search	$n^{o(1)}$	$n^{1/5+o(1)}$
Factoring n	Number field sieve	$L(1.90188\ldots)$	$L(1.94961\ldots)$
Block cipher encryption	All methods (first key)	$n^{1+o(1)}$	$n^{1+o(1)}$
($\|$key space$\| = n$)	Table lookup (nth key)	$n^{o(1)}$	$n^{1/3+o(1)}$
Double encryption	Meet-in-the-middle	$n^{1+o(1)}$	$n^{4/3+o(1)}$
	Parallel collision search	$n^{1+o(1)}$	$n^{6/5+o(1)}$
Two-key triple encryption	Unlimited chosen texts	$n^{1+o(1)}$	$n^{4/3+o(1)}$
($\|$message space$\| = u \geq n$)	w known texts ($w \leq u$)	$n^{1+o(1)}u/w$	$n^{1+o(1)}u/w^{2/3}$
Three-key triple encryption	Meet-in-the-middle	$n^{2+o(1)}$	$n^{7/3+o(1)}$
	Parallel collision search	$n^{2+o(1)}$	$n^{11/5+o(1)}$
Hash collision	Meet-in-the-middle	$n^{1/2+o(1)}$	$n^{2/3+o(1)}$
($\|$output space$\| = n$)	Parallel collision search	$n^{1/2+o(1)}$	$n^{1/2+o(1)}$

References

[1] H.R. Amirazizi and M.E. Hellman, Time–Memory–Processor Trade-Offs, *IEEE Transactions on Information Theory*, vol. IT-34, no. 3 (1988), pp. 505–512.

[2] D. Bernstein, Circuits for Integer Factorization: A Proposal, Manuscript, Nov. 2001, available at http://cr.yp.to/nfscircuit.ps.

[3] D. Coppersmith, Solving Homogeneous Linear Equations over GF(2) via Block Wiedemann Algorithm, *Mathematics of Computation*, vol. 62, no. 205 (Jan. 1994), pp. 333–350.

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

[5] H. Handschuh and B. Preneel, On the Security of Double and 2-Key Triple Modes of Operation, *Fast Software Encryption* '99, 6*th International Workshop* (LNCS 1636), Springer-Verlag, Berlin, 1999, pp. 215–230.

[6] M.E. Hellman, A Cryptanalytic Time–Memory Trade-Off, *IEEE Transactions on Information Theory*, vol. IT-26 (1980), pp. 401–406.

[7] D.E. Knuth, *The Art of Computer Programming*, *vol.* 3: *Sorting and Searching*, second edition, Addison-Wesley, Reading, MA, 1998.

[8] A.K. Lenstra and H.W. Lenstra, Jr. (eds.), *The Development of the Number Field Sieve* (Lecture Notes in Mathematics 1554), Springer-Verlag, Berlin, 1993.

[9] A.K. Lenstra, A. Shamir, J. Tomlinson, and E. Tromer, Analysis of Bernstein's Factorization Circuit, *Advances in Cryptology—Asiacrypt* 2002 (LNCS 2501), Springer-Verlag, Berlin, 2002, pp. 1–26.

[10] S. Lucks, Attacking Triple Encryption, *Fast Software Encryption '*98, 5*th International Workshop* (LNCS 1372), Springer-Verlag, Berlin, 1998, pp. 239–253.

[11] A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, Boca Raton, FL, 1997.

[12] 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.

[13] P.L. Montgomery, A Block Lanczos Algorithm for Finding Dependencies over GF(2), *Advances in Cryptology—Eurocrypt'*95 (LNCS 925), Springer-Verlag, Berlin, 1995, pp. 106–120.

[14] S.C. Pohlig and M.E. Hellman, An Improved Algorithm for Computing Discrete Logarithms over GF(p) and Its Cryptographic Significance, *IEEE Transactions on Information Theory*, vol. IT-24 (1978), pp. 106–110.

[15] J.M. Pollard, Monte Carlo Methods for Index Computation (mod p), *Mathematics of Computation*, vol. 32, no. 143 (July 1978), pp. 918–924.

[16] A.L. Rosenberg, Three-Dimensional VLSI: A Case Study, *Journal of the ACM*, vol. 30 (1983), pp. 397–416.

[17] V. Shoup, Lower Bounds for Discrete Logarithms and Related Problems, *Advances in Cryptology—Eurocrypt* '97 (LNCS 1233), Springer-Verlag, Berlin, 1997, pp. 256–266.

[18] E. Teske, Speeding Up Pollard's Rho Method for Computing Discrete Logarithms, *Algorithmic Number Theory Symposium III* (LNCS 1423), Springer-Verlag, Berlin, 1998, pp. 541–554.

[19] E. Tromer, Personal communication.

[20] P.C. van Oorschot and M.J. Wiener, A Known-Plaintext Attack on Two-Key Triple Encryption, *Advances in Cryptology—Eurocrypt'*90 (LNCS 473), Springer-Verlag, Berlin, 1990, pp. 318–325.

[21] P.C. van Oorschot and M.J. Wiener, Parallel Collision Search with Cryptanalytic Applications, *Journal of Cryptology*, vol. 12, no. 1 (1999), pp. 1–28.

[22] P.M.B. Vitányi, Locality, Communication and Interconnect Length in Multicomputers, *SIAM Journal on Computing*, vol. 17 (1988), pp. 659–672.