Implementing several attacks on plain ElGamal encryption

by

Bryce Allen

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Mathematics

Program of Study Committee:
Clifford Bergman, Major Professor
Paul Sacks
Sung-Yell Song

Iowa State University

Ames, Iowa

2008

Contents

Chapter1. OVERVIEW

In “Why Textbook ElGamal and RSA Encryption are Insecure” [BJN00], several algorithms for attacking the plain ElGamal public-key cryptosystem are described. In this paper I explore the implementation in more detail and discuss the relative efficiency of different approaches. I also explore the use of external storage to reduce the memory requirements and allow the attacks to be run on larger messages.

1.1 Introduction

Public key cryptosystems generally have an elegant mathematical simplicity, and many textbooks describe them in these terms. However it is often the case that a simple implementation is not secure. In particular, [BJN00] describe several attacks on ElGamal and RSA which work well when the encrypted message is short and has not been preprocessed. The attacks on ElGamal also depend on the parameters used when creating the cryptosystem. The required parameters, while not commonly discussed in textbooks, are commonly used in practical implementations.

Computer scientists have been trying to formalize the idea of what makes a secure cryptosystem. However these formal definitions often seem far stronger than necessary for practical security — many cryptosystems used in practice do not satisfy the formal definitions. This includes typical ElGamal implementations. The attacks discussed in this paper suggest that it is worth striving to meet formal definitions of security in actual implementations.

1.2 Hybrid Cryptosystems

A *cryptosystem* describes a way for two or more parties to communicate in secret over a public channel. The content of the communication, which may be human language or anything else, is called the *plaintext*. The heart of a cryptosystem is a cipher, which specifies rules for encryption and decryption. The encryption rule takes the plaintext and a pre-determined key, and produces a *ciphertext* which hides the content of the original plaintext. The decryption rule takes another key, possibly different from the encryption key, and recovers the plaintext from the ciphertext.

There are two basic types of cryptosystems: symmetric key (private key) systems and public key (asymmetric) systems. Public key systems are much slower than symmetric systems, but symmetric systems require key agreement through an existing secure channel. Hybrid cryptosystems combine them to gain the advantages of both.

1.2.1 Symmetric key cryptosystems

Symmetric key systems use the same key for both encryption and decryption. In order to communicate securely using a symmetric system, two partyies must agree on the key using some pre-existing secure channel. When more than two parties are involved key distribution becomes even more complicated, and historically key distribution has been a major obstacle for practical uses of cryptography. Typical symmetric ciphers use very convoluted transformations to obscure any patterns in the original message. The key controls how the transformations operate, and provides a map for reversing the transformations during decryption.

Examples of symmetric key systems include the Data Encryption Standard (DES), the Advanced Encryption Standard (AES), and Skipjack. DES is no longer considered to provide adequate security, and AES is the recommended symmetric key algorithm for new applications. However DES was the recommended standard from 1976 until 1999. Many legacy systems, including embedded systems which cannot be easily updated with software patches, use DES. DES uses a 56-bit key and AES uses keys of 128 bits or more. Skipjack was developed by the U.S. National Security Agency, and was declassified in 1998. It uses 80-bit keys. All of the ciphers mentioned are block ciphers - they encrypt and decrypt data in chunks.

1.2.2 Public key cryptosystems

Public-key cryptosystems help solve the key distribution problem by using separate keys for encryption and decryption, and making the encryption key public. Anyone can then encrypt a message, but only parties in possession of the private key can decrypt messages. Public key systems rely on one-way trap door functions, which are interesting mathematical functions that can be easily computed in one direction but are very difficult to reverse unless a secret key is known (the trap door). Since the encryption key is made public, finding the private decryption key from the public encryption key must be intractable.

One application of public-key cryptography is secure email. Public keys are typically published on a user's website. However if the user's website is compromised, a different public key corresponding to a malicious adversaries private key can be substituted. For this reason, public-key cryptography doesn't completely solve the key distribution problem. However digital signatures can be used to fill the remaining gaps, allowing users to build a web of trusted public keys, and accept new keys if they are signed by an already trusted public key.

ElGamal is a public key system which uses modular exponentiation as the basis for a one-way trap door function. The reverse operation, the discrete logarithm, is considered intractable. ElGamal was never patented, making it an attractive alternative to the more well known RSA system. Public key systems are fundamentally different from symmetric systems, and typically demand much larger keys. 1024 bits is the minimum recommended size for ElGamal, and even larger keys are recommended for some applications.

1.2.3 Combining the two systems

Symmetric systems are far faster than public key systems — AES, for example is roughly 10000 times faster than RSA. Efficiency is important for real-time communication and bulk encryption and decryption of large data sets.

In a hybrid system, a public key system is used to negotiate a session key — one party generates a random session key, encrypts it using the other party's public key, and sends the encrypted message. The receiving party decrypts the session key message with their private key, and now both parties know the session key. The session key is then used to encrypt and decrypt all remaining communication with a symmetric cipher.

1.2.4 Short messages and public key systems

Since symmetric systems like DES pre-date public key systems, there has never been a good reason to use public key systems for encryption and decryption except as part of a hybrid system. For this reason, the primary message payload for public key systems is session key negotiations. It is therefore reasonable to expect a legacy system using DES and ElGamal in a hybrid system to send 56-bit DES keys encrypted using ElGamal.

1.3 Implementation

I implemented two of the attacks discussed in [BJN00]: the basic meet-in-the-middle attack and the two-table attack. Both attacks perform meet-in-the-middle searches of part of the message space. I implemented several variations of the basic meet-in-the-middle attack, one of which uses external storage instead of main memory. Source code is available at `http://www.math.iastate.edu/brycea/thesis2008.html` under the MIT license.

All attacks were implemented in C++ with the GNU Multiple Precision Arithmetic Library (GMP) [gmp]. While GMP has been heavily optimized as a general purpose big number library, it has not been optimized for cryptography. In particular it does not allow the programmer to specify that a certain type of reduction should be used to perform modular exponentiations or multiplications. Depending on the choices made by GMP internally, it may be possible to significantly improve the performance of the attacks.

The different attacks were run on ciphertexts of different size messages to determine how the attacks scale and how they compare to one another. The results are given in Chapter 5.

1.4 Splitting Probabilities

The attacks discussed in this paper start by assuming that the message is a positive integer and can be factored into two or more integers of given sizes. Assuming the message is an integer does not limit the attacks; ElGamal already requires that messages be converted to a positive integer before encryption. A DES key, for example, is a string of 56-bits; however that string of bits can also be interpreted as a 56-bit unsigned integer. We may then assume that the DES key splits into two factors of 28 bits each. Not all 56-bit integers will split in this way, so an attack with that splitting assumption will fail on many messages. However the probability of success is still fairly high.

Table 1.1 lists some splitting probabilities obtained by factoring 100000 random numbers m in the range $1 \ldots 2^b - 1$ and checking for splits $m_1 m_2 = m$ with $m_1 \leq 2^{b_1}$ and $m_2 \leq 2^{b_2}$. The last column includes results from [BJN00], which differ slightly from my results.

For each attack, the choice of b_1 determines the time and space required for the pre-computation step, which only needs to be done once for a given cryptosystem, and b_2 determines the time required to crack a specific message. Thus we can choose different values for b_1 and b_2 to obtain different trade-offs between pre-computation and crack time, memory requirements, and success probability.

See [BJN00] for some analytic results about splitting probabilities.

Table 1.1 Experimental splitting probabilities

b	b_1	b_2	Probability	Probability [BJN00]
40	20	20	20%	18%
	21	21	33%	32%
	22	22	40%	39%
	20	25	54%	50%
56	28	28	18%	
	29	29	30%	
	30	30	36%	
	26	34	49%	
64	32	32	17%	18%
	33	33	29%	29%
	34	34	34%	35%
	30	36	40%	40%

Chapter2. THE ELGAMAL CRYPTOSYSTEM

An ElGamal cryptosystem operates in a finite cyclic group, which by convention is written multiplicatively. For simplicity we restrict our discussion to the two most common choices: the group of integers from 1 to $p-1$ under multiplication mod p for some prime p, commonly called $\mathbb{Z}_p^*$, and subgroups of $\mathbb{Z}_p^*$ of prime order. We will use $|g|$ to denote the order of an element g in $\mathbb{Z}_p^*$ and $\langle g \rangle$ to denote the cyclic subgroup of $\mathbb{Z}_p^*$ generated by g. Unless otherwise noted, assume multiplications and exponentiations involving elements of $\mathbb{Z}_p^*$ are done mod p.

I begin the discussion of ElGamal with the discrete log problem, since its intractability is central to the security of ElGamal.

2.1 The Discrete Log Problem

The standard logarithm is the inverse operation of standard exponentiation. Similarly we define the *discrete logarithm* to be the inverse of modular exponentiation: given a modular exponentiation $y = g^x$ in $\mathbb{Z}_p^*$ and the base g, the discrete logarithm $\log_g y$ is x. This is a discrete logarithm in the cyclic group $\langle g \rangle$ which may or may not be all of $\mathbb{Z}_p^*$. When $|g| = n$ is large and has at least one large prime factor, discrete log problems in $\langle g \rangle$ are considered intractable.

There are three basic types of discrete log algorithms: "square-root" algorithms such as Pollard's rho algorithm, the Pohlig-Hellmen algorithm, and index calculus algorithms.

Pollard's rho algorithm can compute discrete logs in a cyclic group of prime order n in time $O(\sqrt{n})$ and negligible space. If n is not prime and the factorization of n is known, then the Pohlig-Hellman algorithm can be used. If $n = p_1^{e_1} p_2^{e_2} \dots p_c^{e_c}$ is the prime factorization of n, then the Pohlig-Hellman algorithm computes partial solutions by computing discrete logs in subgroups of order p_i for $i = 1 \dots c$. Typically Pollard's rho algorithm is used as a subroutine to compute these logarithms, and the partial solutions are combined to compute the requested discrete log. The runtime of Pohlig-Hellman is $O(\sum_{i=1}^{c} e_i(\log n + \sqrt{p_i}))$, assuming n has the prime factorization given above. In particular, if n is *B-smooth*, meaning that none of it's prime factors are greater than B, the runtime of the Pohlig-Hellman algorithm is $O(\ln\ln n(\log n + \sqrt{B}))$, since the average number of not necessarily distinct prime factors is $\sim \ln\ln n$. If n is at most 256 bits and has no factors of more than 16 bits, i.e. n is $(2^{16} - 1)$-smooth, then we can expect the Pohlig-Hellman algorithm to require only $O(2^{12})$ operations. When Pollard's rho algorithm is used with the Pohlig-Hellman algorithm, the combined algorithm also uses negligible space. If n has a large prime factor neither of these algorithms work well.

Index calculus algorithms do not work in a general cyclic group, but they do work in $\mathbb{Z}_p^*$ and they run in sub-exponential time. For example the number field sieve has an expected running time of $O(e^{(1.923+O(1))(\ln p)^{1/3}(\ln\ln p)^{2/3}})$. Index calculus methods do not work directly on subgroups of $\mathbb{Z}_p^*$; however it can be used to compute logs in subgroups by computing logs in $\mathbb{Z}_p^*$. For this reason, if $n \ll p$ then a square-root algorithm such as Pollard rho (or Pohlig-Hellman if n is composite) may be faster than index calculus methods, depending on the exact relationship between n and p [MVO96].

2.2 Encryption and Decryption

An ElGamal cryptosystem can be described by a 4-tuple (p, g, x, y), where p is a large prime and describes which group $\mathbb{Z}_p^*$ is used, g is an element of order n in $\mathbb{Z}_p^*$, x is a random integer with $1 \leq x \leq n - 1$, and $y = g^x$. If g is primitive, then $n = p - 1$. However this is not required, and n may be chosen to be much smaller than $p - 1$ for efficiency reasons. The public key is (p, g, y), and the private key is x.

Before a message can be encrypted, it must be converted to an integer between 1 and $p-1$, i.e. an element of $\mathbb{Z}_p^*$. If the message is the key for a symmetric cipher, it may already be a number. If the message is larger than $p - 1$, it can be broken into blocks. Many textbooks impose a further restriction that the message is a member of $\langle g \rangle$ [MVO96] [Sti05]; however in practice implementations often ignore this restriction. If g is not primitive, then only n of the $p-1$ members of Z_p^* will be in $\langle g \rangle$. In this case, it is not clear how to convert messages to an element of $\langle g \rangle$. Raising g to the power of the message does not work, since a discrete log would be required to retrieve the original message.

The encryption function requires a random integer $k \in [1, n-1]$ in addition to the public key. The encryption and decryption functions are:

$$E_k(m) = (g^k, my^k) \text{ and } D(u, v) = u^{-x}v.$$

where all operations are done mod p (in $\mathbb{Z}_p^*$). The decryption function will recover the original message: $u^{-x} = g^{-kx} = (g^x)^{-k} = y^{-k}$, so $D(E_k(m)) = u^{-x}v = y^{-k}my^k = m$.

2.3 Security

If we recover the private key x, we can decrypt all past and future messages. Since the public key includes $y = g^x$ and g, finding the private key from the public key amounts to computing a single discrete logarithm in $\langle g \rangle$. For this reason, n and p should be very large — n determines the runtime of square-root discrete log algorithms like Pollard's rho algorithm, and p determined the runtime of index calculus discrete log methods.

The key size typically refers to the size of p; 1024 is the recommended minimum. Legacy implementations running on limited hardware may use 768 bits or even less. If $n < p-1$ is used, it should be large enough that the $O(\sqrt{n})$ discrete log algorithms take at least as long as the index calculus algorithms.

To break a single ciphertext $(u, v) = (g^k, my^k)$, it would suffice to find y^{-k}, since $m = vy^{-k}$. Since inverses can be computed efficiently, we really just need y^k. We can find k by computing the discrete log of $u = g^k$ base g — however this may not be necessary. Cracking a single ciphertext is equivalent to the Diffie-Hellman problem: given g^k and $y = g^x$, determine $g^{kx} = y^k$. Since a discrete log can be used to solve the Diffie-Hellman problem, it is not any more difficult than the discrete log problem; however it is not known whether or not it is less difficult.

2.4 Efficiency

Consider the ElGamal cryptosystem (p, g, x, y) where $n = |g|$ is the order of g. Encryption requires two exponentiations and one multiplication in $\mathbb{Z}_p^*$. Since the exponent k is chosen between 1 and $n-1$, using a large n will slow down exponentiation and therefore encryption. The single multiplication will not be significant compared to the two exponentiations.

Decryption requires one exponentiation, one inversion, and one multiplication in $\mathbb{Z}_p^*$. The private key x is the exponent, and it is also chosen between 1 and $n-1$. Therefore choosing a smaller n could also increase decryption performance; however there are other approaches, like picking an x with very few ones in it's binary representation. Because of the way modular exponentiation is implemented, this will reduce the computation time.

Decreasing p also increases performance, since multiplications in $\mathbb{Z}_p^*$ are faster for smaller p. This gives the implementer with performance constraints a choice: decrease p and use $n = p - 1$, or keep p large and choose $n \ll p - 1$. According to [BJN00] the latter choice is not uncommon in actual implementations, likely because the faster index calculus discrete log methods depend on p and not n.

2.5 Brute Force Attacks on a Hybrid System

Brute force attacks are a class of attacks on cryptosystems characterized by exhaustive searching and very little ingenuity. The classic example is recovering a plaintext from a ciphertext by decrypting the ciphertext with every possible key. This will generate a bunch of false plaintexts which look like gibberish, and with high probability only one which looks like it could be the real plaintext. While it may be very easy for a human to determine what a real plaintext should look like, for this attack to work a computer program must be able to determine if a plaintext looks real. In practice this is often possible if the real plaintexts contain structured data. Decrypting a ciphertext using the wrong key will likely produce a false plaintext which has a roughly uniform distribution of bytes. Decryptions with very irregular distributions are more likely to be the correct plaintext.

I will outline two brute force attacks on the ElGamal cryptosystem. There are $n-1$ possible values for the private key x. Given the ElGamal ciphertext of a 64-bit session key plaintext, we could decrypt it using each possible value of the private key. Every decryption that is at most 64 bits is a potential value for the session key, but if $n \ll p - 1$ there will be very few such decryptions. Even if this is the case and n is only 256 bits, we require $O(2^{256})$ modular exponentiations which is intractable.

We could instead compute all possible encryptions of all possible messages until one is found which matches the ciphertext. However this is even worse. If n is 256 bits and the message is a 64-bit session key, then we must compute at most $2^{64}(n-1)$ encryptions, so the attack will take $O(2^{320})$ encryptions, each taking two modular exponentiations.

If ElGamal is used as part of a hybrid cryptosystem, the actual data is encrypted with a symmetric cipher using the session key. In this case it will likely be easier to attack the symmetric cipher directly. If we have some way of recognizing real plaintexts, we can decrypt the ciphertext with all 2^{64} possible session keys until we find a decryption that looks like plaintext. This attack requires $O(2^{64})$ symmetric cipher decryptions and plaintext tests, which is orders of magnitude faster than the other brute force attacks.

Chapter3. MEET-IN-THE-MIDDLE ATTACK

3.1 Requirements and Assumptions

For this attack we assume that the adversary has intercepted a ciphertext (u, v) and knows which public key (g, p, y) was used to encrypt the message. Only the second part $v = my^k$ of the ciphertext is used.

The attack works well under the following conditions:

1. The original message m is at most b bits, b is small, and the adversary is aware of this limit. For example, the adversary may know that the message is a 56-bit DES key. The attack becomes infeasible for messages much larger than 64 bits — in particular there is no hope of using this attack on encryptions of a Skipjack (80-bit) or AES (128-bit) session key.

2. m can be factored (split) into two factors of at most b_1 and b_2 bits respectively. The probability of different splits is discussed in section 1.4.

3. The order n of g in $\mathbb{Z}_p^*$ is known. If $p - 1$ has only one large prime factor, then it can be factored efficiently using a combination of trial division, Pollard's rho algorithm for factoring, and primality testing. In that case, or if the factorization of $p - 1$ is already known, n can be computed efficiently. Otherwise there is no known efficient algorithm for computing n.

4. Messages are not represented as elements of $\langle g \rangle$. For ElGamal to work, the messages must be represented as members of $\mathbb{Z}_p^*$ — however the restriction to $\langle g \rangle$ is not necessary, although highly recommended based on the success of this attack.

5. $n \leq (p-1)2^{-b}$. This condition ensures that given an element v^n of order dividing $(p-1)/n$ in $\mathbb{Z}_p^*$, the expected number of distinct messages m such that $m^n = v^n$ is small.

The attack will not succeed 100% of the time, since we assume that the message splits in some way. For example if the message is 56 bits and we choose parameters $b_1 = b_2 = 28$, we have a 18% success probability. However this does not lessen the impact of the attack by much — if 18% of the credit cards numbers passing through a credit processor are stolen it's nearly as much of a disaster as 100% stolen.

3.2 The Attack

One of the strength of ElGamal is its non-determinism — encrypting the same plaintext multiple times will result in different ciphertexts, since a random k is chosen each time. However the non-deterministic term y^k has order dividing n, so if we raise $v = my^k$ to the n power we eliminate y^k:

$$v^n = m^n(y^k)^n = m^n(g^{xk})^n = m^n(g^n)^{kx} = m^n.$$

Note that there may be other messages $\tilde{m}$ such that $\tilde{m}^n = m^n = v^n$. However under reasonable assumptions this is unlikely. This is discussed in the next section.

A possible attack is to perform a brute force search for message $\tilde{m}$ such that $\tilde{m}^n = v^n$. However if the message is a 56-bit session key and modular exponentiations can be computed in one microsecond, this search will take over 1000 years on average. This is where the splitting assumption comes into play. We limit our search to message $\tilde{m}$ which can be factored as $\tilde{m} = \tilde{m}_1\tilde{m}_2$ with $\tilde{m}_1 \leq 2^{b_1}$ and $\tilde{m}_2 \leq 2^{b_2}$. In that case:

$$\begin{aligned} v^n &= \tilde{m}^n = \tilde{m}_1^n\tilde{m}_2^n \\ v^n\tilde{m}_2^{-n} &= \tilde{m}_1^n. \end{aligned}$$

The idea of the attack is to compute $\tilde{m}_1^n$ for $\tilde{m}_1 = 1 \ldots 2^{b_1}$, store the (key, value) pairs $(\tilde{m}_1^n, \tilde{m}_1)$ in a dictionary, and then compute $v^n\tilde{m}_2^{-n}$ for $\tilde{m}_2 = 1 \ldots 2^{b_2}$ and look up the values in the dictionary. If a match is found, it means that $v^n\tilde{m}_2^{-n} = \tilde{m}_1^n$, so $\tilde{m} = \tilde{m}_1\tilde{m}_2$ is a candidate for the original message. The dictionary depends only on the public key and b_1, so it can be re-used for multiple messages.

If every message is represented as a member of $\langle g \rangle$, then $v^n = 1$ for every message and this attack fails completely.

3.3 Solution Collisions

If $\tilde{m} \in \mathbb{Z}_p^*$, then $\tilde{m}^n$ will have order dividing $(p-1)/n$, i.e. will be in the subgroup of order $(p-1)/n$. Given v^n, we wish to calculate the number of expected messages $\tilde{m}$ not equal to the actual message m such that $\tilde{m}^n = m^n = v^n$. Let X_c be the random variable representing this quantity. We will assume that there are 2^b possible messages and that the values $\tilde{m}^n$ for $\tilde{m} = 1 \ldots 2^b$ are roughly uniformly distributed in the subgroup of order $(p-1)/n$. In that case, $\tilde{m}^n = v^n$ with probability $\frac{1}{(p-1)/n} = \frac{n}{p-1}$. X_c then has the binomial distribution with associated probability $\frac{n}{p-1}$ and $2^b - 1$ trials. If $n \leq (p-1)2^{-b}$, then

$$E[X_c] = \left(2^b - 1\right)\left(\frac{n}{p-1}\right) < \left(2^b\right)\left(\frac{(p-1)2^{-b}}{p-1}\right) = 1.$$

The attack will only find splitting messages, so the actual expected number of collisions is $E[X_c]$ times the splitting probability. In practice $n \ll (p-1)2^{-b}$, in which case there will be no collisions with high probability. For example if $p-1$ is 1024 bits, n is 512 bits, and $b = 64$ (the messages are 64 bits), $E[X_c] \sim 2^{-448}$.

3.4 Implementation

3.4.1 Dictionary data structure

The attack is fairly straight forward, but we need to select a suitable data structure for the dictionary.

The dictionary needs to support efficient insert and search routines, so the most obvious choice would be to use a hash table. However the approach suggested by [BJN00] is to use a sorted array. Instead of "inserting" into a data structure, we simply store all the (key, value) pairs in the array as we generate them and sort the array (by the keys) at the end. Lookup is implemented in $O(\log n)$ using binary search. I call this basic implementation *mim*, short for meet-in-the-middle. In practice the sort and binary search time are insignificant compared to the modular exponentiations and multiplications, calling into question whether the complexity of a hash table is necessary.

The sort will require $O(n \log n)$ operations, but the operations are much faster than the modular exponentiations used to generate the array keys. The space requirement is also very low: $O(1)$ with heapsort and $O(\log n)$ with a clever implementation of quicksort, for example.

The author used the *qsort* and *bsearch* routines in the standard C library. Custom binary search routines were used for some variations.

3.4.2 Reducing space requirements

Reducing the size of the dictionary will allow us to crack larger message without being forced to use slow external storage. For example, if $b = 64$ and we choose $b_1 = b_2 = 32$, and each entry in the dictionary requires s bytes, the dictionary will require $4s$ gigabytes. If p is 1024 bits, then most elements of $\mathbb{Z}_p^*$, in particular $\tilde{m}_1^n$, take up 128 bytes. The values of $\tilde{m}_1$, however, are only 4 bytes each. This means that if we store entire $(\tilde{m}_1^n, \tilde{m}_1)$ pairs in the array, the dictionary will require over 512GB. This is not going to fit in system memory. [BJN00] suggests storing $(\text{hash}(\tilde{m}_1^n), \tilde{m}_1)$ in the table instead, where hash() is a suitable hash function. To find $u^n \tilde{m}_2^{-n}$, we first compute $\text{hash}(u^n \tilde{m}_2^{-n})$ and then do a binary search. hash() will likely have collisions, so the search may find multiple possible values for $\tilde{m}_1$. For each match, we recompute $\tilde{m}_1^n$ and test for equality with $u^n \tilde{m}_2^{-n}$. If the number of matches is much less than 2^{b_2}, the extra exponentiation calculations required will not make a significant contribution to the run time.

The expected number of matches depends on the size of the hash values. Suppose the hash values are h bits and assume that the values of $\text{hash}(\tilde{m}_1^n)$ are uniformly distributed from 0 to $2^h - 1$. The probability that $\text{hash}(\tilde{m}_2^n)$ matches a specific element in the table for a given value of $\tilde{m}_2$ is 2^{-h}. There are 2^{b_1} entries in the table, so the expected number of matches for a single value of $\tilde{m}_2$ is 2^{b_1-h}, and the total number of expected matches is $2^{b_2} 2^{b_1-h} = 2^{b-h}$. h should therefore be chosen so that $b - h \ll b_2$.

hashmim implements this approach using a hash function which simply outputs the lower 32 bits of its input. With $b = 46$ and $b_2 = 23$, the expected number of extra exponentiations is 2^{14}. This is approximately 0.2% of 2^{23}, so we expect only a 0.2% increase in message crack time when comparing hashmim to plain mim. For $b = 64$ and $b_1 = b_2 = 32$, a hash function with $h \geq 40$ would be needed.

3.4.3 Using external storage

The hashmim implementation still requires over 36GB to store the dictionary when $b_1 = 32$. While there are machines with this much main memory, the author only had access to a computer with 6GB. An efficient implementation using external storage could also be used to crack even larger messages.

diskmim uses the lower 32 bits, just like hashmim, but uses a Tokyo Cabinet [Hir] B+ tree database to implement the dictionary.

3.5 Running Time and Memory Usage

The pre-computation requires 2^{b_1} modular exponentiations, regardless of what data structure is used for the dictionary. The mim and hashmim attacks require an $O(b_1 2^{b_1})$ sort - however for sizes of practical import, the exponentiation dominates since the constants involved are much higher. The space requirement is 2^{b_1} table entries. For the hash implementation, each entry is only 8 bytes. If b_1 is greater than 32 more space will be required for each entry, but by that point we will need to use external memory anyway. For mim, each entry is $\log p + b_1$ bits plus the overhead associated with using a big integer type.

diskmim again requires 2^{b_1} exponentiations, but requires 2^{b_1} inserts instead of the sort. Each insert requires $O(h)$ disk accesses, where h is the height of the $B+$ tree. Since $B+$ trees maintain balance, $h \approx \log_t 2^{b_1} = b_1 \log_t 2$, where t is the minimum branching factor. Therefore we can expect the table build to run in $O(b_1 2^{b_1})$. However the operations involved are now disk accesses as well as in memory searches within a node, so we expect this implementation to run more slowly.

All attacks require $O(2^{b_2})$ modular exponentiations to attack a specific message once the table is built. mim and hashmim require $O(2^{b_2})$ binary searches of the table, each running in $O(\log 2^{b_1}) = O(b_1)$ time, for a total complexity of $O(b_1 2^{b_2})$. diskmim requires $O(b_1 2^{b_2})$ disk accesses, since each B+ tree search requires $O(h) = O(b_1)$ disk accesses.

3.6 Comparison to Brute Force

As discussed in section 2.5, the most effective brute force attack on a hybrid system is usually a direct attack on the symmetric cipher. If a b-bit session key is used, then the expected runtime is $O(2^b)$. If $b_1 = b_2 = b/2$, then the runtime of both phases of the meet-in-the-middle attack will be $O(2^{b/2+1})$.

As an example, consider $b = 56$ and $b_1 = b_2 = 28$. Brute force will take on average 2^{55} symmetric cipher decryptions and plaintext tests. If decryptions and plaintext tests can be done in one nanosecond, it will take more than a year to complete the attack. The meet-in-the-middle attack will only succeed 18% of the time, but it will only take about 5 days to complete.

Chapter4. TWO TABLE ATTACK

4.1 Introduction

The two table attack is a refinement of the meet-in-the-middle attack which works when $\mathbb{Z}_p^*$ has a subgroup in which discrete logs can be computed efficiently. For example if $p-1$ has a B-smooth factor s with B sufficiently small, say 2^{10}, the Pohlig-Hellman algorithm can be used to efficiently compute discrete logarithms in the subgroup of $\mathbb{Z}_p^*$ of order s.

Roughly speaking, this attack uses discrete logarithms in the pre-computation phase to replace modular exponentiation with additions in the message cracking phase. Again the adversary requires only the second part $v = my^k$ of the ciphertext and the public key (p, g, y). All the requirements and assumptions of the basic meet-in-the-middle attack apply to this attack as well, except that now we require $s > 2^b$ to ensure that the expected number of solution collisions is small. A splitting assumption is still used, so b_1 and b_2 can be chosen for different time, space, and success probability trade-offs.

4.2 The Attack

Let $p - 1 = nrs$ with s smooth. s will be easily factorable using trial division, and we can therefore efficiently find an element α of $\mathbb{Z}_p^*$ which generates the subgroup of order s [Sho05].

Instead of raising v to the n power, we raise v to the nr power. If a is any member of $\mathbb{Z}_p^*$, $(a^{nr})^s = a^{p-1} = 1$, so a^{nr} will be an element of the subgroup $\langle\alpha\rangle$ of order s, allowing us to compute the discrete logarithm base α. Again we suppose that $\tilde{m}_1$ and $\tilde{m}_2$ are factors of $\tilde{m}$

with bit size at most b_1 and b_2, respectively. If v is a ciphertext for $\tilde{m}$, then

$$\begin{aligned} v &= y^k \tilde{m} = y^k \tilde{m}_1 \tilde{m}_2 \\ v^{nr} &= (y^k)^{nr} \tilde{m}_1^{nr} \tilde{m}_2^{nr} \\ v^{nr} &= \tilde{m}_1^{nr} \tilde{m}_2^{nr} \\ \log v^{nr} &= \log \tilde{m}_1^{nr} + \log \tilde{m}_2^{nr} \end{aligned}$$

where all logarithms are base α. Solution collisions are discussed in the next section.

For the pre-computation step, we build two tables T_1 and T_2, where T_1 contains pairs $(\log \tilde{m}_1^{nr}, \tilde{m}_1)$ for $\tilde{m}_1 = 1 \ldots 2^{b_1}$, and T_2 contains pairs $(\log \tilde{m}_2^{nr}, \tilde{m}_2)$ for $\tilde{m}_2 = 1 \ldots 2^{b_2}$. In the cracking phase, we wish to find two pairs (t_1, v_1) and (t_2, v_2) in the table such that $\log v^{nr} = t_1 + t_2 \bmod s$. If we find such a pair, $v_1 v_2$ is a possible plaintext. Under the right conditions, this solution will be unique with high probability and $m = v_1 v_2$.

The task of expressing an integer as the sum of k other integers from k different tables is called the k-table problem. This attack uses the specific case where $k = 2$. Note that we no longer need a dictionary data structure.

The basic idea for solving the two table problem is to sort both tables by the first coordinate — T_1 in ascending order and T_2 in descending order — then test if the heads of each list sum to the target, and if not advance the head pointer on one of the lists according to whether the sum was larger or smaller than the target. However here we wish to find equality mod s, so we require some modifications. We still sort T_1 in ascending order and T_2 in descending order, which would be done in the pre-computation phase. Let $t = \log v^{nr}$ be the target, and note that the problem can be rephrased as finding (t_1, v_1) and (t_2, v_2) in T_1 and T_2, respectively, such that $t_1 = t - t_2 \bmod s$. For a fixed t, we can define a virtual table T_2' from T_2 containing the values $(t - t_2 \bmod s, v_2)$ for each $(t_2, v_2) \in T_2$. The smallest element of T_2' will not necessarily be at the first position, but the T_2' will still be in circular order. However since we subtract elements it will be in ascending circular order. Note that if $t_2 = t+1$ is in T_2, then $t-(t+1) \bmod s = s-1$ will be the largest element in T_2'. We perform a binary search for $t+1$ in T_2, and if $t+1$ is found we return the index. If $t+1$ is not found, the binary search will have zeroed in on the indexes between which $t+1$ would occur if it were present. The smaller of these indexes will give us the smallest element $\hat{t_2}$ in T_2 greater than $t+1$, and the corresponding element $t-\hat{t_2}$ in T_2' will be the largest element of T_2'. The following element will be the smallest element - e.g. if t is in T_2, then $t-t=0$ is the smallest element in T_2'.

Having determined the structure of T_2', our task is to find elements (t_1, v_1) in T_1 and (t_2', v_2) in T_2' such that $t_1 = t_2'$. We compare the target t to the sum of the heads of T_1 and T_2'. If $\operatorname{head} T_1 < \operatorname{head} T_2'$, then we advance the head pointer of T_1. If they're equal we have found a potential solution. Otherwise we advance the head of T_2'.

4.3 Solution Collisions

If there are 2^b possible messages and we assume that the values of v^{nr} are uniformly distributed in the subgroup of order s, then the expected number of solution collisions $E[X_c]$ will be:

$$E[X_c] = \left(2^b - 1\right)\left(\frac{nr}{p-1}\right) = \left(2^b - 1\right)\left(\frac{1}{s}\right).$$

In particular if $s > 2^b$ then $E[X_c] < 1$. Again the actual expected number of collisions will be $E[X_c]$ times the splitting probability associated with the b_1 and b_2 used for the attack.

4.4 Implementation

4.4.1 Discrete logs

This attack requires a discrete log algorithm which works well in a group of smooth order. The author used the Pohlig-Hellman algorithm together with Pollard's rho algorithm. Trial exponentiation is used instead of Pollard rho for very small prime powers. See section 2.1 for a brief discussion of these algorithms and their run-time.

4.4.2 Using only one table

If $b_1 = b_2$, then the tables T_1 and T_2 will be identical. In this case there is no need to store multiple copies - if T_1 is sorted in ascending order, we can treat it as a list sorted in descending order by inverting all comparisons, starting indexing at the end of the list, and traversing in the reverse order. This will halve the space requirement and halve the sorting time.

4.5 Running Time and Memory Usage

Let $b_m = \max(b_1, b_2)$. The pre-computation step requires 2^{b_m} modular exponentiations and 2^{b_m} discrete logarithms, and the power for the modular exponentiations is now larger (nr vs n). We therefore expect this pre-computation to run much more slowly than that of the other attacks but to scale similarly for a fixed smooth factor s.

Cracking a message, on the other hand, is much faster. We need to compute one modular exponentiation and one discrete log to compute the target. When searching the tables, will will examine at most $2^{\min(b_1,b_2)}$ candidate pairs t_1, t_2'. For each candidate pair tested, we compute $t_2' = n - t_2 \bmod s$ and perform comparison. These operations are orders of magnitude faster than modular exponentiation when large numbers are involved.

If $b_1 = b_2$, $T_1 = T_2$ will have $2^{b_1} = 2^{b_2}$ entries. Each entry is a pair $(\log a^{nr}, a)$, where a is at most $b_1 = b_2$ bits and the log is $O(\log s)$ bits. Since s may require more bits then the word size, we may require a large integer type to store $\log a^{nr}$. Note that storing just the hash of the log is not sufficient, we need the full value.

When $b_1 \neq b_2$, the size requirement jumps to $2^{b_1} + 2^{b_2}$ table entries.

4.6 Three and Four Table Attacks

This attack can be extended to three and four table attacks [BJN00]. These attacks assume the message splits into three or four factors, so they have much lower probability of success. However the four table attack in particular requires far less memory and computation, making the attack feasible for larger messages.

Chapter5. RESULTS

5.1 Test Data

The ElGamal cryptosystem used in generating attack timings has a 1024 bit prime, a base g with 512-bit order n, and $p-1$ has a 72 bit 619-smooth factor. See Table 5.1 for a complete list of the parameters.

The attacks were run with parameters $b_1 = b_2 = b/2$, for $b = 32, 34, \ldots, 46$. hashmim was also run on a 56 bit message. Messages were generated by picking two random $b/2$ bit number and multiplying, ensuring that the attacks will succeed when cracking the resulting ciphertexts. All attacks searched the entire table to find all possible solutions. In practice this is not necessary, especially since the ElGamal cryptosystem used is such that solution collisions are extremely unlikely. Full table searches were done to ensure that all attacks were operating correctly. This means that on average the first solution will be found in half the time listed in the tables if the message splits. If the solution is unique with high probability, there is no need to continue the search after a solution is found. Non-splitting messages will require an entire table search to discover the failure, so the numbers given will be accurate. While different splits of a single solution occurred, none of ciphertexts had multiple unique solutions, as expected.

5.2 The Test Machine

An AMD Athlon(tm) 64 X2 4400+ with 6GB of RAM running Linux with kernel 2.6.27 was used to run the attacks. This is a dual-core processor, but this implementation makes use of only one core. The code was compiled with GNU g++ v4.3.2.

Table 5.1 1024/512/72-bit smooth ElGamal cryptosystem

p	118381843724717101749461596756646482230905897660463123624560394563807609933956042265392341520956028886446317716642070570538792311686346409424101404111812833160856599353200278320709069863021480695349692087358601640250836457118800932512352680882211491654732513532851546702786190877679512653375709345552713302401
g	59660846376012012299732062167070449913568079369759726390943366472735655747125035179031089451125540857538031073871730574393537580324435989370818322776711385170287961641652843108956199416275969391836776116950834964228187667553031050881716218984733944262206823833143460937854518070649325296332195764146328701846
n	106432791900654366581899866180644064216449650489311237590593999612671885602808381031486165618460173726482764815882812493123891819815192202006792855201655533
y	21724981299327350130492804094624665385311574459503588177022457718102088758917010480569332112803059495077627059746788501514325709884124145938761499931046123297199537456346696963268983551051806775774357646655587458036890153399589255949124528357931618500921631371696087601647428808912263034214228210166729949452
x	104388513511721031015822269269494775753013109228875919590447192380664806119256701591644841306185665539571720651315109147710711154230953299622246764431295529
s	457130418902779295520 0

5.3 Summary of Results

The ciphertext crack times are averages over at least two ciphertexts. Since an entire table search is performed, there is no reason to expect some ciphertexts to take longer than others. The results confirmed this, so averaging over multiple ciphertexts is not really necessary. See figures 5.1 and 5.2 and table 5.2 for the results. A timing resolution of one second was used.

- The 2^{b_1} modular exponentiations dominate the pre-computation phase for mim and hashmim — so much so that sorting is insignificant. With $b = 56$ and $b_1 = b_2 = 28$, the pre-computation phase of hashmim took over 55 hours. The sort took less than 2 minutes.

- Modular exponentiations also dominate the message crack phase for mim and hashmim. However the extra multiplications and inversions required are significant, as can be seen by the longer run time of message cracking vs pre-computation. The gap can be partly closed by combining the exponentiation and inversion into a single operation; however this was discovered too late to include in the final results, and caching (see section 5.5) supersedes this optimization.

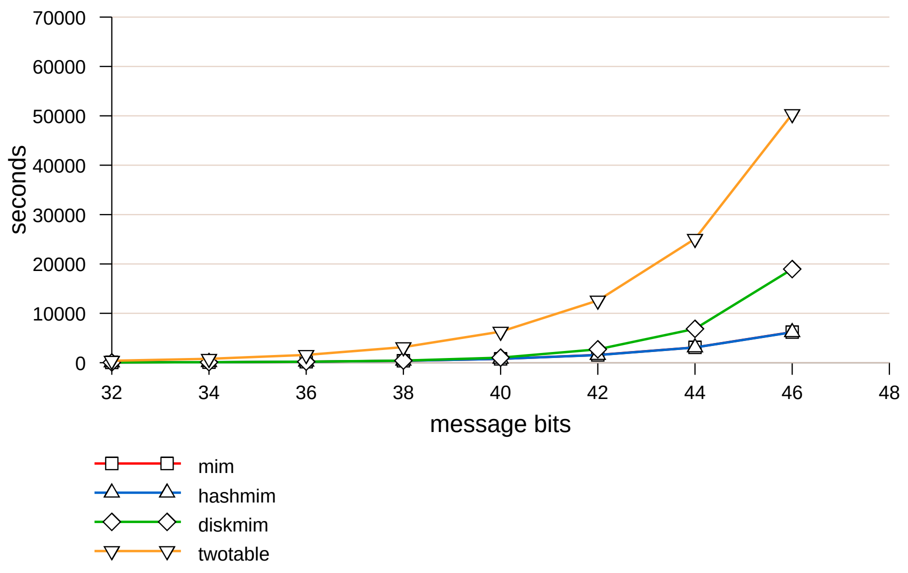
- mim and hashmim are not significantly different in table build time (pre-computation) or message crack time. Moreover for each ciphertext cracked, the actual number of extra exponentiations computed by hashmim was very close to the expected number of matches as discussed in section 3.4.2. hashmim was actually faster than mim in many cases; this was likely caused by the big integer type initializations done by mim.

- The table build times for diskmim start to diverge from the other basic mim attacks for larger messages. Furthermore the times do not scale with the size of the table - the times more than double when the table size doubles, and the difference is significant. Unlike the other attacks, the table insert operations (B+ tree inserts to disk) become a significant part of pre-computation. diskmim also eventually stops using 100% CPU, indicating that it becomes I/O bound.

- The message crack times for diskmim are actually fairly close to the other basic mim attacks, even for larger messages when the table build times start to diverge. This suggests that the node-splitting required when building the B+ tree is causing the long table build times, not the search required to determine the insert location.

- twotable is orders of magnitude faster at cracking messages. For messages less than 44 bits, it took less than a second. Even at 46 bits it took under 4 seconds when the next fastest attack took over 1 hour and 45 minutes. This fast crack time comes at the cost of much longer table build times. For this reason twotable is only faster when more than 7 messages will be attacked.

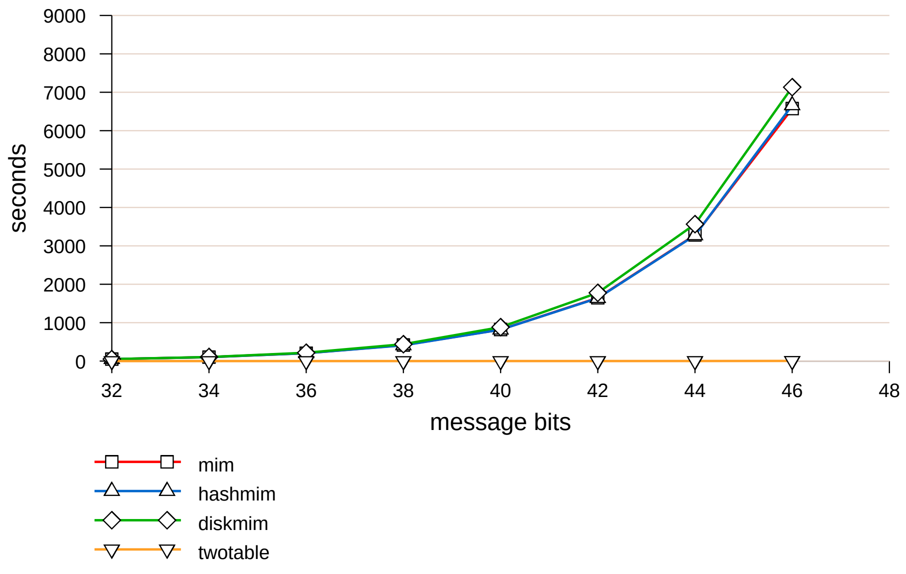
Figure 5.1 Table build times



5.4 The Effect of ElGamal Key Size on Attack Time

Modular exponentiations require $O(\log e)$ modular multiplications, where e is the exponent. In our case, n is the exponent and p determines the group $\mathbb{Z}_p^*$ and the size of the elements being multiplied. Therefore both reducing p and reducing n will reduce the time required for modular exponentiation. In particular we expect halving the bit size of n to approximately

Figure 5.2 Ciphertext crack times



halve the exponentiation time. Since modular exponentiation dominates both pre-computation and message cracking, we expect the attacks to scale similarly.

Table 5.3 lists attack times against a 40 bit message with $b_1 = b_2 = 20$ for several different cryptosystems. All the cryptosystems have a 72-bit B-smooth factor with B near 600, but the smooth factors are not identical. This explains why the twotable times are irregular.

5.5 Caching Exponentiations

If $b_1 \geq b_2$, then the modular exponentiations used during the message crack phase are also calculated during the pre-computation phase. If the values are saved to a file during pre-computation, they can be read sequentially by the message crack phase and need not be re-calculated. Applying this to hashmim with $b = 46$, the message crack time was reduced from 6660 seconds to 420 seconds and only added about 150 seconds to pre-computation. This optimization can also be applied to plain mim and diskmim; however twotable has already eliminated modular exponentiation from the message crack phase.

Caching can still be used if $b_1 < b_2$. The cache can be started during pre-computation,

Table 5.2 Results for a 1024/512/72-bit smooth cryptosystem

			mim		**hashmim**		**diskmim**			**twotable**	
b	$\mathbf{T_0}$	$\mathbf{C_0}$	**T**	**C**	**T**	**C**	**T**	**C**	$\tilde{\mathbf{m}}^{\mathbf{nr}}$	**T**	**C**
32	49	51	49	51	48	51	48	51	88	393	0
34	96	103	97	102	97	102	98	105	175	786	0
36	193	206	193	205	192	204	200	217	349	1573	0
38	385	407	387	416	386	409	425	439	698	3152	0
40	771	813	773	820	772	818	1021	883	1395	6286	0
42	1543	1628	1572	1647	1545	1639	2709	1777	2793	12588	1
44	3080	3260	3097	3284	3085	3274	6860	3565	5581	25091	2
46	6166	6518	6197	6575	6170	6660	18959	7132	11171	50359	4
56	197312*				199500	223468					

b	the assumed bit size of the message; note that $b_1 = b_2 = b/2$
$\mathbf{T_0}$	the time in seconds required to compute $\tilde{m}^n$ for $\tilde{m} = 1 \ldots 2^{b/2}$
$\mathbf{C_0}$	the time in seconds required to compute $(\tilde{m}^n)^{-1}v^n$ for $\tilde{m} = 1 \ldots 2^{b/2}$
T	table build time (pre-computation) in seconds
C	message crack time in seconds
$\tilde{\mathbf{m}}^{\mathbf{nr}}$	the time in seconds required to compute $\tilde{m}^{nr}$ for $\tilde{m} = 1 \ldots 2^{b/2}$
*	T_0 for $b = 56$ was estimated from the value of T_0 for $b = 46$.

which will be significant if $b_2 - b_1$ is small, and completed by the first message crack run.

The message crack phase actually requires the inverse of the modular exponentiations. While inversions are much cheaper than exponentiations, they still make a significant contribution. Caching the inversions during the first message crack would therefore be worthwhile. Future message attacks will then require only a disk read, a multiplication, and a modular reduction for each possible value of $\tilde{m}_2$. The multiplication cannot be cached, since it depends on the particular ciphertext being attacked.

5.6 Larger Messages

If we wish to crack 64 bit messages with $b_1 = b_2 = 32$, we need to calculate 2^{32} modular exponentiations both to build the table(s) and to crack a message. This calculation alone will take about 40 days on the author's machine. Since there is not enough system memory, diskmim is the only attack which can attack messages this large. Since the build times for

Table 5.3 Effect of ElGamal key size on attack time for 40 bit messages

bits		**mim**		**hashmim**		**diskmim**		**twotable**	
p	**n**	**T**	**C**	**T**	**C**	**T**	**C**	**T**	**C**
1024	512	773	820	772	818	1021	883	6286	< 1
1024	256	393	440	390	437	589	508	6703	< 1
768	512	501	536	498	533	709	604	3844	< 1
768	256	256	290	254	288	437	358	4251	< 1
512	256	123	145	121	143	253	212	1843	< 1

p	the bit size of the prime p
n	the bit size of $n = \|g\|$
T	table build time in seconds
C	ciphertext crack time in seconds

diskmim increase by well over a factor of two when doubling the table size, the attack will take well over 120 days. This can likely be improved dramatically however by tuning the B+ tree or using a different data structure. Furthermore lots of data needs to remain secure for years, so if an attack succeeds in months it is still a major problem.

A four table attack could also be run in parallel with the main attack. If the message turns out to split four ways, we will find a solution in far less time.

Chapter6. SUMMARY AND DISCUSSION

6.1 Comparing the Attacks

If $b = 64$ and $b_1 = b_2$, the only viable attack is diskmim, since the table will not fit in memory no matter how little space is used by each entry. However by taking $b_1 < 32$ and $b_2 > 32$, the other attacks will work at the expensive of longer message crack times and possibly success probability. However because of the slow table build times of diskmim, this approach may be faster when only a few messages need to be cracked.

There is no good reason to choose plain mim over hashmim — the size of the hash function can be increased when b and b_1 are large to ensure that the extra computations required by hashmim are insignificant. If hundreds of messages will be attacked and the conditions for the twotable attack are met ($p-1$ has a smooth factor s with $s > 2^b$), then twotable will be faster than hashmim, even with exponentiation caching. However twotable uses more memory than hashmim, so it will require picking $b_1 < b/2$ before hashmim. For this reason hashmim may be faster when b is large.

6.2 Potential Improvements

Since modular exponentiations, multiplications, and inversions dominate both phases of the attack when the table fits in memory, caching and improving the speed of these operations is the best way to improve overall performance. Caching has already been discussed; improving the operations themselves may be possible using special purpose routines instead of the GMP library.

Small gains could be achieved by using a more advanced data structure such as a hash table in mim and hashmim. However it is difficult to beat the performance of hashmim without using more memory. Since the potential gains are so small, this is a poor trade-off.

diskmim could likely be improved significantly by tuning the B+ tree parameters. However I suspects that a disk-based hash table will prove a better data structure for this attack, allowing most inserts and searches to be completed with a single disk access.

6.3 Protecting Against the Attacks

The most direct way to defeat the attacks discussed in this paper is to represent messages as elements of $\langle g \rangle$, either by choosing g primitive so $\langle g \rangle = \mathbb{Z}_p^*$ or by designing an easily computable bijective mapping between messages and the proper subgroup $\langle g \rangle$. However [BJN00] describes a meet-in-the-middle attack which works when $n = p - 1$ and n has a smooth factor at least as large as the message. This demonstrates that meet-in-the-middle methods can work even when all messages are in $\langle g \rangle$.

With the proper choice of parameters, ElGamal is conjectured to be semantically secure — a popular formal definition of security (see section 5.9 in [Sti05] and definition 8.47 in [MVO96]). n is chosen to be a large prime such that $p = 2n + 1$ is also prime, and the base g is selected to have order n as usual. These parameters are recommended in the OpenPGP Message Format (RFC 2440) section 12.5 [CDFT98]. The cyclic subgroup $\langle g \rangle$ will then be the group of quadratic residues mod p, and representing messages as members of this group is relatively easy. If these parameters are used, and messages are represented as quadratic residues, the resulting cryptosystem is conjectured to be semantically secure assuming that the Discrete Log problem in $\mathbb{Z}_p^*$ (and in $\langle g \rangle$) is intractable [Sti05]. Perhaps more importantly, the cryptosystem will not be vulnerable to the meet-in-the-middle attacks discussed in this paper and in [BJN00].

Another way of defeating these attacks is pre-processing the message. For example we can simply pad short messages to say 128 bits, making the attacks infeasible. The modular exponentiation dominates encryption, so having a larger message to multiply will not significantly impact performance. However the reader should be wary of such a simplistic approach. See [ABR99] for a more complete overhaul of ElGamal, called DHAES. DHAES is designed to conform to high standards of formal security while matching the performance of ElGamal.

6.4 Conclusion

Implementing a cryptosystem securely requires far more than an understanding of the basic algorithm. The implementer must be aware of possible attacks on the system, and choose keys and parameters to make those attacks infeasible. This paper discussed attacks which rely on the underlying mathematics - however timing attacks have been discovered against various cryptosystem which gain information based on how long the computer takes to perform encryption or decryption operations. Secure implementation is difficult, and using an existing implementation which has already undergone extensive public review should always be preferred over creating a new implementation.

Bibliography

[ABR99] Michel Abdalla, Mihir Bellare, and Phillip Rogaway, *DHAES: An encryption scheme based on the Diffie-Hellman problem*, Tech. Report 99-07, 1999.

[BJN00] Dan Boneh, Antoine Joux, and Phong Q. Nguyen, *Why textbook ElGamal and RSA encryption are insecure*, ASIACRYPT '00: Proceedings of the 6th International Conference on the Theory and Application of Cryptology and Information Security (London, UK), Springer-Verlag, 2000, pp. 30–43.

[CDFT98] J. Callas, L. Donnerhacke, H. Finney, and R. Thayer, *OpenPGP message format (RFC 2440)*, http://tools.ietf.org/html/rfc2440, November 1998.

[gmp] *GNU multiple precision arithmetic library*, http://www.gmplib.org.

[Hir] Mikio Hirabayashi, *Tokyo cabinet*, http://tokyocabinet.sourceforge.net/index.html.

[MVO96] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot, *Handbook of applied cryptography*, CRC Press, Inc., Boca Raton, FL, USA, 1996.

[Sho05] Victor Shoup, *A computational introduction to number theory and algebra*, Cambridge University Press, New York, NY, USA, 2005, http://www.shoup.net/ntb/.

[Sti05] Douglas R. Stinson, *Cryptography: Theory and practice, third edition*, Chapman & Hall/CRC, 2005.