Attacks on ElGamal

Tihana Britvić, 70078007
University of Ljubljana, Faculty of Mathematics and Physics

Abstract

In this paper, the ElGamal cryptosystem will be defined and analyzed in term of attacks. Three types of attacks will be described: brute force, meet-in-the-middle and two table attack. Furthermore, almost all of them hold discrete log as a base, so it will also be described. This project continues on knowledge that we gain in Cryptography and Computer Security course.

1 Introduction

Public key cryptosystems have an elegant mathematical simplicity, but simple implementations are often insecure. Because of this, I will describe several attacks on ElGamal which work well when the encrypted message is short is not been preprocessed. The attacks on ElGamal depend on the parameters, which are commonly used in practical implementations, and are also used while creating the cryptosystem. The formalization of an idea of what makes a secure cryptosystem is a popular subject among computer scientists. Formal definitions are sometimes a bit stronger than necessary for practical security regarding that many cryptosystems used in practice do not satisfy the formal definitions. ElGamal cryptosystem implementation is something that is often referred to in this context.

2 ElGamal Cryptosystems

ElGamal is a public key system which uses modular exponentiation as the basis for a one-way trapdoor function. The reverse operation is the so-called discrete logarithm and is considered to be 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 and for some applications, even larger keys are recommended. We will now persume, for simplicity, that an ElGamal cryptosystem operates in a finite cyclic group with multiplicativity. The two most common choices are:

1. the group of integers from 1 to $p-1$ under multiplication $mod\, p$, where p is a prime number. Notation: Z_p^*.
2. subgroups of Z_p^* that have the prime order. With $ord(g)$ the order of an element g in Z_p^* is denoted, and with $< g >$ the cyclic subgroup of Z_p^* generated by g is denoted.

2.1 The Discrete Log Problem

The discrete logarithm is defined as the inverse of modular exponentiation: given a modular exponentiation $y = g^x$ in Z_p^* with the base g, the discrete logarithm $log_g y$ is x. This is a discrete logarithm in the cyclic group $< g >$ and it may or may not be the whole Z_p^*. When $ord(g) = n$ is large and has at least one large prime factor, discrete log problem in $< g >$ is considered intractable. There are three basic types of discrete log algorithms:

1. square-root algorithms (i.e. Pollard's rho algorithm)
2. the Pohlig-Hellmen algorithm
3. index calculus algorithms

Pollard's rho algorithm can compute discrete logs in a cyclic group of prime order n in $O(\sqrt{n})$ and negligible space. If n is not prime and the factorization of n is known, then the Pohlig-Hellman algorithm can be used. If the factorization of n is given with $n = p_1^{e_1} p_2^{e_2} ... p_k^{e_k}$, then the Pohlig-Hellman algorithm computes partial solutions by computing discrete logs in subgroups of order p_i for $i = 1, ..., k$. Typically Pollard's rho algorithm is used as a subroutine to compute these loga-

rithms, and the partial solutions are combined to compute the requested discrete log. The runtime of Pohlig-Hellman is $O(\sum_{i=1}^{k} e_i(logn + \sqrt{p_i}))$. If n is a B-smooth number, meaning that none of it's prime factors are greater than B, the runtime of the Pohlig-Hellman algorithm is $O(lnlnn(logn + \sqrt{B})$. 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 Z_p^* and they run in sub-exponential time. Also, they do not work directly on subgroups of Z_p^*, but they can be used to compute logs in subgroups by computing logs in Z_p^*. For this reason, if $n << p$ 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.

2.1.1 Encryption and Decryption

We can represent ElGamal cryptosystem with a 4-tuple (p,g,x,y) where p is a large prime number that group Z_p^* is being represented with, g is an element of Z_p^* such that $ord(g) = n$, random $x \in [1, n-1]$ and $y = g^x$. The tuple (p,g,y) is called a public key, while x is called private key. In addition to the public key, a random integer $k \in [1, n-1]$ is used in encryption function:

$$E_k(m) = (g^k, my^k).$$

Decryption function is defined with:

$$D(u,v) = u^{-x}v.$$

All the operations are done in $mod p$. The decryption function will recover the original message:

$$u^{-x} = g^{-kx} = (g^x)^{-k} = y^{-k}$$

and

$$D(E_k(m)) = u^{-x}v = y^{-k}my^k = m.$$

2.1.2 Security

It is important to choose n, p very large because if we recover private key x, we can decrypt all messages. Since the public key includes y ($y = g^x$) and g, finding a private key equal to computing a single discrete logarithm in $< g >$. For example, if we are using Pollard's rho algorithm then n determines the runtime of square-root discrete logs, while if using index calculus p determines the runtime of the discrete log. Recommended minimum for p is 1024. If $n < p-1$, then n should be large enough such that discrete log algorithms take at least $O(\sqrt{n})$. To break a single cipher $(u,v) = (g^k, my^k)$ we need to find y^{-k}. Since $m = vy^k$ and we can compute inverses efficiently, we only need y^k. Cracking a ciphertext is equivalent to the Diffie-Hellman problem that states: "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 more difficult than the discrete log problem.

3 Attacks

3.1 Brute Force Attacks

Brute force attacks is a class of attacks on cryptosystems that can be characterized with exhaustive searching and very little ingenuity. For example: recovering a plaintext from a ciphertext by decrypting the ciphertext with every possible key. This attack 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 irregular distributions are more likely to be the correct plaintext.

If we look at brute force attack on ElGamal cryptosystem, there are $n-1$ possible values for private key x. If we are given the ciphertext (that was encrypted using a 64-bit key from plaintext), we could decrypt it using each value of the private key. Even if $n << p-1$ there will be few decryptions. So, if we take $n = 256bits$, it requires $O(2^{256})$ modular exponentins, 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 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.

3.2 Meet-in-the-middle

3.2.1 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 ciphertext is used.

The attack works well under following conditions:

1. The original message m is at most b bits, where b is small, and the adversary is aware of this limit.

2. m can be factored (split) into two factors of at most b_1 and b_2 bits respectively.

3. $n = ord(g) \in Z_p^*$ 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 and n can be computed efficiently.

4. Messages are not represented as elements of $< g >$. For ElGamal to work, the messages must be represented with members of Z_p^*.

5. $n < (p-1)2^{-b}$. This ensures that given an element v^n such that $ord(v^n)|\frac{p-1}{n}$ in Z_p^*, the expected number of distinct messages m such that $m^n = v^n$ is very small.

Since we assumed that message splits in some way, the attack will not always work. For example,if we have a 56 bit message and we choose parameters $b_1 = b_2 = 28$, the probability of success equals to 18%.

3.2.2 The Attack

ElGamal cryptosystems are non-deterministic, which means that encryption of the same plaintext multiple times results in different ciphertexts (due to randomness of k). However, the term y^k is non-deterministic and it states that $ord(y^k)|n$, so if we raise $v = my^k$ to the n power we can eliminate y^k:

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

`Note:` There can be a 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 $\tilde{m} = \tilde{m_1}\tilde{m_2}$ where $\tilde{m_1} < 2_1^b$ and $\tilde{m_2} < 2_2^b$. In that case:
$v^n = \tilde{m}^n = \tilde{m_1}^n\tilde{m_2}^n$ and $v^n\tilde{m}^{-n} = \tilde{m_1}^n$.

The idea of the attack is to compute $\tilde{m_1}^n$ for $\tilde{m_1}^n = 1...2^{b_1}$, 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 $\tilde{m_2} = 1...2^{b_2}$ 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 $< g >$, then $v^n = 1$ for every for every message. and this attack fails completely.

3.2.3 Solution Collisions

If $\tilde{m} \in Z_p^*$ then $ord(\tilde{m}^n)|\frac{p-1}{n}$. 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 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 $\tilde{m}^n = 1...2^b$ are roughly uniformly distributed in the subgroup of order $\frac{p-1}{n}$. In that case, $\tilde{m}^n = v^n$ with probability $\frac{1}{\frac{p-1}{n}} = \frac{n}{p-1}$. X_c then has a binomial distribution with probability $\frac{n}{p-1}$ and $2^b - 1$ trials. If $n < (p-1)2^{-b}$ then

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

The attack will only find splitting messages, so the actual expected number of collisions is $E[X_c]$ times the splitting probability. For example, if $p-1$ is 1024 bits , n is 512 bits, and $b = 64$ (message has 64 bits), then $E[X_c] \sim 2^{-448}$.

3.2.4 Implementation

The attack is fairly straight forward so we need a suitable data structure for the dictionary. It needs to support efficient insert and search routines, so the most obvious choices are to use a hash table or 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 $O(logn)$ using binary search. The sort will require $O(nlogn)$ 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 $O(logn)$ with a clever implementation of quicksort, for example.

3.2.5 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 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. If we store $(hash(\tilde{m_1}^n), \tilde{m_1})$ instead, where hash() is the suitable hash function. To find $u^n\tilde{m_2}^{-n}$, we need to compute $hash(u^n\tilde{m_2}^{-n})$ and then do a binary search. Since hash() will likely have collisions the search may find multiple possible values for $\tilde{m_1}$. For each match, we recompute $\tilde{m_1}$ and test for equality with $u^n\tilde{m_2}^{-n}$. If the number of matches is much less than $2^{b_2 n}$, the extra exponentiation calculations required will not make not make a significant contribution to the run time. The expected number of matches depends on the size of the hash val-

ues. Suppose the hash values are h bits and assume that the values of $hash(\tilde{m_1}^n)$ are uniformly distributed over interval $[0, 2^h - 1]$. The probability of $hash(\tilde{m_2}^n)$ matches a specific element in the table for given value of $\tilde{m_2}$ is 2^{b_1-h}, and the total number of expected matches is $2^{b_2}2^{b_1-h} = 2^{b-h}$, where $b-h << b_2$. This implementation requires $36GB$ to store the dictionary when $b_1 = 32$.

3.2.6 Running Time and Memory Usage

No matter is we are using hash dictionary or not, the attack requires $O(b_1 2^{b_1})$ sorts. The space requirement is 2^{b_1} table entries. They both require $O(2^{b_2})$ binary searches of the table, each running in $O(log 2^{b_1}) = O(b_1)$ time, which gives us total complexity of $O(b_1 2^{b_2})$.

3.2.7 Comparison to Brute Force

The most effective brute force attack on a hybrid system is usually a direct attack on the symmetric cipher. If a b-bit key is used, then the expected runtime is $O(2^b)$. If $b_1 = b_2 = \frac{b}{2}$, then the runtime of meet-in-the-middle attack will be $O(2^{\frac{b}{2}+1})$. For example, if $b = 56, b_1 = b_2 = 28$, brute force runtime will take 2^{55}.

3.3 Two table attack

3.3.1 Description

The two table attack is a refinement of the meet-in-the-middle attack which works when Z_p^* has a subgroup in which discrete logs can be computed efficiently. 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 a different time, space, and success probability trade-offs.

3.4 The attack

Let $p - 1 = nrs$, where s is easily factorable using trial division, and we can therefore efficiently find an element $\alpha \in Z_p^*$ that generates the subgroup of order s. Instead of raising v to the n power, we raise v to the nr power. If $a \in Z_p^*, (a^{nr})^s = a^{p-1} = 1$, so $a^{nr} \in < \alpha >$ such that $ord(< \alpha >) = s$. With this we can compute the discrete logarithm with base α. Lets suppose that $\tilde{m_1}$ and $\tilde{m_2}$ are factors of $\tilde{m}$ with bit size b_1 and b_2, respectively. If v is a ciphertext for $\tilde{m}$, then:

$$v = y^k \tilde{m} = y^k \tilde{m_1} \tilde{m_2}$$
$$v^{nr} = (y^k)^{nr} \tilde{m_1}^{nr} \tilde{m_2}^{nr}$$
$$v^{nr} = \tilde{m_1}^{nr} \tilde{m_2}^{nr}$$
$$log v^{nr} = log \tilde{m_1}^{nr} + log \tilde{m_2}^{nr}$$

where log has base α. 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 $\tilde{m_1} = 1...2^{b_1}$, and T_2 contains pairs $(log \tilde{m_2}^{nr}, \tilde{m_2})$ for $\tilde{m_2} = 1...2^{b_2}$. When we try to crack the message, we want to find two pairs (t_1, v_1) and (t_2, v_2) from the tables such that $log v^{nr} = t_1 + t_2 mods$. If we find such pair, we have found candidate (v_1, v_2) for plaintext. Under the 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 integer different tables is called the k-table problem. Lets look at the case when $k = 2$. The basic idea for solving the two table problem is to sort both tables by the first coordinate - T1 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 $mods$, 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 $t_1 = t - t_2 mods$. For a fixed t, we can define a virtual table $T_2^{'}$ from T_2 containing the values $(t - t_2 mods, v_2)$ 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 $t_2 = t + 1 \in T_2$, then $t - (t + 1) mods = s - 1$ 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 $t_2 \in T_2$ greater than $t + 1$, and the corresponding element $t - \hat{\imath}_2 \in T_2^{'}$ 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 head T_1 is less than 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 $T_2^{'}$.

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

3.4.2 Implementation

This attack requires a discrete log algorithm which works well in a group of smooth order. For example, we can use Pohlig-Hellman algorithm together with Pollard's rho algorithm. Trial exponentiation is used instead of Pollard rho for very small prime powers. Let us now observe the case when $b_1 = b_2$, then the tables $T_1 = T_2$. 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 half the space requirement and half the sorting time.

3.4.3 Running Time and Memory Usage

Let us define $b_m := max(b_1, b_2)$. 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 examine at most $2^{min(b_1,b_2)}$ candidate pairs $t_1, t_2^{'}$. For each candidate pair tested, we compute $t_2^{'} = n - t_2 mods$ 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 $(loga^{nr}, a)$, where a is at most $b_1 = b_2$ bits and the log is $O(logs)$ bits. Since s may require more bits than the word size, we may require a large integer type to store $loga^{nr}$. Note that storing just the hash of the log is not sufficient, we need the full value. When b_1 is not equal to b_2, the size requirement jumps to $2^{b_1} + 2^{b_2}$ table entries.

3.4.4 Three and Four Table Attacks

This attack can be extended to three and four table attacks. 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.

4 Results

There is no good reason to choose plain meet-in-the-middle attack implementation over the one that uses hash function — the size of the hash function can be increased when b and b_1 are large to ensure that the extra computations required by hashed meet-in-the-middle are insignificant. If hundreds of messages will be attacked and the conditions for the able attack are met (p-1 has a smooth factor s with $s > 2^b$), then two table will be faster than the implementation of meet-in-the-middle-attack using a hash function, even with exponentiation caching. However two table uses more memory than the version of meet-in-the-middle attack with a hash function, so it will require picking $b_1 < \frac{b}{2}$. For this reason, the meet-in-the-middle attack that uses hash function may be faster when b is large.

4.1 Protecting Against the Attacks

The most direct way to defeat the attacks discussed in this paper is to represent messages as elements $< g >$, either by choosing g primitive $< g >= Z_p^*$ or by designing an easily computable bijective mapping between messages and the proper subgroup $< g >$. However 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 $< g >$. With the proper choice of parameters, ElGamal is conjectured to be semantically secure - a popular formal definition of security. 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. The cyclic subgroup $< g >$ 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 Z_p^* (and in $< g >$) is intractable. Perhaps more importantly, the cryptosystem will not be vulnerable to the meet-in-the-middle attacks discussed in this paper and in. 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.

5 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 gains 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.

6 References

1. Stinson: Cryptography Theory And Practice, 3rd edition, University of Waterloo, Ontario Canada, 2006.

2. Allen, Implementing several attacks on plain ElGamal encryption, Iowa State University , 2008.

3. Abdalla, Bellare, Rogaway: An encryption scheme based on the Diffie-Hellman problem, Tech. Report 99-07, 1999.

4. Menezes, Vanstone, Van Oorschot, Handbook of applied cryptography, CRC Press, Inc., Boca Raton, FL, USA, 1996.