MATHEMATISCHES INSTITUT DER UNIVERSITÄT MÜNCHEN

Individual Bit Security of the Discrete Logarithm: Theory and Implementation Using Oracles

Master Thesis
by Richard Rex McKnight

Supervised by Prof. Dr. Otto Forster
March 2007

Acknowledgments

I would like to thank the Ludwig-Maximilians-Universität München for graciously accepting me into their International Master Program without which this thesis would have been impossible.

Furthermore, I wish to express my sincerest appreciation to my supervisor Prof. Dr. Otto Forster whose incredible patience and helpful insights are directly responsible for the existence of this thesis.

Never to be forgotten are all of my friends, in Germany and abroad, to whom I would like to convey my gratitude for providing me with endless distractions outside of my studies.

Lastly, I am forever indebted to my parents who brought me into this world, raised me with unconditional loving, supported me throughout all of my undertakings in life, and instilled in me the passion to learn. To them, my wonderful sister, and Toni, I dedicate this thesis.

Contents

List of Procedures

Declaration of Authorship

I hereby declare that I wrote this Master Thesis solely by myself and used no resources other than those cited.

München, 30. March 2007 Richard R. McKnight

Abstract

The intent of this paper is to provide a comprehensive overview of the discrete logarithm problem including its properties, its cryptographic significance, the known methods for solving it, and of utmost importance, its individual bit security. The main result of the paper is an elegant proof showing how a perfect oracle on almost any bit of the binary representation of the discrete logarithm can be combined with clever reduction techniques to determine the other bits. This proves that almost all individual bits of the discrete logarithm are hard.

Specifically, the proof yields a procedure to determine the discrete logarithm $r := \log_g(h)$ of any element h in $\mathbb{Z}_p^*$ for any odd prime p, any generator g of $\mathbb{Z}_p^*$, and any perfect oracle on the i^{th} bit where $s \leq i < n-1$ with s the bit border of p and n the binary length of $p-1$. Furthermore, the procedure is extremely efficient in the sense that, on average, it only needs to query the oracle 1.5 times for each bit it recovers.

Chapter 1

Introduction

According to the 2004 E-commerce Multi-sector Report, online purchasing accounted for \$996 billion in manufacturing shipments in 2004 ([Bur06, p. 1]). Furthermore, US retailers' e-commerce sales have increased approximately 25% in 2004 marking the third straight year as more businesses open their virtual shops on the internet with the hopes of taking a bite out of these new profits ([Bur06, p. 2]). However, none of this would be possible without cryptographically secure means of handling the online purchase transactions. Thus, it is of the utmost importance that these online transactions are encapsulated in a provably difficult security layer.

The study and development of this layer of security is known as Cryptography. To be more precise, Cryptography may be thought of as the study of sending data to another person through space and time in such a manner that no other persons can easily comprehend the transmission. In the event that another party is able to intercept the transmission, any modifications to it should be detectable to the receiver.

This paper provides a basic overview of the current state of Cryptography as it relates to the discrete logarithm problem which is at the heart of many important data encryption schemes and key exchange protocols. These key exchanges are prelude to using any one of the many other so called block stream ciphers which then maintain the actual encrypted two-way communication channels. In turn, these encrypted channels facility the transfer of sensitive information such as ones credit card numbers for online purchases or ones private communications.

After developing the basic notations, elementary maths, and concepts of

Cryptography necessary for the paper in Chapter 2, the paper provides an introduction defining the discrete logarithm and its associated properties in Chapter 3. In Chapter 4, the paper underlines the importance of the discrete logarithm problem by covering two practical applications of the discrete logarithm. Chapter 5 provides a detailed overview of the most common means for solving the discrete logarithm problem. The main topic of the paper is then reached in Chapter 6 which provides an elegant proof of the individual bit security of almost all of the discrete logarithm bits. Additionally, Appendix A provides a translation of the proof into a working computer program written in Aribas. Finally, the paper wraps up with some observations about the main results and suggestions for future directions in Chapter 7.

Chapter 2

A Brief Review of Necessary Topics

The next three sections provide a brief overview of the nomenclature, mathematics, and concepts necessary to understand Cryptography as used in this paper.

2.1 Nomenclature

This paper does not subscribe to the view as some papers do that notational shorthand such as *iff*, $\exists$, or $\forall$, improves readability or comprehension. Thus this paper chooses to use English where ever possible to express such ideas. That being said, the paper does make use of the following conventional nomenclature:

1. $a = b$ denotes the standard mathematical concept of equality between a and b.

2. $a := b$ denotes that a is defined as b.

3. $\{s_1, \ldots, s_k\}$ denotes a finite set, *i.e.* a finite collection of distinct elements.

4. $\{s_1, \ldots, s_k, \ldots\}$ denotes an infinite set, *i.e.* an infinite collection of distinct elements.

5. $A \subset B$ denotes that A is included in B. The type of inclusion depends on the properties of A and B.

6. $\mathbb{Z} := \{0, \pm 1, \pm 2, \dots\}$ denotes the ring of integers.

7. $\mathbb{Z}_n := \{0, \dots, n-1\}$ denotes the ring of integers modulo an integer n.

8. $\mathbb{Z}_p := \{0, \dots, p-1\}$ denotes the field of integers modulo a prime p.

9. $\mathbb{Z}_p^* := \mathbb{Z}_p \setminus \{0\} = \{1, \dots, p-1\}$ denotes the multiplicative group of integers modulo a prime p.

10. $\mathbb{F}_{p^f}$ denotes the Galois field of order p^f where p is prime and f is greater than 0. In the case where f is 1, then $\mathbb{F}_p \cong \mathbb{Z}_p$.

11. $\langle g \rangle := \{g^r : r \text{ in } \mathbb{Z}\} \subset G$ denotes the cyclic group generated by the element g of the group G.

12. $\gcd(a, b)$ denotes the greatest common divisor of a and b. If $\gcd(a, b) = 1$, then a and b are called *relatively prime*.

13. $\mathcal{O}(f(n))$ denotes the asymptotic behavior of the function $f(n)$ as n tends toward infinity, also known as Big-O notation [MvOV96, sec. 2.3]. In particular:

    (a) $\mathcal{O}(1)$ denotes the family of constant functions.

    (b) $\mathcal{O}(\log n)$ denotes the family of logarithmic functions.

    (c) $\mathcal{O}(n)$ denotes the family of linear functions.

    (d) $\mathcal{O}(n^c)$ denotes the family of polynomial functions.

    (e) subexponential denotes the family of functions that are asymptotically slower than a polynomial function, but asymptotically faster than an exponential function.

    (f) $\mathcal{O}(c^n)$ denotes the family of exponential functions.

14. $a = (a_{n-1} \dots a_1 a_0)_b$ denotes the base b representation of the positive integer a. Note that it consists of $n := \lceil \log_b(a) \rceil$ digits a_i each in the range $\{0, \dots, b-1\}$.

When referring to the base b representation of an integer a, subscripts are used to denote its individual bits, *i.e.* a_j. This is not be confused with the same notation used to denote a sequence of elements. However, it should be clear from the context which notation is meant.

2.2 Group Theory and Finite Fields

Due to the discrete nature of the problem presented in this paper, almost all of the work will be confined to the mathematics of group theory. As such, it is assumed that the reader is well versed with the concepts of group theory and the foundations on which it is built. An understanding of field theory is recommended, although not explicitly necessary to follow the arguments in this paper.

However, only those definitions and properties of group theory and of finite fields which are frequently used are presented in this section. The reader is advised to refer to the comprehensive review of Number Theory, Abstract Algebra, and Finite Fields given in chapter 2, sections 4, 5, and 6, respectively, of the "Handbook of Applied Cryptography" [MvOV96].

Theorem 2.2.1 (Fermat's Little Theorem). *Let a be an element of $\mathbb{Z}_p^*$, then $a^{p-1} \equiv 1 \pmod{p}$.*

Fact 2.2.1. As $\mathbb{Z}_p$ is a finite field, each element a in $\mathbb{Z}_p^*$ is invertible and its inverse is $a^{-1} := a^{p-2} \pmod{p}$ since $a \cdot a^{p-2} \equiv a^{p-1} \equiv 1 \pmod{p}$ by Fermat's Little Theorem.

Fact 2.2.2. $\mathbb{Z}_p^*$ is a multiplicative cyclic group, *i.e.* there is an element g in $\mathbb{Z}_p^*$ such that $\langle g \rangle = \mathbb{Z}_p^*$.

Theorem 2.2.2 (Chinese Remainder Theorem). *Let $n = n_1 n_2 \cdots n_k$ where the n_i are pairwise relatively prime integers, then there exists a unique integer*

x module n solving the system of simultaneous congruences

$$\begin{aligned} x &\equiv a_1 \pmod{n_1} \\ x &\equiv a_2 \pmod{n_2} \\ &\vdots \\ x &\equiv a_k \pmod{n_k} \end{aligned}$$

Furthermore, this x may calculated explicitly as

$$x \equiv \sum_{i=1}^{k} a_i N_i M_i \pmod{n}$$

where $N_i = n/n_i$ and $M_i \equiv N_i^{-1} \pmod{n_i}$ [MvOV96, p. 68].

Fact 2.2.3. Let p be a prime and let f be an integer greater than 0, then there exists a finite field of order p^f and it is called the Galois field. In the case where f is 1, then $\mathbb{F}_p \cong \mathbb{Z}_p$.

Remark. Although for all primes p, $\mathbb{F}_p \cong \mathbb{Z}_p$, it is not the case that $\mathbb{F}_{p^f} \cong \mathbb{Z}_{p^f}$ when the integer f is greater than 1.

2.3 Binary Representation

This section covers the various definitions and properties associated with treating positive integers as binary numbers. Due to the multitude of definitions in this section, they are marked with an emphasis and simply defined via their context. Let r be a positive integer and $n := \lceil \log_2(r) \rceil$, then r can be represented in *binary* (base 2) by n *bits* (a state which is either 0 or 1) as

$$r = (r_{n-1} \ldots r_1 r_0)_2 = \sum_{k=0}^{n-1} r_k 2^k$$

with r_k from $\{0, 1\}$ being the bits. When r is written in base 2 form, it is called the *binary representation* of r. Note that the bits are enumerated from right to left starting at zero. The i^{th} most right bit is called the *i^{th} least significant bit* or the *least significant bit* when i is 0. The i^{th} most left bit is called the *i^{th} most significant bit* or the *most significant bit* when i is $n - 1$.

Property 2.3.1. The following properties are derived from the definition of the binary representation of $r = (r_{n-1} \dots r_1 r_0)_2$:

1. If r_0 is 0, then r is even otherwise if r_0 is 1, then r is odd. This is known as the *parity* of r.

2. If r_j is 0, then it can be *set*, *i.e.* toggled to 1, by adding 2^j to r.

3. If r_j is 1, then it can be *cleared*, *i.e.* toggled to 0, by subtracting 2^j to r.

4. The j least significant bits of r are all 0 if and only if r is divisible by 2^j.

5. Multiplying r by 2^j shifts all of the bits in the binary representation of r up by j bits, *i.e.* $r \cdot 2^j = (r_{n-1} \dots r_1 r_0 0 \dots 0)_2$ with j trailing zeros. This is known as *shifting up* by j bits.

6. Dividing r by 2^j shifts all of the bits in the binary representation of r down by j bits assuming r is divisible by 2^j. This is known as *shifting down* by j bits.

The reader should take some time to familiarize themselves with these properties as they are used extensively throughout the paper. Furthermore, of particular interest are the following three specific forms of r:

1. If $r = (0 \dots 010 \dots 0)_2$ where the 1 is in the i^{th} bit, then

$$r = 2^i$$

2. If $r = (1 \dots 1)_2$ where all n bits are 1, then

$$r = \sum_{k=0}^{n-1} 2^k = 2^n - 1$$

by taking the sum of the geometric series.

3. If $r = (0 \dots 01 \dots 10 \dots 0)_2$ where the 1s occupy the range from the j^{th} bit up to the $(i-1)^{\text{th}}$ bit with $j \leq i$, then

$$r = 2^i - 2^j$$

since

$$r = \sum_{k=j}^{i-1} 2^k = \sum_{k=0}^{i-1} 2^k - \sum_{k=0}^{j-1} 2^k = (2^i - 1) - (2^j - 1) = 2^i - 2^j$$

by using Case 2. Note that when $i = j$, this case degenerates to $r = 0 = (0 \dots 0)_2$.

It is important to realize that these cases can be combined with other integers via addition and subtraction. Thus, if $c < 2^{j+1}$ is a random, positive integer, then $r := 2^i - 2^{j+1} + c$ has the binary representation

$$r = (0 \dots 01 \dots 1c_j \dots c_1c_0)_2$$

since the $j + 1$ bits of c do not interfere with the range of 1s starting at the $(j+1)^{\text{th}}$ bit in r. This leads to the following fact regarding bits during binary addition.

Lemma 2.3.1 (Bit Propagation)**.** *Given a positive, n bit integer r of the form*

$$r := 2^i - 2^{j+1} + c = (0 \dots 01 \dots 1r_j \dots r_0)_2$$

with $c = (r_j \dots r_0)_2 < 2^{j+1}$ an unknown integer and $j + 1 \leq i$, let

$$\tilde{r} := r + 2^j = (\tilde{r}_{n-1} \dots \tilde{r}_1\tilde{r}_0)_2$$

Then, $\tilde{r}_i = r_j$, that is, adding 2^j to r propagates the unknown j^{th} bit up to the i^{th} bit. Furthermore, the lemma may again be applied to r when $r_j = 1$ and to $\tilde{r}$ when $r_j = 0$.

Proof. Consider the following two cases for r_j:

1. If $r_j = 0$, then $c < 2^j$ and still $r = 2^i - 2^{j+1} + c$. Thus,

$$\tilde{r} = r + 2^j = (2^i - 2^{j+1} + c) + 2^j = 2^i - 2^j + c$$

   Hence, the i^{th} bit is 0 as in Case 3, so $\tilde{r}_i = 0$.

2. If $r_j = 1$, then $c = 2^j + \tilde{c}$ with $\tilde{c} < 2^j$ and hence

$$r = 2^i - 2^{j+1} + c = 2^i - 2^j + \tilde{c}$$

   Thus, $\tilde{r} = r + 2^j = 2^i + \tilde{c}$ and the i^{th} bit is 1 as in Case 1, so $\tilde{r}_i = 1$.

□

Note that Lemma 2.3.1 works regardless of which values the j least significant bits have in r since they do not interfere with the higher bits.

2.4 The Basics of Security by Encryption

The essence of a secure encryption is that knowing the ciphertext gives no additional information about the plaintext. Hence, the probability of a certain plaintext message occurring is the same regardless of whether the ciphertext is known or not. Due to the seminal work of Claude Shannon, the field of Information Theory was created and precise conditions for perfect secrecy were able to be formulated mathematically [Sha49].

However, the only method of encryption which is theoretically capable of this perfect secrecy is the One Time Pad in which the original message to be transmitted is combined with a completely random key of equal length. Of course, this is subject to the main conditions that the key be kept completely secret and be used one time only - often necessitating that multiple keys be distributed beforehand to all parties on a pad (hence its namesake).

Unfortunately, the One Time Pad is not a very practical method as it requires that all parties involved in the encrypted transmission agree ahead of time on the pad key to be utilized. In other words, it is not very useful for public key encryption where the parties who wish to communicate may have never meet as is almost always the case for online purchase transactions. Therefore, modern Cryptography focuses instead on the development of reasonably difficult to break secrecy where the foundation of this difficulty is based around the existence of one-way functions.

Definition 2.4.1 (One Way Functions)**.** A function $f\colon X \to Y$ is considered *one-way* if:

1. Given any $x \in X$, it is computationally "easy" to find $f(x)$.
2. Given "almost any" $y \in Y$, it is computationally "difficult" to find some $x \in X$ such that $f(x) = y$.

Definition 2.4.2 (Computationally Easy)**.** A function is "easy" when there exists an algorithm which can compute the value of the function for all inputs in probabilistic polynomial time. That is when the algorithm is inherently random and runs in a time bounded by some polynomial function of the input size.

Definition 2.4.3 (Computationally Difficult)**.** A function is "difficult" when there exists no such algorithm as described above.

Remark. In the above definition, "almost any" is used to suggest that it may be possible for a few specific elements to be "easy". Another way of phrasing it could be: Given an element $y \in Y$ *at random*, it is computationally "difficult" to find some $x \in X$ such that $f(x) = y$ [MvOV96, p. 8].

In Cryptography, it is often of great importance to possess one-way functions for the purpose of encryption. The primary use of one-way functions is for establishing an encrypted communication between two parties as will be covered later. Unfortunately, regardless of how computationally intractable many one-way functions appear, it is not currently known whether or not such functions really exists.

When working in an infinite field such as the real numbers it is just as easy to exponentiate, *i.e.* $h = g^r$, as it is to take a logarithm, *i.e.* $r = \log_g h$. In the setting of finite cyclic groups, it is still fast to exponentiate modulo the group order using exponentiation by squaring (covered in Section 3.2) where the time is polynomial in the number of bits comprising r. However, calculating the logarithm, that is finding r so that $h = g^r$, is considerably more difficult. It is this property that has lead to the widely held belief that modular exponentiation is a one-way function [Odl00, p. 6]. Along side fast exponentiation by squaring, integer multiplication is the other most common candidate for a one-way function.

The details regarding how a one-way function is used to achieve encryption depends upon the encryption scheme, but the basic concept is that the "easy" way is used to encrypt the data with the knowledge that it is "difficult" to reverse the operation and decrypt the data. In the case of integer factorization, there is usually a trapdoor built into the algorithm which allows the previously "difficult" reverse operation to be performed relatively quickly. In the case of exponentiation by squaring, the recipient of the data reciprocates a similar process on their end and their results taken together and combined with a judicial use of discrete mathematics provides the means for decryption (covered in Chapter 4).

To better understand the relevance of one-way functions, the reader is encouraged to read the seminal paper of Diffie and Hellman [DH76] where the concept was first introduced. The particular case of discrete exponentiation is examined and expanded upon in the following chapters.

Chapter 3

Introduction to the Discrete Logarithm

The Discrete Logarithm Problem (occasionally referred to as the DLP) is an important problem whose solution is believed to be as difficult as that of Integer Factorization. Like Integer Factorization, the Discrete Logarithm is an example of a believed one-way function and as such it is easiest to define it via its inverse - the discrete exponentiation function.

3.1 Discrete Exponentiation

Discrete exponentiation is the analogue of the familiar exponentiation just taking place in a finite cyclic group.

Definition 3.1.1 (Discrete Exponentiation)**.** Let G be a finite cyclic group of order n, written multiplicatively, and let g be a generator of G. Then the discrete exponentiation function is defined as would be expected:

$$\exp_g \colon \mathbb{Z}_n \longrightarrow G,\ r \longmapsto g^r$$

Since g is a generator of G, its order is n and hence for i in $\mathbb{Z}_n$ the elements g^i are distinct from one another. Thus the map $\exp_g$ is injective. However, both the domain $\mathbb{Z}_n$ and the range G of this map contain n elements which means $\exp_g$ is in fact bijective. Now, the naive method of computing this function is to simply calculate $g^r = \overbrace{g \cdots g}^{r}$ via repeatedly multiplying g by itself $r - 1$ times which requires $\mathcal{O}(r)$ group operations. Luckily there is a vastly superior technique which is covered in the next section.

3.2 Exponentiation by Squaring

The trick to developing a faster exponentiation method is based on exploiting the binary representation of the exponent $r = (r_{k-1} \dots r_1 r_0)_2$. First note that r can be written succinctly as

$$r = \sum_{j=0}^{k-1} r_j 2^j = \sum_{\substack{j=0 \\ r_j=1}}^{k-1} 2^j = \sum_{r_j=1} 2^j$$

Thus, g^r can be expressed as

$$g^r = g^{\sum_{r_i=1} 2^i} = \prod_{r_i=1} g^{2^i}$$

Then, taking advantage of the relationship

$$g^{2^i} = (g^{2^{i-1}})^2$$

it is possible to produce the g^{2^i} in succession using a single squaring operation. Hence, the number of group operations required to compute g^r is based only on the binary length of r!

This procedure, known as exponentiation by squaring, can be expressed succinctly in the following recursive form:

$$\exp_g(r) := \begin{cases} 1, & \text{if } r \text{ is } 0 \\ \exp_g^2(r/2), & \text{if } r \text{ is even} \\ g \cdot \exp_g^2((r-1)/2), & \text{if } r \text{ is odd} \end{cases}$$

Because it only needs at most as many multiplication and squaring operations as the length of the binary representation of the input, this method runs in just $\mathcal{O}(\log_2(r))$ time. That means that this improvement allows for the extremely fast calculation of the algorithm even for very large values of r. Furthermore, the properties of groups guarantee that $g^{-r} = (g^r)^{-1}$ which makes it possible to also use this method with negative exponents,

$\exp_g(-r) = \exp_g^{-1}(r)$, provided there is a means to invert elements. Now that the discrete exponentiation function is defined and understood to be efficient to calculate, the next section will cover its inverse, and the essential topic of the paper, the discrete logarithm function.

Procedure 3.2.1 Exponentiation by Squaring

```
Require: $g$ in $G$ and $r$ in $\mathbb{Z}$ with $G$ a multiplicative cyclic group
Ensure: $h = g^r$
h ← 1    /* 1 is the identity element of G */
while $r > 0$ do
    if $r$ is odd then
        $h \leftarrow h \cdot g$
        $r \leftarrow r - 1$
    end if
    $h \leftarrow h^2$
    $r \leftarrow r/2$
end while
```

3.3 The Discrete Logarithm

As mentioned in the beginning of the chapter, the discrete logarithm is simply defined as the inverse of the discrete exponential function:

$$\log_g \colon G \longrightarrow \mathbb{Z}_n$$

While this definition is quite correct, for the sake of clarity, it is defined more verbosely below and some additional formulations of the problem are given as well.

Definition 3.3.1 (Discrete Logarithm)**.** Let G be a finite cyclic group of order n, written multiplicatively, let g be a generator of G, and let h be an element of G. Then the *discrete logarithm* of h to the base g is the unique integer r in $\mathbb{Z}_n$ such that $h = g^r$, denoted $\log_g(h)$.

Note that in defining the discrete logarithm, it actually suffices to chose an element h of large order in an arbitrary multiplicative group G and then work in the generated subgroup $H := \langle h \rangle \subset G$. On the other hand, cryptographic schemes such as the Diffie-Hellman key exchange, covered in Section 4.1, use the discrete logarithm for producing random elements. However, then these elements are not truly random in G as they are confined to the subgroup H. Thus, as explained in [Kob87, pp. 95–96], if these random elements are to have any chance of appearing random in G, it follows that H and G should coincide with each other as in Definition 3.3.1.

Remark. The reader should note that some authors choose to refer to the discrete logarithm as the *index*, denoted ind_g, as this is the classical designation originating from when Gauß constructed index tables of the discrete logarithm for various values of the prime p.

It may help the reader to imagine the multiplicative group $\mathbb{Z}_p^* = \{1, \ldots, p-1\}$ as a concrete instance of the finite cyclic group G. In this case:

$$\log_g \colon \mathbb{Z}_p^* \longrightarrow \mathbb{Z}_{p-1}$$

That is, the discrete logarithm converts multiplication of integers modulo a prime p into addition of integers modulo $p-1$. Additionally, most examples in the paper will be given in this group.

Example 3.3.1. Let $G := \mathbb{Z}_{29}^*$ be the finite cyclic group, let $g := 2$ be the generator of G, and let $h := 17$ be the element of G. Then the discrete logarithm is $r := \log_g(h) = \log_2(17) = 21$ because $g^r = 2^{21} = 2097152 \equiv 17 \pmod{29}$.

Definition 3.3.2 (Generalized Discrete Logarithm Problem)**.** Given a finite cyclic group G of order n, written multiplicatively, a generator g of G, and an element h of G, then the generalized discrete logarithm problem or *GDLP* is the problem of finding the unique r in $\mathbb{Z}_n$ such that $h = g^r$.

Definition 3.3.3 (Discrete Logarithm Problem)**.** Given a prime number p, a generator g of $\mathbb{Z}_p^*$, and an element h of $\mathbb{Z}_p^*$, then the discrete logarithm problem or *DLP* is the problem of finding the unique r in $\mathbb{Z}_n$ such that $h = g^r$.

Since the most common setting for the discrete logarithm has historically been in $\mathbb{Z}_p^*$, the discrete logarithm problem normally only refers to the later case. Thus there is a need to distinguish between the generalized case and the special case of the discrete logarithm problem in modern parlance. Furthermore, it is necessary to restrict the value of r in the previous definitions to the range $0 \leq r < n$ to ensure a *unique* answer to the solution of $g^r = h$. This is due to the fact that $g^n = e$ where e is the identity element of G. Hence if $r := \log_g(h)$ is a solution to $g^r = h$, then so are all infinitely many of $r_j := r + j \cdot n$ for j in $\mathbb{Z}$ since

$$g^{r_j} = g^{r+j \cdot n} = g^r(g^n)^j = g^r(e)^j = g^r = h$$

Therefore, it is always implicitly assumed that when dealing with the discrete logarithm in a finite group G of order n, the corresponding exponents are calculated in $\mathbb{Z}_n$, in other words they are treated as integers modulo n. Now that the discrete logarithm has been defined, its properties are examined in the next section.

3.4 Properties of the Discrete Logarithm

The same properties of the familiar logarithm on the real number line fortunately also hold for the discrete version. For the remainder of this section, let G be a finite cyclic group of order n, written multiplicatively, and let g be a generator of G.

Definition 3.4.1 (Multiplication Property)**.** Given two elements h_1 and h_2 of G, then

$$\log_g(h_1 \cdot h_2) \equiv \log_g(h_1) + \log_g(h_2) \pmod{n}$$

Definition 3.4.2 (Exponentiation Property)**.** Given an element h of G and an integer a in $\mathbb{Z}$, then

$$\log_g(h^a) \equiv a \log_g(h) \pmod{n}$$

Definition 3.4.3 (Change-of-Base Property)**.** Given a second generator $\tilde{g}$ of G and an element h of G, then

$$\log_{\tilde{g}}(h) \equiv \log_g(h) / \log_g(\tilde{g}) \pmod{n}$$

Proof. The proofs of these properties are left as an exercise for the reader □

Remark. The cautious reader should have noticed that since $\mathbb{Z}_n$ is an additive group, division is not generally possible as not all elements are guaranteed to be invertible. In fact, an element r of $\mathbb{Z}_n$ is invertible if and only if $\gcd(r, n) = 1$, that is, when r and n are relatively prime. However, given that $\tilde{g} = g^k$ for some k in $\mathbb{Z}_n$ and that $\tilde{g}$ is a generator of G if and only if $\gcd(k, n) = 1$, it should now be clear that $\log_g(\tilde{g}) = k$ is invertible in $\mathbb{Z}_n$.

As a result of the Change-of-Base Property, the choice of the generator in either definition of the discrete logarithm problem is irrelevant to the actually difficultly of the problem. This can be seen as follows. Assume there is a method for solving the discrete logarithm problem to the base g in the group G. Then given a second generator $\tilde{g}$ of G and an element h of G, it is possible to efficiently convert the problem of finding $\log_{\tilde{g}}(h)$ into a problem involving the discrete logarithm base g via the Change-of-Base Property. Thus, all generators are essentially equivalent.

The next chapter demonstrates how the discrete logarithm is used as the basis of various cryptographic schemes and why its simple definition allows short and concise implementations of these schemes.

Chapter 4

Applications of the Discrete Logarithm

In this section, two applications for the discrete logarithm are presented to motivate the elegance of methods based around the discrete logarithm and also to demonstrate its importance in the field of Cryptography.

4.1 The Diffie-Hellman Key Exchange

Until 1976, the only methods for sending sensitive data between two parties across insecure transmission lines required that both parties shared a previously agreed upon random secret key. One party could use the key to encrypt the data before sending it and the other party could use the same key to decrypt and recover the received data. Unfortunately, this required each party to store a normally lengthy list of secret keys for each of the possibly many parties with whom they wished to communicate. Worse still, it was impossible for two parties to communicate securely if they had never meet and hence not previously agreed on this secret key.

What was needed was a way for both parties to generate the same random secret key by only transmitting partial (public) information while maintaining some additional (private) information. That is, for both parties, a combination of ones private information with the others public information could procure the common secret key that would be used with the usual encryption methods. Nowadays this is called *hybrid encryption.*

The first practical solution to this problem was developed by Diffie and Hellman in their ground breaking paper "New Directions in Cryptography" [DH76] and took advantage of the difficultly of the discrete logarithm problem. The following procedure describes how their original protocol is used to establish the common, random secret key between two parties over an insecure connection. As is customary in most cryptographic literature, the names of the two parties in question are referred to as Alice and Bob.

Procedure 4.1.1 Diffie-Hellman Key Exchange

1: Both Alice and Bob agree on a large prime p and a generator g of $\mathbb{Z}_p^*$ by communicating over the insecure channel.

2: Alice selects a random element x in the range $1 < x < p - 1$, computes $X := g^x \pmod{p}$, keeps x private and makes X public to Bob over the insecure channel.

3: Bob selects a random element y in the range $1 < y < p - 1$, computes $Y := g^y \pmod{p}$, keeps y private and makes Y public to Alice over the insecure channel.

4: Alice determines the secret random key as $K := Y^x \pmod{p}$.

5: Bob determines the secret random key as $K := X^y \pmod{p}$.

Notice that Alice and Bob need only perform Step 1 once ever since, for an appropriately large prime p, the group $\mathbb{Z}_p^*$ provides a huge selection of random elements for use in Steps 2 and 3. Also, since exponentiation by squaring from Section 3.2 can be used to compute X, Y and K, Steps 2 through 5 can always be found efficiently. Most importantly, in Steps 4 and 5, Alice and Bob do in fact both arrive at the same random secret key since

$$K \equiv Y^x \equiv (g^y)^x \equiv (g^x)^y \equiv X^y \pmod{p}$$

Hence, using this procedure allows Alice and Bob to agree upon a random secret key in $\mathbb{Z}_p^*$ without having to communicate it explicitly over their shared insecure channel.

It is conjectured that for an eavesdropper to use $X := g^x \pmod{p}$ and $Y := g^y \pmod{p}$ to determine $K := g^{xy} \pmod{p}$ directly, is difficult. This is known as the Computational Diffie-Hellman (CDH) assumption and it is believed to be as hard as the discrete logarithm problem [MvOV96, sec. 3.7], although this remains unproven as of March 2007. Therefore, the best chance to recover the secret key is to use the discrete logarithm to determine $x = \log_g(X)$, or $y = \log_g(Y)$, and then to calculate $K = Y^x \pmod{p}$, or $K = X^y \pmod{p}$. However, for appropriate primes p, the intractability assumption on the discrete logarithm problem implies that this task is very difficult and hence the Diffie-Hellman key exchange is quite secure.

4.2 The ElGamal Encryption Scheme

Although hybrid encryption systems provide a good balance of speed and security, it is possible to use the same machinery of the previous section to design a full cryptosystem. It turns out that the Diffie-Hellman key exchange can be extended to encrypt messages by utilizing the secret key as a session key as demonstrated in the following encryption scheme developed by ElGamal [ElG85]. In this case, it is assumed that Bob wishes to send a secure message to Alice over an insecure channel.

Procedure 4.2.1 The ElGamal Encryption Scheme

```
1: Both Alice and Bob agree on a large prime $p$ and a generator $g$ of $\mathbb{Z}_p^*$ by communicating over the insecure channel.
2: Alice selects a random element $x$ in the range $1 < x < p - 1$, computes $X := g^x \pmod{p}$, keeps $x$ private and makes $X$ public to Bob over the insecure channel.
3: Bob selects a random element $y$ in the range $1 < y < p - 1$, computes $Y := g^y \pmod{p}$, and keeps $y$ private.
4: Bob then encodes his secret message $m$ in $\mathbb{Z}_p^*$ as $\widetilde{m} := X^y \cdot m$ and sends the pair $(Y, \widetilde{m})$ to Alice.
5: Alice decodes Bob's message as $m = Y^{-x} \cdot \widetilde{m}$ using her private key $x$.
```

Notice that the first three steps of this procedure are virtually identical to those of the Diffie-Hellman key exchange from Procedure 4.1.1, so only the last two steps are examined in detail. Concerning Step 5, note that since

$$Y^{-x} \equiv Y^{p-1-x} \equiv g^{-xy} \pmod{p}$$

Alice does in fact recover Bob's secret message correctly because

$$Y^{-x} \cdot \widetilde{m} \equiv Y^{-x} \cdot X^y \cdot m \equiv g^{-xy} \cdot g^{xy} \cdot m \equiv m \pmod{p}$$

Now, a few comments regarding Step 4. First, if Bob's secret message m when viewed as a number is too large to be an element of $\mathbb{Z}_p^*$, then it can be broken into smaller, equal-sized chunks m_j with j in $\mathbb{Z}$ which when viewed as numbers are elements of $\mathbb{Z}_p^*$. Then the procedure is simply performed on each m_j. Second, since Bob transmits a pair of elements from $\mathbb{Z}_p^*$, the encrypted message is twice the size of the original message which can be seen as a disadvantage when compared to other cryptosystems.

The security of this encryption scheme is clearly dependent on the security of the Diffie-Hellman key exchange and hence on the Computational Diffie-Hellman assumption. However, whereas the key generated by the Diffie-Hellman key exchange is normally used as a seed for a symmetric encryption system, in this case it is used immediately to encrypt Bob's secret message. In fact, this scheme can be viewed as Bob merely "masking" his secret message m with the common secret key $K := X^y \pmod{p}$ and providing the "clue" Y to Alice which together with her secret x allow her to remove the mask and decrypt the message ([Kob87, p. 97] and [MvOV96, note 8.23]).

Unfortunately, the Legendre symbol (covered in Section 6.1.2) of X and Y exposes the Legendre symbol of K. Thus, K leaks some information which an eavesdropper could use in gleaning some information about m from $\widetilde{m} = K \cdot m$ [Bon98, p. 10]. Therefore, the security of the ElGamal encryption scheme rests on the Decisional Diffie-Hellman (DDH) assumption which states that a tuple (g, g^x, g^y, g^{xy}) with x and y chosen at random is computationally indistinguishable from a tuple (g, g^x, g^y, g^z) with x, y and z chosen randomly. This assumption is believed to be stronger than the Computational Diffie-Hellman assumption.

This section detailed how the discrete logarithm provides an incredibly versatile basis for use in cryptographic applications. The next section covers various methods for solving the discrete logarithm problem and hence indirectly demonstrates the reasons why the discrete logarithm is considered computationally intractable.

Chapter 5

Determining the Discrete Logarithm

It is important to cover the various techniques currently known for solving the discrete logarithm problem both to understand better the reasons it is believed to be computationally intractable as well as to develop a background for the next section of the paper.

5.1 Trial Multiplication

The most obvious approach to solving the discrete logarithm in G is to continually exponentiate the generator g until the desired group element is reached, *i.e.* Try using $i = 0, \ldots, n-1$ until $g^i = h$ at which point the solution is i. This method is clearly not effective when the group order gets too large, as is generally the case, since it requires $\mathcal{O}(n)$ group operations to run.

5.2 Explicit Form

A surprising result regarding the discrete logarithm is that there does in fact exist an explicit formula over the multiplicative group of fields of prime power order. As was shown by Niederreiter in [Nid90], given any h in $\mathbb{F}^*_{p^f}$ with $p^f \geq 3$, the following holds:

$$\log_g(h) \equiv -1 + \sum_{i=1}^{p^f-2} \frac{y^i}{g^{-i}-1} \pmod{p}$$

Since the equation holds modulo p, it is only an explicit formula when dealing with prime fields (*i.e.* $f = 1$), in which case it does yield a useful result. However, while this is an elegant solution, it is no better than the trial multiplication method above as it must take the sum across all of the elements of the group which necessitates $\mathcal{O}(n)$ group operations.

5.3 Shanks' Baby-Step Giant-Step Method

The Baby-Step Giant-Step [Sha71] is the most straight forward and intuitive of the algorithms for solving the discrete logarithm problem in an arbitrary finite cyclic group of order n. It is based on the premise that r may be expressed as $r = s \cdot m + t$ with $0 \leq s, t \leq m - 1$ for the constant $m := \lceil \sqrt{n} \rceil$. This yields

$$h = g^r = g^{s \cdot m + t} \Rightarrow g^{s \cdot m} = hg^{-t}$$

This insight gives Procedure 5.3.1, attributed to Shanks.

Procedure 5.3.1 Shanks' Baby-Step Giant-Step Method

```
Require: $G$ a multiplicative cyclic group of order $n$ with generator $g$ and $h$
    an element of $G$.
Ensure: $r = \log_g(h)$
$H \leftarrow \emptyset$    /* $H$ is an associative array or hash table */
for $s = 0$ to $m - 1$ do
    $H[g^{s \cdot m}] \leftarrow s$    /* Store the pair $(g^{s \cdot m}, s)$ in $H$ */
end for
for $t = 0$ to $m - 1$ do
    if $H[hg^{-t}]$ exists then    /* If $(hg^{-t}, \cdot)$ matches an entry in $H$ */
        $s \leftarrow H[hg^{-t}]$
        $r \leftarrow s \cdot m + t$
    end if
end for
```

In practice, it is best to use a hash table to store the tabulated results from the first loop which allows a near constant time lookup in the second loop. Further improvements are attained by simply pre-computing g^{-1} before the loop and by keeping a running variable $\tilde{h}$ which is initialized to h and updated in the loop as $\tilde{h} := \tilde{h}g^{-1}$ giving $\tilde{h} = hg^{-t}$ to use inside the second loop.

Example 5.3.1. Let $G := \mathbb{Z}_{29}^*$ with generator $g := 2$, find $log_g(17)$. First, let $h := 17$, $m := \lceil\sqrt{29}\rceil = 6$, $g^{-1} = 2^{-1} \equiv 15 \pmod{29}$. Build the pairs for the first step.

s	0	1	2	3	4	5
$g^{s\cdot m} \equiv 2^{6\cdot s} \pmod{29}$	1	6	7	13	20	4

Begin the loop for the second step.

loop iteration, t	0	1	2	3
$hg^{-t} \equiv 17 \cdot 2^{-t} \pmod{29}$	17	23	26	13

When $t = 3$ the algorithm finds $17 \cdot 2^{-6\cdot 3} = 13$ in the table paired with $s = 3$, it equates $2^{6\cdot 3} = 17 \cdot 2^{-3}$. Hence $17 = 2^{21}$ and so it computes $r := s \cdot m + t = 3 \cdot 6 + 3 = 21$ as the answer.

The most important aspect of this algorithm is that the practical implementation solves the discrete logarithm problem deterministically in $\mathcal{O}(\sqrt{n})$ time and $\mathcal{O}(\sqrt{n})$ space. For the curious, the name of the algorithm is derived from the fact that the first loop calculates the powers of g^i in "giant" steps $i = 0, m, 2m, \ldots, m(m-1)$ whereas the second loop effectively calculates the powers of g^i in "baby" steps $i = 0, 1, 2 \ldots, m-1$.

5.4 Pollard's Rho Method

While the previous method is able to solve the discrete logarithm problem in $\mathcal{O}(\sqrt{n})$ space and time, as n becomes very large the requirement of $\mathcal{O}(\sqrt{n})$ space becomes unreasonable for computers. On the other hand, even for the same large n, the $\mathcal{O}(\sqrt{n})$ time requirement is far more accessible to computers. For this reason, it would be nice to possess an algorithm that reduces the space complexity to a constant $\mathcal{O}(1)$ which is precisely what the following algorithm achieves.

The Pollard Rho Method [Pol78] is a probabilistic algorithm based on Floyd's cycle-finding algorithm for detecting cycles in arbitrary sequences in just $\mathcal{O}(1)$ space. Given the problem of solving for r in the equation $h = g^r$, the idea is to generate a pseudo-random sequence of the form $s_i := h^{x_i} g^{y_i}$ with x_i and y_i in $\mathbb{Z}$. When a cycle $s_i = s_j$ is found in the sequence, it gives $h^{x_i} g^{y_i} = h^{x_j} g^{y_j}$ and hence $g^{y_i - y_j} = h^{x_j - x_i} = g^{r \cdot (x_j - x_i)}$. Therefore, the sought after r can be found be solving the equation $y_i - y_j \equiv r \cdot (x_j - x_i) \pmod{n}$.

One implementation of the pseudo-random sequence generator is as follows. First, given the cyclic group G of order n, partition the group into three subsets S_0, S_1 and S_2 of roughly equal size such that S_1 does not contain 1 and such that it is quick to determine set membership for all elements. Second, define the pseudo-random sequence as:

$$s_0 := 1 \quad \text{and} \quad s_{i+1} := \begin{cases} h \cdot s_i, & s_i \text{ in } S_0; \\ s_i^2, & s_i \text{ in } S_1; \\ g \cdot s_i, & s_i \text{ in } S_2 \end{cases}$$

which indirectly defines the following two sequences reflecting the exponents of h and of g in $s_i := h^{x_i} g^{y_i}$ generated via the above:

$$x_0 := 0 \quad \text{and} \quad x_{i+1} := \begin{cases} x_i + 1 \pmod{n}, & s_i \text{ in } S_0; \\ 2x_i \pmod{n}, & s_i \text{ in } S_1; \\ x_i, & s_i \text{ in } S_2 \end{cases}$$

$$y_0 := 0 \quad \text{and} \quad y_{i+1} := \begin{cases} y_i, & s_i \text{ in } S_0; \\ 2y_i \pmod{n}, & s_i \text{ in } S_1; \\ y_i + 1 \pmod{n}, & s_i \text{ in } S_2 \end{cases}$$

Third, begin generating the sequence s_i as well as a second sequence s_{2i} (accomplished by merely iterating the formulas twice each step) until $s_i = s_{2i}$ which is quite likely to happen since the group is finite. Then $h^{x_i} g^{y_i} = h^{x_{2i}} g^{y_{2i}}$ and hence $g^{y_{2i} - y_i} = h^{x_i - x_{2i}} = g^{r \cdot (x_i - x_{2i})}$ which yields $r \cdot (x_i - x_{2i}) \equiv (y_{2i} - y_i) \pmod{n}$. Therefore, this procedure produces a congruence that can be solved to give the desired value of r for $\log_g(h)$.

In the highly improbable case that $x_i \equiv x_{2i} \pmod{n}$, it is necessary to restart the algorithm using a different initial condition $s_0 := h^{x_0} g^{y_0}$ with x_0 and y_0 randomly chosen from $\{1, \ldots, n-1\}$. Otherwise, let $a := x_i - x_{2i} \neq 0$ and let $b := y_{2i} - y_i$, then the congruence

$$ar \equiv b \pmod{n} \tag{5.1}$$

has a solution only when $\gcd(a, n) = 1$, *i.e.* when a and n are relatively prime. If this is the case, then the r can be found and the process ends.

If this is not the case, then $d := \gcd(a, n) > 1$. So, let $\tilde{n} := n/d$, let $\tilde{a} := a/d$, and let $\tilde{b} := b/d$ which is valid since $b = ar - nk$ for some integer k. Then note that $\gcd(\tilde{a}, \tilde{n}) = 1$ and so

$$\tilde{a}\tilde{r} \equiv \tilde{b} \pmod{\tilde{n}} \tag{5.2}$$

has a solution. Given that $\tilde{r}$ is the solution to (5.2), then the solution to (5.1) must satisfy $r \equiv \tilde{r} \pmod{\tilde{n}}$. This means that $r = \tilde{r} + l \cdot \tilde{n}$ for some l in the range $0 \leq l < d$. Hence, by testing all of these possible values of l, the solution r can be discovered. In the event that d is too large for this technique to be practical, it is necessary to repeat the entire procedure from the start with different initial conditions and hope that the situation improves.

Assuming that the sequences act truly random, that is, they yield all group elements with equal probability, then the probability that the two sequences both generate the same group element on the same step is governed by the principal of the Birthday Paradox. It states that after individually selecting m random elements from a group of n elements, the probability that two are the same is approximately $1 - e^{-m^2/2 \cdot n}$. Thus, in practice, the algorithm only requires approximately $0.6267\sqrt{n}$ steps to run and due its probabilistic nature only a constant amount of space.

5.5 The Pohlig-Hellman Method

The previous two methods for solving the discrete logarithm gave explicit instructions on how to tackle the problem in a group G. However, when the order of the group becomes exceedingly large, neither method is computationally feasible any longer. This section introduces the Pohlig-Hellman method [PH78] which is a process for reducing the discrete logarithm problem in a group of large order into many discrete logarithm problems in groups of smaller order, under certain circumstances. This allows other techniques to be used in the smaller groups such as Baby-Step Giant-Step from Section 5.3 or Pollard Rho from Section 5.4.

In the following arguments adapted from [Sti02, ch. 5], assume that G is a multiplicative cyclic group of large finite order n with a generator g, that

h is an element of G, and that $r = \log_g(h)$ is the solution to the discrete logarithm problem in question, that is $h = g^r$.

Now, suppose that

$$n = p_0^{e_0} p_1^{e_1} \cdots p_k^{e_k} = \prod_{i=0}^{k} p_i^{e_i}$$

is the unique prime factorization of the group order n with $e_i \geq 1$. If the solutions r_i to the equations

$$r \equiv r_i \pmod{p_i^{e_i}}$$

were known for all i in $\{0, \dots, k\}$, then by using the Chinese Remainder Theorem, it would be possible to construct the desired solution $r \pmod{n}$. Hence, the analysis may be restricted to finding a particular solution $r_i \pmod{p_i^{e_i}}$.

Thus, assume that i is fixed and to simplify the notation that

$$b := r_i, \qquad e := e_i, \qquad \text{and} \qquad p := p_i$$

Then, viewing b as a number written base p gives

$$b = (b_{e-1} \dots b_1 b_0)_p = \sum_{j=0}^{e-1} b_j p^j$$

with b_j in $\{0, \dots, p-1\}$. Further, note that the desired solution r can be expressed in terms of b as

$$r = b + p^e m$$

for some integer m. Hence the following chain of equations modulo n holds

$$\begin{aligned}
\frac{n}{p} \cdot r &\equiv \frac{n}{p} \cdot (b + p^e m) \\
&\equiv \frac{n}{p} \cdot \left(\sum_{j=0}^{e-1} b_j p^j + p^e m \right) \\
&\equiv \frac{n}{p} \cdot \left(\sum_{j=1}^{e-1} b_j p^j + p^e m + b_0 \right) \\
&\equiv n \cdot \left(\sum_{j=1}^{e-1} b_j p^{j-1} + p^{e-1} m \right) + \frac{n}{p} \cdot b_0 \\
&\equiv \frac{n}{p} \cdot b_0 \pmod{n}
\end{aligned}$$

This formula combined with the fact that $g^n \equiv 1 \pmod{n}$ yields the key insight for this section

$$h^{n/p} \equiv g^{(n/p)\cdot r} \equiv g^{(n/p)\cdot b} \equiv g^{(n/p)\cdot b_0} \pmod{n} \tag{5.3}$$

Again, cleaning up the notation, let

$$\tilde{h} := h^{n/p} \quad \text{and} \quad \tilde{g} := g^{n/p}$$

Next note that $\tilde{g}$ is a generator for the cyclic subgroup of order p in G. Thus Equation (5.3) can be expressed as

$$\tilde{h} \equiv \tilde{g}^{b_0} \pmod{n}$$

and so $b_0 = \log_{\tilde{g}}(\tilde{h})$ gives a new discrete logarithm problem to be solved in a group of order p which depending on the size of p is likely far more accessible than in G.

Of course, if $e > 1$, then the remaining values for b_1 up to b_{e-1} still need to be computed. To handle this, note that

$$p \text{ divides } \sum_{j=1}^{e-1} b_j p^j = (b_{e-1} \dots b_1 0)_p = b - b_0$$

and hence b can be redefined as

$$b := (b - b_0)/p$$

which effectively redefines h as

$$h := (hg^{-b_0})^{n/p}$$

in order to maintain the relation $h \equiv g^b \pmod{n}$. This shifts the base p representation of b down by one, leaving b_1 as the least significant digit where b_0 was before. By (5.3), this results in

$$h^{n/p} \equiv g^{(n/p)\cdot b} \equiv g^{(n/p)\cdot b_1} \pmod{n}$$

which means the process used to recover b_0 can again be used for b_1. Continuing the process of redefining b and h, applying (5.3), and solving the new discrete logarithm problem, allows all of the digits b_j of b to be found.

Once all of the digits of $b := r_i$ have been determined, the entire method can be repeated to find r_i for all i and then these results recombined via the Chinese Remainder Theorem to yield the final answer for $r = \log_g(h)$ where h is its original value. This is formally expressed in Procedure 5.5.1.

Procedure 5.5.1 The Pohlig-Hellman Method

```
Require: $G$ a multiplicative cyclic group of order $n$ with generator $g$ and $h$
    an element of $G$.
Ensure: $r = \log_g(h)$
Find the prime factorization of $n = p_0^{e_0} p_1^{e_1} \cdots p_k^{e_k}$ with $e_i \geq 1$.
for $i = 0$ to $k$ do
    $e \leftarrow e_i$
    $p \leftarrow p_i$
    $\tilde{g} \leftarrow g^{n/p}$
    for $j = 0$ to $e - 1$ do
        $\tilde{h} \leftarrow h^{n/p}$
        $b_j \leftarrow \log_{\tilde{g}}(\tilde{h})$    /* i.e. using the Pollard Rho method (5.4) */
        $h \leftarrow (hg^{-b_j})^{n/p}$
    end for
    $r_i \leftarrow \sum_{j=0}^{e-1} b_j p^j$
end for
Use the Chinese Remainder Theorem to compute the $r$ such that $r \equiv r_i$
    $\pmod{p_i^{e_i}}$ for all $i$.
```

As this procedure only requires a polynomial number of group operations to reduce and recombine the results, its running time is realistically bounded by the specific method used to solve the discrete logarithm problem in the smaller groups. Thus, it is possible to select whichever method is best suited for the particular group to obtain optimal performance. Unfortunately, this procedure is only suitable when the prime factorization of n contains mostly small primes.

5.6 The Index Calculus Method

Until now, all of the methods for solving the discrete logarithm problem required, in the worst case, at least $\mathcal{O}(\sqrt{n})$ group operations when working in a cyclic group of order n. This section briefly describes a novel method for solving the discrete logarithm problem which has a subexponential running time. Known as the Index Calculus method, it is divided into two distinct stages. The first stage consists of pre-computing the solutions to a number of discrete logarithm problems in the group while the second stage then attempts to solve a specific discrete logarithm problem by utilizing these pre-computed solutions.

1. The First Stage

   Begin by forming a small subset $P := \{p_1, \ldots, p_k\}$ of elements from G, called the *factor base*, such that a substantial portion of the elements in G can be represented as a product of elements from P. Next, continue to select random integers b from $\mathbb{Z}_n$ until g^b can be represented as

   $$g^b = \prod_{i=1}^{k} p_i^{e_i} \tag{5.4}$$

   with $e_i \geq 0$. Then taking the discrete logarithm of both sides of (5.4) yields

   $$b \equiv \sum_{i=1}^{k} e_i \log_g(p_i) \pmod{n} \tag{5.5}$$

   where the $\log_g(p_i)$ are unknown.

   Repeating this process $k + c$ times for some constant $c \geq 0$ gives $k + c$ relations (5.5) in k variables, the $\log_g(p_i)$, which can then be solved modulo n via many available techniques (refer to [Coh96, ch. 2]). This allows the discrete logarithm problem to be determined for each element of the factor base P.

   Note that the size of the factor base k must be chosen large enough that Equation (5.4) exists for most of the randomly selected b, but small enough that solving for the $\log_g(p_i)$ can be done efficiently. Also, the constant c must be carefully tuned to guarantee that the $k+c$ relations have a good chance of yielding a unique solution, although $c \approx 10$ is recommended by many authors ([MvOV96, p. 110] and [Sti02, p. 171]).

2. The Second Stage

In order to find $r := \log_g(h)$ given an element h of G, do the following. Continue to choose random integers b in $\mathbb{Z}_n$, until hg^b can be represented as

$$hg^b = \prod_{i=1}^{k} p_i^{f_i} \tag{5.6}$$

with $f_i \geq 0$. Then, as before, taking the discrete logarithm of both sides of (5.6) and rearranging terms yields

$$\log_g(h) \equiv \sum_{i=1}^{k} f_i \log_g(p_i) - b \pmod{n} \tag{5.7}$$

where this time the $\log_g(p_i)$ are known from the pre-computation in the first step and hence $r := \log_g(h)$ can be found quickly.

Unfortunately, as the cautious reader may have recognized, no technique was mentioned for how to select the factor base or for how to generate the relations in (5.4) and (5.6). That is because efficient means for both these problems are only known for some cyclic groups, notably $\mathbb{Z}_p^*$, $\mathbb{Z}_{2^n}^*$, and $\mathbb{Z}_{p^n}^*$ [MvOV96, sec. 3.6.5]. Luckily, these groups are among the most common in contemporary usage.

Furthermore, compensating for these limitations, is the fact that the Index Calculus method and its variants are the only known subexponential procedures for solving the discrete logarithm problem. More precisely, heuristics have shown that the asymptotic running time for the pre-computation stage is $\mathcal{O}(e^{(1+o(1)\sqrt{\ln p \ln \ln p})})$ and for the second stage is $\mathcal{O}(e^{(1/2+o(1)\sqrt{\ln p \ln \ln p})})$ ([Sti02, p. 172]).

Chapter 6

Individual Bit Security

The reader should now be quite familiar with the definition of the discrete logarithm, its properties, its uses in Cryptography and the various techniques for solving it. The discrete logarithm's distinction as a suspected one-way function clearly implies the elusive nature of an efficient solution for it, if one exists at all. However, it may still be possible to glean some partial information about its solution. This chapter focuses on what information can be easily found, what parts remain secure, and most importantly, what conditions dictate the division between these two extremes.

As working in $\mathbb{Z}_p^*$ provides a simple setting for the presentation of the following sections, it is assumed for the remainder of this chapter that the discrete logarithm takes place in the multiplicative group $\mathbb{Z}_p^*$. Furthermore, the results are accompanied by computer implementations which benefit greatly from the fact that elements of $\mathbb{Z}_p^*$ are integers, a type natively supported in most all programming languages. The reader may wish to refer to the Source Code Listing in Appendix A for concrete demonstrations as they proceed through this chapter.

6.1 Survey of Required Concepts

At the heart of the this chapter is the concept of quadratic residues which, as will be seen later, prove instrumental in developing the results throughout this chapter and ultimately provide a key to inverting the discrete logarithm.

6.1.1 Quadratic Residues

Definition 6.1.1 (Quadratic Residues)**.** Let p be an odd prime number. An integer a is called a *quadratic residue* modulo p, abbreviated QR, if there exists an integer b such that $b^2 \equiv a \pmod{p}$. If no such integer exists, then a is called a *quadratic non-residue* modulo p, abbreviated QNR. This characterization of a is called the *quadratic character* of a.

Since the notion of quadratic residues given above inherently depends on working in the integers modulo an odd prime p, it is logical to restrict their analysis to $\mathbb{Z}_p$, the integers modulo p. Additionally, as the null element, 0, always satisfies the condition for being a quadratic residue modulo any prime p, it is practical to further restrict the analysis of quadratic residues to the cyclic group $\mathbb{Z}_p^* = \mathbb{Z}_p \setminus \{0\}$.

In order to determine the number of quadratic residues in $\mathbb{Z}_p^*$, the following argument taken from [For96, p. 85] is used. First note that that map

$$\mathrm{sq}\colon \mathbb{Z}_p^* \longrightarrow \mathbb{Z}_p^*,\ x \longmapsto x^2$$

is a group homomorphism since $(xy)^2 = x^2y^2$ and that its image coincides with the quadratic residues modulo p. Next note that

$$\mathrm{sq}(x) = x^2 \equiv 1 \Rightarrow (x-1)(x+1) \equiv 0 \Rightarrow x \equiv \pm 1 \pmod{p} \tag{6.1}$$

since $\mathbb{Z}_p$ is an integral domain and so $\mathrm{Ker(sq)} = \{1, p-1\}$ is the kernel of this map. Then, by the first isomorphism theorem of groups:

$$\mathbb{Z}_p^* \ / \ \mathrm{Ker(sq)} \cong \mathrm{Im(sq)} \subset \mathbb{Z}_p^*$$

where Im(sq) is the image of the map. Thus the quadratic residues modulo p form a proper subgroup of $\mathbb{Z}_p^*$, denoted QR_p, of order $|\mathbb{Z}_p^* \ / \ \mathrm{Ker(sq)}| = (p-1)/2$. Therefore, precisely half of the elements in $\mathbb{Z}_p^*$ are quadratic residues and the other half are quadratic non-residues.

Remark. It is possible to enumerate all of the quadratic residues of $\mathbb{Z}_p^*$ by starting with $x := 1$ which is always a quadratic residue and then iterating the relationship $(x+1)^2 \equiv x^2 + (2x+1) \pmod{p}$ until all $(p-1)/2$ quadratic residues are found.

Example 6.1.1. In the multiplicative group $\mathbb{Z}_{19}^*$, the group of quadratic residues is $\mathrm{QR}_{19} = \{1, 4, 5, 6, 7, 9, 11, 16, 17\}$.

Recall that since $\mathbb{Z}_p^*$ is a multiplicative cyclic group, it has a generator and its elements can be represented by powers of this generator. The next two results provide a connection between the quadratic character of an element in $\mathbb{Z}_p^*$ and this representation of the element as a power of a generator of $\mathbb{Z}_p^*$.

Lemma 6.1.1. *Let p be an odd prime and let g be a generator of $\mathbb{Z}_p^*$. Then an element a of $\mathbb{Z}_p^*$ is a quadratic residue if and only if there exists an even integer s in $\mathbb{Z}_{p-1}$ such that $a \equiv g^s \pmod{p}$.*

Proof.

($\Rightarrow$) If a is a quadratic residue it means there exists an element b in $\mathbb{Z}_p^*$ such that $a \equiv b^2 \pmod{p}$. Since $b \equiv g^r \pmod{p}$ for some integer r, it follows that $a \equiv b^2 \equiv g^{2r} \pmod{p}$. Then taking $s := 2r \pmod{p-1}$ gives the desired integer.

($\Leftarrow$) Let $s := 2r$ be the even integer in $\mathbb{Z}_{p-1}$ such that $a \equiv g^s \pmod{p}$. Then, it is clear that $a \equiv g^s \equiv (g^r)^2 \pmod{p}$ and so a is a quadratic residue.

□

Corollary. *Let p be an odd prime and let g be a generator of $\mathbb{Z}_p^*$. Then an element a of $\mathbb{Z}_p^*$ is a quadratic non-residue if and only if there exists an odd integer s in $\mathbb{Z}_{p-1}$ such that $a \equiv g^s \pmod{p}$.*

As a consequence of Lemma 6.1.1 and Corollary 6.1.1, the product of two quadratic residues is again a quadratic residue as is the product of two quadratic non-residues while the product of a quadratic residue and a quadratic non-residue is a quadratic non-residue. This is also somewhat evident from the fact that QR_p is a subgroup of $\mathbb{Z}_p^*$. The next section presents a compact notation for the quadratic character of an element in $\mathbb{Z}_p^*$.

6.1.2 The Legendre Symbol

Due to the importance of quadratic residues in number theory, the concept is expressed by the following succinct notation.

Definition 6.1.2 (The Legendre symbol)**.** Let p be an odd prime number and let a be an integer such that p does not divide a, then the *Legendre symbol* of a over p is defined as follows:

$$\left(\frac{a}{p}\right) := \begin{cases} 1, & \text{if } a \text{ is a quadratic residue modulo } p \\ -1, & \text{if } a \text{ is a quadratic non-residue modulo } p \end{cases}$$

Remark. In the case where the prime p *does* divide the integer a, the Legendre symbol is in fact defined to be 0. This paper, however, chooses to simplify the definition as this case is neither needed nor encountered in the sections which follow.

The next section covers an important result from number theory which provides an explicit formula for evaluating the Legendre symbol.

6.1.3 Euler's Criterion

This section introduces a critical result from Euler which allows for the efficient computation of the Legendre symbol and thus determination of quadratic character.

Theorem 6.1.2 (Euler's Criterion)**.** *Let p be an odd prime and let a be an integer such that p does not divide a, then*

$$\left(\frac{a}{p}\right) \equiv a^{\frac{p-1}{2}} \pmod{p} \tag{6.2}$$

Proof.

1. Case a QR: If a is a QR, then there exists an integer b such that $a \equiv b^2 \pmod{p}$ and hence

   $$a^{\frac{p-1}{2}} \equiv b^{p-1} \equiv 1 \equiv \left(\frac{a}{p}\right) \pmod{p}$$

   by Fermat's Little Theorem.

2. Case a QNR: To remove clutter, the modulus p is occasionally dropped. Let g be a generator of $\mathbb{Z}_p^*$ and let s be the odd integer such that $a \equiv g^s$ by Corollary 6.1.1. Now let $b := g^{\frac{p-1}{2}}$ so that $b^2 \equiv g^{p-1} \equiv 1$. Then, as

shown in Equation (6.1), $b \equiv \pm 1$, but since g has order $p-1$, it follows that $b \not\equiv 1$. Thus, $b \equiv -1$ and hence

$$a^{\frac{p-1}{2}} \equiv g^{s\left(\frac{p-1}{2}\right)} \equiv b^s \equiv (-1)^s \equiv -1 \equiv \left(\frac{a}{p}\right) \pmod{p}$$

as s is odd.

□

It is possible to utilize the method of exponentiation by squaring developed in Section 3.2 to quickly calculate the right-hand side of Equation (6.2). Thus determining whether or not an element a of $\mathbb{Z}_p^*$ is a quadratic residue can be found in polynomial time. This fact will be used extensively in the remainder of this chapter.

Remark. Though elegant in its simplicity, Euler's Criterion is not the only the method for evaluating the Legendre symbol. The Law of Quadratic Reciprocity, together with its supplementary laws, in fact provides the fastest means for finding the Legendre symbol (for more information, consult [For96, ch. 11]). However, the goal of this section is only to demonstrate that the Legendre symbol, and hence quadratic character, *can* be determined in polynomial time.

6.1.4 Taking Square Roots in $\mathbb{Z}_p^*$

Having introduced the concept of quadratic residues modulo an odd prime p, it is now natural to ask about taking the square roots of these elements modulo p.

Definition 6.1.3 (Square Root)**.** Let p be an odd prime and let a be an element of $\mathbb{Z}_p^*$. If b is an integer such that $b^2 \equiv a \pmod{p}$, then b is called a *square root* of a modulo p, denoted $\sqrt{a}$.

Given two elements x and y of $\mathbb{Z}_p^*$ such that $x^2 \equiv y^2 \pmod{p}$, it follows that

$$0 \equiv x^2 - y^2 \equiv (x-y)(x+y) \Rightarrow x \equiv \pm y \pmod{p}$$

since $\mathbb{Z}_p$ is an integral domain. Thus in $\mathbb{Z}_p^*$ each quadratic residue a has precisely two solutions for the unknown x in the equation $x^2 \equiv a \pmod{p}$. Therefore, if b is a square root of a modulo p, then $-b$ is the other square root.

Lemma 6.1.3. *Let g be a generator of $\mathbb{Z}_p^*$ and let a be a quadratic residue in $\mathbb{Z}_p^*$ with $a \equiv g^{2r} \pmod{p}$ for the integer r in the range $0 \leq r < \frac{p-1}{2}$ as per Lemma 6.1.1. Then $x := g^r \pmod{p}$ and $y := g^{r+\frac{p-1}{2}} \pmod{p}$ are the two square roots of a modulo p.*

Proof.

$$x^2 \equiv g^{2r} \equiv a \equiv g^{2r} \cdot 1 \equiv g^{2r} \cdot g^{p-1} \equiv g^{2r+(p-1)} \equiv y^2 \pmod{p}$$

□

Definition 6.1.4 (Principal and Non-Principal Square Roots)**.** Keeping the definitions from Lemma 6.1.3, $x := g^r \pmod{p}$ is called the *principal square root* of a modulo p where as $y := g^{r+\frac{p-1}{2}} \pmod{p}$ is called the *non-principal square root* of a modulo p. (Definitions taken from [BM86]).

The following corollary provides an incredibly useful relationship between the principal square root of an element and its corresponding discrete logarithm. This connection is exploited extensively throughout the remaining sections of this chapter.

Corollary (To Lemma 6.1.3)**.** *If x is the principal square root of a, then $\log_g(x)$ is the right bit shift of $\log_g(a)$*

Unfortunately, Lemma 6.1.3 and Definition 6.1.4 require the previous knowledge of the specific integer r in the exponent to be of any practical use. Thus, utilizing these methods to find the square roots, would necessarily mean calculating the discrete logarithm first! Luckily, there are probabilistic polynomial time algorithms for calculating square roots in $\mathbb{Z}_p^*$, many of which are presented in [MvOV96, ch. 3].

For the purposes of the paper, it suffices to know that square roots *can* be found in probabilistic polynomial time and thus only the trivial case for primes of the form $p \equiv 3 \pmod{4}$ is demonstrated now. Assuming that a is a quadratic residue in $\mathbb{Z}_p^*$, then the two square roots of a modulo p are

$$x := a^{\frac{p+1}{4}} \qquad \text{and} \qquad y := p - x$$

which can be calculated efficiently using Procedure 3.2.1. These are indeed the two square roots since

$$y^2 \equiv x^2 \equiv a^{\frac{p+1}{2}} \equiv a \cdot a^{\frac{p-1}{2}} \equiv a \pmod{p}$$

where $a^{\frac{p-1}{2}} \equiv 1 \pmod{p}$ by Theorem 6.1.2 (Euler's Criterion). Note that this case occurs for half of all primes as a result of Dirichlet's theorem on primes in arithmetic progressions combined with the prime number theorem.

Remark (Warning). Despite being able to find the square roots in probabilistic polynomial time, distinguishing the principal square root from the non-principal square root is still quite difficult. As can be seen in the above example, this is because these techniques work without determining information about the exponents that is required to define the principal and non-principal square roots.

6.2 The Idea of Bit Security

As stated at the beginning of this chapter, just because the discrete logarithm is difficult to calculate, does not mean that each bit of the discrete logarithm is also difficult. This section explores the security of individual bits in the binary representation of the discrete logarithm.

Definition 6.2.1 (The i^{th} Bit Discrete Logarithm Problem)**.** Given an odd prime p, a generator g of $\mathbb{Z}_p^*$, an element h of $\mathbb{Z}_p^*$, and an integer i in the range $0 \leq i < n := \lceil \log_2(p-1) \rceil$, the i^{th} *bit discrete logarithm problem* is the problem of determining the i^{th} bit in the binary representation of the discrete logarithm $\log_g(h)$. That is, if $(r_{n-1} \ldots r_1 r_0)_2 = r := \log_g(h)$ is the discrete logarithm, then what is the value of r_i. It is common in this setting to refer to i as the *bit index*.

It should be clear that the difficulty of solving the discrete logarithm problem guarantees that the i^{th} bit discrete logarithm problem is also difficult for at least some values of i. To be exact, at least $\mathcal{O}(\log_2(n))$ bits of the discrete logarithm problem should be hard for almost all discrete logarithm problems. Otherwise, the easy bits could be determined and then all $2^{\mathcal{O}(\log_2(n))} = \mathcal{O}(n)$ possible permutations of the hard bits could be tried in polynomial time until the correct values were found.

The concept of a hard bit will be made more rigorous once the appropriate mathematical machinery is developed in Section 6.4. Until then, Section 6.3 will focus on investigating which bits of the discrete logarithm problem are easy bits.

6.3 The Easy Bits

The following definition makes precise what is meant by the notion of an easy bit of the discrete logarithm problem.

Definition 6.3.1 (Easy Bit)**.** Given an odd prime p, a generator g of $\mathbb{Z}_p^*$, an element h of $\mathbb{Z}_p^*$, and a bit index i, if the i^{th} bit discrete logarithm problem of $\log_g(h)$ can be solved in polynomial time, then i is called an *easy bit.*

In this section it is shown that the 0^{th} bit is always easy. Additionally, depending on the odd prime p, it is possible that the first $k < n - 1$ bits are also easy where $n := \lceil \log_2(p-1) \rceil$.

6.3.1 The 0^{th} Bit

Recall that given an odd prime p and a generator g of $\mathbb{Z}_p^*$, the Legendre symbol can be used to determine if an element h of $\mathbb{Z}_p^*$ is a quadratic residue in polynomial time. If h is a quadratic residue, then by Lemma 6.1.1 there is an integer r with $2r$ in $\mathbb{Z}_{p-1}$ such that h can be expressed as $g^{2r} \pmod{p}$. Hence, the discrete logarithm is $\log_g(h) = 2r$ and so its 0^{th} bit is 0. Conversely, if h is a quadratic non-residue, then the 0^{th} bit of $\log_g(h)$ is 1. Therefore, the 0^{th} bit discrete logarithm problem is always easy as demonstrated in Procedure 6.3.1.

Procedure 6.3.1 Determining the 0^{th} Bit : easy_bit(p, g, h)

```
Require: $p$ odd prime, $g$ generator of $\mathbb{Z}_p^*$, and $h$ element of $\mathbb{Z}_p^*$
Ensure: $r_0$ is $0^{\text{th}}$ bit of $r := \log_g(h)$
if $\left(\frac{h}{p}\right) = 1$ then    /* $h$ is a quadratic residue */
    $r_0 \leftarrow 0$
else
    $r_0 \leftarrow 1$
end if
```

6.3.2 Converting Quadratic Non-Residues

As a result of Subsection 6.3.1, if $h \equiv g^r \pmod{p}$ is a quadratic non-residue, then r is odd and hence the 0^{th} bit of r is 1. Thus h can be converted into a quadratic residue by multiplying it by g^{-1}, *i.e.* $h := hg^{-1} \Rightarrow r := r - 1$, which effectively toggles the 0^{th} bit of r from 1 to 0 making r even. This process in summarized in Procedure 6.3.2 which always guarantees its output is a quadratic residue, by converting its input as necessary.

Procedure 6.3.2 Converting Quadratic Non-Residues : ensure_QR(p, g, h)

```
Require: $p$ odd prime, $g$ generator of $\mathbb{Z}_p^*$, and $h$ element of $\mathbb{Z}_p^*$
Ensure: $h$ is a quadratic residue.
if $\left(\frac{h}{p}\right) = -1$ then    /* $h$ is a quadratic non-residue */
    $h \leftarrow h \cdot g^{-1} \pmod{p}$    /* Convert $h$ to a quadratic residue */
end if
```

Therefore converting a quadratic non-residue h into a quadratic residue can be done quickly, a fact which is exploited throughout the rest of this paper.

6.3.3 The Bit Border

Given an odd prime p, it can be written as $p = 2^s k + 1$ where k is an odd integer and s is an integer strictly greater than 0 (since $p-1$ is always even). This representation is unique and can be found quickly by repeatedly dividing $p-1$ through by 2 until only k remains as demonstrated by Procedure 6.3.3.

Definition 6.3.2 (Bit Border)**.** Let p be an odd prime number and let $p = 2^s k + 1$ be the unique representation of p as described above with k an odd integer. Then the integer s greater than 0 will be called the *bit border* of p.

Procedure 6.3.3 Determining the Bit Border : bit_border(p)

```
Require: $p$ odd prime
Ensure: $s$ is the bit border of $p$, i.e. $p = 2^s k + 1$ with $k$ odd
$s \leftarrow 0$
$n \leftarrow p - 1$
while $n$ is even do
    $n \leftarrow n/2$
    $s \leftarrow s + 1$
end while
```

Notice that since 2^s divides $p-1$, the first $s-1$ bits of the binary representation of $\frac{p-1}{2}$ are all 0s. Let $h \equiv g^{2r} \pmod{p}$ be a quadratic residue and let $x := g^r \pmod{p}$ and $y := g^{r+\frac{p-1}{2}} \pmod{p}$ be the two square roots of h by Lemma 6.1.3. Then $\log_g(x)$ and $\log_g(y)$ first differ in their $(s-1)^{\text{th}}$ bit since $|\log_g(x) - \log_g(y)| = \frac{p-1}{2} = 2^{s-1}k$. Thus both $\log_g(x)$ and $\log_g(y)$ can be bit shifted down $s-1$ times before their least significant bit differs. This observation provides a means to partially extend the 0^{th} bit idea.

6.3.4 Extending the 0^{th} Bit Idea

As it was thoroughly detailed in Chapter 3, the discrete logarithm is the inverse of the discrete exponential function. Thus, it may have occurred to the observant reader that if the exponentiation by squaring technique developed in Section 3.2 can be used to calculate $\exp_g(r) := g^r \pmod{p}$, then perhaps the inverse of this technique could be used to solve the discrete logarithm.

Recall that calculating the principal square root of $h \equiv g^r \pmod{p}$ with r in $\mathbb{Z}_{p-1}$ is equivalent to shifting the exponent r down one bit by Corollary 6.1.4. Thus, repeatedly determining the 0^{th} bit by applying Procedure 6.3.1, converting the intermediate quadratic non-residues via Procedure 6.3.2, and calculating the principal square root, would theoretically unveil all of the bits of r. Of course the problem with this plan, as noted in Remark 6.1.4, is that while it is possible to find both square roots, it is not known which one is the principal one.

However, as a result of the analysis in Subsection 6.3.3, the first s square roots of h will always have the same quadratic character. Therefore this process will work for the first s iterations, regardless of which square root is picked. This leads to Procedure 6.3.4 which finds the first s bits of the discrete logarithm problem thus proving that the first s bits are easy.

Procedure 6.3.4 Determining the Easy Bits : easy_bits(p, g, h)

Require: p odd prime, g generator of $\mathbb{Z}_p^*$, and h element of $\mathbb{Z}_p^*$

Ensure: $r_{s-1}, \ldots, r_0$ are the s least significant bits of $r := \log_g(h)$

```
$s \leftarrow$ bit_border($p$)    /* Procedure 6.3.3 */
for $j \leftarrow 0$ to $s - 1$ do
    $r_j \leftarrow$ easy_bit($p, g, h$)    /* Procedure 6.3.1 */
    $h \leftarrow$ ensure_QR($p, g, h$)    /* Procedure 6.3.2 */
    $h \leftarrow \sqrt{h}$    /* Find either square root of $h$ modulo $p$ */
end for
```

6.4 The Hard Bits

The previous section showed that the first s bits of the discrete logarithm are easy where s is the bit border. The main result of this section is that, of the remaining bits, all bits but the most significant bit of the discrete logarithm are hard (as defined below). In fact, the most significant bit is also hard as proved in [BM86]. However, the proof technique developed in this section does not handle this case and in the interest of keeping the proof elegant, this one case is left out.

To begin, an informal definition of a hard bit is given to help the reader understand the necessity for the definitions of oracles and reductions given below. The basic idea is to assume that within in a certain cyclic group, the i^{th} bit discrete logarithm problem is, through some means, always known with 100% accuracy for all elements of the group. If this information can allow a polynomial time algorithm to solve the discrete logarithm problem of any element in the group, then this bit is said to be *hard*. That is, if assuming the i^{th} bit discrete logarithm problem is easy implies that the discrete logarithm problem is also easy, then the i^{th} bit must be hard since the discrete logarithm problem is hard.

Definition 6.4.1 (Oracle)**.** Given an odd prime p, a generator g of $\mathbb{Z}_p^*$, an element h of $\mathbb{Z}_p^*$, and a bit index i, it is common to refer to the ability to query the state of the i^{th} bit of $\log_g(h)$ as having an *oracle* on the i^{th} bit, denoted $\text{Oracle}_i(p, g, h)$.

Remark. Note that oracles are a theoretical construct and generally do not exists. Thus, to implement an oracle on the i^{th} bit for testing purposes, one of the methods from Chapter 5 is used to solve the discrete logarithm and then only the i^{th} bit is returned. Of course, it is necessary to choose the size of the odd prime p in such a way that the queries to the oracle can be done quickly. Therefore, when applicable, it is best to use the Index Calculus method of Section 5.6 since it is the fastest overall due to the pre-computation stage.

However, having an oracle for some fixed bit of all elements is only part of the solution, as some reduction method is still required to break a specific discrete logarithm problem into multiple varied discrete logarithm problems whose queries to the oracle can be recombined to recover the desired solution.

Definition 6.4.2 (Reduction)**.** The process of reducing a given problem into multiple varied problems whose solutions are then recombined to solve the original problem is called a *reduction*.

Now that the necessary background for oracles and reductions has been covered, the formal definition of a hard bit is stated.

Definition 6.4.3 (Hard Bit)**.** Given an odd prime p, a generator g of $\mathbb{Z}_p^*$, an element h of $\mathbb{Z}_p^*$, and a bit index i, if knowing $\text{Oracle}_i(\tilde{h})$ for all elements $\tilde{h}$ of $\mathbb{Z}_p^*$ allows a polynomial time algorithm to determine $\log_g(h)$, then i is called a *hard bit*.

Since all bits below the bit border are easy as shown in Section 6.3, the analysis of hard bits is restricted to the bits on or above the bit border, except the most significant bit as detailed in the beginning of this section. In the next few sections, two reduction procedures are covered which together allow an oracle on the i^{th} bit to completely determine any discrete logarithm problem in the group.

For the remainder of this section, always assume that p is an odd prime, that g is a generator of $\mathbb{Z}_p^*$, that h is an element of $\mathbb{Z}_p^*$, that s is the bit border of p, that $n := \lceil \log_2(p-1) \rceil$ is the maximum bit length for elements of $\mathbb{Z}_p^*$, that i is a fixed bit index in the range $s \leq i < n-1$ with an associated oracle, and that $r := \log_g(h)$ is the solution to the discrete logarithm problem.

6.4.1 A Useful Metaphor

It may aid the reader throughout the chapter to visualize the following metaphor for the discrete logarithm, the oracles, and the reductions. The fundamental approach of this chapter is to assume there exists an oracle on the i^{th} bit of the binary representation of the discrete logarithm $r := \log_g(h)$ and then attempt to use this information to reveal the other bits of r by using reductions. So, view the binary representation of r as a long string a paper consisting of n cells each containing either 0 or 1 which, while fixed, cannot be seen.

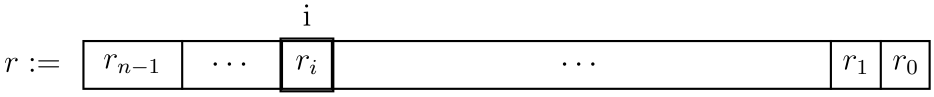

Figure 6.1: Metaphorical representation of $r := \log_g(h)$

Then, the oracle on the i^{th} bit acts as a fixed window into the i^{th} cell, exactly one cell wide, which allows this cell to be viewed. Now although the window cannot be moved, it is possible to reveal the cells above and below the window by sliding the paper to the right or to the left. This is achieved by manipulating the paper via the reductions.

6.4.2 Wrap-Around in the Exponent

Recall that for all prime p and for all elements a of $\mathbb{Z}_p^*$, Fermat's Little Theorem states that $a^{p-1} \equiv 1 \pmod{p}$ always holds. Now note that $r := \log_g(h)$ is an element of $\mathbb{Z}_{p-1}$ and let $\tilde{t} := r + c$ with c a positive integer. Then it is possible that $p - 1 \leq \tilde{t}$ in which case $\tilde{t} = (p-1) \cdot k + t$ for an integer $k > 0$ and an integer t in the range $0 \leq t < p - 1$. Thus,

$$g^{\tilde{t}} \equiv g^{(p-1)\cdot k+t} \equiv (g^{p-1})^k \cdot g^t \equiv 1^k \cdot g^t \equiv g^t \pmod{p}$$

by Fermat's Little Theorem since g is an element of $\mathbb{Z}_p^*$.

Definition 6.4.4 (Wrap-Around)**.** When the situation described above occurs, it is called a *wrap-around* in the exponent of $\tilde{t}$ to t.

Since p and r are fixed for a given problem instance, the likelihood of a wrap-around occurring depends exclusively on whether $p - 1 - r \leq c$. Unfortunately, r is the unknown solution to the discrete logarithm, so in practice, wrap-around is a rather troublesome issue to predict. Normally, this situation arises in the context of multiplying h by another element g^c which indirectly manipulates the exponents causing the wrap-around

$$h \cdot g^c \equiv g^r \cdot g^c \equiv g^{r+c} \equiv g^{\tilde{t}} \equiv g^t \pmod{p}$$

This can be a serious problem when a method depends on the inequality $r < r+c = \tilde{t}$ holding in the exponent. To make matters worse is the fact that it is not possible to detect when the wrap-around occurs since that would require solving the discrete logarithm. Even when possessing an oracle on the i^{th} bit, it is not always possible to detect whether a wrap-around occurred since it may occur it such a manner that the i^{th} bit does not in fact change its value.

Therefore, in the proofs of the reductions which follow in Subsection 6.4.3 and in Subsection 6.4.5, it is assumed the no wrap-arounds take place. The consequences of this assumption are addressed in Subsection 6.5.2.

6.4.3 The Right Reduction Technique

This subsection introduces a novel reduction technique inspired by the work in [O'C90] that takes advantage of the properties of binary addition to allow an oracle on the i^{th} bit to discover all bits below i. The technique is based on the notion of bit propagation which was proved in Lemma 2.3.1 and it proceeds as follows.

If the oracle reveals the i^{th} bit of r to be 1, then it can be cleared by subtracting 2^i from r, which is achieved by multiplying h by g^{-2^i}. Otherwise, if the i^{th} bit of r is 0, then leave h as it is. Now, let $j = i-1$ be a second bit index and note that r satisfies the conditions of the bit propagation lemma for the case where $i = j+1$. Thus, adding 2^j to r, call it $\tilde{r} := r + 2^j$, which is achieved by multiplying h by g^{2^j}, will cause the unknown j^{th} bit to propagate into the i^{th} bit where the oracle can reveal its value.

If the oracle reveals the i^{th} bit of $\tilde{r}$ to be 0, then the j^{th} bit of r is 0 and $\tilde{r}$ satisfies the conditions for the bit propagation lemma. Thus, the bit index j may be decremented and the process repeated using $\tilde{r}$. Otherwise, if the oracle reveals the i^{th} bit of $\tilde{r}$ to be 1, then the j^{th} bit of r is 1 and r already satisfies the conditions for the bit propagation lemma. Thus, the bit index j may be decremented and the process repeated using r.

Note that since the technique always clears the i^{th} bit before beginning, it is not possible for the j^{th} bit to propagate above the i^{th} bit. Hence, this process never alters the bits above the i^{th} bit. Therefore, through this careful manipulation of $h \equiv g^r \pmod{p}$, all of the bits below the bit index i may be determined, when given an oracle on the i^{th} bit, as demonstrated in Procedure 6.4.1.

Procedure 6.4.1 The Right Reduction : right_reduction(p, g, h, i)

```
Require: $p$ odd prime, $g$ generator of $\mathbb{Z}_p^*$, $h$ element of $\mathbb{Z}_p^*$, and $i$ bit index
Ensure: $r_i, \ldots, r_s$ are the bits of $r := \log_g(h)$ between the $s^{\text{th}}$ bit and the $i^{\text{th}}$ bit
$s \leftarrow$ bit_border$(p)$    /* Procedure 6.3.3 */
for $j \leftarrow i$ to $s$ by $-1$ do
    if Oracle$_i(p, g, h) = 1$ then    /* If the $j^{\text{th}}$ bit is 1 */
        $r_j \leftarrow 1$
        $h \leftarrow h \cdot g^{-2^j} \pmod{p}$    /* Reset $h$ to the right form */
    else
        $r_j \leftarrow 0$    /* Otherwise $h$ is the right form */
    end if
    $h \leftarrow h \cdot g^{2^{j-1}} \pmod{p}$    /* Apply Bit Propagation Lemma 2.3.1 */
end for
```

6.4.4 Distinguishing Square Roots

An important consequence of the right reduction technique developed in Subsection 6.4.3, is that possessing an oracle on the i^{th} bit always allows the s^{th} bit of $\log_g(h)$ to be determined. Next note, as explained in Subsection 6.3.3, that if h is a quadratic residue with principal square root x and non-principal square root y, then $\log_g(x)$ and $\log_g(y)$ first differ in their $(s-1)^{\text{th}}$ bit. These two observations are connected by the fact that the $(s-1)^{\text{th}}$ bit of $\log_g(x)$ is in fact the s^{th} bit of $\log_g(h)$ as per Corollary 6.1.4. Therefore, comparing the s^{th} bit of $\log_g(h)$ to the $(s-1)^{\text{th}}$ bit of both square roots of h allows the principal square root to be identified.

Furthermore, the $(s-1)^{\text{th}}$ bit of both $\log_g(x)$ and $\log_g(y)$ can be calculated without an oracle since it is an easy bit. Hence, distinguishing the principal square root can be accomplished in polynomial time, provided the s^{th} bit of

$\log_g(h)$ is known. This insight leads to Procedure 6.4.2.

Procedure 6.4.2 Distinguishing Square Roots : principal_root(p, g, h, b)

```
Require: $p$ odd prime, $g$ generator of $\mathbb{Z}_p^*$, $h$ quadratic residue in $\mathbb{Z}_p^*$, and $b$
    the $s^{\text{th}}$ bit of $\log_g(h)$ with $s$ the bit border
Ensure: $x$ is the principal square root of $h$
$s \leftarrow$ bit_border$(p)$    /* Procedure 6.3.3 */
$x \leftarrow \sqrt{h}$    /* Find either square root of $h$ modulo $p$ */
$r_{s-1}, \ldots, r_0 \leftarrow$ easy_bits$(p, g, x)$    /* Procedure 6.3.4 */
if $b \neq r_{s-1}$ then    /* If $x$ is the non-principal square root */
    $x \leftarrow p - x$    /* Swap $x$ for the principal square root */
end if
```

6.4.5 The Left Reduction Technique

It remains to be shown how an oracle on the i^{th} bit can determine the $(i+1)^{\text{th}}$ bit up to the $(n-1)^{\text{th}}$ bit. Recall that in Subsection 6.3.4, a method was developed to calculate the s least significant bits of $r := \log_g(h)$ by taking advantage of the relationship between principal square roots and right bit shifts as given in Corollary 6.1.4. The method was however restricted to the bits below the bit border since finding the bits on or above the bit border required a means to distinguish the square roots.

Luckily, as just proved in Subsection 6.4.4, such a means *does* exists when an oracle is present. Thus the same approach that worked for finding the easy bits below the bit border can, with slight modification, also find the bits above the bit border. Of course, in the process of using the right reduction technique to obtain the s^{th} bit of $\log_g(h)$, needed for selecting the principal square root, the hard bits between the s^{th} bit and the i^{th} bit are determined as well. Hence, the left reduction technique is only needed to reveal the bits above the i^{th} bit.

As this technique is based on Procedure 6.3.4, its form is almost identical. Each iteration of the procedure ensures x is a quadratic residue and then replaces x with its principal square root. This effectively performs a right bit shift of the exponent which moves the $(i+1)^{\text{th}}$ bit into the i^{th} bit where the oracle may be used to reveal its value. Thus, after j iterations, the s^{th} bit of $\log_g(x)$ is in fact the $(s+j)^{\text{th}}$ bit of $r := \log_g(h)$. Hence, the procedure requires previous knowledge of the s^{th} bit up to the i^{th} bit before starting in order distinguish the first s principal square roots, after which point, the process is self sustaining.

This analysis leads to Procedure 6.4.3 which, when given an oracle on the i^{th} bit, reveals all of the bits above i^{th} bit.

Procedure 6.4.3 The Left Reduction : left_reduction(p, g, h, i)

```
Require: $p$ odd prime, $g$ generator of $\mathbb{Z}_p^*$, $h$ element of $\mathbb{Z}_p^*$, and $i$ bit index
Ensure: $r_{n-1}, \ldots, r_{i+1}$ are the bits of $r := \log_g(h)$ above the $i^{\text{th}}$ bit
$s \leftarrow$ bit_border($p$)    /* Procedure 6.3.3 */
$n \leftarrow \lceil \log_2(p-1) \rceil$    /* Calculate the binary length of $p-1$ */
$x \leftarrow h$
$r_i, \ldots, r_s \leftarrow$ right_reduction($p, g, h, i$)    /* Procedure 6.4.1 */
for $j \leftarrow i+1$ to $n-1$ do
    $x \leftarrow$ ensure_QR($p, g, x$)    /* Procedure 6.3.2 */
    $x \leftarrow$ principal_root($p, g, x, r_{s+j-(i+1)}$)    /* Procedure 6.4.2 */
    $r_j \leftarrow \text{Oracle}_i(p, g, x)$
end for
```

6.5 Putting It All Together

Section 6.3 developed the concept of the bit border and then showed why all bits of the discrete logarithm below the bit border are easy. Section 6.4 developed two reduction techniques that could be used together with an oracle on a single bit index i to reveal all the other bits of the discrete logarithm above and below i. This showed that all bits of the discrete logarithm, other than the most significant bit, are hard. The next two subsections explain how to use these results to solve an arbitrary discrete logarithm problem given an oracle for almost any bit and offer fixes for problems not handled earlier.

6.5.1 Combining the Reductions

Many procedures were developed over the course of this chapter in preparation for the main result - a procedure to determine the discrete logarithm problem given an oracle on almost any bit index i. Notable among these were:

1. Procedure 6.3.3 was relevant in defining the border between the easy bits and the hard bits of the discrete logarithm.

2. Procedure 6.3.4 was created to quickly determine the easy bits of the discrete logarithm.

3. Procedure 6.4.1 was designed to reveal all of the bits of the discrete logarithm on or below i for a given bit index i with an associated oracle.

4. Procedure 6.4.3 was made to be used in conjunction with the previous procedure to unveil the bits of the discrete logarithm above i.

Together, these four procedures can be combined to determine all of the bits of the discrete logarithm as demonstrated in Procedure 6.5.1.

Procedure 6.5.1 Combining the Reductions : combine(p, g, h, i)

```
Require: $p$ odd prime, $g$ generator of $\mathbb{Z}_p^*$, $h$ element of $\mathbb{Z}_p^*$, and $i$ bit index
Ensure: $r = (r_{n-1} \ldots r_1 r_0)_2 = \log_g(h)$
$s \leftarrow$ bit_border$(p)$   /* Procedure 6.3.3 */
$n \leftarrow \lceil \log_2(p-1) \rceil$   /* Calculate the binary length of $p-1$ */
$r_{s-1}, \ldots, r_0 \leftarrow$ easy_bits$(p, g, h)$   /* Procedure 6.3.4 */
$r_i, \ldots, r_s \leftarrow$ right_reduction$(p, g, h, i)$   /* Procedure 6.4.1 */
$r_{n-1}, \ldots, r_{s+1} \leftarrow$ left_reduction$(p, g, h, i)$   /* Procedure 6.4.3 */
```

Note that it is possible to implement this procedure so that the number of queries to the oracle required to recover all of the bits of the discrete logarithm is *optimal*. In other words, for each bit on or above the bit border, the oracle only needs to be queried once to determine the bit. This is easy to see since the left reduction technique utilizes the right reduction technique, internally, exactly once and hence it can be modified to simply return all of the bits on or above the bit border.

6.5.2 Handling Wrap-Around

Despite appearing to be the main result, Procedure 6.5.1 does not always yield correct solutions to the discrete logarithm. This is due to fact that wrap-arounds, covered in Subsection 6.4.2, were incorrectly assumed not to occur during the reductions. Conveniently, the following lemmas provide a fix to this problem.

Lemma 6.5.1. *If* combine(p, g, h, i) *yields an incorrect solution to the discrete logarithm, then a wrap-around must have occurred.*

Proof. The only assumption made throughout the entire chapter was that wrap-arounds did not occur. Therefore, if any results are incorrect, the error stems from this single, faulty assumption. □

Lemma 6.5.2. *If a wrap-around occurred, then it must have occurred in* right_reduction(p, g, h, i)*.*

Proof. Note that proofs which manipulated the exponent by taking principal square roots, could not have caused a wrap-around since they divide the exponent by 2 making it always smaller. Furthermore, the proof for converting quadratic non-residues only subtracts 1 from the exponent when it is guaranteed to be possible and hence it could not have caused a wrap-around. Therefore, the only proof remaining which manipulated the exponents was for the right reduction technique which does in fact explicitly add powers of two to the exponent. □

Lemma 6.5.3. *If* right_reduction(p, g, h, i) *causes a wrap-around, then the* $(n-1)^{\text{th}}$ *bit of* $r := \log_g(h)$ *must have been* 1*, where* $n := \lceil \log_2(p-1) \rceil$*.*

Proof.

Let l be the largest integer such that $2^l < p - 1$. Now, assume for a contradiction that there was a wrap-around and that the $(n-1)^{\text{th}}$ bit of r was 0. Note that regardless of r, the right reduction technique can never alter the $(n-1)^{\text{th}}$ bit. This is because the oracle is restricted to never be on the $(n-1)^{\text{th}}$ bit and, as explained in Subsection 6.4.3, the technique does not alter the bits above the oracle. Thus, no matter how the technique manipulates r, the $(n-1)^{\text{th}}$ bit of r remains 0 and hence r always remains less than $2^l < p - 1$. Therefore, right_reduction(p, g, h, i) could not have caused r to wrap-around. □

The proof of Lemma 6.5.3 also explains why the right reduction technique can never work with an oracle on the most significant bit.

Corollary. *The procedure* right_reduction(p, g, h, i) *causes a wrap-around for less than half of the elements h in $\mathbb{Z}_p^*$.*

Proof. Note that $\frac{p-1}{2} \leq 2^l < p - 1$ otherwise $2^{l+1} < p - 1$ which is a contradiction to the maximality of l. Furthermore, note that if $r < 2^l$, then the $(n-1)^{\text{th}}$ bit of the exponent r is 0. Together, these imply that the $(n-1)^{\text{th}}$ bit is 0 for more than half of the exponents in $\mathbb{Z}_{p-1}$. Now since the discrete logarithm is bijective, each element h in $\mathbb{Z}_p^*$ corresponds to precisely one exponent r in $\mathbb{Z}_{p-1}$. Thus, by Lemma 6.5.3, for more than half of the elements h in $\mathbb{Z}_p^*$, right_reduction(p, g, h, i) does not cause a wrap-around. □

Finally, these lemmas can be combined into the following theorem.

Theorem 6.5.4. *If* combine(p, g, h, i) *yields an incorrect solution to the discrete logarithm, then the $(n-1)^{\text{th}}$ bit of $r := \log_g(h)$ is 1.*

Proof. This theorem follows directly from the previous three lemmas. □

Corollary. *If the $(n-1)^{\text{th}}$ bit of $r := \log_g(h)$ is 0, then* combine(p, g, h, i) *yields a correct solution to the discrete logarithm.*

These last two results provide instructions on how to fix the problems caused by the erroneous wrap-around assumption. First, use combine(p, g, h, i) to find $r := \log_g(h)$. Utilizing the exponentiation by squaring method, it is possible to quickly verify whether or not r is indeed the correct solution by checking that $g^r \equiv h \pmod{p}$. If it is correct, then simply return r.

Otherwise, if the result is incorrect, then the $(n-1)^{\text{th}}$ bit of r must be 1 by Theorem 6.5.4. Thus, clear this bit via $\tilde{h} := h \cdot g^{-2^{n-1}}$. Then find $\tilde{r} := \log_g(\tilde{h})$ using combine$(p, g, \tilde{h}, i)$ which is guaranteed to be correct by Corollary 6.5.2. Last, set the $(n-1)^{\text{th}}$ bit of $\tilde{r}$ back to 1 via $r := \tilde{r} + 2^{n-1}$ and return r. This idea is implemented in Procedure 6.5.2.

Procedure 6.5.2 Determining the Discrete Logarithm : dlog(p, g, h, i)

```
Require: $p$ odd prime, $g$ generator of $\mathbb{Z}_p^*$, $h$ element of $\mathbb{Z}_p^*$, and $i$ bit index
Ensure: $r = (r_{n-1} \dots r_1 r_0)_2 = \log_g(h)$
$n \leftarrow \lceil \log_2(p-1) \rceil$   /* Calculate the binary length of $p-1$ */
$r \leftarrow \text{combine}(p, g, h, i)$   /* Procedure 6.5.1 */
if $h \not\equiv g^r \pmod{p}$ then   /* $r$ is incorrect */
    $h \leftarrow h \cdot g^{-2^{n-1}}$   /* Clear the $(n-1)^{\text{th}}$ bit to prevent wrap around */
    $r \leftarrow \text{combine}(p, g, h, i)$   /* Procedure 6.5.1 */
    $r_{n-1} \leftarrow 1$   /* Set the $(n-1)^{\text{th}}$ bit back to 1 */
end if
```

The reader may wish to peruse the Source Code Listing in Appendix A for implementations of the proofs presented throughout the chapter in order to gain more appreciation for why and how they work.

Chapter 7

Conclusions

The intent of this paper was to provide a thorough investigation into the discrete logarithm problem, its properties, its applications, methods for solving it, and most of all, its individual bit security. In particular, the focus was on producing the most efficient reduction techniques possible in Section 6.4 based on the assumption of a perfect oracle. This research culminated at the end of Chapter 6 with an elegant and extremely efficient proof that, in $\mathbb{Z}_p^*$, all bits of the discrete logarithm on or above the bit border are individually hard, with the exception of the most significant bit.

Under the assumption that the oracles were perfect, the right reduction and left reduction techniques were shown to be optimal in the sense that the number of queries to the oracle was exactly equal to the number of bits revealed. However, due to the issue of wrap-around in the exponent which occurs for less than half the elements, it was sometimes necessary to perform the reductions a second time. This implies that the approach used to determine the discrete logarithm requires, on average, at most 1.5 times as many oracle queries as the number of bits revealed.

Although perfect oracles are an invaluable tool for the analysis of theoretical computations, they are often unavailable for many problems in practice. Thus imperfect oracles, which only answer correctly for over half of the inputs, can be seen as a closer approximation to the reality of the situation. Many papers such as [LW83] and [Per85] first present procedures for determining the discrete logarithm using perfect oracles and later adapt their procedures to use imperfect oracles. These methods tend to take a majority vote of the oracle across multiple queries with known relationships that can be exploited in polynomial time to ensure a correct response with a negligible probability of error.

On the other hand, the proofs developed in these papers are not concerned with the efficiency of their implementations and hence are organized around different approaches which make them somewhat difficult to adapt completely. However, the reductions given in this paper can utilize a similar randomized method to that of [Per85], but restricted to oracles on the s^{th} bit up to the $(s+l)^{\text{th}}$ bit where s is the bit border of p and $l = c \cdot \log_2 \log_2 (p-1)$ for a constant $c > 0$. The trade-off between the accuracy of the oracles and the efficiency of the corresponding reduction techniques would be a topic that merits closer attention in the future.

Appendix A

Source Code Listing

The source code presented in this listing is comprised of the implementations for the results of Section 6.3, of Section 6.4, and of Section 6.5. It is written in the programming language Aribas from Prof. Dr. Otto Forster which can be found on the internet at:

`http://www.mathematik.uni-muenchen.de/~forster/sw/aribas.html`

Included with this thesis is a CD-ROM containing the files necessary to run Aribas as well as this source code listing and the additional files needed by it.

```
(****************************************************************
*
*     Filename:  dlog_using_oracle.ari
*       Author:  Richard McKnight, rmcknigh@cs.hmc.edu
*
*      Created:  Wed  7 Mar 2007 19:13:35 CET
*      Revised:  Fri 23 Mar 2007 18:20:49 CET
*
*  Description:  Implements the proofs presented in my Masters
*                Thesis:
*
*         Individual Bit Security of the Discrete Logarithm:
*              Theory and Implementation Using Oracles
*
*    Copyright:  This code released under the GNU GPL.
*
****************************************************************)

(*
**  Load the necessary functions for solving
**  the discrete logarithm via index calculus.
*)
load("index");

(*
**  The following assumptions are made for all functions:
**  -  p is an odd prime number
**  -  g is a generator of (Z/p)*
**  -  h is an integer in the range 0 < h < p
*)

(*
**  Calculates the border between the easy bits
**  and the hard bits.
*)
function bit_border(p: integer): integer;
var
    s, n: integer;
begin
    s := 0;
    n := p-1;

    while even(n) do
        n := bit_shift(n, -1);
        inc(s);
    end;

    return s;
end;

(*
**  Calculates all easy bits of the discrete logarithm.
*)
function easy_bits(p, g, h: integer): integer;
```

```
var
    s, g_inv, r, j: integer;
begin
    s := bit_border(p);
    g_inv := mod_inverse(g, p);
    r := 0;

    for j := 0 to s-1 do
        if jacobi(h, p) = -1 then
            h := h*g_inv mod p;
            r := bit_set(r, j);
        end;

        h := gfp_sqrt(p, h);
    end;

    return r;
end;

(*
**  Checks that the given bit index is valid;
**  i.e. Above the border of the easy bits,
**  but not the most significant bit.
*)
function in_range(p, i: integer): boolean;
var
    s, n: integer;
begin
    s := bit_border(p);
    n := bit_length(p-1);

    if s <= i and i < n then
        return true;
    end;

    writeln();
    write("Bit index i=", i, " not in range ");
    writeln("[", s, ", ", n-1, "]");
    return false;
end;

(*
**  Implements an i^{th} Bit Oracle for
**  the discrete logarithm.
*)
function oracle(p, g, h, i: integer): integer;
begin
    if in_range(p, i) then
        return bit_test(dlog(p, g, h), i);
    end;
end;

(*
**  Uses the i^{th} Bit Oracle to find all hard bits
**  less than or equal to i.
*)
function right_reduction(p, g, h, i: integer): integer;
var
    s, g_inv, r, j: integer;
begin
    if not in_range(p, i) then
        return;
    end;

    s := bit_border(p);
    g_inv := mod_inverse(g, p);
    r := easy_bits(p, g, h);

    for j := i to s by -1 do
        if oracle(p, g, h, i) = 1 then
            h := h*(g_inv**(2**j mod (p-1)) mod p) mod p;
            r := bit_set(r, j);
        end;

        h := h*(g**((2**(j-1)) mod (p-1)) mod p) mod p;
    end;
    return r;
end;

(*
**  Uses right_reduction() and easy_bits_msb() to find
**  all hard bits greater than i.  Assumes 0 < h < p.
*)
function left_reduction(p, g, h, i: integer): integer;
var
    s, n, g_inv, r, j: integer;
begin
    s := bit_border(p);
    n := bit_length(p-1);
    g_inv := mod_inverse(g, p);
    r := right_reduction(p, g, h, i);

    for j := i+1 to n-1 do
        if jacobi(h, p) = -1 then
            h := h*g_inv mod p
        end;

        h := gfp_sqrt(p, h);

        if bit_test(r, s+j-(i+1)) /=
           bit_test(easy_bits(p, g, h), s-1) then
            h := p-h;
        end;

        if oracle(p, g, h, i) = 1 then
            r := bit_set(r, j);
        end;
    end;

    return r;
end;

(*
**  Uses left_reduction() to find the discrete logarithm.
*)
function dlog_using_oracle(p, g, h, i: integer): boolean;
var
    r, n, g_inv: integer;
begin
    if not in_range(p, i) then
       return;
    end;

    r := left_reduction(p, g, h, i);

    if g**(r mod (p-1)) mod p = h then
        return r;
    else
        n := bit_length(p-1);
        g_inv := mod_inverse(g, p);
        h := h*(g_inv**(2**(n-1) mod (p-1)) mod p) mod p;

        r := left_reduction(p, g, h, i);

        r := bit_set(r, n-1);
    end;
end;
```

References

[BM86] M. Blum and S. Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM Journal on Computing*, 13:850–864, 1986.

[Bon98] Dan Boneh. The decision diffie-hellman problem. In *Lecture Notes in Computer Science*, volume 1423, pages 48–63. Third Algorithmic Number Theory Symposium, Springer-Verlag, 1998.

[Bur06] U.S. Census Bureau. 2004 e-commerce multi-sector report, 2006. `http://www.census.gov/estats/`.

[Coh96] Henri Cohen. *A Course in Computational Algebraic Number Theory.* Springer-Verlag, 1996. ISBN 0-387-55640-0.

[DH76] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, 1976.

[ElG85] T. ElGamal. A public-key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31:469–472, 1985.

[For96] Otto Forster. *Algorithmische Zahlentheorie*. Vieweg-Verlag, 1996. ISBN 3-528-06580-X.

[Kob87] Neal Koblitz. *A Course in Number Theory and Cryptography.* Number 114 in Graduate Texts in Mathematics. Springer-Verlag, 1987.

[LW83] Douglas L. Long and Avi Wigderson. How discreet is the discrete log? In *STOC '83: Proceedings of the fifteenth annual ACM* *symposium on Theory of computing*, pages 413–420, New York, NY, USA, 1983. ACM Press.

[MvOV96] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996.

[Nid90] H. Niderreiter. A short proof for explicit formulas for the discrete logarithms in finite fields. *Applicable Algebra in Eng., Comm., and Comp.*, 1:55–57, 1990.

[O'C90] Luke O'Connor. Every bit of the discrete logarithm is either hard or easy. In *5th Annual University at Buffalo Graduate Conference*, pages 23–30, 1990.

[Odl00] Andrew M. Odlyzko. Discrete logarithms: The past and the future. *Des. Codes Cryptography*, 19(2/3):129–145, 2000.

[Per85] René Peralta. Simultaneous security of bits in the discrete log. In *EUROCRYPT*, pages 62–72, 1985.

[PH78] S. C. Pohlig and M. E. Hellman. An improved algorithm for computing logarithms over GF(p) and its cryptographic significance. *IEEE Trans. Inform. Theory*, 24(1):106–110, 1978.

[Pol78] J. Pollard. Monte carlo methods for index computation (mod p). *Mathematics of Computation*, 32(143):918–924, 1978.

[Sha49] C. E. Shannon. Communication theory of secrecy systems. *Bell Sys. Tech. J.*, 28:657–715, 1949.

[Sha71] D. Shanks. Class number, a theory of factorization and genera. *Proc. Symp. Pure Math.*, 20:415–440, 1971.

[Sti02] Douglas R. Stinson. *Cryptography: Theory and Practice.* CRC Press, 2nd edition, 2002.