2010-03

An analysis of algorithms for solving discrete logarithms in fixed groups

Mihalcik, Joseph P.

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

AN ANALYSIS OF ALGORITHMS FOR SOLVING DISCRETE LOGARITHMS IN FIXED GROUPS

by

Joseph Mihalcik

March 2010

Thesis Advisor:	Dennis Volpano
Second Reader:	Harold Fredricksen

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 2010	Master's Thesis

4. TITLE AND SUBTITLE An Analysis of Algorithms for Solving Discrete Logarithms in Fixed Groups	5. FUNDING NUMBERS
6. AUTHOR(S) Joseph Mihalcik	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) Department of Defense	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT	12b. DISTRIBUTION CODE
Approved for public release; distribution is unlimited	

13. ABSTRACT (maximum 200 words)

Internet protocols such as Secure Shell and Internet Protocol Security rely on the assumption that finding discrete logarithms is hard. The protocols specify fixed groups for Diffie-Hellman key exchange that must be supported. Although the protocols allow flexibility in the choice of group, it is highly likely that the specific groups required by the standards will be used in most cases. There are security implications to using a fixed group, because solving any discrete logarithm within a group is comparatively easier after a group-specific precomputation has been completed. In this work, we more accurately model real-world cryptographic applications with fixed groups. We use an analysis of algorithms to place an upper bound on the complexity of solving discrete logarithms given a group-specific precomputation.

14. SUBJECT TERMS Discrete Logarithms, Analysis of Algorithms, Advice Strings, Diffie-Hellman Key Exchange	15. NUMBER OF PAGES 71
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

AN ANALYSIS OF ALGORITHMS FOR SOLVING DISCRETE LOGARITHMS IN FIXED GROUPS

Joseph P. Mihalcik
Civilian, Department of Defense
B.S., University of Maryland, College Park, 2000

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2010

Author: Joseph P. Mihalcik

Approved by: Dennis Volpano
Thesis Advisor

Harold Fredricksen
Second Reader

Peter Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Internet protocols such as Secure Shell and Internet Protocol Security rely on the assumption that finding discrete logarithms is hard. The protocols specify fixed groups for Diffie-Hellman key exchange that must be supported. Although the protocols allow flexibility in the choice of group, it is highly likely that the specific groups required by the standards will be used in most cases. There are security implications to using a fixed group, because solving any discrete logarithm within a group is comparatively easier after a group-specific precomputation has been completed. In this work, we more accurately model real-world cryptographic applications with fixed groups. We use an analysis of algorithms to place an upper bound on the complexity of solving discrete logarithms given a group-specific precomputation.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ALGORITHMS AND DEFINITIONS

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgements

Thanks to Jonathan Herzog, Harold Fredricksen, and Dennis Volpano for stimulating, informative discussions and helpful suggestions.

Professor Herzog was the actual primary advisor for this thesis. Unfortunately, I finished my thesis after Professor Herzog left the Naval Postgraduate School or his name would officially appear as advisor. His continued help in completing the thesis after his professional obligation ended is greatly appreciated.

I especially want to thank my wife, Alexandria, and my daughter, Kaitlyn. I never would have been able to finish this without their constant love and support.

THIS PAGE INTENTIONALLY LEFT BLANK

I. Introduction

Thirty years ago, the field of cryptography was revolutionized when Whitfield Diffie and Martin Hellman published *New Directions in Cryptography* [3]. In this seminal paper, they introduced the idea of *public key cryptography*, a concept that now provides the foundation for secure communications and secure financial transactions over the Internet. In the same paper, they also described a method for exchanging secret keys over an insecure network. Now known as the Diffie-Hellman *key exchange*, this method is used within common network security protocols including Secure Shell (SSH) [28] and Internet Protocol Security (IPsec) [12].

The Diffie-Hellman key exchange is an application of group theory. Computing the secret key requires *modular exponentiation*: raising a number to an exponent within a group of integers modulo a prime number. The inverse operation of modular exponentiation is called finding the *discrete logarithm*. Exponentiation is computationally easy, while finding discrete logarithms is believed to be hard. The key exchange depends on this asymmetry in computational complexity for its security. If an adversary can compute discrete logarithms, the adversary can break Diffie-Hellman and recover the secret key. This situation has lead to a vast amount of research toward finding efficient algorithms to solve discrete logarithms and also towards understanding the computational complexity of the discrete logarithm problem.

Algorithms solving discrete logarithms generally can be divided into two phases: a precomputation phase and a search phase. The precomputation phase is run first and the result is stored in memory. The stored result is used in the search phase to speed up computation of the discrete logarithm. Often, the precomputation algorithm requires only the group description. This means that the first phase is independent of any particular instance of a discrete logarithm. Additional discrete logarithms over the same group can be solved by running just the search phase.

Our work focuses on the efficiency of solving multiple discrete logarithms over the same group. The practical importance of this investigation can be seen when we examine how the Diffie-Hellman key exchange is used in real applications, such as the SSH and IPsec security

protocols. Within these protocols, a small number of standard groups are defined. For example, the standard for SSH only defines two groups that must be supported. There are valid reasons to use standard groups. In particular, when two users exchange keys, using a standard group relieves one user from the computational burden of creating a secure group and the other user from the need to trust that it has been done securely. Choosing a secure group requires avoiding certain groups with characteristics that make them easier to solve. Leaving group choice to a standards committee saves the user significant computation time, but the result will be many key exchanges occurring over the same fixed groups. This provides an advantage to the attacker in that the cost of precomputation for a group can now be amortized over many key exchanges. As more exchanges occur under a group, the group precomputation increases in value to an attacker. Therefore, our analysis must take into account an attacker that can dedicate large parallel systems to the precomputation.

Typically, a security analysis of discrete logarithm cryptography would consider the complexity of the discrete logarithm problem (DLP). However, the DLP is an incomplete model for cryptographic applications with fixed groups. In these applications, the group is constant, but the DLP treats the group as a variable input to the problem. In the DLP, the problem is to find a single discrete logarithm in a given group, however, a precomputation provides no benefit when solving only one instance. Group-specific precomputation is most valuable when the group is reused often, which is the case for standards that specify fixed groups. Current security proofs based on the DLP do not account for group-specific precomputation and, therefore, underestimate the difficulty of attacking applications that specify fixed groups.

In this work, we present a more conservative security model for fixed groups that shows that such real-world applications provide less cryptographic strength than previously acknowledged. In particular, we introduce the *para-discrete logarithm problem* (PDLP), a variant of the DLP where the group is not an input, but rather dependent only on the input size. This allows us to model the result of a group-specific precomputation as an advice string. In complexity theory, an *advice string* is roughly a piece of data provided to a help solve a computational problem, and the data can be dependent on the size of the input, but not on the input itself. In the standard DLP, the precomputation is not an advice string, because it is based on an input: the group. Once the precomputation has been completed for a standard group, the DLP is reduced to our PDLP with an advice string.

We use an analysis of algorithms to place an upper bound on the complexity of the para-discrete logarithm problem with an advice string. In particular, we provide an analysis of the common algorithms for solving discrete logarithms, focusing on the relationship between the

asymptotic running times of the two phases and the asymptotic bit-length of the advice string. Given a group of order N, we show that the generalized para-discrete logarithm problem can be solved in $O(N^{1/3})$ group operations with an advice string of size $O(N^{1/3})$. The precomputation of such an advice string requires $O(N^{2/3})$ group operations.

The rest of the work is as follows. In the next chapter, we review both the technical background and the prior research in the field of cryptography that is relevant to understanding our work. In Chapter III, we survey the known algorithms for the discrete logarithm problem and perform a traditional analysis of their complexity. In Chapter IV, we consider the complexity of discrete logarithms over fixed groups and reanalyze the discrete logarithm algorithms in that context.

THIS PAGE INTENTIONALLY LEFT BLANK

II. Background

In this chapter, we review both the technical background and the prior research in the field of cryptography that is relevant to understanding our work. In particular, we begin by describing discrete logarithms. Next, we examine their importance in cryptology. Lastly, we look at the use of fixed groups in cryptographic protocols.

A. Discrete Logarithms Explained

In this section, we describe discrete logarithms. In particular, we first relate discrete logarithms to standard logarithms in real numbers. Then, we provide mathematical definitions for group exponentiation and discrete logarithms. Next, we provide a simple concrete example of discrete logarithms. Lastly, we first define the standard computational problems regarding discrete logarithms; that is, the discrete logarithm problem (DLP) and the generalized discrete logarithm problem (GDLP). Throughout, we assume the reader is familiar with the concept of groups from abstract algebra.

Discrete logarithms are so named because they are analogous to standard logarithms with real numbers. Just as the logarithm is the inverse operation of exponentiation, the discrete logarithm is the inverse operation of group exponentiation. In the real numbers, $\log_g a = x$ if $g^x = a$. The same is true for discrete logarithms, except g and a are elements of a multiplicative cyclic group, G, with generator g. A *cyclic group* is a group where all the elements of the group can be generated by raising one element, a *generator*, to successive powers.

Group exponentiation to a power $x \in \mathbb{N}$, can be defined as repeated group multiplication,

$$g^x = \prod_{1}^{x} g$$

Methods such as repeated-squaring [16, Algorithm 2.143] allow group exponentiation to be done efficiently, with just $\lg x$ multiplications. In the group $\mathbb{Z}_n^*$, where the group operation is

multiplication modulo an integer, n, group exponentiation is called modular exponentiation. In this setting, the value of g^x is a if and only if $g^x \equiv a \bmod n$. We can compute g^x by raising g to the power x in the integers, then finding the remainder modulo n. (There are more practical algorithms as well [8].)

Finding a discrete logarithm means inverting the exponentiation and finding the exponent x given the value, a. That is, given g, n, and a, find a value of x, $0 \leq x < n-1$, such that $g^x \equiv a \bmod n$. While efficient algorithms exist for group exponentiation, no efficient algorithm is known for computing discrete logarithms. This asymmetry is what makes exponentiation useful in public key cryptography.

1. Discrete Logarithm Example

To further clarify, we will use a concrete example in $\mathbb{Z}_p^*$. The group $\mathbb{Z}_p^*$ is the multiplicative group of integers modulo a prime, p. The elements of $\mathbb{Z}_p^*$ are the integers $1, 2, \ldots, p-1$. In this example, a is an element of the group that can be represented as $a = g^x$, where x is an integer, $0 \leq x < p-1$. The discrete logarithm of a to the base g, can be written as $\log_g a = \log_g g^x = x$. For this example, if we let $p = 11$ and $g = 2$, Table 1 shows the powers of g. If we look in the table at the row $x = 4$ we see $a = g^x = 2^4 = 16 \equiv 5 \bmod 11$. All ten elements of $\mathbb{Z}_p^*$ are generated before we see another 1 in the table. For every $g \in \mathbb{Z}_p^*$, $g^{p-1} \equiv 1 \bmod p$, and, if g is a generator, then there is no element $0 \leq x < p-1$ such that $g^x \equiv 1 \bmod p$. There is no $x < 10$ such that $2^x \equiv 1 \bmod 11$, so 2 is a generator of $\mathbb{Z}_{11}^*$. Also note that the values for greater exponents repeat, $2^0 = 2^{10} = 1 \bmod 11$.

When we invert this table we have the discrete logarithms in $\mathbb{Z}_{11}^*$. Table 2 shows us the discrete logarithms. For example, looking in the table at $a = 5$ we find $\log_2 5 = 4$.

2. Discrete Logarithm Problem

Before we can analyze the security of cryptography, it is helpful to formally define the computational problems upon which that security relies. The DLP is the problem of solving discrete logarithms over the group of integers modulo a prime and can be formalized as follows [16],

Definition 1 The Discrete Logarithm Problem (DLP)

Input: Prime: p, Generator of $\mathbb{Z}_p^*$: g, Element of $\mathbb{Z}_p^*$: a

Output: Exponent: x satisfying $g^x \equiv a \mod p$, where $0 \leq x < p-1$.

x	$2^x \equiv a \bmod 11$	
0	2^0	1
1	2^1	2
2	2^2	4
3	2^3	8
4	2^4	5
5	2^5	10
6	2^6	9
7	2^7	7
8	2^8	3
9	2^9	6
10	2^{10}	1
11	2^{11}	2
⋮	⋮	⋮

Table 1: Powers of $g = 2$ in $\mathbb{Z}_{11}^*$

Discrete logarithms can be defined over any cyclic group; they need not be restricted to $\mathbb{Z}_p^*$. Therefore, the discrete logarithm problem can be generalized to apply to any cyclic group [16].

Definition 2 The Generalized Discrete Logarithm Problem (GDLP)

Input: Cyclic Group: G, Generator of G: g, Element of G : a

Output: Exponent: x satisfying $g^x = a$, where $0 \leq x < |G|$.

B. Cryptography and Discrete Logarithms

The difficulty of solving discrete logarithms relative to exponentiation makes them very useful in cryptographic applications. The security of many common cryptographic applications depends on the assumption that solving discrete logarithms is infeasible. The first published cryptographic use of discrete logarithms was in the Diffie-Hellman key agreement protocol [3].

a	$\log_g a = x$	
1	$\log_2 1$	0
2	$\log_2 2$	1
3	$\log_2 3$	8
4	$\log_2 4$	2
5	$\log_2 5$	4
6	$\log_2 6$	9
7	$\log_2 7$	7
8	$\log_2 8$	3
9	$\log_2 9$	6
10	$\log_2 10$	5

Table 2: Discrete Logarithms to the Base $g = 2$ in $\mathbb{Z}_{11}^*$

The first public key cryptosystem relying on discrete logarithms was the ElGamal cryptosystem [4]. ElGamal also developed the first signature scheme based on discrete logarithms, a variant of which is the Digital Signature Algorithm (DSA) [19].

In this section, we present several cryptographic algorithms to demonstrate the practical importance of discrete logarithms. In particular, we first examine the Diffie-Hellman key agreement scheme. Next, we focus on the public key encryption system known as ElGamal. Finally, we examine the Digital Signature Algorithm (DSA).

1. Diffie-Hellman Key Agreement

The Diffie-Hellman key agreement enables two parties to agree on a secret key over an insecure channel without revealing the key to an attacker. In this scheme, two participants, A and B, agree on a cyclic group, G, and generator of the group, g. We must assume the attacker will know the details of the group, as they will be sent over the same insecure channel. A and B independently and randomly choose their own secret exponents, a and b, respectively. User A computes and transmits g^a; B computes and transmits g^b. The secret key they agree on is g^{ab}.

User A computes the secret key by raising g^b (received from B) to the power, a (A's secret),

$$(g^b)^a = g^{ba} = g^{ab}$$

Equivalently, user B raises the value g^a to the power, b,

$$(g^a)^b = g^{ab}$$

Now both users know the secret key, g^{ab}, while the attacker has only seen g^a and g^b. Clearly, however, if the attacker could compute discrete logarithms in the group, G, then the attacker could solve for either a or b and compute the secret key. It is an open question whether there is an easier way to find g^{ab} than to compute discrete logarithms. This is called the *Diffie-Hellman problem*.

It should also be noted that the Diffie-Hellman key agreement does not provide authentication. An attacker with the ability to modify and insert messages could be in the middle of an exchange between users A and B. If this occurs, A and B could unknowingly be sharing keys with the attacker and not each other. To avoid this attack, Diffie-Hellman must be part of a larger protocol that provides authentication.

2. ElGamal

The ElGamal cryptosystem is a method of public key encryption (PKE) that is based on Diffie-Hellman [4]. In this subsection, we show that the security of ElGamal is dependent on the difficulty of finding discrete logarithms. In particular, we begin with a formal definition of PKE. Then we explain what it means for a PKE to be secure. Next we describe the ElGamal algorithms. Finally, we demonstrate how the security of ElGamal would be compromised if an efficient discrete logarithm method is discovered.

Definition 3 Public Key Encryption (PKE)

A *public key encryption* system [6] is a triple of PPT algorithms (G, E, D) such that,

1. G is a *key generation algorithm* that on input 1^k computes output (e, d).
2. E is an *encryption algorithm* that on input $(1^k, e, m)$ computes output c.
3. D is a *decryption algorithm* that on input $(1^k, d, c)$ computes output m.

where 1^k is the security parameter, e is the public encryption key, d is the secret decryption key, $m \in \{0,1\}^k$ is the plaintext message and $c \in \{0,1\}^*$ is the encrypted ciphertext such that if $G \rightarrow (e, d)$ then $D(E(e, m), d) = m$.

The security of a PKE system has been defined in terms of *semantic security* [6]. Informally, a PKE system is semantically secure if an adversary with access to the encryption key, e, and ciphertext, c, has no more than a negligible advantage in guessing the plaintext over an adversary without acesss to e or c.

Algorithm 4 ElGamal Key Generation

Input: Security parameter: 1^k
Output: Public encryption key: e, Secret decryption key: d
$p \Leftarrow k$-bit prime such that $p - 1$ has a large prime factor
$g \Leftarrow$ generator of $\mathbb{Z}_p^*$
$a \Leftarrow$ randomly selected exponent between 0 and $p - 1$
$d \Leftarrow (p, g, a)$
$e \Leftarrow (p, g, g^a)$
return (e, d)

In ElGamal key generation, a user A selects a prime, p, that defines the multiplicative group, $\mathbb{Z}_p^*$, a generator of that group, g, and a secret exponent a. User A also computes g^a. (In this, all arithmetic is mod p.) A's private key, d is (p, g, a), and A's public key, e, is (p, g, g^a). As ElGamal initially presented the scheme, the prime and generator be fixed for all users, and the public key would be only (g^a). He acknowledged that having each user select a prime "is preferable from the security point of view although that will triple the size of the public file." [4]

To encrypt a message for A, user B must represent his message as m, an element of $\mathbb{Z}_p^*$. User B must choose a random exponent b and compute $c_1 = g^b$ and $c_2 = (g^a)^b m = g^{ab} m$. B sends the encrypted message, (c_1, c_2), to A. To decrypt, A uses the private key, a, to compute

Algorithm 5 ElGamal Encryption

Input: Public encryption key: $e = (p, g, g^a)$, Plaintext message: m where $0 \leq m \leq p-1$
Output: Encrypted ciphertext: c

$b \Leftarrow$ randomly selected exponent between 0 and $p-1$
$c_1 \Leftarrow g^b \mod p$
$c_2 \Leftarrow (g^a)^b m \equiv g^{ab} m \mod p$
$c \Leftarrow (c_1, c_2)$
return c

Algorithm 6 ElGamal Decryption

Input: Private decryption key: $d = (p, g, a)$, Encrypted ciphertext: $c = (c_1, c_2) = (g^b, g^{ab} m)$
Output: Decrypted plaintext: m

$g^{ab} \Leftarrow (c_1)^a \equiv (g^b)^a \mod p$
$(g^{ab})^{-1} \Leftarrow$ inverse of g^{ab} using extended Euclidean algorithm
$m \Leftarrow (g^{ab})^{-1} c_2 \equiv (g^{ab})^{-1} g^{ab} m \mod p$
return m

c_2 / c_1^a. Because inverses can be efficiently computed mod p, A can quickly find m.

$$\frac{c_2}{c_1^a} = \frac{g^{ab} m}{(g^b)^a} = \frac{g^{ab} m}{g^{ab}} = m$$

As with Diffie-Hellman, ElGamal would be insecure if discrete logarithms could be solved efficiently. An adversary with the ability to find discrete logarithms in $\mathbb{Z}_p^*$ could recover the private key, a, from the public key, g^a. The adversary could then decrypt messages just as the valid user can.

3. Digital Signature Algorithm

ElGamal also proposed a method for digital signatures in his 1984 paper, *A public key cryptosystem and a signature scheme based on discrete logarithms* [4]. A variation of that method, the Digital Signature Algorithm (DSA), was adopted in 1994 as the Digital Signature Standard (DSS) [19] and is in common use. In this subsection, we show that the security of DSA is dependent on the difficulty of finding discrete logarithms. In particular, we begin with a formal definition of a digital signature system. Then we define security for a signature scheme. Next, we describe DSA. Finally, we demonstrate how the security of DSA would be compromised if an efficient discrete logarithm method is discovered.

Definition 7 Digital Signature System

A *digital signature* system is a triple of PPT algorithms (G, S, V) such that,

1. G is a *key generation algorithm* that on input 1^k computes output (e, d).
2. S is a *signature generation algorithm* that on input $(1^k, d, m)$ computes output s.
3. V is a *verification algorithm* that on input (e, s, m) computes output v.

where 1^k is the security parameter, e is the public verification key, d is the secret signing key, $m \in \{0,1\}^k$ is the message to be signed, $s \in \{0,1\}^k$ is the signature string and $v \in \{true, false\}$ is the boolean value indicating the validity of the signature, such that if $G \rightarrow (e, d)$ then $V(e, S(d, m), m) = true$.

A strong definition of security for a digital signature system is a system that is secure against existential forgery under chosen message attack [7]. In a chosen message attack, the adversary can choose messages to be signed by the signer. A signature can be existentially forged if, in polynomial time, an adversary can create a message and signature that verifies with greater than negligible probability even though the message may not be the adversary's choice.

Algorithm 8 DSA Key Generation

Input: Security parameter: 1^k
Output: Public verification key: e, Secret signing key: d

$L, N \Leftarrow$ bit-lengths of p and q, respectively, to provide security equivalent to k
$p \Leftarrow L$-bit prime modulus
$q \Leftarrow N$-bit prime such that $q|(p-1)$
$g \Leftarrow$ generator of subgroup of $\mathbb{Z}_p^*$ of order q such that $1 < g < p$
$x \Leftarrow$ randomly selected exponent between 0 and q
$y \Leftarrow g^x \mod p$
$d \Leftarrow (p, q, g, x)$
$e \Leftarrow (p, q, g, y)$
return (e, d)

In DSA, a private key is (p, q, g, x) and a public key is (p, q, g, y), where p, q are prime with $q|(p-1)$, $g \in \mathbb{Z}_p^*$ is an element of order q, x is a secret exponent, and $y = g^x \mod p$. This looks similar to keys in ElGamal with the addition of the prime, q. The element g is chosen so that it generates the cyclic subgroup of $\mathbb{Z}_p^*$ of order q. Note that while the group, $\mathbb{Z}_p^*$, is not fixed for all of DSA, it is also not different for every user. Instead, the values (p, q, g) are called *domain parameters* and are generated and fixed for a particular domain of users.

Algorithm 9 DSA Signature Generation

Input: Message: m, Secret signing key: $d = (p, q, g, x)$, Approved hash function: Hash()
Output: Signature of m: $s = (s', r')$

$k \Leftarrow$ randomly selected exponent between 0 and q
$k^{-1} \Leftarrow$ inverse of $k \mod q$ using extended Euclidean algorithm
$r' \Leftarrow (g^k \bmod p) \bmod q$
$s' \Leftarrow (k^{-1}(\text{Hash}(m) + xr')) \bmod q$
$s \Leftarrow (s', r')$
return s

Algorithm 10 DSA Signature Verification

Input: Message: m, Signature: $s = (s', r')$, Public verification key: $e = (p, q, g, y)$, Approved hash function: Hash()
Output: Validity: v, such that $v =$ **true** $\iff$ s is a valid signature of m

$w \Leftarrow (s')^{-1} \mod q$ // using extended Euclidean algorithm
$z \Leftarrow \text{Hash}(m)$
$u_1 \Leftarrow zw \bmod q$
$u_2 \Leftarrow r'w \bmod q$
$v' \Leftarrow (g^{u_1} y^{u_2} \bmod p) \bmod q$
if $v' = r'$ **then**
  $v \Leftarrow$ **true**
else
  $v \Leftarrow$ **false**
end if
return v

DSA security depends on the difficulty of solving discrete logarithms. An efficient algorithm for finding discrete logarithms would result in a complete break of DSA. Recovering the secret signing key, x, from the public key, $y \equiv g^x \bmod p$, can be achieved by solving the discrete logarithm in $\mathbb{Z}_p^*$ or in the subgroup of $\mathbb{Z}_p^*$ of order q.

C. Fixed Groups in Cryptographic Protocols

Previous algorithms selected a new group for every exchange or key pair, but in practice the group is often chosen from a small list of predefined groups. For example, consider a Diffie-Hellman key exchange. The two participants must first agree upon a group and a generator. In theory, one of the participants could always start by randomly selecting a group at the time of the exchange. However, in reality, using common security protocols, the participants will likely

agree to use a group that is specified in their protocol standard.

In this section, we look at the use of fixed groups in cryptography. In particular, we first provide examples of two commonly used cryptographic protocols that define specific groups. These are Secure Shell [29] and Internet Protocol Security [10]. Then, we examine the motivations for specifying fixed groups in protocol standards. Following this, we discuss the security risks of reusing groups.

1. Groups in SSH

Protocol standards often specify just a few fixed groups for Diffie-Hellman key exchanges. The standard for Secure Shell (SSH) [29] only defines two groups. The two predefined groups are subgroups of $\mathbb{Z}_p^*$ where p is a specific 1024-bit prime and a 2048-bit prime, respectively. The primes were selected by a method described in the OAKLEY key determination protocol [20]. In addition to the two required groups, an SSH implementation is free to add additional groups. But since both client and server implementations must have a specific group predefined, this is essentially a mechanism to add additional standard groups. The SSH standard does not require support for on-the-fly group generation.

There is, however, a proposed Internet standard, *Diffie-Hellman Group Exchange for the Secure Shell (SSH) Transport Layer Protocol* [5], that extends SSH to allow new private groups. The standard defines a method for an SSH server to propose a new group to the client. For this Diffie-Hellman group exchange extension to be effective, it must be supported by implementations and new private groups must actually be created. The popular OpenSSH implements the group exchange, but does not automatically generate new groups. Instead, a utility is included that allows a server administrator to generate new groups from the command line. Without new group generation being automatic and transparent to the user, it is likely that standard groups will still be used even between implementations supporting this extension.

2. Groups in IKE

The Internet Key Exchange (IKE) [10] is the key exchange protocol used in Internet Protocol Security (IPsec). IKE uses the Diffie-Hellman key exchange and specifies just four fixed groups. The first two are subgroups of $\mathbb{Z}_p^*$ where p is a 768-bit prime and 1024-bit prime respectively. The two primes are chosen by the same Oakley method as in SSH. The other two standard groups in IKE are a 155-bit and a 185-bit elliptic curve group. The disparity in

bit-lengths is because more efficient algorithms are known for solving discrete logarithms in $\mathbb{Z}_p^*$ groups than in well chosen elliptic curve groups. Therefore a smaller elliptic curve group is believed to provide security equivalent to a larger $\mathbb{Z}_p^*$ group.

3. Advantages of Fixed Groups

There are many valid reasons to specify fixed groups for Diffie-Hellman key exchanges in a protocol standard. In this subsection, we discuss two major advantages of specifying fixed groups. The first advantage we consider is the reduction in protocol complexity. The second advantage we examine is that the standard groups can be carefully selected to be secure, saving the user the computational expense of creating new secure groups.

Reduced Protocol Complexity

If a protocol is shorter and less complex, its security is easier to analyze and there are fewer opportunities for flaws. A simpler protocol also makes implementation easier with less chance of errors or incompatibilities with other implementations. Using fixed groups reduces the protocol complexity. In particular, it eliminates the need to communicate a description of the group before the key exchange. Additionally, it eliminates the need for clients to implement a method of secure group selection, which as we see in the next section, can be a complicated process.

Securely Selected Groups

If a group is predefined, it can be carefully selected for desired security properties, and the selection is not bound by the computational limitations that would exist if the group selection was done during a live protocol transaction. A protocol standard must ensure that the key exchange provides an appropriate level of security, and the security provided by a group depends on more than just bit-length. Certain groups are weak and must be avoided. Specifically, if the group order is the product of only small prime factors, discrete logarithms can be computed efficiently in this group [21]. (See Section F on page 24.)

In both SSH and IKE, the groups were selected with the Oakley method to achieve goals of efficiency, security, and trust that there is no back-door. In particular, for an n-bit prime, p, the Oakley method fixes the first and last 64-bits to all ones to speedup modular exponentiation. Then the interior bits of p are set to $(c + m)$, where c is the first $(n - 128)$ bits of π and m is the

smallest positive integer such that p and $(p-1)/2$ are both prime. The reason for using π as the source of randomness is to avoid "any suspicion that the primes have secretly been selected to be weak" [20].

Additionally, using standard groups eliminates the need for the computationally intensive process of group creation within the protocol. Creating a new group requires finding a prime, p, and a generator, g of $\mathbb{Z}_p^*$. No efficient method is known for finding a generator of $\mathbb{Z}_p^*$ for a random prime, p. This is because efficiently determining that a number is a generator requires knowing the factorization of $p-1$, the order of the group, and factorization is believed to be a hard problem. Therefore, instead of first choosing a random p, we must generate $N = p-1$ with a known factorization and then test that p is a prime.

To avoid creating a weak group, we want N to have a large prime factor. (Again, see Section F on page 24.) Because N is even, our best case is if $N = 2q$ for q prime. Thus, to create a secure group $\mathbb{Z}_p^*$, we must select random primes, q_i, until $p = 2q_i + 1$ is prime. Many iterations of primality testing make this a computationally intensive process. If this had to be done at the start of each transaction, the user may find the long delay unacceptable. If the group creation was performed automatically on the server it could potentially enable a denial of service attack.

4. Risks of Using Fixed Groups

The downside of using a fixed group is that it places a high premium on attacking a single group. There will be many key exchanges over the same group over many years. To an adversary, the value of solving all discrete logarithms over this fixed group will be much higher than the value of solving all discrete logarithms over a random group that may be used only once. For example, while the value of decrypting a single bank transaction may be small, the value of attacking many simultaneously would be great.

The computational cost of computing multiple logarithms in a single group is much less than computing the same number of logarithms in separate groups. This is because algorithms to find discrete logarithms often require a precomputation dependent only on the group. Once the precomputation is complete for a group, finding additional discrete logarithms in that group is comparatively easy.

Using fixed groups also allows the time-consuming precomputation to occur before a specific key-exchange occurs. Consider a hypothetical attack where the precomputation takes one year, but then solving each instance takes just one hour. (The attacker can trade off instance-

time for precomputation-time, so such a disparity is not unreasonable.) Given a random group, the adversary would always take one year from key exchange to solving the key. However, once an adversary has completed the precomputation for a standard group, a key in that group could be solved just one hour after the exchange occurs. If the encrypted information is only valuable to the attacker for a short period of time, only the second attack is worthwhile.

THIS PAGE INTENTIONALLY LEFT BLANK

III. Survey of Discrete Logarithm Algorithms

In this chapter, we survey the known algorithms for solving discrete logarithms and perform a traditional analysis of their complexity. In particular, we begin by distinguishing between generic algorithms, which work in all cyclic groups, and group-specific algorithms, which apply only in certain families of groups. Then we define the model of computation on which we will base our analysis. After that, we survey several generic algorithms. Lastly, we consider the index calculus algorithm, which is group-specific. For each algorithm we find the asymptotic running time and space requirements.

A. Generic vs. Group-Specific Algorithms

There are several known algorithms for solving discrete logarithms. In this section, we divide the algorithms into two categories, *generic algorithms* and *group-specific algorithms*. The first category we call generic algorithms, because they apply generally over any type of cyclic group. A generic algorithm solves the generalized discrete logarithm problem (GDLP). The second category of algorithms are the group-specific algorithms. These are specialized algorithms that make use of the structure in the group elements and apply only within certain families of groups.

The generic algorithms we will consider include Shank's algorithm [16], which is also called the Baby-Step Giant-Step algorithm, Pollard's Rho and Pollard's Kangaroo algorithms [22]. These algorithms apply over any cyclic group including elliptic curve groups and subgroups of $\mathbb{Z}_p^*$, where better methods do not apply. The group-specific algorithms we discuss are *index calculus algorithms*. They apply in $\mathbb{Z}_p^*$. Therefore, index calculus algorithms solve the standard discrete logarithm problem (DLP).

B. Model of Computation

In this section, we define our model of computation. In particular, we begin by defining the abstract machine that which will execute the algorithms. Then we define the notation used in our analysis. Finally, we explain the format we will use for our analysis of each algorithm.

For our analysis of algorithms to be consistent we need to define a model of computation. Central to that is defining a standard abstract machine for finding the asymptotic running time of each algorithm. Our model uses a multitape Turing machine, which is a good model of a standard computer. We provide our runtime complexity in terms of the number of group operations. We do this because the complexity of the group operation varies among different group families.

Now we define the standard notation we use in our analysis. For each algorithm, we have a cyclic group G and a generator g of that group. We let N be the order of g,

$$N = |\langle g \rangle|,$$

and let n be the bit-length of N,

$$2^{n-1} \leq N < 2^n,$$

$$n = \lceil \log_2 N \rceil.$$

When describing the asymptotic performance of these algorithms, we do so in terms of n, as is common practice. In terms of storage, we assume that elements of G can be represented in $O(n)$ bits. This assumption is reasonable because there are less than 2^n elements in G.

In the following sections, we perform a traditional complexity analysis of several known algorithms for solving discrete logarithms. Each analysis will follow a standard format. For each algorithm we begin with a description of the algorithm itself. Next, we analyze the algorithm's runtime complexity. Then we analyze the asymptotic space requirements of the algorithm. We conclude the chapter with a table summarizing the space and runtime complexity of each algorithm.

C. Brute-Force Search

We begin our survey of generic algorithms, with the simplest method, brute-force or exhaustive search. That is simply trying every possible exponent $(g^0, g^1, g^2, ...)$ until a match is found.

Algorithm 11 Brute-Force Search

Input: Cyclic Group: G, Generator: g, Group Element: a
Output: Exponent: x such that $g^x = a$

```
1: b ⇐ 1
2: x ⇐ 0
3: while a ≠ b do
4:     b ⇐ b × g
5:     x ⇐ x + 1
6: end while
7: return x
```

In the worst case, where $a = g^{N-1}$, every exponent would be tested, requiring a total of N tests. In terms of the bit-length of the input, n, this requires 2^n group operations and comparisons in the worst case. In the average case, one can expect to find the correct exponent after searching half the space, or 2^{n-1} group operations. In either case, the running time of the algorithm is exponential, $O(2^n)$, and will quickly become intractable for increasing n.

On the other hand, the space requirements are minimal. At each step we need only to store x and b, and both can be represented in n bits. Therefore, the asymptotic space requirement of the brute-force algorithm is $O(n)$.

D. Precomputed Table Algorithm

Just two average-case runs of the brute-force search algorithm requires an amount of work equivalent to computing all N exponents. Consider, instead, if one first computed all N exponents and stored them. That is the idea behind our next algorithm, the precomputed table algorithm. We build a table holding every discrete logarithm for the group. After computing the table, finding an individual discrete logarithm requires just a single table lookup.

The running time of the algorithm is dominated by the precomputation, which requires N group operations. The asymptotic running time of the precomputed table algorithm is $O(2^n)$. The advantage of the algorithm is the instant solutions of subsequent discrete logarithms in the

Algorithm 12 Precomputed Table Algorithm

Input: Cyclic Group: G, Generator: g, Group Element: a
Output: Exponent: x such that $g^x = a$

```
// First build the table such that hash[g^x] = x for 0 ≤ x < N
b ⇐ 1
for x = 0 to N − 1 do
    hash[b] ⇐ x
    b ⇐ b × g
end for
// Now perform the table lookup
x ⇐ hash[a]
return x
```

same group; only a single table lookup is required.

Of course, this algorithm is infeasible for values of n of cryptologic significance, as it is exponential in both time and space complexity. The lookup table holds N values of size n, giving an asymptotic size of $O(n2^n)$.

E. Shank's Algorithm

Solving discrete logarithms using brute-force search requires $O(2^n)$ group operations. With a precomputed table, we can do it in constant time but require $O(n2^n)$ bits of storage. What if we could find an optimal point between these two extremes? Shank's Algorithm gives us a way to achieve such a balance.

Shank's Algorithm is also known as the baby-step giant-step algorithm. The algorithm has two stages. In the first stage of the algorithm, we step consecutively through the first X powers of $g^i : g^0, g^1, g^2, ...g^{X-1}$. These are the "baby-steps". At each step we store the exponent, i, in a hash table indexed by g^i. After X steps we have a table of discrete logarithms, but only for the first X elements of the cyclic group.

In the second stage, we want to transform the input $a = g^x$ into a value that is in our range of precomputed discrete logarithms. Starting from g^x, we step X elements at a time through the cyclic group until we reach the beginning of the cycle where we have precomputed the logarithms. To take these "giant-steps", we simply multiply by g^X,

$$g^x g^X = g^{x+X},$$

Algorithm 13 Shank's Algorithm

Input: Cyclic Group: G, Generator: g, Group Element: a, Number of exponents to precompute: X

Output: Exponent: x such that $g^x = a$

1: // Build table *hash* such that *hash*$[g^i] = i$ for $0 \leq i < X$
2: $b \Leftarrow 1$
3: **for** $i = 0$ to $X - 1$ **do**
4: hash$[b] \Leftarrow i$
5: $b \Leftarrow b \times g$
6: **end for**
7: // Now compute successive exponents until you find one in the *hash*
8: $b \Leftarrow a$
9: $y \Leftarrow 0$
10: $h \Leftarrow$ hash$[b]$
11: **while** $g^h \neq b$ **do**
12: $b \Leftarrow b \times g^X$
13: $y \Leftarrow y + 1$
14: $h \Leftarrow$ hash$[b]$
15: **end while**
16: $x \Leftarrow h - yX \mod N$
17: **return** x

$$g^{x+X} g^X = g^{x+2X}$$

$$\vdots$$

When we find a value in the precomputed range we will have the equation,

$$g^h = g^{x+yX}$$

Now we can solve for x,

$$h = x + yX \mod N$$

$$x = h - yX \mod N$$

We are certain to hit a logarithm in the precomputed a range of X consecutive exponents, because we are stepping by exactly X exponents at a time.

Now we will consider the runtime of the algorithm. The first stage requires X group operations. The runtime of the second stage will vary depending on the number of giant steps to reach the precomputed range of exponents. The most steps will be needed when $X < x < 2X$,

putting x just outside the range of precomputed exponents. In this worst case, the second stage will take $\lceil \frac{N}{X} \rceil$ group operations (multiplications by g^X). (We can store the precomputed exponents in a hash table to avoid the cost of sorting the table.)

To minimize the total computation time, we must choose X so that the number of baby-steps equal the number of giant-steps. That is when $X = \lceil \frac{N}{X} \rceil$.

$$X = \frac{N}{X},$$

$$X^2 = N,$$

$$X = \sqrt{N},$$

$$X = \sqrt{2^n},$$

$$X = 2^{\frac{n}{2}}.$$

Given $X = 2^{\frac{n}{2}}$, both stages of the algorithm take $2^{\frac{n}{2}}$ group operations. Therefore the runtime complexity of Shank's algorithm is $O(2^{\frac{n}{2}})$. Although the running time is still exponential, it is a significant improvement over the brute-force search.

The space requirements are a middle ground between the brute-force search and the precomputed table algorithms. The table in Shank's algorithm will require X entries of size n-bits. Therefore the space complexity of Shank's algorithm is $O(n2^{\frac{n}{2}})$.

F. Pohlig-Hellman Algorithm

The Pohlig-Hellman algorithm makes use of the prime factorization of N, the order of the group. For groups of prime order this algorithm provides no advantage and is equivalent to Shank's algorithm. Our analysis will focus on the case where the order, N, has only small prime factors. This is where the algorithm is most efficient, and this is why some groups are weaker than others, motivating standards bodies to include specific "secure" groups in their standards.

The first step of the Pohlig-Hellman algorithm is to factor, N, the order of the group. When N has only small prime factors the factorization can be found easily. Let the factorization of $N = \prod_{i=1}^{k} p_i^{n_i}$.

For each unique prime factor, p_i, we solve for $x_i \equiv x \bmod p_i^{n_i}$. Once each x_i is found they can be combined using the Chinese Remainder Theorem to find x, requiring $O(k \log N)$ group operations and $O(k \log N)$ space [21].

Algorithm 14 Pohlig-Hellman Algorithm

Input: Cyclic Group: G, Generator: g, Group Element: a, Order of group: N
Output: Exponent: x such that $g^x = a$

Find the factorization of $N = \prod_{i=1}^{k} p_i^{n_i}$
// For each factor $p_i^{n_i}$ find $x_i \equiv x \bmod p_i^{n_i}$
for $i = 1$ to k **do**
    $z \Leftarrow a$
    $h \Leftarrow g^{-1}$
    $q \Leftarrow (p-1)/p_i$
    $g_i \Leftarrow g^q$
    for $j = 0$ to $(n_i - 1)$ **do**
        $w \Leftarrow z^q$
        $b_j \Leftarrow \log_{g_i} w$ // Solve this discrete logarithm using Algorithm 13
        $z \Leftarrow z h^{b_j}$
        $h \Leftarrow h^{p_i}$
        $q \Leftarrow q/p_i$
    end for
    $x_i \Leftarrow \sum_j b_j p_i^j$
end for
Solve for x given $x_1, ...x_k$ using the Chinese Remainder Theorem
return x

To find an x_i, we find each coefficient, b_j, from the representation of $x_i = \sum_j^{n_i - 1} b_j p_i^j$. From Algorithm 14, $b_j = \log_{g_i} w$, where $\log$ is a discrete logarithm. The base $g_i = g^{(p-1)/p_i}$, so the order of the g_i is p_i. For a group with a large prime factor, p_i, the dominant step will be finding the discrete logarithm in the subgroup of order p_i using Shanks algorithm.

For small p_i, discrete logarithms can be solved with precomputed tables. In the case where all p_i are small relative to N, the dominant step of the algorithm is computing $w = z^n$, requiring $O(\log N)$ group operations [21]. The number of times z^n must be computed is $\sum_1^k n_i$, which is $O(\log N)$ when the prime factors are small. This gives a total running time of $O(\log N)^2$ or $O(n^2)$.

G. Pollard's Rho Algorithm

The next algorithm we present, Pollard's Rho algorithm, has a running time on the same order as Shank's, but does so while avoiding a large stored table. The rho algorithm takes advantage of the *birthday paradox*; that is there is greater than 50% probability that 2 people

out of 23 chosen randomly will share a birthday. More generally, when selecting elements at random from N elements, a collision will be found after an expected $\sqrt{\pi N/2}$ selections [27].

To find the discrete logarithm of an element a to the base g, Pollard's Rho algorithm steps through a random sequence of group elements s^i that can be represented as products of powers of a and g.

$$s_i = a^{a_i} g^{g_i} = g^{xa_i} g^{g_i} = g^{xa_i + g_i}.$$

The algorithm searches for a cycle in the sequence, two elements $s_u, s_v, u \neq v$ such that $s_u = s_v$. Solving this equation for x gives the discrete logarithm of a.

$$s_u = s_v$$

$$g^{xa_u + g_u} = g^{xa_v + g_v}$$

$$xa_u + g_u \equiv xa_v + g_v \mod N$$

$$xa_u - xa_v \equiv g_v - g_u \mod N$$

$$x(a_u - a_v) \equiv g_v - g_u \mod N$$

$$x \equiv (a_u - a_v)^{-1} g_v - g_u \mod N$$

The running time is dominated by a search for a cycle in the sequence. Finding a cycle could be accomplished by storing each element in the sequence until one is repeated. This would require a large amount of storage, so instead Pollard uses the Floyd cycle-finding algorithm which requires storing just two sequence elements s_i and s_{2i}. The element, s_{2i}, is always twice as far into the sequence as s_i and a cycle is found when $s_i = s_{2i}$. To advance both sequences, one step of the algorithm requires a total of three steps of the sequences. Pollard's [22] calculations gave a mean value for i of $1.08\sqrt{N}$. The asymptotic running time is $O(2^{\frac{n}{2}})$ group operations and storage of just $O(n)$.

Algorithm 15 is an improved version of Pollard's Rho method due to van Oorschot and Wiener [27]. Their method finds the cycle by stepping just once through the sequences, providing a speedup by a factor of 3. This is possible because they store *distinguished points*. Distinguished points are elements of the group with an easily distinguished property, for example, elements where the first c bits of their binary representation are zeros. We start at a random location and step through the sequence until we reach a distinguished point. We store the distinguished point and start again from a new random location. When we reach a distinguished point that we already have stored, we have found a cycle and can solve for the logarithm.

Algorithm 15 Pollard's Rho Algorithm

Input: Cyclic Group: G, Generator: g, Group Element: a, Order of g: N
Output: Exponent: x such that $g^x = a$

// Search for a cycle in the random sequence $S = s_0, s_1, ...$ defined by Algorithm 16
success $\Leftarrow$ **false**
$D \Leftarrow$ a subset of distinguished points from G
while (success = **false**) **do**
  $i \Leftarrow 0$
  $a_i \Leftarrow$ randomly selected exponent between 0 and $p-1$
  $g_i \Leftarrow$ randomly selected exponent between 0 and $p-1$
  $s_i \Leftarrow g^{g_i} a^{a_i}$
  repeat
    $i \Leftarrow i+1$
    Calculate s_i, a_i, g_i applying Algorithm 16
  until $(s_i \in D)$
  // If we have already stored this point before
  if $((a_j, g_j) \Leftarrow \text{hash}(s_i))$ **then**
    success $\Leftarrow$ **true**
  else
    $\text{hash}(s_i) \Leftarrow (a_i, g_i)$
  end if
end while
$m \Leftarrow a_i - a_j \mod N$
$x \Leftarrow m^{-1}(g_j - g_i) \mod N$
return x

The running time of this version of the rho algorithm is the sum of the time to find a collision, T_c, plus the time to reach a distinguished point, T_d. If we assume the sequence is a random mapping, then the expected time to a collision will be $T_c = \sqrt{\pi N/2}$. The time to reach a distinguished point depends on the frequency of distinguished points. Given that there are $c\sqrt{N}$ distinguished points in the group for some constant $c \gg 1$, one of every $\frac{N}{c\sqrt{N}} = \frac{\sqrt{N}}{c}$ elements is a distinguished point. The sequence reaches a distinguished point after an expected $T_d = \frac{\sqrt{N}}{c}$ steps. The total expected running time of the rho algorithm is

$$T_c + T_d = \sqrt{\frac{\pi N}{2}} + \frac{\sqrt{N}}{c} = \left(\sqrt{\frac{\pi}{2}} + \frac{1}{c}\right)\sqrt{N} \approx \sqrt{\frac{\pi N}{2}}.$$

The asymptotic running time in terms of the bit-length n is $O(2^{\frac{n}{2}})$.

The algorithm needs storage for the distinguished points. The expected number of dis-

Algorithm 16 Pollard's Rho - Random Sequence Algorithm

Input: Element: s_i, Exponents: a_i, g_i, such that $s_i = a^{a_i} g^{g_i}$
Output: Element: s_{i+1}, Exponents: a_{i+1}, g_{i+1}, such that $s_{i+1} = a^{a_{i+1}} g^{g_{i+1}}$

// Given a partitioning of G into three equal-sized subsets S_1, S_2, S_3
if $s_i \in S_1$ **then**
    $s_{i+1} = a s_i$
    $a_{i+1} = a_i + 1 \mod N$
    $g_{i+1} = g_i$
else if $s_i \in S_2$ **then**
    $s_{i+1} = {s_i}^2$
    $a_{i+1} = 2a_i \mod N$
    $g_{i+1} = 2g_i \mod N$
else if $s_i \in S_3$ **then**
    $s_{i+1} = g s_i$
    $a_{i+1} = a_i$
    $g_{i+1} = g_i + 1 \mod N$
end if
return $s_{i+1}, a_{i+1}, g_{i+1}$

tinguished points will be the expected number of steps multiplied by the fraction of elements that are distinguished points,

$$\left(\left(\sqrt{\frac{\pi}{2}} + \frac{1}{c}\right)\sqrt{N}\right)\left(\frac{c}{\sqrt{N}}\right) = c\sqrt{\frac{\pi}{2}} + 1$$

For each distinguished point, we store a pair of n-bit exponents. Thus the total expected storage required by the algorithm is $(c\sqrt{2\pi} + 2)n$ bits. Because c is a constant, the total storage requirement is $O(n)$.

H. Pollard's Kangaroo Algorithm

Another generic algorithm discovered by Pollard [22] is the kangaroo or lambda method. It has a runtime that differs from the Pollard's Rho method by only a constant. It can also be used to find discrete logarithms when the exponent is known to lie in a smaller interval. We present an improved version, due to van Oorschot and Weiner [27], that uses distinguished points.

The kangaroo-method gets its name because it can be described with an analogy of two kangaroos hopping. If we imagine each element of the the cyclic group as being steps on a

Algorithm 17 Pollard's Kangaroo Algorithm

Input: Cyclic Group: G, Generator: g, Group Element: a, Order of g: N
Output: Exponent: x such that $g^x = a$

// Select a small sequence of possible step sizes
$S \Leftarrow (s_0, s_1, \ldots, s_{k-1})$ where $s_i = 2^i$ and k such that the mean of the entries is $\sqrt{N}$
$R \Leftarrow (r_0, r_1, \ldots, r_{k-1})$ where $r_i = g^{s_i}$
// Select a hash function to map a group element to a particular step size, s_i
$h(x) \Leftarrow$ hash function mapping G into the interval $[1..k]$
$D \Leftarrow$ a subset of distinguished points from G
// The "tame" kangaroo starts off half way through the cycle
$x_t \Leftarrow g^{\frac{N}{2}}$
$d_t \Leftarrow 0$
// The "wild" kangaroo starts off from $a = g^x$
$x_w \Leftarrow a$
$d_w \Leftarrow 0$
success $\Leftarrow$ **false**
while (success = **false**) **do**
  // Step the tame kangaroo one hop
  $i \Leftarrow h(x_t)$
  $x_t \Leftarrow x_t r_i$
  $d_t \Leftarrow d_t + s_i$
  if $x_t \in D$ **then**
    // If we have already stored this point for a wild kangaroo
    if $((m, x_i, d_i) \Leftarrow$ hash$(x_t))$ && (m = 'wild') **then**
      $x \Leftarrow \frac{N}{2} + d_t - d_i$
      success $\Leftarrow$ **true**
    else
      hash$(x_t) \Leftarrow$ ('tame', x_t, d_t)
    end if
  end if
  // Step the wild kangaroo one hop
  $i \Leftarrow h(w_w)$
  $x_w \Leftarrow x_w r_i$
  $d_w \Leftarrow d_w + s_i$
  if $x_w \in D$ **then**
    // If we have already stored this point for a tame kangaroo
    if $((m, x_i, d_i) \Leftarrow$ hash$(x_w))$ && (m = 'tame') **then**
      $x \Leftarrow \frac{N}{2} + d_i - d_w$
      success $\Leftarrow$ **true**
    else
      hash$(x_w) \Leftarrow$ ('wild', x_w, d_w)
    end if
  end if
end while
return x

path, ordered by exponent, $(g^0, g^1, g^2, \ldots)$, then each hop of the kangaroo is from one element of the group to another. The distance of the hop, s_i, is selected from a small set of possible hop distances S. The choice of s_i is based only on the current position, x, using a hash function $h(x) = i$. This means that any kangaroo that lands on a particular element will always take the same sequence of hops from then on.

The algorithm uses two kangaroos (sequences), one wild and one tame. The wild one starts on element $a = g^x$. Its starting position exponent, x is unknown; it is the discrete logarithm that we are trying to find. We start the tame kangaroo at a known position, halfway through the cycle, at $g^{\frac{N}{2}}$. We alternate stepping the wild and tame kangaroos. We keep track of their respective positions, x_w, x_t, and their respective distances traveled, d_w, d_t.

We want the wild kangaroo to land on the path of the tame kangaroo. Since we know the exponent of the tame kangaroo, $\frac{N}{2} + d_t$, we can calculate the discrete logarithm by subtracting the distance traveled by the wild kangaroo,

$$\log_g a = x \equiv \frac{N}{2} + d_t - d_w \bmod N$$

As the two kangaroos jump, their paths will eventually converge.

Anytime a kangaroo lands on a distinguished point, we store which kangaroo, the point, and the distance traveled in a hash table. If the other kangaroo has already stored this point in the hash table, then the paths have converged, and we can solve for the discrete logarithm. The use of distinguished points allows us to discover the convergence point quickly while reducing memory accesses and storage requirements. Memory only needs to be read or written on the small percentage of steps that land on distinguished points.

To find the runtime and storage requirements, we follow the approximate analysis of Pollard [23]. We consider the algorithm as three stages:

1. The kangaroo in back must catch up with the starting point of the other kangaroo.

2. The back kangaroo must then land on the path of the other kangaroo.

3. The back kangaroo must continue until it reaches a distinguished point.

Throughout each stage, the back kangaroo could be either the wild or the tame kangaroo.

At the start, the back kangaroo can be at most half a cycle behind and on average will be a quarter of a cycle behind or $\frac{N}{4}$. The mean step size is $m = \frac{\sqrt{N}}{2}$, so the back kangaroo needs $\frac{N}{4} / \frac{\sqrt{N}}{2} = \frac{\sqrt{N}}{2}$ steps, on average, to catch up to the starting point of the front kangaroo. Given that the kangaroos alternate steps, the average running time of Stage 1 is $\sqrt{N}$ group operations.

Once the back kangaroo has caught up, it must land on the front kangaroo's path. Given a mean step size, m, one out of every m elements will be on the kangaroo's path, on average. Each hop of the back kangaroo has a $\frac{1}{m}$ chance of landing on the other kangaroo's path. Thus the kangaroo will land on the path after an expected m hops or $2m$ total steps of both kangaroos. The average running time of Stage 2 is $2m = 2\frac{\sqrt{N}}{2} = \sqrt{N}$.

Now that the back kangaroo is on the same path, it must step until it reaches a distinguished point. Given that there are $c\sqrt{N}$ distinguished points in the group for some constant $c \gg 1$, one of every $\frac{N}{c\sqrt{N}} = \frac{\sqrt{N}}{c}$ elements is a distinguished point. The kangaroo will land on a distinguished point after an expected $\frac{\sqrt{N}}{c}$ hops. The expected running time of Stage 3 is $2\frac{\sqrt{N}}{c}$ group operations.

Summing the running times of the three stages gives a total expected running time of

$$\sqrt{N} + \sqrt{N} + 2\frac{\sqrt{N}}{c} = 2\sqrt{N}(1 + \frac{1}{c}).$$

Given that c is large, the running time is approximately $2\sqrt{N}$. The asymptotic running times in terms of the bit-length n is $O(2^{\frac{n}{2}})$.

The algorithm needs storage for the distinguished points. The expected number of distinguished points will be the expected number of steps multiplied by the fraction of elements that are distinguished points,

$$(2\sqrt{N}(1 + \frac{1}{c}))(\frac{c}{\sqrt{N}}) = 2c(1 + \frac{1}{c}) = 2(c + 1)$$

For each distinguished point, we store a pair of n-bit quantities: a group element and an integer distance. Thus the total expected storage required by the algorithm is $4(c + 1)n$ bits. Because c is a constant, the total storage requirement is $O(n)$.

I. Index Calculus Algorithm

The index calculus algorithm takes advantage of the structure of the group elements, specifically the fact that group elements can be factored into a product of primes. Unlike the generic algorithms that treat the group as a black box and work in any group, the index calculus algorithm only applies to groups with the necessary structure, like $\mathbb{Z}_p^*$. The algorithm is divided into three phases. In the first phase, a number of linear relations are found. In the second phase, a solution is found to the system of linear relations. In the final phase, an individual discrete

logarithm instance is solved.

The index calculus algorithm (Algorithm 18) depends on the fact that many elements of the group can be represented as the product of a small number of group elements. In the case of $\mathbb{Z}_p^*$, many integers can be represented as a product of small primes. An integer is called B-*smooth* if it has no prime factors larger than B. The primes less than B make up a *factor base*, $S = (p_1, p_2, \ldots, p_k)$ where there are k primes less than B. A B-smooth integer is one that can be represented as a product of the elements of S.

With an optimal choice for the bound, B, the running time of the index calculus algorithm is *subexponential*. That is, it is faster than any algorithm that is exponential in the input size. We will use the standard notation for subexponential running times [16],

$$L_p(\alpha, c) = O(\exp((c + o(1))(\ln p)^{\alpha}(\ln \ln p)^{1-\alpha})),$$

where $0 \leq \alpha \leq 1$ and $c > 0$. If the first parameter, α, is 0, the algorithm is polynomial with degree equal to the second parameter, c. If α is 1, the algorithm is fully exponential. For α between 0 and 1, the algorithm is called subexponential.

When comparing two algorithms using the $L_p(\alpha, c)$ notation, the smaller the value α, the shorter the asymptotic running time. If both algorithms have the same α, then the one with the smaller value of c will be faster.

During the first phase of the algorithm, we generate random group elements, g^y, by randomly selecting exponents, y. We test each element to find any that are B-smooth and factor those we find. Then we take the logarithm of the factorization, giving us a linear equation in terms of the discrete logarithms of the primes in the factor base. We continue until we have more relations than there are unknowns.

In the second phase, we solve for the k unknowns among the linear relations found in Phase 1. Phase 2 is complete when we have a table that holds the discrete logarithms of each of the primes in the factor base, table$(i) = \log_g p_i$ for $0 \leq i \leq k$.

The third phase proceeds much like the first, except now we need to find just one B-smooth integer. It will be of the form g^{x+y} where $x = \log_g a$ is the discrete logarithm we are trying to solve and y is between 0 and $p-1$. We randomly select y, until $u = ag^y = g^x g^y = g^{x+y}$ is smooth. Next we find the factorization of u,

$$g^{x+y} = \prod_{i=1}^{k} p_i^{c_i}.$$

Algorithm 18 Index Calculus Algorithm

Input: Prime: p, Generator of $\mathbb{Z}_p^*$: g, Element of $\mathbb{Z}_p^*$: a

Output: Exponent: x satisfying $g^x \equiv a \mod p$, where $0 \leq x < p-1$.

// **Setup**: Select a factor base
$B \Leftarrow$ a bound for the largest prime in the factor base, S
$S \Leftarrow (p_1, p_2, \ldots, p_k)$ where S contains all primes, $p_i < B$
// **Phase 1**: Find linear relations of the factor base
// Find a few more relations than the size of the factor base to ensure a unique solution
for $j = 0$ to $k + c$ **do**
    repeat
        $y \Leftarrow$ randomly selected exponent between 0 and $p-1$
        $u \Leftarrow g^y \mod p$
    until (u is B-smooth)
    // Find the factorization of u
    $u = \prod_{1 \leq i \leq k} p_i^{c_i}$
    // Take logarithms and store the linear relation
    $y = \sum_{1 \leq i \leq k} c_i \log_g p_i \pmod{p-1}$
end for
// **Phase 2**: Solve system of linear relations
Given the $k + c$ relations from Phase 1, solve for the k unknown discrete logarithms of the factor base.
Store the logarithms, such that table$(i) = \log_g p_i, 1 \leq i \leq k$
// **Phase 3**: Solve for the individual discrete logarithm
repeat
    $y \Leftarrow$ randomly selected exponent between 0 and $p-1$
    $u \Leftarrow ag^y \mod p$
until (u is B-smooth)
// Find the factorization of u
$u = \prod_{1 \leq i \leq k} p_i^{c_i}$
// Take logarithms of both sides
$x + y = \sum_{1 \leq i \leq k} c_i \log_g p_i \pmod{p-1}$
$x \Leftarrow \sum_{1 \leq i \leq k} c_i \text{ table}(i) - y \pmod{p-1}$
return x

Then taking the discrete logarithm of both sides,

$$x + y = \sum_{i=1}^{k} c_i \log_g p_i \pmod{p-1}.$$

The discrete logarithms for any of the small primes, p_i, can be read from the table created in Phase 2, and thus we can simply solve for x,

$$x = \sum_{i=1}^{k} (c_i \text{table}(i)) - y \pmod{p-1}.$$

To analyze the running time, we consider the time of each phase separately. The running time of the first phase, T_1, will be the number of smooth elements that need to be found, k, multiplied by E_s, the expected number of elements to test to find one B-smooth element, multiplied by the time, T_s, to test one element for smoothness. That is

$$T_1 = kE_sT_s.$$

Solving the linear system requires having as many linear equations as there are unknowns. The unknowns are the logarithms of the factor base. Thus, we need to find as many B-smooth elements as there are primes in our factor base. The size of the factor base is the number of primes less than B,

$$k = \pi(B) \approx \frac{B}{\log B}.$$

The expected number of integers to test to find one B-smooth integer depends on the distribution of smooth integers. The probability that a random element of $\mathbb{Z}_p^*$ is B-smooth is $p/\psi(p, B)$ where $\psi(p, B)$ is the number of B-smooth numbers less than p. Thus, we expect to find a B-smooth element after testing $E_s = p/\psi(p, B)$ random elements. An approximation for the number of B-smooth numbers less than p is

$$\psi(p, B) = pu^{-u},$$

where $u = \log p / \log B$. Therefore,

$$E_s = p/\psi(p, B) = \frac{p}{pu^{-u}} = u^u = (\log p / \log B)^{\log p / \log B}.$$

The time required to test an element for smoothness depends on the method used. The simplest method, trial division, will require k divisions where k is the size of the factor base, S. A sieving method where many values are tested simultaneously is much more efficient, giving a time $T_s = \log\log B$ [24]. The total runtime of the Phase 1 is

$$T_1 = kE_sT_s = \frac{B}{\log B}(\log p/\log B)^{\log p/\log B}\log\log B.$$

The running time of Phase 2 is the time to solve a $k \times k$ linear system. Using Gaussian elimination would take $O(k^3)$ time, but because the system is very sparse there are methods that work in $O(k^2)$ time [14]. Recall that $k \approx \frac{B}{\log B}$. Thus the running time of Phase 2 is

$$T_3 = k^2 = (\frac{B}{\log B})^2$$

The calculation for the running time of Phase 3, T_3, is very similar to that of Phase 1. The biggest difference is that in Phase 3 only one smooth element needs to be found. That means the sieving approach used in Phase 1 to find many smooth elements simultaneously is not applicable. Instead, the most efficient method is elliptic curve factorization in time $L_B(\frac{1}{2}, \sqrt{2})$ [16]. This gives a total running time for Phase 3 of

$$T_3 = E_sT_s = (\log p/\log B)^{\log p/\log B}L_B(\frac{1}{2}, \sqrt{2})$$

With an optimal choice for the bound, B, Adleman [1] showed that the index calculus algorithm is subexponential with a running time of $L_p(\frac{1}{2}, c)$. Coppersmith, Odlyzko, and Schroeppel [2] showed that by using sieving methods in Phase 1 a running time of $L_p(\frac{1}{2}, 1)$ could be achieved. Currently the fastest algorithm for solving discrete logarithms in $\mathbb{Z}_p^*$ is a complex variant of the index calculus algorithm called the number field sieve. The running time of the number field sieve for discrete logarithms is $L_p(\frac{1}{3}, 1.923)$, but a discussion of this algorithm is beyond the scope of this thesis. The storage requirement of the index calculus

algorithm is the space needed to represent the system of linear equations. Although the system being solved is $k \times k$, the system is very sparse and the zero entries need not be stored. Each equation will have fewer than $\log B$ non-zero entries of size n. So the asymptotic size of the index calculus algorithm is

$$n\frac{B}{\log B}\log B = nB.$$

To achieve the optimal running time for the index calculus algorithm the choice of B is $L_p(\frac{1}{2}, \frac{1}{2})$.

J. Summary

The asymptotic space and running times of the algorithms presented in this chapter are summarized in Table 3. The running times of all the generic algorithms are exponential, with the three best being $O(2^{\frac{n}{2}})$. Pollard's Rho and Pollard's Kangaroo algorithms achieve this running time with only linear storage requirements. The kangaroo method is about 1.60 times slower than the best variants of the rho-method, which achieve an expected running time of $\sqrt{\frac{\pi N}{2}}$ [27]. The running time of the only group-specific algorithm presented, the index calculus algorithm, is subexponential in both running time and in space.

Algorithm	Space	Running Time
Brute-Force Search	$O(n)$	$O(2^n)$
Precomputed Table	$O(n2^n)$	$O(2^n)$
Shank's	$O(n2^{\frac{n}{2}})$	$O(2^{\frac{n}{2}})$
Pollard's Rho	$O(n)$	$O(2^{\frac{n}{2}})$
Pollard's Kangaroo	$O(n)$	$O(2^{\frac{n}{2}})$
Index Calculus	$L_p(\frac{1}{2}, \frac{1}{2})$	$L_p(\frac{1}{2}, 1)$

Table 3: Complexity of Discrete Logarithm Algorithms

Not included in the table is the Pohlig-Hellman algorithm. The dominant step of that algorithm is to compute the discrete logarithm in the subgroup of order q, where q is the largest prime factor of N. The runtime of the algorithm will therefore be that of the algorithm to solve the discrete logarithm in a prime subgroup. Any generic algorithm could be used for this step. Pohlig and Hellman initially suggested Shank's algorithm [21], but the rho algorithm would be the superior choice today.

IV. Complexity of Discrete Logarithms over Fixed Groups

In this chapter, we examine the complexity of discrete logarithms over fixed groups. In particular, we introduce the para-discrete logarithm problem, a variant of the discrete logarithm problem that more closely models cryptologic applications over fixed groups. Next, we discuss how to model an adversary with access to a group-specific precomputation. Then we re-examine each algorithm from the previous chapter as a para-discrete logarithm solver. We summarize the analysis with a chart showing the run times and precomputation sizes for each algorithm. We then apply our analysis of generic algorithms to place an upper bound on the complexity of the generalized para-discrete logarithm problem. Finally, we use our analysis of index calculus algorithms to place an upper bound on the para-discrete logarithm problem.

A. The Para-Discrete Logarithm Problem

Complexity theoretic models are useful for evaluating the security of real world cryptographic applications. However, a model can also provide a false sense of security if it oversimplifies the implementation details or makes bad assumptions about the capabilities of the adversary. To illustrate this point, imagine a protocol that implements ElGamal public key encryption. This hypothetical protocol requires a user to prove they know their private key by responding with the plaintext after receiving an encrypted random number as a challenge. This allows an adversary to mount a chosen ciphertext attack. Consider an adversary who intercepts a ciphertext, $(c_1 = g^b, c_2 = g^{ab}m)$, encrypted with user A's public key, g^a, and wants to read the secret message, m. The adversary selects a random, r, and sends $(c_1' = c_1 = g^b, c_2' = c_2 r = g^{ab}mr)$ to the user A as a random number challenge. User A will decrypt and return the seemingly random message, $m' = mr$. From m', the attacker easily solves for m by multiplying by r^{-1}, giving $m = mrr^{-1}$. Thus, if ElGamal is used within a flawed protocol, the difficulty of the discrete logarithm problem is irrelevant. A security model must consider the protocol as a whole and not just the underlying cryptography.

The use of fixed groups in security protocols inspires the question: Do our existing models sufficiently capture these applications? In the DLP and GDLP, the group and generator are inputs to the problem along with a particular instance to solve. Yet in a fixed group implementation, every instance takes place over the same small set of groups. Does this provide an advantage to an adversary? We propose a new complexity problem that more closely models these fixed group protocols, the para-discrete logarithm problem.

Definition 19 The Para-Discrete Logarithm Problem (PDLP)

Setup: Let $p = p_2, p_3, p_4, \ldots$ be an infinite sequence of primes, where p_i is a prime of bit-length, i. Let $g = g_2, g_3, g_4, \ldots$ be an infinite sequence of integers, where $0 < g_i < p_i$ and g_i generates $\mathbb{Z}^*_{p_i}$.

Input: Security Parameter: 1^n, Group Element: $a \in \mathbb{Z}^*_{p_n}$, where $p_n \in p$.

Output: Exponent: x satisfying ${g_n}^x \equiv a \mod p_n$, where $g_n \in g$

Unlike in the standard discrete logarithm problem, the group and generator are not inputs to the para-discrete logarithm problem. Instead, there is just a security parameter, 1^n, that specifies the bit-length of the prime modulus. The prime modulus, p_n, comes from an infinite sequence of primes, with exactly one prime for a given bit-length. We contend this problem is a better computational model for discrete logarithms over fixed groups, because for a given security parameter there is a single defined group.

Just as the discrete logarithm problem can be generalized from $\mathbb{Z}^*_p$ to any cyclic group G, we can generalize the para-discrete logarithm problem.

Definition 20 The Generalized Para-Discrete Logarithm Problem (GPDLP)

Setup: Let $G = G_1, G_2, G_3, \ldots$ be an infinite sequence of groups, where G_i is a cyclic group of order N_i, such that $2^{i-1} < N_i \leq 2^i$. Let $g = g_1, g_2, g_3, \ldots$ be an infinite sequence of group elements, where $g_i \in G_i$ and g_i generates G_i.

Input: Security Parameter: 1^n, Group Element: $a \in G_n$, where $G_n \in G$.

Output: Exponent: x satisfying ${g_n}^x = a$, where $g_n \in g$

Just as in the PDLP, the group is not an input to the GPDLP problem. Instead, the group is determined by the security parameter 1^n. For a given n, the group is fixed to G_n where G_n is an element of an infinite sequence of groups $G_1, G_2, ...$

B. The Para-Discrete Logarithm Problem with an Advice String

By removing the group as an input to the problem, the PDLP more closely models fixed group applications. In applications where groups are generated on the fly and used once, the adversary gains no advantage through a precomputation; the precomputation can only be used once. In contrast, with fixed groups a precomputation can provide the adversary an advantage for all instances over the life of the cryptographic application.

We model this precomputation as an *advice string*. In computational complexity theory an advice string is an extra input to a computational problem that depends only on the length of the input. By the definition of the PDLP, the group is fixed for a given input length. This allows us to consider a group-specific computation as producing an advice string for the PDLP. (In the standard DLP setting, the precomputation could not be considered an advice string because it is dependent on an input to the problem, the specific group.)

We assert that a conservative approach to evaluating the security of a protocol is to consider an attack where the adversary has access to a precomputation based only on the protocol standard. In the case of a protocol with fixed groups, we should consider an adversary with access to a group-specific precomputation. Using the advice-string formalism allows us to better consider the difficulty of solving discrete logarithms once a group-specific precomputation has been completed. We can consider the time and space complexity of solving the instance separately from the time to create the precomputation.

C. Para-Discrete Logarithm Algorithms

In this section, we re-examine each algorithm from the previous chapter as a para-discrete logarithm solver. We want to analyze the complexity of the PDLP with an advice string. To assist this, we explicitly divide each algorithm into two sub-algorithms: the advice-generator (precomputation phase) and the instance-solver (search phase). The advice-generator performs a precomputation based only on the group and generator, not the specific problem instance. That allows us to treat the precomputation's output as an advice string for the PDLP. The instance-solver searches for the solution of a specific problem instance making use of the advice string. For each algorithm, we determine the asymptotic runtime of both phases and the size of the advice string.

1. Brute-Force Search and Precomputed Table Algorithms

We begin our re-examination, with the two simplest algorithms. In the brute-force search, there is no precomputation done. If we try to modify the brute-force search to have a precomputation, we end up with the precomputed table algorithm. The precomputed table algorithm very naturally divides into the two algorithms we are looking for. In the advice-generator algorithm, the table of all logarithms is built. In the instance-solver algorithm a single lookup into the table returns the discrete logarithm.

Algorithm 21 Precomputed Table: Advice Generator

Input: Cyclic Group: G, Generator: g
Output: Advice string: *hash* such that *hash*$[g^x] = x$ for $0 \leq x < N$

1: $b \Leftarrow 1$
2: **for** $x = 0$ to $N - 1$ **do**
3: hash$[b] \Leftarrow x$
4: $b \Leftarrow b \times g$
5: **end for**
6: **return** hash

Algorithm 22 Precomputed Table: Instance Solver

Input: Group Element: a, Advice string: *hash*
Output: Exponent: x such that $g^x = a$

1: $x \Leftarrow$ hash$[a]$
2: **return** x

The advice-generator will require N group multiplications to preform the precomputation. The asymptotic running time of the advice-generator is $O(2^n)$. The size of the advice string (the precomputed hash table containing N exponents) is exponential in n, $O(n2^n)$. After the precomputation, the instance solver requires a single table lookup to solve an individual discrete log.

2. Shank's Algorithm

The hash table built in Shank's algorithm is independent of a particular discrete logarithm instance, so the hash table can act as the advice string. That is, the advice-generator builds the hash table. The instance-solver then giant-steps from the input a until it reaches an element with a precomputed discrete logarithm in the hash table.

Algorithm 23 Shank's Algorithm: Advice Generator

Input: Cyclic Group: G, Generator: g, Number of exponents to precompute: X
Output: Advice string: *hash* such that *hash*$[g^i] = i$ for $0 \leq i < X$

```
1: b ⇐ 1
2: for i = 0 to X − 1 do
3:     hash[b] ⇐ i
4:     b ⇐ b × g
5: end for
6: return hash
```

Algorithm 24 Shank's Algorithm: Instance Solver

Input: Group Element: a, Advice string: *hash*, Number of precomputed logarithms: X
Output: Exponent: x such that $g^x = a$

```
1: b ⇐ a
2: y ⇐ 0
3: h ⇐ hash[b]
4: while g^h ≠ b do
5:     b ⇐ b × g^X
6:     y ⇐ y + 1
7:     h ⇐ hash[b]
8: end while
9: x ⇐ h − yX  mod N
10: return x
```

Now we consider the runtime of each algorithm. For general X, the advice-generator takes X group operations to generate an advice string with X entries of size n-bits, resulting in an advice string of $O(nX)$. The instance-solver takes, on average, $\frac{N}{2X}$ operations. For $X = 2^{\frac{n}{2}}$, both the advice generator and instance-solver take $2^{\frac{n}{2}}$ group operations. Therefore, the runtime complexity of both algorithms is $O(2^{\frac{n}{2}})$ with an advice string of $O(n2^{\frac{n}{2}})$.

The best choice of X varies depending on the particular trade-offs of a given application. A smaller choice for X means a quicker precomputation and a smaller advice string, but at the expense of a longer runtime for the instance solver. Likewise, a larger X means quicker instance solving, but at the expense of a larger advice-string and a longer runtime for the advice generator. Note that for $X = 1$ we have essentially the brute-force search, and for $X = N$ we have the precomputed table algorithm.

Given a desired number, k, of instances to solve in a particular group, we can select an X to minimize overall computation time. The total computation time is that of one precomputation

plus k instance computations,

$$f(X) = X + k\frac{N}{2X}.$$

To minimize $f(X)$, we find the positive zero of the derivative,

$$\frac{d}{dX}f(X) = 1 - \frac{kN}{2X^2} = 0,$$

$$\frac{kN}{2X^2} = 1,$$

$$X^2 = \frac{kN}{2},$$

$$X = \sqrt{\frac{kN}{2}}.$$

Therefore, the best choice of X to minimize computation time when solving k instances is $\sqrt{\frac{kN}{2}}$ or, in terms of the bit-length, n, $X = \sqrt{k}2^{\frac{n-1}{2}}$. This gives a total computation time of $\sqrt{2kN}$ and an average time per solution of $\frac{\sqrt{2kN}}{k} = \sqrt{\frac{2N}{k}}$.

3. Pollard's Rho Algorithm

As we examine Pollard's Rho algorithm, we see that it does not fit the two-phase pattern. The algorithm only requires a small amount of storage while running and does not make use of a precomputation. The random sequence depends on a particular instance, $a = g^x$, we are trying to solve. It it not immediately clear how a instance-independent precomputation could assist this algorithm. Using our terminology there is only an instance-solver algorithm and it uses no advice string.

However, [13] analyzes how work can be saved from each instance of the rho algorithm, speeding the solution of subsequent instances over the same group. They note that the table of distinguished points stored during computation of one logarithm becomes a table of known logarithms once that instance has a solution. Those distinguished points can be saved and used to assist the next instance and so on. The saved distinguished points are effectively a precomputation for solving the next instance.

This idea can be extended to create a useful advice string, a database of logarithms of distinguished points. The advice generator selects random exponents, i, to create random elements, g^i, and stores the discrete logarithm each time a distinguished point is found. Additional

distinguished points are found until the desired advice size has been reached. Let d be the number of distinguished point logarithms we precompute. In one extreme, where $d = 0$, we have the standard rho algorithm. At the other extreme, where d equals the total number of distinguished points in the group, we completely remove the benefit of finding cycles in the random sequence; the first time we reach a distinguished point, we can solve the logarithm. This extreme is clearly inferior to Shank's algorithm where precomputing each logarithm requires only a single group operation.

Algorithm 25 Pollard's Rho Algorithm: Advice Generator

Input: Cyclic Group: G, Generator: g, Order of g: N, Number of logarithms of distinguished points to precompute: d

Output: Advice string: *hash* such that *hash*$[a^{a_i}g^{g_i}] = (a_i, g_i)$ for some $a^{a_i}g^{g_i} \in D$

1: $D \Leftarrow$ a subset of distinguished points from G
2: // Randomly chose exponents and store logarithms of d distinguished points
3: **for** $j = 1$ to d **do**
4: **repeat**
5: $i \Leftarrow$ randomly selected exponent between 0 and $p - 1$
6: $s_i \Leftarrow g^i$
7: **until** $(s_i \in D)$
8: hash$(s_i) \Leftarrow (0, i)$
9: **end for**
10: **return** hash

Kuhn and Struik [13] show that computing a total of X logarithms in the same group takes time $\sqrt{2NX}$ and that the $X + 1$ logarithm can be computed in time $\sqrt{N/2X}$ for $X << N^{1/4}$.

We design our advice generator so that it creates the number of distinguished points, d, equivalent to having solved X logarithms. Because the results of Kuhn and Struik are limited to $X << N^{1/4}$, we let $X = N^{1/5}$. The advice generator will take time

$$T_{advice} = \sqrt{2NX} = \sqrt{2N(N^{1/5})} = \sqrt{2N^{6/5}} = \sqrt{2}N^{3/5}.$$

The instance solver will take time

$$T_{instance} = \sqrt{N/2X} = \sqrt{N/2N^{1/5}} = \sqrt{N^{4/5}/2} = \frac{1}{\sqrt{2}}N^{2/5}.$$

The size of advice depends on θ, the proportion of elements that are distinguished points. For the time estimate of the instance solver to be accurate, it must reach many distinguished points.

Algorithm 26 Pollard's Rho Algorithm: Instance Solver

Input: Group Element: a, Order of g: N, Advice string: *hash*
Output: Exponent: x such that $g^x = a$

// Search for a cycle in the random sequence $S = s_0, s_1, ...$ defined by Algorithm 16
success $\Leftarrow$ **false**
$D \Leftarrow$ a subset of distinguished points from G
while (success = **false**) **do**
  $i \Leftarrow 0$
  $a_i \Leftarrow$ randomly selected exponent between 0 and $p-1$
  $g_i \Leftarrow$ randomly selected exponent between 0 and $p-1$
  $s_i \Leftarrow g^{g_i} a^{a_i}$
  repeat
    $i \Leftarrow i+1$
    Calculate s_i, a_i, g_i applying Algorithm 16
  until $(s_i \in D)$
  // If we have already stored this point before
  if $((a_j, g_j) \Leftarrow \text{hash}(s_i))$ **then**
    success $\Leftarrow$ **true**
  else
    $\text{hash}(s_i) \Leftarrow (a_i, g_i)$
  end if
end while
$m \Leftarrow a_i - a_j \mod N$
$x \Leftarrow m^{-1}(g_j - g_i) \mod N$
return x

Thus we select $\theta = c/N^{2/5}$. The number of distinguished points stored equals the runtime of the advice-generator multiplied by the proportion of distinguished points,

$$T_{advice}\theta = \sqrt{2}N^{3/5}c/N^{2/5} = \sqrt{2}cN^{1/5}.$$

In terms of the bit-length n, the asymptotic runtime is $O(2^{\frac{3n}{5}})$ for the advice-generator and $O(2^{\frac{2n}{5}})$ for the instance-solver, given an advice-string of size $O(n2^{\frac{n}{5}})$.

4. Pollard's Kangaroo Algorithm

Pollard's Kangaroo method does not fit the two-phase model, but we can adapt it to this setting. Recall that in this method, there are two kangaroos. The wild kangaroo starts at the instance we are trying to solve. The tame kangaroo starts from a fixed point on the cycle. Since

the tame kangaroo's behavior is independent of a particular instance, we can step through the tame kangaroo in the advice-generator phase. Then in the instance-solver algorithm, we only need to step the wild kangaroo.

Algorithm 27 Pollard's Kangaroo Algorithm: Advice Generator

Input: Cyclic Group: G, Generator: g, Order of g: N, Mean step size: m
Output: Advice string: *hash* such that *hash*$[g^i] = i$ for some $g^i \in D$

// Select a small sequence of possible step sizes
$S \Leftarrow (s_0, s_1, \ldots, s_{k-1})$ where $s_i = 2^i$ and k such that the mean of the entries is m
$R \Leftarrow (r_0, r_1, \ldots, r_{k-1})$ where $r_i = g^{s_i}$
// Select a hash function to map a group element to a particular step size, s_i
$h(x) \Leftarrow$ hash function mapping G into the interval $[1..k]$
$D \Leftarrow$ a subset of distinguished points from G
// The "tame" kangaroo starts off at the beginning of the cycle
$x_t \Leftarrow g^0$
$d_t \Leftarrow 0$
while $(d_t < N)$ **do**
    // Step the tame kangaroo one hop
    $i \Leftarrow h(x_t)$
    $x_t \Leftarrow x_t r_i$
    $d_t \Leftarrow d_t + s_i$
    if $x_t \in D$ **then**
        // Store the exponent of the distinguished point in the hash table
        hash$(x_t) \Leftarrow d_t$
    end if
end while
return hash

In the advice-generator, the tame kangaroo steps through the entire cycle, building a hash table of all the distinguished points along its path. For our analysis we will use m for the mean step size of the values in set S and c for the mean distance between distinguished points. The time of the first phase will be $\frac{N}{m}$ and the storage will be $\frac{N}{mc}$.

In the instance-solver, the wild kangaroo steps through the cycle until it reaches a distinguished point stored in the advice string. To analyze the runtime we can break down the instance solver in to two stages:

1. The wild kangaroo must first land on the path of the tame kangaroo.
2. The wild kangaroo must then continue until it reaches a distinguished point.

In Stage 1, the wild kangaroo must land on the tame kangaroo's path. Given a mean step size,

Algorithm 28 Pollard's Kangaroo Algorithm: Instance Solver

Input: Group Element: a, Order of g: N, Advice string: *hash*
Output: Exponent: x such that $g^x = a$

// Use the same set S as in the advice-generator
$S \Leftarrow (s_0, s_1, \ldots, s_{k-1})$ where $s_i = 2^i$ and k such that the mean of the entries is m
$R \Leftarrow (r_0, r_1, \ldots, r_{k-1})$ where $r_i = g^{s_i}$
// Use the same hash function as in the advice-generator
$h(x) \Leftarrow$ hash function mapping G into the interval $[1..k]$
$D \Leftarrow$ a subset of distinguished points from G
// The "wild" kangaroo starts off from $a = g^x$
$x_w \Leftarrow a$
$d_w \Leftarrow 0$
success $\Leftarrow$ **false**
while (success = **false**) **do**
    // Step the wild kangaroo one hop
    $i \Leftarrow h(w_w)$
    $x_w \Leftarrow x_w r_i$
    $d_w \Leftarrow d_w + s_i$
    if $x_w \in D$ **then**
        // If we have already stored this point for a tame kangaroo
        if $(d_i \Leftarrow \text{hash}(x_w))$ **then**
            $x \Leftarrow d_i - d_w \bmod N$
            success $\Leftarrow$ **true**
        end if
    end if
end while
return x

m, each hop of the wild kangaroo has a $\frac{1}{m}$ chance of landing on the tame kangaroo's path. Thus the kangaroo will land on the path after an expected m hops.

In Stage 2, the wild kangaroo is on the path of the tame kangaroo and must reach a distinguished point. Given that one of every c elements is a distinguished point, the wild kangaroo will land on a distinguished point after an expected c hops.

Combining the times from both stages gives a total runtime of $m + c$ for the instance-solver. If we select $c = m$, the runtime becomes $2m$ and the size of the advice string is $\frac{N}{m^2}$. We can perform a trade-off between the runtime and the size of the advice string by varying m. One interesting trade-off is to balance the size of the advice string with the runtime,

$$\frac{N}{m^2} = 2m.$$

Solving for m,

$$m^3 = \frac{N}{2},$$

$$m = \sqrt[3]{\frac{N}{2}}.$$

This gives us a advice size and instance-solver time of $2m = \sqrt[3]{4N}$. The advice-generator time is

$$\frac{N}{m} = \frac{\sqrt[3]{2}N}{\sqrt[3]{N}} = \sqrt[3]{2}N^{\frac{2}{3}}.$$

In terms of n, the bit-length of N, the instance-solver runs in $O(2^{\frac{n}{3}})$ time with an $O(2^{\frac{n}{3}})$ size advice string. The advice-generator runs in $O(2^{\frac{2n}{3}})$ time.

5. Index Calculus Algorithm

The index calculus algorithm needs essentially no changes to match our desired two algorithm pattern. Phases 1 & 2 already use only the group description to create a precomputation result. Combined, the first two phases become the advice-generator. The table of logarithms of the factor base becomes the advice string, and final phase becomes our instance-solver algorithm.

The runtime of the advice generator will be the runtime of Phase 1 and Phase 2 of the standard index calculus algorithm,

$$t_{advice} = T_1 + T_2 = \frac{B}{\log B}(\log p/\log B)^{\log p/\log B}\log\log B + (\frac{B}{\log B})^2.$$

The runtime of the instance generator is that of Phase 3,

$$t_{instance} = T_3 = (\log p/\log B)^{\log p/\log B} L_B(\frac{1}{2}, \sqrt{2}).$$

The size of the advice string will be the size of the table of logarithms of the factor base,

$$nk = n\frac{B}{\log B}.$$

Looking at the structure of the algorithm, there is clearly a large imbalance between the runtime of the advice-generator and that of the instance-solver. The instance solver has to find only a single B-smooth integer, while the advice-generator must find $k + c \approx k$ B-smooth

Algorithm 29 Index Calculus Algorithm: Advice Generator

Input: Prime: p, Generator of $\mathbb{Z}_p^*$: g
Output: Advice string: *table* such that *table*$(i) = \log_g p_i$ for $0 \leq i < k$.

// Setup: Select a factor base, S
$B \Leftarrow$ a bound for the largest prime in the factor base
$S \Leftarrow (p_1, p_2, \ldots, p_k)$ where S contains all primes, $p_i < B$
// Find linear relations of the factor base
// Find a few more relations than the size of the factor base to ensure a unique solution
for $j = 0$ to $k + c$ **do**
  repeat
    $y \Leftarrow$ randomly selected exponent between 0 and $p - 1$
    $u \Leftarrow g^y \bmod p$
  until (u is B-smooth)
  // Find the factorization of u
  $u = \prod_{1 \leq i \leq k} p_i^{c_i}$
  // Take logarithms and store the linear relation
  $y = \sum_{1 \leq i \leq k} c_i \log_g p_i \pmod{p-1}$
end for

// Solve system of linear relations
Given the $k + c$ linear relations, solve for the k unknown discrete logarithms of the factor base.
Store the logarithms, such that table$(i) = \log_g p_i, 1 \leq i \leq k$
return table

integers. In addition, half the time of the advice-generator is spent solving the system of linear relations.

With B optimized to minimize the precomputation, $B = L_p(\frac{1}{2}, \frac{1}{2})$, the running time of the instance solver is $t_{instance} = L_p(\frac{1}{2}, \frac{1}{2})$ while the runtime of the advice-generator is $t_{advice} = L_p(\frac{1}{2}, 1)$. Recall

$$L_p(\alpha, c) = O(\exp((c + o(1))(\ln p)^{\alpha}(\ln \ln p)^{1-\alpha})),$$

Thus, asymptotically, the instance time runs in just the square root of the advice time.

The choice of the bound, B, allows tradeoffs between the size of the advice and the runtime of the instance solver. A larger B will result in a larger advice string, but make the search for B-smooth elements faster for the instance-solver.

Algorithm 30 Index Calculus Algorithm: Instance Solver

Input: Element of $\mathbb{Z}_p^*$: a, Advice string: *table*
Output: Exponent: x satisfying $g^x \equiv a \mod p$, where $0 \leq x < p-1$.

// Solve for the individual discrete logarithm
repeat
  $y \Leftarrow$ randomly selected exponent between 0 and $p-1$
  $u \Leftarrow ag^y \bmod p$
until (u is B-smooth)
// Find the factorization of u
$u = \prod_{1 \leq i \leq k} p_i^{c_i}$
// Take logarithms of both sides
$x + y = \sum_{1 \leq i \leq k} c_i \log_g p_i \pmod{p-1}$
$x \Leftarrow \sum_{1 \leq i \leq k} c_i \, \text{table}(i) - y \pmod{p-1}$
return x

D. Summary

In this section, we summarize the runtimes of the advice-generator and instance-solver algorithms for the para-discrete logarithm problem. For the algorithms that allow a time-memory trade-off, the formulas governing the trade-offs are shown in Table 4.

Algorithm	Advice-Generator Time	Advice Size	Instance-Solver Time
Shank's	X	nX	$N/2X$
Pollard's Rho	$\sqrt{2NX}$	cnX	$\sqrt{N/2X}$
Pollard's Kangaroo	N/m	nN/mc	$m+c$
Index Calculus	$\frac{B}{\log B}(\frac{\log p}{\log B})^{\frac{\log p}{\log B}} \log\log B + (\frac{B}{\log B})^2$	$n\frac{B}{\log B}$	$(\frac{\log p}{\log B})^{\frac{\log p}{\log B}} L_B(\frac{1}{2}, \sqrt{2})$

Table 4: Time-Memory Trade-Offs of Para-Discrete Logarithm Algorithms

The entries of Table 5 represent a specific trade-off point where advice size and instance time are roughly balanced. Each of the generic algorithms solve the GPDLP. The result from the precomputed-table algorithm shows that the GPDLP can be solved in constant time given an advice string exponential in size. More interestingly, using Pollard's Kangaroo algorithm, the GPDLP can be solved in $O(\sqrt[3]{N})$ operations with access to advice of size $O(\sqrt[3]{N})$ elements.

The index calculus algorithm places an upper bound on the complexity of the PDLP in $\mathbb{Z}_p^*$. The PDLP can be solved in subexponential time, $L_p(\frac{1}{2}, \frac{1}{2})$, with an advice string of subexponential size, $L_p(\frac{1}{2}, \frac{1}{2})$.

Algorithm	Advice-Generator Time	Advice Size	Instance-Solver Time
Precomputed Table	$O(2^n)$	$O(n2^n)$	$O(1)$
Shank's	$O(2^{\frac{n}{2}})$	$O(n2^{\frac{n}{2}})$	$O(2^{\frac{n}{2}})$
Pollard's Rho	$O(2^{\frac{3n}{5}})$	$O(2^n{}^{\frac{n}{5}})$	$O(2^{\frac{2n}{5}})$
Pollard's Kangaroo	$O(2^{\frac{2n}{3}})$	$O(n2^{\frac{n}{3}})$	$O(2^{\frac{n}{3}})$
Index Calculus	$L_p(\frac{1}{2}, 1)$	$L_p(\frac{1}{2}, \frac{1}{2})$	$L_p(\frac{1}{2}, \frac{1}{2})$

Table 5: Complexity of Para-Discrete Logarithm Algorithms

In our conservative model of security for protocols over fixed groups, we consider only the instance-solver time, assuming a group specific precomputation is available. Under this model, the cryptographic strength provided by fixed groups is significantly less than that of one-time groups. This disparity is demonstrated when we compare the instance-solver runtimes with the standard runtimes from the previous chapter.

By comparing the results of Table 5 with those of Table 3, we see that the discrete logarithm in a particular group is significantly easier given a group-specific advice string. In the case of the generic algorithms, the GDLP can be solved in $O(2^{\frac{n}{2}})$, but given an advice string of $O(n2^{\frac{n}{3}})$ can be solved in $O(2^{\frac{n}{3}})$ time using the kangaroo instance-solver algorithm. Similarly, the time to solve the DLP in $\mathbb{Z}_p^*$ with the index calculus algorithm improves from $L_p(\frac{1}{2}, 1)$ to $L_p(\frac{1}{2}, \frac{1}{2})$ when given an advice string of size $L_p(\frac{1}{2}, \frac{1}{2})$.

LIST OF REFERENCES

[1] L. Adleman. "A subexponential algorithm for the discrete logarithm problem with applications to cryptography." In *20th Annual Symposium on Foundations of Computer Science(SFCS 1979)*, pages 55–60. IEEE Computer Society, Los Alamitos, CA, USA, 1979. ISSN 0272-5428.

[2] D. Coppersmith, A. M. Odlyzko, and R. Schroeppel. "Discrete Logarithms in GF(p)," *Algorithmica*, 1(1):1–15, 1986.

[3] W. Diffie and M. Hellman. "New directions in cryptography," *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[4] T. El Gamal. "A public key cryptosystem and a signature scheme based on discrete logarithms." In *Proceedings of Advances in Cryptology (CRYPTO 84)*, number 196 in Lecture Notes in Computer Science, pages 10–18. Springer, New York, NY, 1985.

[5] M. Friedl, N. Provos, and W. Simpson. "Diffie-Hellman Group Exchange for the Secure Shell (SSH) Transport Layer Protocol." RFC 4419 (Proposed Standard), March 2006. Accessed March 25, 2010. http://www.ietf.org/rfc/rfc4419.txt, IETF.

[6] S. Goldwasser and S. Micali. "Probabilistic encryption," *Journal of Computer and Systems Sciences*, 28(2):270–299, 1984.

[7] S. Goldwasser, S. Micali, and R. Rivest. "A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks.", *SIAM Journal on Computing*, 17:281–308, 1988.

[8] D. M. Gordon. "A survey of fast exponentiation methods," *Journal of algorithms*, 27(1): 129–146, 1998.

[9] D. M. Gordon and K. S. McCurley. "Massively parallel computation of discrete logarithms." In *Advances in Cryptology CRYPTO 92*, Lecture Notes in Computer Science, page 312. Springer, New York, NY, 1993.

[10] D. Harkins and D. Carrel. "The Internet Key Exchange (IKE)." RFC 2409 (Proposed Standard), November 1998. Accessed March 25, 2010. http://www.ietf.org/rfc/rfc2409.txt, IETF. Obsoleted by RFC 4306, updated by RFC 4109.

[11] A. Hildebrand and G. Tenenbaum. "On Integers Free of Large Prime Factors.", *Transactions of the American Mathematical Society*, 296(1):265–290.

[12] S. Kent and K. Seo. "Security Architecture for the Internet Protocol." RFC 4301 (Proposed Standard), December 2005. Accessed March 25, 2010. http://www.ietf.org/rfc/rfc4301.txt, IETF.

[13] F. Kuhn and R. Struik. "Random walks revisited: Extensions of Pollard's rho algorithm for computing multiple discrete logarithms," *Lecture Notes in Computer Science*, pages 212–229, 2001.

[14] B. A. LaMacchia and A. M. Odlyzko. "Solving large sparse linear systems over finite fields." In *Advances in Cryptology*, pages 109–133. Springer, New York, NY, 1990.

[15] K. S. McCurley. "The discrete logarithm problem." In *Proceedings of Symposia in Applied Mathematics*, volume 42 of *Cryptology and computational number theory*, pages 49–74. American Mathematical Society, 1990.

[16] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of applied cryptography*. CRC press, 1996.

[17] D. Naccache and I. E. Shparlinski. "Divisibility, Smoothness and Cryptographic Applications," *Arxiv preprint arXiv:0810.2067*, 2008.

[18] A. M. Odlyzko. "Discrete logarithms in finite fields and their cryptographic significance." In *Proceedings of Advances in Cryptology (EUROCRYPT 84)*, volume 84 of *Lecture Notes in Computer Science*, pages 224–314. Springer, New York, NY, 1985.

[19] National Institute of Standards and Technology. "Digital Signature Standard (DSS)." Federal Information Processing Standard (FIPS), Publication 186-2, 2000.

[20] H. Orman. "The OAKLEY Key Determination Protocol." RFC 2412 (Informational), November 1998. Accessed March 25, 2010. http://www.ietf.org/rfc/rfc2412.txt, IETF.

[21] S. Pohlig and M. Hellman. "An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance (Correspondence)," *IEEE Transactions on Information Theory*, 24(1):106–110, 1978.

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

[23] J. M. Pollard. "Kangaroos, monopoly and discrete logarithms," *Journal of cryptology*, 13 (4):437–447, 2000.

[24] C. Pomerance. *Algorithmic Number Theory: Lattices, Number Fields, Curves and Cryptography*, volume 44 of *Mathematical Sciences Research Institute Publications*, chapter Smooth numbers and the quadratic sieve, pages 69–81. Cambridge University Press, 2008.

[25] O. Schirokauer, D. Weber, and T. Denny. "Discrete logarithms: the effectiveness of the index calculus method." In *Algorithmic Number Theory*, volume 1122 of *Lecture Notes in Computer Science*, pages 337–362. Springer, New York, NY, 1996.

[26] V. Shoup. "Lower bounds for discrete logarithms and related problems." In *Proceedings of Advances in Cryptology (EUROCRYPT 97)*, number 1233 in Lecture Notes in Computer Science, pages 256–266. Springer, New York, NY, 1997.

[27] P. C. Van Oorschot and M. J. Wiener. "Parallel collision search with cryptanalytic applications," *Journal of Cryptology*, 12(1):1–28, 1999.

[28] T. Ylonen and C. Lonvick. "The Secure Shell (SSH) Protocol Architecture." RFC 4251 (Proposed Standard), January 2006. Accessed March 25, 2010. http://www.ietf.org/rfc/rfc4251.txt, IETF.

[29] T. Ylonen and C. Lonvick. "The Secure Shell (SSH) Transport Layer Protocol." RFC 4253 (Proposed Standard), January 2006. Accessed March 25, 2010. http://www.ietf.org/rfc/rfc4253.txt, IETF.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudly Knox Library
   Naval Postgraduate School
   Monterey, California