Digital Signature Schemes

Nick Carruthers
Department of Math and Computer Science
Middlebury College
December 9, 1997

Abstract: This thesis will attempt to describe in detail the concepts of digital signatures and the related background issues. We will begin with a general introduction to cryptography and digital signatures and follow with an overview of the requisite math involved in cryptographic applications. We will then discuss in detail two digital signature schemes based on the discrete logarithm problem: the ElGamal scheme and its derivative, the United States Digital Signature Standard. Having done this, we will explore the problems and insecurities involved in their use.

1 Introduction

As long as there has been communication, there have been issues of privacy and authenticity. With the advent of the digital computer, these concerns have moved to the forefront. What many people do not realize is just how much information about our lives is stored on various computers spread across the country, or even the world. Most of this information is fairly innocuous, like your spending patterns at your friendly local grocery store (ever wonder why those "super bonus cards" were introduced?), but some is not so harmless. If someone were to break in to the computers of the IRS with anything more than mischievous intentions, much damage could be done. As much as we'd like to not believe it, Big Brother *is* watching, and everything he knows about you is stored on a computer somewhere. What's more, Big Brother is quite benign compared to some of the people or organizations that would like to get hold of your information. Thus, it would be in your best interests to make it as hard as possible for these people to access it.

In this thesis, I will discuss the concepts of *Digital Signatures*, a field of cryptography that "may prove to be one of the most fundamental and useful inventions of modern cryptography [11]." Specifically, I will examine the *ElGamal Signature Scheme* [9] and its variant, the *United States Digital Signature Standard (DSS)* [20]. I will attempt to fully describe these schemes, as well as discuss the security issues involved in their use. In this introduction, I will present some background information important to the rest of the paper.

1.1 Standards

Since one of the main purposes behind cryptography is communication, we will be discussing protocols of communication between several parties. In order to make this paper a bit more readable, I will be using a set of names to represent the participants in a cryptographic protocol. Not to break from the apparent standard in cryptography papers, I will be using the ubiquitous Alice and Bob. My group of pseudonyms is a subset of the one presented in [24].

Alice	First participant in all the protocols (the signer)
Bob	Second participant in all the protocols (the verifier)
Eve	Eavesdropper (passive attacker)
Mallory	Malicious attacker (active or passive)
Walter	Warden who guards Alice and Bob in some protocols

Table 1.1

In addition, the term *message* will come up frequently. When we are dealing with digital signature schemes, as we will be doing for the bulk of this paper, it will mean exactly that, a message sent from one person to another, usually Alice to Bob. There will

be other cases where I will use it to mean anything one might want to encrypt, be it a word processing document or the formula for a super-efficient gasoline alternative. The algorithms we will be presenting are blind to the nature of what is being encrypted; it's all just binary data to them. Whenever I mention the DSS, I mean the United States Digital Signature Standard. DSA means the Digital Signature Algorithm, which is the algorithm specified by the DSS. When the term *plaintext* and *ciphertext* appear, we take them to mean the message before and after encryption, respectively.

1.2 Digital Signatures – A Quick Overview

Handwritten signatures have been used for years to prove a certain document came from, or was approved by a specific person. You always sign credit card bills to prove that you are who you say you are. In the event of a dispute, the credit card company or the store issuing the receipt could use your signature to prove you made the purchase. There are several reasons why a signature is compelling [24]:

- The signature is authentic.
- The signature cannot be forged.
- The signature is not reusable.
- The signed document can not be altered.
- The signature cannot be repudiated.

With conventional signatures, someone with a little whiteout and a lot of time could invalidate at least one of these statements. What's more, with a little practice it is fairly easy to copy someone's signature.

Attaching signatures to a message on a computer could be a very handy thing. If you sent someone a signed e-mail message with sensitive instructions on say, the handling of your will, they could take this message with a reasonable assumption of authenticity. Unfortunately, the statements listed above become even more problematic with you are dealing with computers. It is trivial to modify a computer file after it has been signed, and unlike the case of whiteout, there will be no evidence of this modification [24]. Also, since the digital signature cannot possible be a physical part of the message it signs, we somehow need to "bind" the signature to the message [29] (fulfilling the "reusable" clause above). We will take all these issues into consideration when we discuss digital signatures in more detail in Chapter 3.

1.3 Concepts of Security

What do we mean by security and how do we provide it? Bruce Schneier makes the following analogy in his textbook *Applied Cryptography* [24]:

> If I take a letter, lock it in a safe, hide the safe somewhere in New York, then tell you not to read the letter, that's not security. That's obscurity. On the other hand if I take a letter and lock it in a safe, and then give you the safe along with the design specifications of the safe and a hundred identical safes with their combinations so that you and the world's best safecrackers can study the locking mechanism—and you still can't open the safe and read the letter—that's security.

In years past, most cryptography schemes were based on what Schneier refers to as *obscurity*; their security was based on the fact that no one really knew how they worked. The algorithm was essentially a sealed black box (I suppose the color doesn't really matter) that one fed a plaintext into one end and got a ciphertext out the other. If anyone ever managed to pry the box open, the algorithm would be completely broken. This black-box approach may have worked with the Caesar Cipher two thousand years ago, but with the resources available to even the least savvy computer user nowadays, this is not a secure way to go about designing an algorithm. Granted the Caesar Cipher is an overly simple example, but no matter how obtuse your method, someone will eventually pick it apart (i.e. pry open the box) and cryptanalyze it, resulting in a complete break of the cryptosystem. Thus, no self-respecting modern cryptologist will ask others to put faith into his or her algorithm unless the design is free for anyone who wants to analyze it.

It is upon this paradigm of open design that most cryptographic algorithms are based these days. In general, no one will trust it if they can't see how it works and be satisfied that it can not be broken in a feasible amount of time. It should be mentioned that a cryptosystem with perfect secrecy does exist. The *Vernam One-time Pad*, invented by Gilbert Vernam in 1917 for encrypting telegraph messages provides this kind of security. For a formal description of perfect security, see [29]. Unfortunately, this system is not practical due to the size of the keys and the difficulty involved in communicating them, similar to our discussion in section 1.4.2 below.

1.4 Public-Key Cryptography

When most people who are not versed in cryptographic concepts think of encryption, they generally think of *symmetric cryptography* (though they may not know that terminology!). That is, cryptography where you encrypt a message using a secret key, and use that exact same key (or one easily derived from it) to decrypt the message. For a general description of symmetric algorithms, see Figure 1.1. However, all the schemes discussed in this paper will concern *asymmetric cryptography*, better known as *public-key cryptography*. Figure 1.2 below gives a general description. The basic idea behind public-key cryptography is as follows: Rather than having one key to encrypt messages, each user has two related keys. One is used for encryption, published in a

General Description of Symmetric (Private-Key) Cryptography

k is the key agreed on beforehand by Alice and Bob,

m is the message to be sent from Alice to Bob,

E_k is the encryption algorithm using key k,

D_k is the decryption algorithm using key k

Alice encrypts message M using $E_k(m)$ and sends it to Bob.

Bob decrypts using $D_k(E_k(m))$ and recovers the message M.

Figure 1.1

General Description of Asymmetric (Public-Key) Cryptography

m is the message to be sent from Alice to Bob,

E_B is the encryption algorithm using Bob's public key,

D_B is the decryption algorithm using Bob's private key.

Alice encrypts the message *m* using $E_B(m)$ and transmits the results to Bob

Bob decrypts using $D_B(E_B(m))$ and recovers the message *m.*.

Figure 1.2

public directory (perhaps a server on the Internet), the other is the private key that is used to decrypt messages that were encrypted using the public key. Alice, wishing to send Bob a message, must look up his public encryption key and encrypt her message using it. Once she has encrypted the message, neither she nor anyone else (including Bob) can decrypt it using the public encryption key. Bob, using his private key that only he should know can easily retrieve the original message from the ciphertext sent to him. These algorithms are based on *one-way functions*, which we will now discuss.

1.4.1 One-Way Functions

The security we discussed in section 1.3 with regards to public-key cryptography is largely based on the notion of a one-way function. That is, a function that is easy to compute in one direction, but very difficult to compute going the other direction. By "difficult," we mean that given *f(x)*, it would take millions of years and all the computers in the world, to compute *x* [24]. The concept of "hard" problems will be discussed in more detail later in section 2.1. An example (not quite real world) of a one way function would be an infinitely deep well. It is very easy to throw something down the well, but another matter altogether to get it back out. In [24], Schneier asserts that in the strictly mathematical sense, there is no proof that one-way functions actually exist. We can come close, though, for there are many functions that seem to be one-way. For example:

- Computing square roots in a finite field.
- Factoring in a finite field.
- Calculating discrete logs in a finite field.

Calculating in a finite field essentially means calculating modulo a number. Thus, given a modulus *q*, your results are restricted to between 0 and *q – 1*. This will be discussed more in Chapter 2.

One-way functions in their pure form are of no use in cryptography, because there would be no way to decrypt a message. Take the unlikely infinitely deep well for example. Alice could write a message to Bob and throw it down the well, but all Bob could do would be to look longingly down into the abyss, wondering what it was that Alice had written. Was it a love letter? Was it a check for a million dollars? He would never know. What Bob needs is something called a *trapdoor*. Perhaps he could build an infinitely long elevator that could bring the message back up to the surface, designed in such a way that only he could operate it. Ignoring the obvious lack of respect for the

laws of physics, this could work as a *trapdoor one-way function*. Anyone could easily throw a message down the well (analogous to *f(x)*), but only Bob, who has access to the elevator, can retrieve it (calculate *x* from *f(x)*). Public-Key cryptosystems utilize these principles; anyone who wants to send Bob a message encrypts it using his public key, and he retrieves it using his infinitely long elevator, also known as his private key.

1.4.2 Symmetric vs. Public-Key

Without going into too much detail, problems arise when using symmetric algorithms for communications between Alice and Bob. The main complication is what is known as *key exchange*; in order for Bob to read messages sent by Alice, they must first agree on a *session key* to encrypt all communication between them for a certain period of time (a *session*). This exchange of keys is very vulnerable to an eavesdropper, because the key must pass over potentially lengthy channels (generally electronic ones) in either unencrypted form, or encrypted using a publicly known key. The latter is essentially equivalent to no encryption at all for all but the most casual of eavesdroppers. Say Alice and Bob want to have a conversation, Alice is in Boston and Bob is on business in Kuala Lumpur, Malaysia. Alice comes up with a random session key and transmits it to Bob. Eve, who is sitting at her computer in Boise, intercepts Alice's message to Bob and can now do what she does best; eavesdrop on all of Alice and Bob's communications. Public-key cryptography, to some extent circumvents the key exchange problem, though there are still issues. For example, the decidedly politically incorrectly named *Man-in-the-Middle Attack*. Since this is not a paper dealing with key exchange in depth, I will simply refer you to Schneier [24] or Stinson's [29] excellent survey textbooks on cryptography.

In a similar vein, there are also problems with the sheer number of keys needed for symmetric algorithms when communicating in a large network [24]. Assuming each pair of users in the network want to be able to communicate privately, a separate key is needed for each pair. Thus, a network of n users will require $n(n-1)/2$ keys. It is easy to see that as the network grows, the number of keys grows at a fairly quick rate. The same network using public-key cryptography would need only $2n$ keys (one public and one private key for each user) for all pairs of users to communicate in private.

Probably one of the strongest arguments *against* public-key algorithms is their bulkiness, both in storage space and computational power required. One popular system, RSA (named after its inventors, Rivest, Shamir, and Adleman), runs about a thousand times slower than the symmetric DES (Data Encryption Standard) and uses keys that are about ten times as large [8]. This is basically due to the need for computational security while using trapdoor one-way functions. In order to provide the necessary measure of security, the keys must be on the order of hundreds of digits long. Hardly like an eight-digit password that you can memorize!

Overall, it is difficult to say which is better, because they are designed to solve different sorts of problems. Symmetric algorithms are best at encrypting data, and as we discussed above, it is many times faster than their public-key counterparts. Public-key algorithms, though slow, are best for key management [24] and digital signatures. As

amazing as public-key cryptography seems, it would not be a wise choice to forgo symmetric cryptography altogether. For our purposes, i.e. digital signatures, public-key cryptography is undoubtedly the right choice.

1.4.3 Applicability to Digital Signatures

So what does all this have to do with digital signatures? One thing that I did not mention in my earlier discussion of public-key cryptography (see Figure 1.2 above) was the fact that, for many public-key systems, you can reverse the public and private keys of the scheme. That is, Bob can encrypt a message using his private key, and Alice, or anyone knowing Bob's public key, can decrypt the message. Using the same terminology as Figure 1.2, we have:

$$E_B(D_B(m)) = M = D_B(E_B(m))$$

This allows Bob to prove that he knows his secret key without actually divulging it. This figures very heavily into the concepts of digital signatures, and will be discussed in greater detail in Chapter 3. This property applies to the RSA cryptosystem, allowing it to be used both for signing and encryption. The two public-key signature schemes we will be discussing however, are designed exclusively for signing. Therefore, E_B and D_B are the secret signing key and the public verification key, respectively.

2 Mathematical Background

Cryptography has always relied on mathematics of some sort, whether it be adding three to an "A" to create the ciphertext "D" (as in the Caesar Cipher) or computing trapdoor one-way functions with numbers hundreds of digits long. In this section, I will present some of the concepts and algorithms necessary for a better understanding of the digital signature algorithms discussed in Chapter 3 and attacks against them in Chapter 4.

2.1 Complexity Theory

Complexity theory concerns itself with the *computational complexity* of algorithms, giving us an upper bound on their computing requirements. The computational complexity of an algorithm involves both the CPU time requirements and the memory requirements of said algorithm. Even if an algorithm is very fast (i.e. its time requirements are low), if it requires a prodigious amount of memory, it will not be practical. One of the advantages of describing the complexity of an algorithm in this manner is that it is not tied to any particular computer system. The complexity of an algorithm is the same whether you are running it on a Commodore 64 or a Cray supercomputer. It will become apparent that seemingly huge differences in CPU power are negligible compared to the order-of-magnitude complexity [24].

When determining the complexity of an algorithm, we are measuring how fast the computing requirements grow as the size of the input grows. We express this in "Big O" notation, usually using the variable n as the size of the input. We only pay attention to the fastest growing element of the algorithm, ignoring all constants and slower growing elements. So if you discover that an algorithm takes $2n^3 - 3n + 1$ steps to complete, we say that its complexity is $O(n^3)$. For this algorithm, if the length of input n increases by one bit, the running time will increase by approximately $3n^2 + 3n + 1$ time units. In Table 2.1, Schneier [24] illustrates the various classes of algorithms and examples of their running times. From this table, it is apparent that if you have a cryptographic algorithm that requires exponential time to break, you can rest assured that, barring any fantastic leaps in the computer industry, it will not be broken.

Class	Complexity	# of Operations for $n = 10^6$	Time at 10^6 O/S
Constant	O(1)	1	1 μsec.
Linear	O(n)	10^6	1 sec.
Quadratic	O(n^2)	10^{12}	11.6 days
Cubic	O(n^3)	10^{18}	32,000 yrs.
Exponential	O(2^n)	$10^{301,030}$	$10^{301,006}$ times the age of the universe

Table 2.1

The first four classes in the table above are said to be *polynomial*, i.e. they can be solved in polynomial time. As you can see, problems requiring exponential time grow at astronomical rates. There is a subset of exponential algorithms, ones that are called *superpolynomial*. Their complexities are O($c^{f(n)}$) where c is a constant and $f(n)$ is more complex than constant but less than linear [24]

What it all boils down to is that for cryptographic purposes, we want to design algorithms where all currently known cracking algorithms are of superpolynomial-time complexity or greater. Algorithms that require polynomial running time are easy; ones that can not run in polynomial (i.e. superpolynomial or exponential) time are not. This is what is meant when we say a problem is "hard." If our algorithm is believed to be of superpolynomial order, provided we use key large enough, it will not be computationally feasible to break. According to Schneier [24], "advances in computational complexity may some day make it possible to design algorithms for which the existence of polynomial-time cracking algorithms can be ruled out with mathematical certainty." As it stands, we will have to make do with the "all currently known algorithms" assertion.

2.2 Primality

Simply put, prime numbers are of great importance in public-key cryptography. Both of the digital signature schemes that we will be discussing rely heavily on very large prime numbers for their security. This is a vast field in and unto itself, and one could easily write a whole thesis on prime number generation and primality testing. For this paper, though, it will suffice to present a few of the basic issues. An integer is said to be prime if its only factors are 1 and itself. We can trivially throw out all even numbers greater than two. As there are an infinite number of integers, there are also an infinite number of primes (see [25] for a proof). Two numbers are said to be *relatively prime* if their greatest common divisor is equal to 1. That is, they share no factors in common other than 1 [24]. This is very important in computing inverses mod a number (see section 2.3.2). There has been a method for computing *gcd(a, b)* for over two thousand years now, first written down by Euclid in his book, *Elements*, around 300 BC, and is thus known *Euclid's Algorithm*. Whoever originally came up with it is not getting their due, for it is believed that the algorithm could be 200 years older [24]. See Figure 2.1 for the algorithm in C.

```
Long FindGCD(long x, long y)
{
     long remainder;
     while (y != 0)
     {
          remainder = x % y;
          x = y;
          y = remainder;
     }
     return x;
}
```

Figure 2.1 – Euclid's GCD Algorithm

All the primes used in cryptography have to come from somewhere, and are usually generated randomly, then tested using a probabilistic algorithm such as the Solovay-Strassen or Miller-Rabin algorithms. For our purposes, I will refer you to [24] or [29] for detailed discussions of these methods. We will assume that the primes used in the digital signature schemes we discuss in Chapter 3 are valid. Unfortunately, not all primes are created equal relative to cryptography. I will show in section 4.7 that you must take great care when choosing primes for use in these schemes as to avoid potential insecurities.

2.3 Modular Arithmetic

Modular arithmetic is a fairly fancy way of stating a simple concept. Any reasonably well educated nine-year-old can perform modular arithmetic to some extent. If it is ten o'clock in the morning, simply ask them what time it will be in five hours. They will perform the calculations and (hopefully) tell you "three o'clock in the afternoon." The calculations they did were the following:

$$(10 + 5) \bmod 12 = 15 \bmod 12 = 3 \bmod 12$$

This is simply known as "clock arithmetic" [24] and is something we do nearly every day. For the equation above you would say that fifteen is *congruent* to three, mod twelve. You write this formally as

$$15 \equiv 3 \pmod{12}$$

You could also say that three is the *residue* of fifteen mod twelve. The *complete set of residues* mod n is simply all the integers from 0 to $(n-1)$ inclusive, i.e. the set of integers that you could get when performing arithmetic mod n.

Modular arithmetic is used extensively in cryptography for several reasons. Calculating many functions mod n is a difficult problem, for example, the discrete logarithm case with ElGamal and the DSA. Computers are also finite machines, and performing calculations mod n for all intermediate steps helps to restrict the range of the results. For a k- bit modulus, the intermediate results of most simple arithmetic will be no more than $2k$-bits long [24]. Even the previously mentioned example of the Caesar Cipher works in *mod 26*. In this simple case, a constant k is added to each number to produce the ciphertext, with letters at the end of the alphabet wrapping back to the beginning. Thus, if $k = 3$, the plaintext 'Y' would become 'B' in the ciphertext. The calculation $c = m + k \bmod 26$ is being performed, where c is the ciphertext and m is the plaintext. Solving addition mod n is not that difficult of a problem, so we will focus on slightly more secure methods.

2.3.1 Fast Exponentiation

It is apparent that when performing exponentiation, i.e. $a^x \bmod n$, we would not want to do x multiplications and then take the mod of the result; this would have an undesirably high complexity. Specifically, it will have a $O(n)$ complexity that is

exponential in terms of the bits of n. In addition, the intermediate values would quickly get way out of hand, overflowing the finite bounds of the computer. It is easy to see that x in binary represents x as the sum of powers of two. Using an example from [24], 25 in binary is 11001, thus $29 = 2^4 + 2^3 + 2^0$. Thus

$$a^{25} \bmod n = (a * a^{24}) \bmod n = (a * a^8 * a^{16}) \bmod n$$
$$= (a * ((a^2)^2)^2 * (((a^2)^2)^2)^2) \bmod n = ((((a^2 * a)^2)^2)^2 * a) \bmod n$$

If you add a *mod n* in at every step, we have succeeded in our goal of limiting the range of the intermediate values to twice the bit length of the exponent x. Using this method, called *addition chaining*, we can reduce the number of operations to about $1.5 * k$, where

```
Long FastExponentiation(long a, long x, long n)
{
    long p = 1;

    while (z > 0)
    {
        if (z & 1)              // If the rightmost bit is set
            p = (p * a) % n;    // then update the sum.
        z >>= 1;                // Shift 'z' 1 bit to the right.
        a = (a * a) % n;        // Update the multiplier.
    }
    return p;
}
```

Figure 2.2 – Fast Exponentiation

k is $(\log_2 x) + 1$, or the number of bits in x [24]. This is a polynomial-time algorithm in the linear class, and thus will not become much slower as the length of the exponent increases. For a representation in C (with C++ comments), see Figure 2.2.

2.3.2 Inverses Mod a Number

The multiplicative inverse of a number is simply a value that results in one when you multiply it by the original number. In the world of non-modular arithmetic, finding the multiplicative inverse of an integer is trivial; simply take its reciprocal. You hardly need to prove that, for all integers x not equal to zero, the following holds:

$$x * \frac{1}{x} = 1$$

Things get a bit hairier when you are dealing with inverses modulo a number. Trying to find $a^{-1} \equiv x \pmod n$ is significantly harder than in the non-modular case. For one thing, not all numbers will have an inverse modulo another number. Generally, the equation $a^{-1} \equiv x \pmod n$ has a unique solution only if *gcd(a, n)* = 1, i.e. they are relatively prime [24] (see section 2.2). One of the most common ways of calculating the inverse of a number mod n is the *Extended Euclidian Algorithm*. Figure 2.3 below, taken from [29], gives a description of this algorithm. Schneier makes the observation that the algorithm

is "iterative and can be slow for large numbers [24]." The average number of divisions performed by the Extended Euclidian Algorithm was show by Knuth [17] to be

$$.843 * \log_2(n) + 1.47$$

There are other ways of computing inverses mod a number, such as the *Euler Totient Function*, also known as the *Euler Phi Function*. This method is somewhat specialized and applies more to the RSA public-key system, which we will not be discussing.

$n_0 = n$
$b_0 = b$
$t_0 = 0$
$t = 1$
$q = \left\lfloor \frac{n_0}{b_0} \right\rfloor$
$r = n_0 - q * b_0$
while $r > 0$ **do**
    $temp = t_0 - q * t$
    if $temp \geq 0$ **then**
        $temp = temp \bmod n$
    if $temp < 0$ **then**
        $temp = n - ((-temp) \bmod n)$
    $t_0 = t$
    $t = temp$
    $n_0 = b_0$
    $q = \left\lfloor \frac{n_0}{b_0} \right\rfloor$
    $r = n_0 - q * b_0$
end while
if $b \neq 1$ **then**
    b has no inverse modulo n
else
    $b^{-1} = t \bmod n$
end if

Figure 2.3 – The Extended Euclidian Algorithm

2.3.3 Chinese Remainder Theorem

The Chinese remainder theorem allows you to solve a system of congruences in different moduli. In general, if $n = p * q$ where p and q are prime, then a congruence of the form $x \equiv g \pmod{n}$ can be represented by the two congruences $x \equiv a \pmod{p}$ and

$x \equiv b \pmod{q}$. In the cryptographic setting, we often need to go in the reverse direction. That is, given $x \equiv a \pmod{p}$ and $x \equiv b \pmod{q}$, we would like to find a single congruence mod *n* that is consistent with the originals mod *p* and mod *q*. Schneier [24] states the general theorem, discovered by the first-century Chinese mathematician, Sun Tse, as follows: If the prime factorization of *n* is $p_1 * p_2 * ... * p_t$, then the system of equations

$$x \equiv a_i \pmod{p_i}, \text{ where } 1 \le i \le t$$

has a unique solution *x*, where *x* is less than *n*. So, for an arbitrary $a < p < q$ where *p* and *q* are primes, there exists a unique $x < pq$ such that

$$x \equiv a \pmod{p} \text{ and } x \equiv b \pmod{q}$$

In order to solve this, you fist find *u* such that

$$u = q^{-1} \bmod p$$

You then compute the following:

$$x = (((a - b) * u) \bmod p) * q + b$$

This can easily be illustrated with an example, also from [24]. Suppose $a = 2, b = 4$, $p = 3$, and $q = 5$. That is, we are trying to solve for *x* where we have the equations $x \bmod 3 = 2$ and $x \bmod 5 = 4$. We first calculate $u = 5^{-1} \bmod 3$ to be 2, then simply plug into the Chinese remainder theorem equation. We get

$$(((2 - 4) * 2) \bmod 3) * 5 + 4 = 14,$$

which is easy to check for correctness.

2.4 Generators

Schneier gives the following definition for generators in [24]: If *p* is a prime, and *g* is less than *p*, then *g* is a generator mod *p* if

for each *b* from 1 to $p - 1$, there exists some *a* where $g^a \equiv b \pmod{p}$.

In other words, every number in the complete set of residues (0 excepted) of *p* can be expressed as $g^a \pmod{p}$. For example, *p* = 11 has generators 2, 6, 7, and 8. 3 is not a generator because there is no solution to the equation

$$3^a \equiv 2 \pmod{11}.$$

It is not easy to test whether a given number is a generator mod *p* unless you know the factorization of *p* – 1, in which case it is relatively easy to solve. Again, from [24]:

> Let $q_1, q_2, ..., q_n$ be the distinct prime factors of *p* – 1. To test whether a number *g* is a generator mod *p*, calculate
>
> $$g^{(p-1)/q} \bmod p$$
>
> for all values of $q = q_1, q_2, ..., q_n$. If, for any value of *q*, that number equals 1, then *g* is not a generator. Otherwise, *g* is a generator mod *p*.

Generators will come up later in our discussion of the DSS in section 3.4, because one of the parameters of the DSA must be a generator mod one of the other parameters.

2.5 Quadratic Residues

Quadratic residues will come in to play in our discussion of the *subliminal channels* present in digital signature schemes (see section 4.8). Given a prime p, and a between 1 and $p - 1$ inclusive, a is a quadratic residue mod p if you can find an x such that

$$x^2 \equiv a \pmod{p}$$

has a solution. A *quadratic nonresidue* is simply a number that does not satisfy the above equation for any x.

2.5.1 The Legendre Symbol

The *Legendre symbol* allows us to efficiently determine whether a number a is a quadratic residue mod p, so long as p is prime [24]. So, given p as a prime greater than 2, and any integer a, the Legendre Symbol is calculated by

$$L(a, p) = a^{(p-1)/2} \bmod p .$$

There are three possible values that this can result in:

- 0 if a is divisible by p.
- 1 if a is a quadratic residue mod p.
- -1 if a is a quadratic nonresidue mod p.

2.6 The Discrete Logarithm Problem

In our introduction, we discussed one-way functions in the abstract (see section 1.4.1); we will now describe the one-way function that is used for both digital signature schemes we will be discussing. Our one-way function is exponentiation mod p, as discussed in section 2.3.1. Stinson states that given a suitable prime p, exponentiation mod p is a one-way function [29]. Calculating the inverse of an exponentiation mod p is what is known as the *Discrete Logarithm Problem*. That is, knowing a, b, and n, find x where $a^x = b \pmod{n}$ [24]. This is a hard problem, and as of yet, there is no known polynomial-time algorithm to solve it [29]. There are algorithms to solve it in special cases, which we will discuss in section 4.4. We can consider this as a one-way function because, as we showed in section 2.3.1, it is relatively easy to exponentiate modulo a number, but it is believed to be very hard to do the inverse.

2.6.1 Galois Field

When we are operating modulo a prime p, we are working in what is called a *Galois Field*, written as GF(p). The Galois field is a *finite* field, meaning that the results of any calculations performed within it will fall inside a well defined set (0 through $p - 1$

in our case). This keeps numbers a finite size [24] (good, since computers are finite machines) and also has the added benefit that any $q \in GF(p), q \neq 0$ has a unique inverse. This is easy to see, because the only stipulation for $a^{-1} \equiv x \pmod{n}$ to have a unique solution is that $gcd(a, n) = 1$ (see section 2.3.2). Since p is prime, and GF(p) is composed of the residues of p, then every integer in GF(p) is trivially relatively prime to p.

3 Digital Signatures

We discussed the concept of Digital Signatures in general in section 1.2; we will now attempt to formalize these concepts before we go on to present the two examples that we will be focusing on.

3.1 Definition of a Signature Scheme

Stinson [29] formalizes a signature scheme as a five-tuple (P, A, K, S, V) with the following conditions:

1. P is a finite set of possible messages
2. A is a finite set of possible signatures
3. K, the keyspace, is a finite set of possible keys
4. For each $k \in K$, there is a signing algorithm $sig_k \in S$ and a corresponding verification algorithm $ver_k \in V$. Each $sig_k : P \to A$ and $ver_k : P \times A \to \{true, false\}$ are functions such that the following equation is satisfied for every message $x \in P$ and for every signature $y \in A$.

$$ver(x, y) = \begin{cases} \text{true if } y = sig(x) \\ \text{false if } y \neq sig(x) \end{cases}$$

In addition, for every $k \in K$, the secret function sig_k and public function ver_k should be polynomial-time. It should be computationally infeasible for Mallory to forge Alice's signature on a message. It is impossible for a signature scheme to be unconditionally secure, since Mallory can try all possible signatures $y \in A$ for a message $x \in P$ using the public algorithm *ver*. Given sufficient time, Mallory will find a y that matches the message x, and thus have forged Alice's signature for the message x. We want to make it so that the number of possibilities Mallory must check is so large that there is no way he will find a match before the end of time. In any case, we would like for a digital signature scheme to conform to the stipulations laid out in section 1.2.

3.2 Role of Hash Functions

A hash function maps a large collection of messages into a small set of message digests, digests that can be used to produce a fixed-length digital signature that depends on the whole message [3]. In other words, a hash function takes in a message of arbitrary length and outputs a fixed length "fingerprint" of that message (see Figure 3.1). Both of the schemes we will discuss make use of hash functions to sign a relatively small (160 bits in the case of DSA) message that is representative of the entire message. There are two main requirements we have for a hash function used in our digital signature schemes: that it be *one-way* and *collision free*. The concept of one-way-ness with respect to hash functions is similar to the concept of a one-way function in public-key cryptography (see section 1.4.1), but there is no requirement for a trapdoor. Given a message m, it should be easy to compute the its hash value, h, but given h, it should be hard to compute a m

such that H(m) = h. In addition, given m, it is hard to find another message m' such that $H(m) = H(m')$. The collision free stipulation adds one more requirement: it should be hard to find two random messages m and m' such that $H(m) = H(m')$.

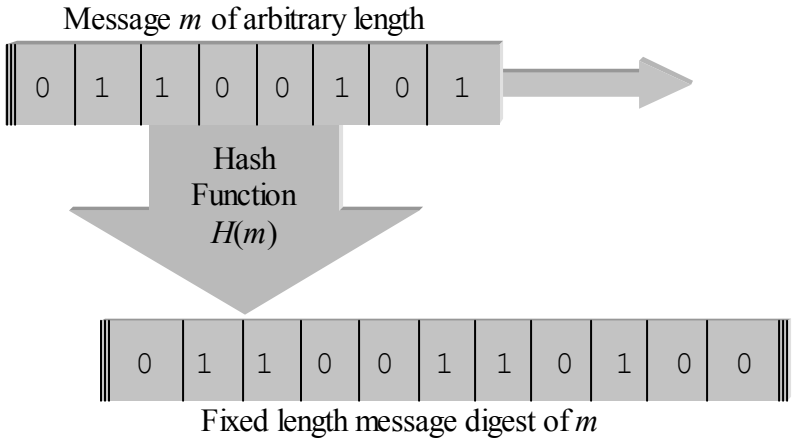

Figure 3.1 – Hash Functions

Hash functions are useful in signature schemes for several reasons. First of all, public-key signing algorithms tend to be very slow, so taking the hash value of a large message and signing the (small) resulting value is much more efficient than simply signing the whole message. Secondly, if you wanted to add multiple signatures to a document, without the use of hash functions, the resulting signed message would be many times the size of the original. If you did the same thing, only signing the hash value of the message, you would only be adding a small (320 bit for DSS) overhead for each signature added. We want our hash function to be collision free so that "any change to a message in transit will, with very high probability, result in a different message digest, and the signature will fail to verify [19]."

There is no particular hash function specified for the ElGamal Scheme, but the DSS requires a hash function called the Secure Hash Algorithm (SHA) [19]. We will fully describe the SHA in section 3.4.5.

3.3 ElGamal Digital Signature Scheme

The ElGamal Signature scheme was first proposed in 1985, and is fully described in [9]. Unlike RSA, it is a scheme designed exclusively for signing not for encryption. In fact, it can be used for encryption, but due to the random k parameter, it is non-deterministic. This scheme is the basis for the Digital Signature Algorithm, which will be described in the next section. Its security is based on the previously discussed discrete log problem. The following description is adapted from [29].

Parameter		
	p	A Prime such that the discrete log problem in the finite field GF(p) is hard
	g	A generator mod p
	x	A number less than p
	y	$y \equiv g^x \bmod p$

The values p, g, and y are the public key, and x is the secret key, and stay constant throughout the scheme. The values p and g can be shared among a group of users. For Alice to sign the message m, she first picks a random number k such that k is relatively prime to $p-1$. The signature signing algorithm sig_k outputs the double (r,s) where

$$r = g^k \bmod p$$
$$s = (m - xr)k^{-1} \bmod (p-1)$$

Alice sends the triple (m,r,s) along to Bob. In order to verify the signature, Bob must check to make sure that the following congruence holds:

$$y^r r^s \equiv g^m \bmod p$$

Thus

$$ver_k(m,r,s) = \text{true} \Leftrightarrow y^r r^s \equiv g^m \bmod p$$

Owing to the random parameter k, the ElGamal scheme is non-deterministic. That is, there are many possible valid signatures for any given message m.

3.3.1 An Example

We will illustrate the scheme with a simple example, modified from [29]. We use the m that we do for a specific reason, as will be shown in section 4.3.

$$p = 467, g = 2, x = 127$$

Thus

$$y = g^x \bmod p$$
$$= 2^{127} \bmod 467$$
$$= 132$$

So, Alice's public key is $(p = 467, g = 2, y = 132)$, and her private key is $x = 127$. Suppose Alice wants to sign the message $m = 101$ and chooses the random value $k = 213$. Note that gcd(213, 466) = 1 and $213^{-1} \bmod 466 = 431$. She then calculates the pair (r,s) to be the following:

$$r = 2^{213} \bmod 467 = 29$$
$$s = (101 - 127 * 29)431 \bmod 466 = 16$$

Alice sends the triple (m, r, s) on to Bob, who can easily verify the signature by calculating

$$132^{29}29^{16} \equiv 378 \pmod{467}$$

Since

$$2^{101} \equiv 378 \pmod{467}$$

Bob has confirmed that the triple is a valid signature for the message m from Alice.

We will now go on to describe the DSA, a variant of the ElGamal scheme. The security of both schemes will be discussed in Chapter 4.

3.4 The Digital Signature Standard (DSS)

The Digital Signature Algorithm (DSA) is part of the National Institute of Standards and Technology's (NIST) Digital Signature Standard (DSS). The DSA "authenticates the integrity of the signed data and the identity of the signatory" and is "intended for use in electronic mail, electronic fund transfer, electronic data interchange, software distribution, data storage, and other applications which require data integrity assurance and data origin authentication [20]." It is "applicable to all Federal departments and agencies for the protection of unclassified information" and is available for adoption to private and commercial organizations on a royalty-free basis [20]. The standard was first proposed in 1991.

Reaction to the proposal was anything but subdued. The prominent cryptographer Martin Hellman was "deeply concerned by faults in the technical specifications of the proposed DSS and by its development process [13]." By "development process," he was most likely referring to the role of the NSA. Many people were worried that the National Security Agency (NSA), who had a large part in designing the algorithm, would build a trapdoor into the algorithm, thus allowing them to break the scheme. This paranoia is to be expected, considering the track record of the NSA in the past. Concerns were heightened by the fact that the NSA would not allow the design and selection process to be public. A group called "Computer Professionals for Social Responsibility" filed suit against NIST, claiming the Freedom of Information Act (see [30]).

There was also a lot of complaining from RSA Data Security Inc., the owners of the RSA algorithm. This was mainly a result of the fact that they stood to lose a lot of money if a standard other than RSA, especially one that was royalty-free, were to be implemented. What's more, RSA was already the *de facto* standard, and many companies had already invested millions of dollars in it. They were understandably wary about losing this investment, and campaigned for the adoption of ISO 9796, an international standard for digital signatures that uses RSA [24]. In addition, there are questions about whether DSS infringes on certain patents. Schneier discusses this in some detail in [24]. In any case, the selection process was completed and the DSS went into effect on December 1, 1994.

3.4.1 Modifications to the ElGamal Scheme

The DSA modifies the ElGamal scheme in several interesting ways that we will discuss before we present the actual algorithm. These modifications are discussed fully in [29]; I will attempt to summarize them here. Since the security of the ElGamal scheme rests in the discrete logarithm problem, a very large modulus p must be used; at least 512 bits, but to ensure security into the future, the length of p should be 1024 bits. The problem with the ElGamal scheme is that the use of a 512-bit modulus leads to a 1024 bit signature (since both r and s are calculated mod p). This is not acceptable for use in some applications where a short signature is desired, such as smart cards. The memory

on such devices is very precious, and every byte saved keeps the price down. The DSS modifies the ElGamal scheme in such a way that a 160-bit message is signed using a 320-bit signature, but the computations are done using a modulus p that is between 512 and 1024 bits.

The first change to the ElGamal scheme we make is flipping the "-" to a "+" in the definition of s, resulting in

$$s = (m + xr)\,k^{-1} \bmod (p-1)$$

The verification condition changes accordingly to the following:

$$g^m y^r \equiv r^s \pmod{p}$$

If $(m + xr)$ and $(p-1)$ are relatively prime, then $s^{-1} \bmod (p-1)$ exists, and we can modify the above verification condition to the following:

$$g^{ms^{-1}} y^{rs^{-1}} \equiv r \pmod{p}$$

The "major innovation in the DSS" as presented in [29] is as follows. Suppose that p is a 160-bit prime such that q divides $(p-1)$, and g is a qth root of 1 modulo p. To construct this g, we let h be a generator mod p, and define $g = h^{(p-1)/q} \bmod p$. Thus, y and r will also be qth roots of 1. It follows that any exponent of g, y, and r can be reduced modulo q without affecting the new verification condition. Notice that r appears not only as an exponent on the left side, but also not as an exponent on the right side. Thus, if r is reduced modulo q, then we must also reduce the whole left side of the new verification function modulo q. Just as an aside, the DSS was designed exclusively for signing, but Schneier shows in [24] that through some creative toying with the parameters, it can used for both ElGamal and RSA encryption. A description of the DSA follows.

3.4.2 Description of DSA

To sign a message m, Alice first picks a random number k, such that $k < q$, and then computes the pair (r, s) as follows:

$$r = (g^k \bmod p) \bmod q$$
$$s = (k^{-1}(m + xr)) \bmod q$$

As with the ElGamal scheme, she then sends the triple (m, r, s) to Bob. In order to verify the signature, Bob computes the inverse of s and stores it (for efficiency's sake) in a variable.

$$w = s^{-1} \bmod q$$

He then computes the two exponents of the verification function as follows:

$$e_1 = (mw) \bmod q$$
$$e_2 = (rw) \bmod q$$

Finally, he makes sure the following equation holds:

$$((g^{e_1} * y^{e_2}) \bmod p) \bmod q = r$$

If the equation does hold, then he has verified that it is indeed a valid signature of Alice on the message *m*. For a proof of this verification condition, see Appendix 1 of the DSS specification [20].

In [29] Stinson points out that it is necessary that *s* not be congruent to zero mod *q*, since in order to verify the signature, we need to compute the value $s^{-1} \bmod q$. Thus, if Alice ends up computing a value where $s \equiv 0 \pmod q$, she should simply reject it and start over using a different random *k*. There is only a 1 in 2^{160} chance of this ever happing, so it's not really a concern.

3.4.3 Example

We will now present a small example, also from [29]. We will take $q = 101$ and $p = 78q + 1 = 7879$ (notice *q* divides $p - 1$). 3 is a generator mod 7879, so we can take

$$g = 3^{78} \bmod 7879 = 170.$$

Alice now picks a private key $x = 75$ and computes her public key *y* like so:

$$y = g^x \bmod 7879 = 4567$$

Alice wants to sign the message $m = 22$; she chooses a random number $k = 50$ and finds the inverse to be $k^{-1} \bmod 101 = 99$. She then calculates (r, s):

$$r = (170^{50} \bmod 7879) \bmod 101 = 2518 \bmod 101 = 94$$
$$s = (22 + 75 * 94)\ 99 \bmod 101 = 97$$

Alice then sends the triple (22, 94, 97) on to Bob, who performs the following computations to verify the signature:

$$w = 97^{-1} \bmod 101$$
$$e_1 = 22 * 25 \bmod 101 = 45$$
$$e_2 = 94 * 25 \bmod 101 = 27$$
$$(170^{45} 4567^{27} \bmod 7879) \bmod 101 = 2518 \bmod 101 = 94 = r$$

Thus, Bob has verified that the signature came from Alice.

3.4.4 Speedups

All public-key algorithms tend to be relatively slow, but one of the major complaints leveled against the DSA is that it is quite slow compared to RSA. Schneier [24] asserts that while signature generation speeds are the same, the verification process can be 10 to 40 times slower with DSA. To speed up DSA, he proposes adding two precomputations to the process. Notice that *r* depends on the random value *k*, but not on the message itself. A DSA implementation could create a string of random *k* values, and compute and store their corresponding *r* values. Obviously, *p* and *q* must remain constant throughout the lifetime of these precomputed values, but *p* and *q* generally don't change, so this is not really an issue. You could also compute $k^{-1} \bmod q$ for each of the *k* values

you come up with. So when Alice wants to sign a message, she uses one of these precomputed values. It is interesting to note that these precomputations only speed up the signature generation process, which Schneier already said was equivalent to RSA. These speedups in no way affect the verification computations, which appears to be the biggest slowdown to the scheme.

3.4.5 The Secure Hash Algorithm (SHA)

The message *m* we have been referring to up until now is really the output of the SHA run on the original message. In the specification for the DSS [20], NIST requires the use of the Secure Hash Algorithm, laid out in [19]. It is based on a hash function proposed by Ron Rivest called MD4, but is more complicated and produces a longer hash value. According to [19], the SHA takes any message of length $< 2^{64}$ bits and outputs a 160-bit message digest. The algorithm is fairly simple, just a long series of simple bit operations on the message. It is designed in such a way that every bit in the message has some effect on the resulting message digest. The algorithm works like this:

The message is padded so that it becomes a multiple of 512 bits long. Let's say *k* is the length of message *m* in bits. You first append a 1, then $((512 * z) - k - 64 - 1)$ zeros, where

$$z = \lceil k / 512 \rceil$$

That is, append enough zeros to make the message 64 bits short of a multiple of 512. In the 64 bits left, you append a 64-bit representation of the length of the original message. Once you have padded the message, you initialize five 32-bit variables as follows:

$$H_0 = 0\text{x}67452301$$
$$H_1 = 0\text{xEFCDAB89}$$
$$H_2 = 0\text{x}98\text{BADCFE}$$
$$H_3 = 0\text{x}10325476$$
$$H_4 = 0\text{xC3D2E1F0}$$

These five variables will eventually be concatenated into the 160-bit output of the algorithm. The main loop runs from $t = 0$ to 79 . Before describing the main loop, we will set forth the functions and constants used at each iteration.

- For $0 \le t \le 19$ $\begin{cases} f_t(B,C,D) = ((B \wedge C) \vee (\neg B \wedge D)) \\ K_t = 0\text{x}5\text{A}827999 \end{cases}$
- For $20 \le t \le 39$ $\begin{cases} f_t(B,C,D) = B \oplus C \oplus D \\ K_t = 0\text{x}6\text{ED}9\text{EBA}1 \end{cases}$
- For $40 \le t \le 59$ $\begin{cases} f_t(B,C,D) = ((B \wedge C) \vee (B \wedge D) \vee (C \wedge D)) \\ K_t = 0\text{x}8\text{F}1\text{BBCDC} \end{cases}$
- For $60 \le t \le 79$ $\begin{cases} f_t(B,C,D) = B \oplus C \oplus D \\ K_t = 0\text{xCA}62\text{C}1\text{D}6 \end{cases}$

For each message block M_i, we perform the following algorithm (The "<<<" represents a circular bit shift):

a) Transform M_i from 16 32-bit words $\{m_0 ... m_{15}\}$ to 80 32-bit words $\{w_0 ... w_{79}\}$.

$$w_j = \begin{cases} m_j \text{ for } 0 \le j \le 15 \\ ((w_{t-3} \oplus w_{t-8} \oplus w_{t-14} \oplus w_{t-16}) <<< 1) \text{ for } 16 \le j \le 79 \end{cases}$$

b) Let $a = H_0, b = H_1, c = H_2, d = H_3, e = H_4$

c) For $t = 0$ to 79 do

$$TEMP = (a <<< 5) + f_t(b,c,d) + e + w_t + K_t$$
$$e = d;\ d = c;\ c = (b <<< 30);\ b = a;\ a = TEMP$$

d) Let $H_0 = H_0 + a;\ H_1 = H_1 + b;\ H_2 = H_2 + c;\ H_3 = H_3 + d;\ H_4 = H_4 + e$

After all the blocks in the padded message have been processed with the above algorithm, concatenating the five 32-bit *H* values into one 160-bit value forms the message digest.

The NIST standard specifies an alternate method of computation for use when memory is at a premium, like on smart cards, for instance. They claim that it saves 64 32-bit words, but at the cost of slower execution. For a description of this method, see section 8 of [19]. Appendix A and B of the same document present examples of the SHA in all its glory.

As I mentioned previously, SHA is an enhancement of Ron Rivest's MD4 hashing algorithm. Ron Rivest did some enhancing of his own to produce the MD5 algorithm, but while he made the motivations for each change to MD4 public, NIST did not. You can refer to [24] for a list of those changes and how they compare to the ones made for SHA. According to Schneier [24], there are no known cryptographic attacks against SHA, and [3] confirms this. In addition, due to its 160-bit hash, it is not as susceptible to brute force attacks.

3.4.6 Parameter Generation

Up until now, no mention has been made of how we are to come up with appropriate values for the parameters of the DSA. The specification of the DSS [20] lays out methods for generating both suitable primes and random or pseudo-random numbers. For the latter, the document proposes several methods in its Appendix 3. In section 4.6, we will see that the choice of random numbers for the DSA is very important to its security. For primality testing, they suggest the use a simplified version of the Miller-Rabin algorithm (see [29]). They claim that running this algorithm *n* times will produce a false prime with probability no greater than $1/4^n$. They then go on to propose a specific algorithm to generate appropriate *p* and *q* values. Remember that we need two primes:

$$\{p \mid 2^{L-1} < p < 2^L, L = 512 + 64j, 0 \le j \le 8\}$$
$$\{q \mid 2^{159} < q < 2^{160}, q \text{ divides } p - 1\}$$

We pick a desired length *L* (according to the above restrictions), and then let $L - 1 = 160n + b$, where $0 \le b < 160$. We then proceed according to the following steps:

[a] Let *SEED* equal an arbitrary bit string of at least 160 bits.
Let *g* equal the length of *SEED* in bits. ($g = \log_2 SEED + 1$)

[b] $U = \text{SHA}(SEED) \oplus \text{SHA}((SEED + 1) \bmod 2^g)$

[c] $q = U \vee 2^{159} \vee 1$. This ensures that $2^{159} < q < 2^{160}$.

[d] Check to see if *q* is prime. If not, go back to step (a).

[e] $counter = 0$, $offset = 2$

[f] For $k = 0...n$, let $V_k = \text{SHA}((SEED + offset + k) \bmod 2^g)$

[g] Let $W = V_0 + 2^{160} V_1 + ... + 2^{160(n-1)} V_{n-1} + 2^{160n} (V_n \bmod 2^b)$ and
$X = W + 2^{L-1}$.

[h] $c = X \bmod 2q$, and $p = X - (c - 1)$. Note that $p \equiv 1 \bmod 2q$.

[i] If $p < 2^{L-1}$, go to step [j]. If not, check to see if *p* is prime. If *p* passes the primality test, we have succeeded, so go to step [l].

[j] $counter = counter + 1$; $offset = offset + n + 1$.

[k] If $counter \geq 4096$, start over. Otherwise, go back to step [f].

[l] Save the value of *SEED* and the variable *counter*.

The whole point of the public prime generation algorithm was to allay concerns about the use of "weak" primes in DSS. If a public authority hands you primes *p* and *q*, as well as the values of *SEED* and *counter* that were used to generate the primes, you can run the algorithm yourself to ensure that *p* and *q* were randomly generated. The use of the one-way hash function SHA prevents someone from figuring out a *SEED* and *counter* by working backwards from a "weak" *p* and *q* [24]. In section 4.7, we will discuss the implications of the poor choices of *p* and *q*.

3.5 Derivative Schemes

There have been many schemes proposed that use the signature schemes described above. In [4], a method is set forth for accelerating DSS signature operations in the setting of smart cards. It concerns using an insecure server to perform off-card computation of the exponentiations used in the DSA. Since the card cannot implicitly trust the server, it must verify all computations performed by the server to ensure that the results are valid. Much research has been done to find more efficient versions of the ElGamal scheme and the DSS. Both [14] and [15] propose a "highly efficient" generalized ElGamal scheme. In a recent paper from MIT and IBM [10], a system was proposed for threshold DSS signatures. A threshold signature scheme is one where you divide a secret (in this case, the signing key) into *n* shares in such a way that $m < n$ of those shares are needed to reconstruct the secret [24]. In particular, the scheme specifies a parameter $t < n/2$ such that $m = 2t + 1$ can collaborate to produce a valid DSS signature, but no *t* corrupted participants can forge a signature.

4 Security Issues

We will now give a review of the security issues related to the signature schemes we discussed in the previous chapter. In general cryptography, the goal of the attacker Mallory is to break an encryption scheme so that he may read any messages encrypted using that scheme. In the context of digital signatures, Mallory is still trying to break the system, but to a different end. Michael Conn, Chief of Information Policy at the NSA said the following in a letter to *The Houston Chronicle* [7]: "The DSS does not encrypt any data. The real issue is whether the DSS is susceptible to someone forging a signature and therefore discrediting the entire system. We state categorically that the chances of anyone—including NSA—forging a signature with the DSS when it is properly used and implemented is infinitesimally small." We will see that most of the supposed insecurities arise from improper use or dishonest implementation of the DSS.

4.1 Classification of Attacks

It is apparent that the ultimate goal of Mallory is to forge a signature on a message of his choice. While this is the most dangerous form of forging, it is by no means the only one. There are three basic kinds of attacks, as set forth in [11].

- *Key-Only Attack*: Mallory knows only the public key of the signer and therefore only has the capability of checking the validity of signatures given to him.
- *Known Signature Attack*: Mallory knows the public key of the signer and has seen message/signature pairs chosen and produced by the actual signer. This is the minimum Mallory can do.
- *Chosen Message Attack*: Mallory can ask the signer to sign a number of messages of Mallory's choice. A notary public that signs documents on demand would be vulnerable to this type of attack.

In addition, there are different levels of success for Mallory, also from [11]

- *Existential Forgery*: Mallory succeeds in forging the signature of one message, but he cannot necessarily choose the message.
- *Selective Forgery*: Mallory succeeds in forging the signature of some message of his choice.
- *Universal Forgery*: Mallory is able to forge the signature for any message, but still does not know the secret signing key of the real signer.
- *Total Break*: Mallory can compute the signer's secret key, and thus can forge his signature on any message.

4.2 Existential Forgery Using ElGamal

It is well known that the ElGamal scheme cannot be shown to be fully secure because it is susceptible to existential forgery [21]. There are two known methods, both of which are described in [29]. The first forgery involves simultaneously choosing *r*, *s*, and *m*. Mallory chooses *i* and *j* as integers where $0 \le i, j \le p-2$, and *j* is relatively prime to $p-1$. Mallory then computes the following:

$$r = g^i y^j \bmod p$$
$$s = -rj^{-1} \bmod (p-1)$$
$$m = -rij^{-1} \bmod (p-1)$$

Thus, Mallory has computed a valid ElGamal signature pair (r, s) on the message *m*. We can prove this by performing the verification algorithm as follows:

$$\begin{aligned} y^r r^s &\equiv y^r (g^i y^j)^{-rj^{-1}} \pmod p \\ &\equiv y^r g^{-irj^{-1}} y^{-r} \pmod p \\ &\equiv g^{-irj^{-1}} \pmod p \\ &\equiv g^m \pmod p \end{aligned}$$

Thus, Mallory has created a valid signature for the message *m*. There is another way for Mallory to forge a signature where he begins with a signature pair (r, s) that resulted from Alice signing the message *m*. Mallory chooses integers *h*, *i*, and *j* such that $0 \le h, i, j \le p-2$ and $(hr - js)$ is relatively prime to $p-1$. He then computes the following:

$$r' = r^h g^i y^j \bmod p$$
$$s' = sr'(hr - js)^{-1} \bmod (p-1)$$
$$m' = r'(hm + is)(hr - js)^{-1} \bmod (p-1)$$

Again, to verify, we plug our results into the equation $y^{r'}(r')^{s'} \equiv g^{m'}$. According to Stinson, this is straightforward, but tedious, so we will not go completely through the verification. While both of these methods allow Mallory to forge a valid signature, he has no control over what message he can sign. Thus, they don't really seem to pose a threat to the security of the system [29].

4.3 Universal Forgery Using ElGamal

In [6], Bleichenbacher shows a way in which Mallory could perform a known-signature attack if the verifier is not careful. Specifically, the verifier must make sure that $1 \le r < p$ is satisfied. If Bob accepts signatures without checking that condition, Mallory could easily forge a signature on a message of his choice, without knowing the secret key of the signer. Say Mallory has seen a valid signature (r_1, s_1) from Alice on the message m_1. It is interesting to note that m_1 must be relatively prime to $p-1$, which Bleichenbacher does not mention in his paper. This is so because of the inversion mod

$p-1$ performed in the calculation of u. This does not pose a huge problem for Mallory, he would just need to wait for Alice to sign a message m_1 that was relatively prime to $p-1$. Once Mallory has an appropriate message and signature, he can construct a new signature (r_2, s_2) on a message m_2 of his choosing. He first sets $u \equiv m_2 (m_1)^{-1} \bmod p-1$, which implies the following:

$$g^{m_2} \equiv g^{m_1 u} \equiv y^{r_1 u} (r_1)^{s_1 u} \pmod{p}.$$

Mallory now forms (r_2, s_2) by setting $s_2 \equiv s_1 u \pmod{p-1}$ and finding a r_2 such that $r_2 \equiv r_1 u \pmod{p-1}$ and $r_2 \equiv r \pmod{p}$ by using the Chinese Remainder Theorem. Notice that by the Chinese Remainder Theorem, $N = (p-1) * p$, so r_2 can be greater than p.

4.3.1 Example

No example was given, so I came up with one based on the example of the ElGamal scheme given in section 3.3.1. Recall that $p = 467$, $g = 2$, $x = 127$, and $y = 132$ and the signature for message $m_1 = 101$ was $(r_1, s_1) = (29,16)$. Also note that 101 is relatively prime to 466 and $101^{-1} \bmod 466 = 263$. Mallory gets this signature from Alice and wants to sign the message $m_2 = 211$ and convince Bob that Alice signed it. He first computes $u \equiv 211 * 263 \bmod 466 = 39$, so $s_2 \equiv 16 * 39 \bmod 466 = 158$. Mallory must now find a r_2 such that both of the following hold:

$$r_2 \equiv 29 * 39 \bmod 466 \equiv 199 \bmod 466$$
$$r_2 \equiv 29 \bmod 467$$

Mallory can easily solve for r_2 by using the Chinese Remainder Theorem:

$$v = 467^{-1} \bmod 466 = 1$$
$$r_2 = (((199 - 29) * v) \bmod 466) * 467 + 29 = 79419$$

He then sends the message 211 along with the signature $(79419, 158)$ to Bob, who stupidly does not check to make sure $1 \le r < p$ holds. Instead, he goes ahead and performs the verification computations as follows:

$$132^{79419}\ 79419^{158} \bmod 467$$
$$= 99 * 289 \bmod 467 = 124$$

And since $2^{211} \bmod 467 \equiv 124$, Mallory has fooled Bob into thinking that Alice signed the message $m_2 = 211$. Thus, it is very important for an implementation of the ElGamal scheme to check to make sure that r is not larger than p.

4.4 Brute Force Attacks

Just to recap, the discrete logarithm problem is given as follows: given a prime p, g a generator mod p, and $0 \le y \le p-1$, find a unique exponent $0 \le x \le p-2$ such that $g^x \equiv y \pmod{p}$. Clearly, signature schemes based on the discrete logarithm problem can be broken in $O(p)$ time and $O(1)$ space simply by trying all the possible a's. If you

precompute all possible values of g^x, and sort the ordered pairs $(x, g^x \bmod p)$ according to their second coordinates, the discrete log problem can be solved in $O(1)$ time with $O(p)$ precomputation and $O(p)$ memory [29].

4.4.1 Shank's Algorithm

Shank's algorithm, described in [29], is a non-trivial algorithm for solving the discrete logarithm problem, using the precomputation method described above. The algorithm works like this:

1. Define *m* equal to the ceiling of $\sqrt{p-1}$.
2. Compute $g^{mj} \bmod p$ where *j* ranges from 0 to $m-1$.
3. Sort the *m* ordered pairs $(j, g^{mj} \bmod p)$ according to their second coordinates to obtain a list L_1.
4. Compute $yg^{-i} \bmod p$ where *i* ranges from 0 to $m-1$.
5. Sort the *m* ordered pairs $(i, yg^{-i} \bmod p)$ according to their second coordinates to obtain a list L_2.
6. Find a pair $(j, z) \in L_1$ and a pair $(i, z) \in L_2$ (i.e. a pair with identical second coordinates).
7. Define $\log_g y = mj + i \bmod (p-1)$.

It is easy to see that if $(j, z) \in L_1$ and $(i, z) \in L_2$, then the following is true:

$$g^{mj} = z = y^{-i}$$
$$g^{mj+i} = y$$

This algorithm can be implemented in $O(m)$ time and $O(m)$ memory, which given a prime *p* large enough, would still require massive amounts of computing power and time. You can see [29] for a simple example of this algorithm.

4.4.2 Pohlig-Hellman Algorithm

We will now describe the Pohlig-Hellman algorithm for computing discrete logs, leaving the proof to [29]. This algorithm is important in the issues discussed in sections 4.7.1 and 4.8.4. The algorithm only works when $p-1$ factors to a series of distinct primes. We state this formally as

$$p-1 = \prod_{i=1}^{k} (p_i)^{c_i}$$

where the $p_i's$ are distinct primes. For instance, if $p = 29$, $p-1 = 28 = 2^2 7^1$. Since we can compute $a \bmod (p_i)^{c_i}$ for each *i*, where $1 \le i \le k$, we can use the Chinese Remainder Theorem to compute $a \bmod (p-1)$. Each step of the Pohlig-Hellman algorithm calculates

$$\log_g B \bmod q^c.$$

The algorithm performs the following steps for each i with $1 \le i \le k$:

1) Set $q = p_i$ and $c = c_i$. So for $p = 29$ and $i = 2$, $p = 7$ and $c = 1$.
2) Compute $z = B^{(p-1)/q} \bmod p$.
3) If $z \equiv 1 \pmod{p}$, then $a_0 = 0$. If not, successively compute $r^i \bmod p$ where $i > 0$ and $r = g^{(p-1)/q} \bmod p$ until $r^i \equiv z \bmod p$. Then we set $a_0 = i$.
4) If $c = 1$, we are finished. Otherwise, we repeat the following three steps $c - 1$ more times, where $1 \le j \le c - 1$.
5) Define $B_j = B_{j-1} g^{-a_{j-1}}$.
6) Compute $z = B_j^{(p-1)/q^{j+1}} \bmod p$.
7) Similar to step 3, find an i such that $r^i \equiv z \bmod q$. We then set $a_j = i$.
8) Define $d = \sum_{i=0}^{c-1} a_i q^i$ and set up the congruence $a \equiv d \pmod{q^c}$.

At the end of running the algorithm k times, we are left with a system of k modular equations of the form $a \equiv d_i \pmod{(q_i)^{c_i}}$ that we can use the Chinese Remainder Theorem to solve. See [29] for an example of the algorithm. For an extensive discussion, including a complexity analysis, see [12].

4.5 The Random Parameter k

Both the ElGamal Scheme and the DSA employ a random value k when signing a message. In the case of ElGamal, k must be relatively prime to $(p-1)$ due to the inverse mod $(p-1)$ involved in the calculation of the value of s. With the DSA, k must simply be less than q. It also must be relatively prime to q, but since q is a prime, simply requiring that $k < q$ is adequate (all residues mod q are relatively prime to q when q is a prime). In both cases, it is essential to the security of the system that the k values remain secret, and that a single k value never be reused. Stinson [29] shows that if k is ever revealed, it is very simple to compute the secret key x. Recall the equation for the s portion of the (r, s) signature pair:

$$s = (m - xr)k^{-1} \bmod (p-1)$$
$$ks = (m - xr) \bmod (p-1)$$
$$xr = m - ks \bmod (p-1)$$
$$x = (m - ks)r^{-1} \bmod (p-1)$$

Thus, if Mallory ever discovers the secret k value, he can very easily compute Alice's secret key and have performed a complete break of the system. The situation is almost identical in case of the DSA, with only a few subtle differences due to the small differences in the signature computations (see section 3.4.1).

The situation is a bit more complicated, though no less dire, if two messages are signed using the same k value. Mallory doesn't even need to know the k to perform the attack. It is easy for him to tell if the same value of k has been used for a pair of

signatures, because the *r*-value will be the same (remember $r = g^k \bmod p$). Once again, a good explanation of the attack on the ElGamal scheme is found in [29]. Suppose (r, s_1) is a signature on m_1 and (r, s_2) is a signature on m_2. Thus, according to the ElGamal verification condition, we have both of the following equations:

$$y^r r^{s_1} \equiv g^{m_1} \pmod{p}$$
$$y^r r^{s_2} \equiv g^{m_2} \pmod{p}$$

Thus

$$g^{m_1 - m_2} \equiv r^{s_1 - s_2} \pmod{p}$$

Since $r = g^k$, we substitute in to get the following equation:

$$g^{m_1 - m_2} \equiv g^{k(s_1 - s_2)} \pmod{p}$$

This is equivalent to

$$m_1 - m_2 \equiv k(s_1 - s_2) \pmod{p-1}.$$

You then let $d = \gcd(s_1 - s_2, p-1)$. Since *d* obviously divides $(p-1)$ and $(s_1 - s_2)$, it follows that *d* divides $m_1 - m_2$. We then make the following definitions:

$$m' = \frac{m_1 - m_2}{d}, \quad s' = \frac{s_1 - s_2}{d}, \quad p' = \frac{p-1}{d}.$$

We can now define the congruence as

$$m' \equiv ks' \pmod{p'},$$

and since s' and p' are relatively prime, we can compute

$$e = (s')^{-1} \bmod p'.$$

Then we can determine the value of *k* modulo p' as

$$k = m'e \bmod p',$$

which yields *d* candidate values for *k*:

$$k = m'e + ip' \pmod{p-1} \text{ for some } i, \ 0 \le i \le d-1.$$

It is a simple matter to test all *d* candidate values to see which one fulfils the condition

$$r \equiv g^k \bmod p.$$

Once Mallory has found a *k* value that fits the above equation, he simply uses the previously described method for finding the secret key *x* when the secret random value *k* is known. Mallory has once again achieved a total break of the ElGamal system. Schneier [24] claims (without proving it) that the same attack is valid against the DSA.

4.6 Random Number Weaknesses

In a paper published earlier this year, Bellare, Goldwasser, and Micciancio [5] illustrate "the high vulnerability of the DSS to weaknesses in the underling random number generation process." They claim that if the random numbers used in the DSS are generated with a linear congruential pseudorandom number generator (LCG), then the secret key can be recovered after seeing only a few signatures, resulting in a total break of the system. Here, I will attempt to give an overview of the attack they describe. The DSS does specify several methods for random number generation as discussed in section 3.4.6, none of which this attack applies to. It is important nonetheless to show the extent that the security of the DSS relies on the underlying random number generation process.

4.6.1 Linear Congruential Generators

A LCG is a pseudo-random number generator based on the linear recurrence $X_{n+1} = a X_n + b \bmod M$ where *a*, *b*, and *M* are randomly chosen and then fixed. The initial seed value is X_0. These methods are very fast and have good statistical properties if *a*, *b*, and *M* are chosen well, but they are quite predictable. It is obvious that if Mallory knows the parameters *a*, *b*, and *M*, as well as the initial seed X_0, he can quickly compute any X_i. Even if he doesn't know these parameters, the sequence of numbers produced by the LCG is still predictable if he knows some of the X_i's.

The random *k*'s produced by a LCG, while predictable, seem to be well protected by the equation $r = (g^k \bmod p) \bmod q$. Mallory could not recover *k* "short of computing discrete logarithms, and in fact not even then, due to the second mod operation [5]." So even if we use a predictable LCG where we can find k_3 if we are given k_1 and k_2, it doesn't appear to be a risk if k_1 and k_2 are sufficiently masked by the *r* equation. It turns out that this masking is not sufficient to protect the values of *k*. In the paper, they claim that given just three valid signatures produced with a LCG, the secret key can be recovered.

4.6.2 Uniqueness Lemma

The Uniqueness Lemma shows that the idea that the DSS protects its random *k* values with the *r* equation is false. It says that so long as the *k* values are pseudorandomly generated, we can safely ignore the relation $r_i = (g^{k_i} \bmod p) \bmod q$. The DSS signature equations $s_i k_i - r_i x = m_i$, to a high degree of probability, uniquely determine the secret key *x*. Thus, Mallory can ignore the masking provided by the *r* equation. For a full proof of this lemma, see section 3.2 of [5].

4.6.3 The Attack

The attack hinges on the relationship $sk - rx = m \bmod q$ for any signature (r, s) produced by the DSA. First, assume that *p*, *q*, and *g* are fixed. Suppose Mallory receives two messages m_1 and m_2 and their signatures $(r_1, s_1) = DSA(x, k_1, m_1)$ and $(r_2, s_2) = DSA(x, k_2, m_2)$. Mallory knows the values of $m_1, r_1, s_1, m_2, r_2, s_2$ (they were sent to him) as well as the public parameters *p*, *q*, and *g* and the signer's public key

$y = g^x \bmod p$. Mallory also knows that $s_1 k_1 - r_1 x = m_1 \bmod q$ and $s_2 k_2 - r_2 x = m_2 \bmod q$. Mallory does not know the secret key *x* or the two random values k_1 and k_2. Now we examine what Mallory knows about the LCG used to generate the *k* values. He knows the parameters *a*, *b*, and *M* that define the LCG, but the initial seed value is still unknown to him. The paper proposes that Mallory solve the system of three modular equations in three unknowns as follows:

$$\begin{cases} s_1 k_1 - r_1 x = m_1 & (\bmod q) \\ s_2 k_2 - r_2 x = m_2 & (\bmod q) \\ -a k_1 + k_2 = b & (\bmod M) \end{cases}$$

Since this is a simultaneous system of modular equations in different moduli, unless $M = q$, Mallory can't use linear algebra to solve this system. So it is apparent that it is incredibly insecure to use *q* as the modulus *M* in the LCG. The Uniqueness Lemma allows us to focus on this system without worrying about the extra constraints placed by the non-linear relationships $r_i = (g^{k_i} \bmod p) \bmod q$. Though I won't go in to details, the paper describes a method called *Lattice Reduction* that can be used to solve the above system of equations, revealing the secret key *x*. For a more complete and certainly more coherent description of their methods, see [5].

4.7 Public Parameters

Both the ElGamal scheme and the DSS employ public parameters that can be shared. The specification of the DSS [20] specifically states "the integers *p*, *w*, and *g* can be public and be common to a group of users." Both ElGamal and DSS have been widely shown to be somewhat susceptible to attacks where a dishonest authority hands out "cooked" public parameters. That is, parameters that have been generated in such a way that, given some special information about the parameters, it is fairly easy to forge a user's signature. Three papers were published at around the same time dealing with this subject ([1], [6], and [31]). In describing the insecurities in the ElGamal scheme, some of the design decisions of the DSS discussed in section 3.4.1 make more sense.

4.7.1 Weak Public Parameters in ElGamal

Bleichenbacher [6] details an attack that allows Mallory to generate a valid ElGamal signature if the public parameter *p* is not chosen properly. Suppose $p = qw + 1$ where *w* is smooth (all its prime factors are small) and $y \equiv g^x \bmod p$ as usual be the public key of Alice. If Mallory knows *r* and *k* such that $r \equiv g^k \equiv cq \pmod p$, with $0 < c < w$, then Mallory can generate a valid ElGamal signature (r, s) for all messages $m \equiv xr \pmod{\gcd(k, p-1)}$. First, Mallory solves the equation

$$(g^q)^z \equiv y^q \bmod p$$

for *z*. Since *w* is smooth, Mallory can do this using the Pohlig-Hellman algorithm (see section 4.4.2). He then defines $f = \gcd(k, p-1)$ and

$$s \equiv \frac{m - cqz}{f}\left(\frac{k}{f}\right)^{-1} \pmod{(p-1)/f}.$$

The requirement that $m \equiv xr \pmod{\gcd(k, p-1)}$ ensures that f divides $m - cwz$. We notice that it follows that $sk \equiv m - cqz \pmod{p-1}$. Therefore

$$\begin{aligned} y^r r^s &\equiv y^{cq} g^{sk} \\ &\equiv g^{cqz} g^{m-cqz} \equiv g^m \bmod p, \end{aligned}$$

and the signature has been verified.

I have one comment to make on this discovery. Notice that only messages satisfying $m \equiv xr \pmod{\gcd(k, p-1)}$ can be signed by Mallory. It appears that in order for Mallory to choose a message he knows he can sign using this method, he needs to know Alice's private signing key x. If he knew this already, there would be no need to forge her signature.

Bleichenbacher goes on to show other ways in which weak public parameters can render the ElGamal system breakable. Suppose that the generator g in the scheme is smooth and divides $p-1$. He shows that if $p \equiv 1 \pmod 4$, Mallory can forge Alice's signature on an arbitrary message m. If $p \equiv 3 \pmod 4$, he will be successful on one half of the messages $0 \le m < p$. Bleichenbacher proves this by fist setting $k = (p-3)/2$. Thus,

$$\begin{aligned} g^k &\equiv g^{(p-1)/2} g^{-1} \\ &\equiv (-1) g^{-1} \quad \text{(by the Legendre Symbol, perhaps?)} \\ &\equiv (p-1)/g \pmod p \end{aligned}$$

Recall the previous requirement that $m \equiv xr \pmod{\gcd(k, p-1)}$; if $p \equiv 1 \pmod 4$, then $\gcd((p-3)/2, p-1) = 1$, and thus Mallory can compute signatures for all m. If $p \equiv 3 \pmod 4$, then $\gcd((p-3)/2, p-1) = 2$. This implies that Mallory will be able to sign one half of all messages m.

In section 4 of [6], Bleichenbacher shows a way for a dishonest authority to generate public parameters p and g in such a way as to make the attacks described above feasible.

4.7.2 Weak Public Parameters in the DSS

Vaudenay [31] points out that the DSS does not specify any checks for the parameter g, as it does for the parameters p and q (see section 3.4.6). It does state that g should be greater than one, but does not explicitly say that an implementation must check to make sure this holds. If Mallory were a dishonest authority, he could hand out $g = 1$ to Alice. Using this g value, Alice would then accept any signature forged by

$$\begin{aligned} r &= (y^k \bmod p) \bmod q \\ s &= rk^{-1} \bmod q \end{aligned}$$

This is fairly trivial to prove, but hinges on the fact that the public key $y = g^x \bmod p = 1$ when $g = 1$. In a slightly less obvious attack, Vaudenay suggests that Mallory provide $g = y^\alpha \bmod p$ to Alice, where *y* is Bob's public key. Mallory can now forge Bob's signature by

$$r = (y^k \bmod p) \bmod q$$
$$s = (\alpha\, m + r)\, k^{-1} \bmod q$$

These two attacks could easily be avoided by adding a generation algorithm for *g* to the DSS, and requiring that the verifier check for "cooked" parameters.

4.7.3 Collisions in the DSS

This attack, described by Serge Vaudenay [31], is not applicable to the DSS when it is properly used, but serves to highlight the importance of following the standard to the letter. The attack is based on the concept that the DSA does not sign the output of the SHA, but rather the output of the SHA mod *q*. This is easy to see since the *s* portion of the signature is reduced modulo *q*. Since SHA is considered a secure hash function, it is not feasible to find a collision using the SHA, or even the SHA mod *q*. However, it is possible to construct a valid public parameter *q* that will cause the DSA to produce identical signatures for two messages m_1 and m_2.

To this end, Mallory could check whether or not $q = \left| \mathrm{SHA}(m_1) - \mathrm{SHA}(m_2) \right|$ is a 160-bit prime. It is obvious that the probability that the integer $\left| \mathrm{SHA}(m_1) - \mathrm{SHA}(m_2) \right|$ is a 160-bit integer is ½. According to the prime number theorem, the probability that a 160-bit integer is prime is approximately $1/(160 * \ln 2) \approx 1/111$. Thus, we can generate a valid prime *q* that produces the desired collision with an average of 222 tries. Once the *q* has been created, it is a simple matter to find valid values for *p* and *q* by using the generation algorithm recommended by the specification of the DSS (see section 3.4.6).

This attack might not seem terribly pernicious until you see the example given by Vaudenay in his paper. Suppose Alice wants to join an Internet service that promises to provide useful services. They provide her with the public parameters *p* and *q* and ask her to sign the message "This is just a test" using the DSS to ensure everything is set up correctly. The online service could easily be swindling Alice by the following means. Take *m* = "This is just a test", which hashes to

SHA(*m*) = 653095248274681173784622345203351158554318835 88.

Unbeknownst to Alice, the "service" is really Mallory, and he has picked the second message m' = "Transfer $9,302 on account XYZ", which hashes to

SHA(m') = 14129973124864981439056178719272707214461104315 87

Mallory then picks

$$q = SHA(m) - SHA(m')$$

which happens to be 160-bit prime. He can now fairly easily pick a *p* such that *q* divides $p - 1$, and give these values to Alice. So in signing the message "This is just a test"

using the cooked public parameters, Alice has provided Mallory with a valid signature for the message "Transfer $9,302 on account XYZ." Mallory can now steal that amount from Alice and whomever else he dupes with his devious plan.

4.7.4 Design Motivations of the DSS

Anderson and Vaudenay claim that the "DSS is simply ElGamal with many of the bugs fixed, and with a rather simple optimization (the reduction of r modulo q) that reduces the length of the signature and the amount of arithmetic needed to verify it [1]." One simple way in which the DSS improves on ElGamal is the addition of the check to ensure $0 \le r < p$, which foils the universal forgery of ElGamal discussed in section 4.3. In addition, the requirement that $g = h^{(p-1)/q} \bmod p$ generates a prime order subgroup of the group created by p, which helps to avoid the attacks described by Bleichenbacher (see section 4.7.1) as well as the Newton subliminal channel, which we will discuss in section 4.8.4.

4.8 Subliminal Channels

Subliminal channels are a fascinating twist to digital signatures. Schneier [24] makes a good analogy that I will build on here. Suppose Alice and Bob have been arrested for breaking into the IRS computers and changing their tax records. Since there is no co-ed prison in their state, Alice and Bob go to separate prisons. Walter is the warden, and will allow Alice and Bob to exchange messages so long as they are not encrypted, since he obviously wants to monitor their communication. Alice and Bob want to establish a subliminal channel so they can communicate right in front of Walter without him realizing it. One easy way would be to have the first letter of each sentence from the message. Thus, the seemingly innocuous (though nonsensical) message from Bob

Dear Alice,

Elvis' singing can ameliorate Peter's emotions! Tom offered me orange Rolaids! Randall opened Will's aging T-bird's engine in Georgia. Hot tamales!

Sincerely,
Bob

is actually telling Alice "escape tomorrow at eight." This is not a very interesting method, it is just stenography; there is no secret key and its security depends on keeping the algorithm secret from Walter. We already established in section 1.3 that this type of obfuscation is not secure. Alice and Bob want to make it so that Walter will not be able to read their subliminal communication even if he knows that it is present in a message.

To this end, they persuade Walter to allow them to sign the messages they exchange between one anther. The concept of subliminal channels in digital signature schemes was invented by Gustavus Simmons in 1983 [26] . The basic protocol (from [24]) goes like this:

1) Alice generates an innocuous message.

2) Using a secret key she shares with Bob, Alice signs the message in such a way that she hides her subliminal message in the signature.
3) Alice sends the message and its signature through Walter to Bob.
4) Walter checks the validity of the signature and makes sure there are no overt escape plans in the message. Since he has no idea the subliminal message is there, he passes the message along to Bob.
5) Bob confirms that the signature of the message is a valid signature of Alice.
6) Bob tosses the message in the trash and uses the secret key that he and Alice share to extract the subliminal message.

It is very interesting to note that the issue of subliminal channels in digital signature schemes first arose when the USA and USSR were deciding on a nuclear arms limitation treaty. Both nations agreed to place sensors in the other's nuclear facilities to check for compliance with the treaty. The following is from [2], which was in turn paraphrased from [28] :

> This was a special concern with systems used to monitor not just the occurrence of nuclear tests, but the number of fielded nuclear weapons. If a Russian sensor designed to relay merely the presence of absence of an American missile in a silo could covertly communicate the silo's location, then this information could have been used to facilitate a first strike. One of the early designs for equipment to verify treaty compliance had just such a weakness: the sensor's location could have been transmitted using a subliminal channel in an early authentication scheme based on discrete logarithms.

Since signatures with a subliminal message embedded in them look no different than regular signatures, Walter should never even know the channel exists. Even if he did know it was there, he doesn't know the secret key, and thus could not read the message. There are two types of these channels: broadband and narrowband. A broadband channel allows Alice to hide a subliminal message on the order of 160 bits, while a narrowband channel usually only hides a few bits per signature. We will now move on to discuss subliminal channels in the signature schemes we have been discussing.

4.8.1 Broadband Channel in ElGamal

There is a broadband channel in the ElGamal scheme (described in [27]) that is quite easy to set up. The main drawback is that it requires Alice to share her secret signing key x with Bob. Doing so allows Bob to recover the random k value used in the signing process, but also allows him to create undetectable forgeries of her signature. Walter cannot do this, since he doesn't know the signing key. Bob gets the message m along with the signature (r, s) and uses the s equation to recover k like so:

$$k = (m - xr)s^{-1} \bmod (p-1)$$

The thing is, according to the specification, k must be relatively prime to $p-1$ so that the inversion can be performed. This reduces the number of possible messages to $\varphi(p-1)$, or the number of integers less than and relatively prime to $p-1$. In addition, there is a

chance that Bob will not be able to recover *k*. If you look at the equation to recover *k*, you will notice that we must take the inverse of $s \bmod p-1$, which means *s* must be relatively prime to $p-1$. Thus, the probability that Bob will be able to recover the subliminal message *k* is $\varphi(p-1)/(p-1)$ [16]. There is also the matter of mapping the *k* values to some sort of meaningful message; the *k* value in itself probably won't mean much to Bob. Simmons [27] asserts that for large primes, primes *p* where $p-1$ has several factors, this problem "is so difficult to do that it limits the potential usefulness of the subliminal channel."

If Walter guesses the correct subliminal message k', he can easily check to see if the channel is being used by calculating $r' \equiv g^{k'} \bmod p$. If this matches the signature parameter *r*, then Walter can be certain he has found the channel. One easy way to defeat this possibility is to use a Vernam cipher to mask the *k* value with a one-time key that Alice and Bob share. Due to the perfect secrecy afforded by the Vernam cipher, Walter can no longer eavesdrop on the channel [16].

4.8.2 Broadband Channel in the DSA

The broadband subliminal channel in the DSA is similar to the one in the ElGamal scheme, only Alice is now unfettered by the $\varphi(p-1)$ restriction on the number of *k* values she can use. Since *q* is prime, and the standard specifies only that $k < q$, we have the whole range of residues mod *q* to use as subliminal messages. Obviously *k* must still be relatively prime to *q*, but since *q* is prime, it is inherently relatively prime to every integer less than it is. This lifts the requirement of a special function to map values of *k* onto meaningful message, but surprisingly reduces the number of possible messages. With this channel, Alice has $q-1$ possible messages available to her, but according to [16], it is always the case that $\varphi(p-1) \geq q-1$. She still must give up her private signing key in order for Bob to read the subliminal message. This channel also removes the problem where Bob could only recover the *k* value if *s* was relatively prime to $p-1$. If you notice that the *s* portion of a DSA signature is calculated mod *q*, it is apparent that Bob will always be able to find the inverse of *s* mod *q* [16]. Thus, it appears that the DSA is far more conducive to hiding subliminal messages than ElGamal. We will show in section 4.8.4 that this in fact is not the case

4.8.3 Narrowband Channels in the DSA

There are ways for Alice to transmit subliminal messages to Bob using the DSA where she does not have to reveal her secret key. The drawback to this is that the channel is very narrowband; only one bit may be hidden in each signature that Alice transmits. In [16], three narrowband channels are described. In the first one, Alice and Bob agree on a common secret prime *P* where $P > q$. Alice first chooses any message she wants. She then chooses a random *k* value in such a way that the resulting *r* portion of the signature is either a quadratic residue mod *P* or a quadratic nonresidue mod *P*, depending on the bit she wants to send. She could have a 1 correspond to the former case and a 0 correspond to the latter. Bob first verifies the signature to make sure the message came from Alice,

then checks to see whether *r* is a quadratic residue or a quadratic residue mod *P*. Thus, he can easily recover the subliminal bit hidden by Alice.

Schneier [24] expands on this channel and describes a case where Mallory, through an unscrupulous implementation of DSA, can have the algorithm reveal 10 bits of Alice's secret key for each signature she issues. The attacks proceeds as follows:

1) Mallory implements his dishonest version of DSA and puts it in a tamperproof chip, so as to prevent anyone from discovering his malfeasance. Using a variant on the channel described above, he creates a 14-bit subliminal channel by choosing 14 random primes $P_1 ... P_{14}$. The chip will choose a value of *k* such that *r* ends up being either a quadratic residue or a quadratic nonresidue mod each of those 14 primes, depending on the subliminal message. Bit *i* of the subliminal message would be a 1 if *r* was a quadratic residue mod P_i, and a 0 otherwise.
2) Mallory gives the chips to Alice and Bob, or whoever else wants one.
3) Alice signs a message normally using her 160-bit private key *x*.
4) Mallory's chip chooses a random 10-bit block of *x*, where each block starts on bit $10n$ where $0 \le n \le 15$. The number *n* is the block number. Since there are 16 possible blocks of this size, a 4-bit integer can identify the block number *n*. Thus, the subliminal message becomes the 10-bit block plus the 4-bit identifier.
5) The chip now tries random value of *k* until it finds one that produces a *r* that has the correct quadratic residue properties relative to the primes $P_1 ... P_{14}$. This is not a particularly daunting task, since statistically, the chip will pick a suitable *k* within $2^{14} = 16384$ tries. If we assume that Mallory's chip can test 10,000 values of *k* per second, it will be successful in less than two seconds. Furthermore, since this computation does not involve the message, the chip can perform this computation before Alice tries to sign a message.
6) The chip uses the *k* value it found in the previous step to normally sign the message.
7) Alice sends the resulting signature to Bob.
8) Mallory intercepts the signature and recovers *r*. Since he obviously knows the primes $P_1 ... P_{14}$ he built into the chip, he can easily decrypt the subliminal message. After receiving a sufficient number of signatures from Alice, Mallory can reconstruct her secret key, and has achieved a total break of the system.

The other two narrowband subliminal channels described in [16] could probably be exploited in a similar manner. One of these solutions proposes that Alice and Bob share a random binary sequence $b_1 ... b_t$, which is used as a one-time key. Once again, only one subliminal bit per signature can be hidden. Subliminal bit *i* of the subliminal message can be found by taking by taking the *exclusive or* of the lowest bit of *r* and the random bit b_i. All Alice would have to do would be choose a *k* value such that the lowest bit of *r* turns out to be the desired subliminal message bit, which is trivial to do.

In order for Bob to recover the subliminal message, he must receive the signatures in the order that Alice sent them. Thus, if Walter suspects something is going on, he could easily foil the scheme by switching the order of the signatures sent to Bob.

The last narrowband channel described in [16] is due to the inventor of the subliminal channel concept, Gustavus Simmons. This channel allows Alice to encode *l* subliminal bits in each signature. The feasible length of *l* depends on the computing power Bob has available to him. Alice chooses an additional random parameter k' such that $k' < q$. She then computes $r' = g^{k'} \bmod p$ and secretly sends r' to Bob. Let's say m' is the *l*-bit subliminal message Alice wishes to send to Bob. To sign, Alice computes $k = m' + k' \pmod{p}$, signs the innocuous message *m*, and sends the resulting signature (r, s) along with the message to Bob. Bob knows that $r \equiv (r' g^{m'} \pmod{p}) \pmod{q}$, and since *l* (the length of m') is bounded, he can find m' by trying every single possible value of m' until it satisfies the above equation. Notice that unless Alice computes a new k' and r' (and transmits r' to Bob) for each subliminal message, she is using the same value of *r* if she signs the same innocuous message *m*. Thus, using the method described in 4.5, Mallory could compute her private signing key.

4.8.4 The Newton Channel

It was a widely held belief that the DSA provided a far better setting for subliminal channels than the ElGamal scheme. This was in fact one of the arguments against the DSS when it was proposed by the NIST. People thought that the NSA had designed it in such a way to allow it to be easily used for broadband subliminal channels. While the DSA removed the message-mapping requirement (see section 4.8.1), it also sealed a gaping subliminal channel in ElGamal. This broadband subliminal channel, called the Newton Channel, is outlined in [2]. The reason this channel is more significant than the broadband channel discussed in section 4.8.1 is that it does not require the signer Alice to give up her secret signing key. In this system, we assume that the public parameter $p = qw + 1$ where *w* is smooth (all its prime factors are small) and that calculating the discrete logarithm problem using this *p* is hard. Let's say that the subliminal message Alice wishes to transmit is m'. She finds a random *k* value such that $k \equiv m' \pmod{w}$. In other words, she sets $k = m' + k'w$ for some randomly chosen k'. She signs the message in the normal fashion and sends the signature (r, s) and the innocuous message m' along to Bob. Since Bob knows the factorization of $p - 1$ (he and Alice agreed on this earlier), he forms r^q and solves the following equation for *z*:

$$(g^q)^z \equiv r^q \pmod{p}$$

In other words, he is trying to solve the problem $\log_{g^q}(r^q) \pmod{p}$, which they claim is feasible due to the special choice of parameter *p*. This is a bit beyond the scope of this paper, but they suggest using the Pohlig-Hellman algorithm (see section 4.4.2) in combination with something called Pollard's rho method [22]. Using this combination, Bob will then have $m' \equiv z \pmod{w}$, so he has recovered the subliminal message.

After presenting this channel in [2], the authors comment that this channel is a broadcast one; there is no secret information required to recover the subliminal message,

aside from the factorization of p. They then go on to present narrowcast channels where someone can only recover the subliminal message if they possess some shared secret. The crux of the discovery of the Newton Channel was that, in the words of its discoverers, "the DSA does not maximize the covert utility of a signature, but minimizes it—by eliminating the Newton channel." The DSA does so by operating in a group of prime order; replacing g with $g^{(p-1)/q}$. To an extent, this takes some of the heat off the NSA by invalidating one of the major complaints against it.

4.8.5 Defeating Subliminal Channels

You may have noticed that all the subliminal channels we have discussed rely on Alice choosing an appropriate k to hide her subliminal message. Schneier [24] describes a method to foil the subliminal channel where Alice is not allowed to independently choose k. She obviously can't let Bob choose k for her; if she did, he could easily recover her secret signing key after seeing the signature she generated using the value of k he chose (see section 4.5). Instead, we need a system where Alice and Bob collaborate to generate k in such a way that Alice can't control a single bit of k and Bob does not know a single bit of k. The protocol proceeds like this:

1) Alice chooses a random k' and sends Bob $u = g^{k'} \bmod p$.
2) Bob chooses a random k'' and sends it to Alice.
3) Alice calculates $k = k'k'' \bmod (p-1)$. She then uses this k value to sign her message m and sends Bob the signature (r, s).
4) Bob verifies that $((u^{k''} \bmod p) \bmod q) = r$.

Having performed this protocol, Bob can be certain that Alice has not embedded any subliminal information in the signature. Though I won't go in to details, this protocol also allows Bob to embed subliminal information of his own in a signature of Alice's by choosing k'' with certain characteristics. Using a different generation method can prevent this attack.

4.9 Timing Attacks

Though I will only give a brief overview on this attack, I feel that it should be mentioned. In [18] Paul Kocher developed an attack on several public-key cryptosystems, including RSA and the DSS. He asserted that these schemes could be broken by "carefully measuring the amount of time required to perform private key operations." Since the time to perform these operations varies with the input, Mallory could possibly recover the secret signing key. In this attack, Mallory would need a way to take extremely accurate measurements of Alice's signature operations. In a press brief [23], RSA Data Security Inc. responded to Kocher's discovery, saying that "the attack Paul Kocher describes is academically interesting, but it is easy to defend systems against his attack using a technique called 'blinding,' developed by Dr. Ron Rivest of RSA." This blinding they suggest involves equalizing the time needed for various operations by

introducing a random number into these operations, “making it impossible to get any useful data out of timing these transactions.

5 Conclusions

Anderson and Vaudenay state that "the large majority of actual attacks on conventional cryptosystems are due to blunders in design, implementation, and operation, which are typically discovered by chance and exploited in an opportunistic way [1]." We have seen that this is frequently the case. Most of the attacks we discussed were not due to flaws in the design of the algorithm, but rather in the implementation. In section 4.3, we showed the dangers of not checking to make sure that $1 \leq r < p$ in the ElGamal scheme. The DSS corrected this problem by specifically stating that the verifier should check this condition, but a careless implementation might leave this out, leaving the system vulnerable to attack.

Both of the signature schemes we discussed allow some of the public parameters to be shared among a group of users. As discussed in section 4.7, a user should be very careful about what values they accept for those parameters. One should never implicitly trust an authority that hands these parameters out. We showed in section 3.4.6 how the DSS recommends generating these primes. Thus, an honest authority would hand out the values of p and q along with the seed and counter variable used to generate them. The user could then run these values through the generation algorithm to ensure that they were valid. Since the generation process uses the SHA in several spots, a dishonest authority cannot reverse the process in order to generate valid seed and counter values for "cooked" values of p and q. The DSS does seem to have a slight semantic weakness, in that it does not specify a way for a user to ensure that the generator g is not a fake value. As a result of this, we showed in section 4.7.2 that a user who is not careful might accept g values that allow the authority to forge signatures.

Another potential weakness in both algorithms is the random value of k used to sign each message. We showed in section 4.5 that a user must guard this value very closely, for if even one were to be revealed, their secret signing key would be compromised. Similarly, in an attack that is less apparent, a user should never sign two messages using the same random k value. If they do, an attacker can recover their signing key without even knowing the k they used. While this is not really a design flaw, it is a potential weakness if users are not careful. In the same vein, [5] showed that the security of the DSS depends greatly on the underlying random number generator. If an implementation of the DSS were to employ a predictable random number generator, an attacker could exploit this to break the system. Once again, as long as the implementation follows the standard to the letter by using one of the suggested random number generators, it will not be vulnerable to this type of attack.

You may notice a pattern developing. It seems that most of the attacks on the DSS do not apply to a properly implemented version. If the implementer is honest, the authority handing out the public values is honest, and the users are careful, the system is secure. The existential attacks described in section 4.2 are possible no matter what, but

since the attacker can't choose the message, it does not appear to be much of a security threat.

We discussed the subliminal channels present in both schemes in section 4.8 and concluded that, though there are subliminal channels in the DSS, it lacks one major broadband channel that is present in the ElGamal scheme; the Newton channel. Thus, contrary to initial criticisms, the DSS is not more conducive for hiding subliminal message, but less so.

So, all in all, we conclude that the DSS is a significant improvement over the original ElGamal scheme laid out in [9]. While it is not perfect, it is far less prone to both forgery attacks and subliminal messages than the ElGamal scheme. Whether it should have been chosen as a national standard over the veteran RSA is to some extent a matter of opinion. In any case, it is the standard and appears to be a good one.

References

[1] R. Anderson and S. Vaudenay, "Minding your p's and q's", *Advances in Cryptology – ASIACRYPT '96*, Springer-Verlag, 1996, pp. 26-35.

[2] R. Anderson, S. Vaudenay, B. Preneel, K. Nyberg, "The Newton Channel," *Information Hiding 1996*, 1996, pp. 134-148.

[3] S. Bakhtiari, R. Safavi-Naini, and J. Pieprzyk, "Cryptographic Hash Functions: A Survey", Technical Report 95-09, Department of Computer Science, University of Wollongong, July 1995.

[4] P. Béguin and J. Quisquater, "Secure Acceleration of DSS Signatures Using Insecure Server," *Advances in Cryptology – ASIACRYPT '94*, Springer-Verlag, 1994, pp. 249-259.

[5] M. Bellare, S. Goldwasser, and D. Micciancio, "'Pseudo-Random' Number Generation within Cryptographic Algorithms: the DSS Case," Advances *in Cryptology – Crypto 97 Proceedings*, Springer-Verlag, 1997.

[6] D. Bleichenbacher, "Generating ElGamal Signatures Without Knowing the Secret Key," *Advances in Cryptology – EUROCRYPT '96*, Springer-Verlag, 1996, pp. 10-18.

[7] M. S. Conn, Letter to Joe Abernathy of the *Houston Chronicle*, National Security Agency, Ser: Q43-111-92, June 1992.

[8] W. Diffie, "The First Ten Years of Public-Key Cryptography," *Proceedings of the IEEE*, v. 76, n. 5, May 1988, pp. 560-577.

[9] T. ElGamal, "A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms," *Advances in Cryptology: Proceedings of CRYPTO 84*, Springer-Verlag, 1985, pp. 10-18.

[10] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, "Robust Threshold DSS Signatures," *Proceedings of EUROCRYPT '96*, Springer-Verlag, 1996, pp. 354-371.

[11] S. Goldwasser and M. Bellare, "Digital Signatures," *Lecture Notes on Cryptography*, 1997, pp. 96-118.

[12] J. Guajardo and P. Soria-Rodriguez, "Implementation of the Pohlig-Hellman Algorithm for the Discrete Logarithm Problem," Available on the Web at *http://www.wpi.edu/~sorrodp/crypto/*, 1995.

[13] M. Hellman, Letter to James Burrows of NIST, November 1991.

[14] P. Horster, M. Michels, H. Peterson, "Generalized ElGamal Signatures for One Message Block," Technical Report 94-3, Theoretical Computer Science and Information Security, University of Technology Chemnitz-Zwickau, 1994.

[15] P. Horster, H. Peterson, and M. Michels, "Meta-Elgamal Signature Schemes," Theoretical Computer Science and Information Security, University of Technology Chemnitz-Zwickau, 1994.

[16] P. Horster, M. Michels, and H. Petersen, "Subliminal Channels in Discrete Logarithm Based Signature Schemes and How to Avoid Them," Technical Report 94-13-D, Department of Computer Science, University of Technology Chemnitz-Zwickau, 1994.

[17] D. Knuth, *The Art of Computer Programming: Volume 2, Seminumerical Algorithms*, 2nd edition, Addison-Wesley, 1981.

[18] P. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," *Proceeding of CRYPTO '96*, Springer-Verlag, 1996, pp. 104-113.

[19] National Institute of Standards and Technology, NIST FIPS PUB 180-1, "Secure Hash Standard," U.S. Department of Commerce, April 1995.

[20] National Institute of Standards and Technology, NIST FIPS PUB 186, "Digital Signature Standard," U.S. Department of Commerce, May 1994.

[21] D. Pointcheval and J. Stern, "Security Proofs for Signature Schemes," *Proceedings of EUROCRYPT '96*, Springer-Verlag, 1996, pp. 387-398.

[22] J. Pollard, "Monte Carlo Methods for Index Computation (mod p)," *Mathematics of Computation, v 32 no 143*, 1978, pp. 918-924.

[23] RSA Data Security Inc., "Questions and Answers Regarding the Paul Kocher Timing Attack on Public-Key Cryptosystems," *An RSA Press Brief*, 1995.

[24] B. Schneier, *Applied Cryptography (Second Edition)*, John Wiley & Sons, Inc., 1996.

[25] P. Schumer, *Introduction to Number Theory*, PWS Publishing, 1996.

[26] G. Simmons, "The Prisoner's Problem and the Subliminal Channel," *Advances in Cryptology: Proceedings of CRYPTO '83*, Plenum Press, 1984, pp. 51-67.

[27] G. Simmons, "The Subliminal Channels in the U.S. Digital Signature Algorithm (DSA)," *Proceedings of the Third Symposium on: State and Progress of Research in Cryptography*, Rome: Fondazone Ugo Bordoni, 1993, pp. 33-54.

[28] G. Simmons, "Subliminal Channels: Past and Present," *European Transactions on Telecommunications v 5 no 4*, 1994, pp. 459-473.

[29] D.R. Stinson, Cryptology: Theory and Practice, CRC Press, Inc., 1995.

[30] United States District Court for the District of Columbia, C.A. 92-0972-RCL, "Computer Professionals for Social Responsibility vs. National Institute of Standards and Technology, et al.", 1992.

[31] S. Vaudenay, "Hidden Collisions on DSS," *Advances in Cryptology – CRYPTO '96*, Springer-Verlag, 1996, pp. 83-88.

This thesis was processed using the LºTEC macro package with MSW97 style.