CPSC 467b: Cryptography and Computer Security
Lecture 16

Michael J. Fischer

Department of Computer Science
Yale University

March 22, 2010

Brief Review of Squares and Square Roots

Summary of results mod primes and composites

We have discussed three different cases of moduli with respect to quadratic residues, square roots, primitive roots, and other number-theoretic functions.

1. Primes.
2. Composites that are the product of two distinct odd primes.
3. General composite numbers

I want to summarize some of these results to help you keep them straight and to give a greater perspective on how they relate to each other.

Testing versus computing

Testing for the existence of an element with specific properties is never harder than finding it, and it can be much easier (assuming you can tell for a specific element whether it has those properties).

Finding a proper prime divisor

Testing for the existence of a proper prime divisor of n is equivalent to testing if n is prime, a problem for which we have shown feasible probabilistic solutions.

Finding a proper prime divisor of a number n is equivalent to the factoring problem, a problem believed to be intractable.

Finding square roots

Testing if a number a is a quadratic residue modulo n is the same thing as testing if it has a square root modulo n.

This problem is never harder than the problem of finding a square root since, given an algorithm to find square roots, one can test if a has a square root by trying to find it!

- If the algorithm succeeds, then a is definitely a quadratic residue.
- If the algorithm fails, then either a doesn't have a square root or the algorithm doesn't always work (and therefore isn't really a solution to the problem).

Why do algorithms fail?

Assume we have an algorithm that always returns a square root of a whenever a is a quadratic residue.

In order to infer that a is *not* a quadratic residue, we must be able to detect that the algorithm has failed.

Possible indications of failure are:

- it has explicitly halted with a failure indication,
- it has produced an incorrect answer that we can verify is wrong,
- or it has already run longer on the given input than it runs on any quadratic residue of the same length.

Square roots with prime modulus

For p an odd prime, testing and finding square roots are both easy.

Testing	Use Euler Criterion: $a \in \mathrm{QR}_p$ iff $a^{(p-1)/2} \equiv 1 \pmod{p}$, p odd prime.
Finding	Use Shank's algorithm.

(See lecture 13.)

Square roots with $n = pq$ modulus, factorization known

For modulus $n = pq$ the product of two *known* distinct odd primes p and q, both can be reduced to corresponding cases for primes.

Testing Use fact that a is a quadratic residue modulo n iff it is a quadratic residue modulo both p and q.

Finding Use the Chinese Remainder theorem to find a square root of $a \pmod{n}$ from square roots of $a \pmod{p}$ and $a \pmod{q}$.

(See lecture 13 and lecture 14.)

Square roots with $n = pq$ modulus, factorization unknown

For modulus $n = pq$ the product of two *unknown* distinct odd primes, no feasible algorithm is known for testing existence of or finding square roots modulo n of *arbitrary* numbers a.

Nevertheless, 1/2 of the numbers in $\mathbf{Z}_n^*$ *are* easily shown *not* to be quadratic residues modulo n.

Namely, numbers with Jacobi symbol $\left(\frac{a}{n}\right) = -1$ are not quadratic residues modulo n.

These are exactly the numbers $a \in Q_n^{01} \cup Q_n^{10}$ which are quadratic residues modulo one of p or q but not both.

The Jacobi symbol is easily computed. (See lecture 14.)

Numbers with Jacobi symbol 1

The numbers in $a \in Q_n^{00} \cup Q_n^{11}$ all have Jacobi symbol 1.

Those in Q^{11} are quadratic residues; those in Q^{00} are not.

There is no known feasible algorithm for distinguishing these two classes.

This is the basis for the Goldwasser-Micali cryptosystem presented in lecture 13.

Combining Signatures with Encryption

Signed encrypted messages

One often wants to encrypt messages for privacy and sign them for integrity and authenticity.

Let Alice have cryptosystem (E, D) and signature system (S, V). Some possibilities for encrypting and signing a message m:

1. Alice signs the encrypted message and sends the pair $(E(m), S(E(m)))$.
2. Alice encrypts the signed message and sends the result $E(m \circ S(m))$. Here we assume a standard way of representing the ordered pair $(m, S(m))$ as a string, which we denote by $m \circ S(m)$.
3. Alice separately encrypts and signs message and sends the pair $(E(m), S(m))$.

Weakness of encrypt-and-sign

Note that method 3, sending the pair $(E(m), S(m))$, is quite problematic since *signature functions make no guarantee of privacy*.

We can construct a signature scheme (S', V') in which the plaintext message is part of the signature itself.

If (S', V') is used as the signature scheme in method 3, there is no privacy, for the plaintext message can be read directly from the signature itself.

A forgery-resistant signature scheme with no privacy

Example: Let (S, V) be an RSA signature scheme. Define

$$S'(m) = m \circ S(m) \;;$$

$$V'(m, s) = \exists t(s = m \circ t \wedge V(m, t)) \;.$$

Fact

(S', V') is at least as secure as (S, V).

A forgery-resistant signature scheme with no privacy

Example: Let (S, V) be an RSA signature scheme. Define

$$S'(m) = m \circ S(m) ;$$

$$V'(m, s) = \exists t (s = m \circ t \wedge V(m, t)) .$$

Fact

(S', V') is at least as secure as (S, V).

Why?

A forgery-resistant signature scheme with no privacy

Example: Let (S, V) be an RSA signature scheme. Define

$$S'(m) = m \circ S(m) \;;$$

$$V'(m, s) = \exists t(s = m \circ t \wedge V(m, t)) \;.$$

Fact

(S', V') is at least as secure as (S, V).

Suppose a forger produces a valid signed message (m, s) in (S', V'), so $s = m \circ t$ for some t and $V(m, t)$ holds..

Then (m, t) is a valid signed message in (S, V).

Encrypt or sign first?

Method 1 (encrypt first) allows Eve to verify that the signed message was sent by Alice, even though Eve cannot read it. Whether or not this is desirable is application-dependent.

Method 2 (sign first) forces Bob to decrypt a bogus message before discovering that it wasn't sent by Alice.

Subtleties emerge when cryptographic protocols are combined, even in a simple case like this where it is only desired to combine privacy with authentication.

Encrypt or sign first?

Method 1 (encrypt first) allows Eve to verify that the signed message was sent by Alice, even though Eve cannot read it. Whether or not this is desirable is application-dependent.

Method 2 (sign first) forces Bob to decrypt a bogus message before discovering that it wasn't sent by Alice.

Subtleties emerge when cryptographic protocols are combined, even in a simple case like this where it is only desired to combine privacy with authentication.

Think about the pros and cons of other possibilities, such as sign-encrypt-sign, i.e., $(E(m \circ S(m)), S(E(m \circ S(m))))$.

ElGamal Signatures

ElGamal signature scheme

The *private signing key* consists of a primitive root g of a prime p and an exponent x.
The public verification key consists of g, p, and $a = g^x \bmod p$.

To sign m:
1. *Choose random* $y \in \mathbf{Z}^*_{\phi(p)}$.
2. *Compute* $b = g^y \bmod p$.
3. *Compute* $c = (m - xb)y^{-1} \bmod \phi(p)$.
4. *Output signature* $s = (b, c)$.

To verify (m, s), *where* $s = (b, c)$*:*
1. *Check that* $a^b b^c \equiv g^m \pmod{p}$.

ElGamal signature scheme

The *private signing key* consists of a primitive root g of a prime p and an exponent x.
The public verification key consists of g, p, and $a = g^x \bmod p$.

To sign m:

1. *Choose random* $y \in \mathbf{Z}^*_{\phi(p)}$.
2. *Compute* $b = g^y \bmod p$.
3. *Compute* $c = (m - xb)y^{-1} \bmod \phi(p)$.
4. *Output signature* $s = (b, c)$.

To verify (m, s)*, where* $s = (b, c)$*:*

1. *Check that* $a^b b^c \equiv g^m \pmod p$.

Why does this work?

ElGamal signature scheme

The *private signing key* consists of a primitive root g of a prime p and an exponent x.
The public verification key consists of g, p, and $a = g^x \bmod p$.

To sign m:

1. *Choose random* $y \in \mathbf{Z}^*_{\phi(p)}$.
2. *Compute* $b = g^y \bmod p$.
3. *Compute* $c = (m - xb)y^{-1} \bmod \phi(p)$.
4. *Output signature* $s = (b, c)$.

To verify (m, s)*, where* $s = (b, c)$*:*

1. *Check that* $a^b b^c \equiv g^m \pmod{p}$.

$$a^b b^c \equiv (g^x)^b (g^y)^c \equiv g^{xb+yc} \equiv g^m \pmod{p}$$

since $xb + yc \equiv m \pmod{\phi(p)}$.

Digital Signature Algorithm (DSA)

Digital signature standard

The commonly-used Digital Signature Algorithm (DSA) is a variant of ElGamal signatures.

Also called the Digital Signature Standard (DSS), it is described in U.S. Federal Information Processing Standard FIPS 186–3.[1].

It uses two primes: p, which is 1024 bits long,[2] and q, which is 160 bits long and satisfies $q \mid (p-1)$. Here's how to find them: Choose q first, then search for z such that $zq+1$ is prime and of the right length. Choose $p = zq+1$ for such a z.

[1] Available at http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf

[2] The original standard specified that the length L of p should be a multiple of 64 lying between 512 and 1024, and the length N of q should be 160. Revision 2, Change Notice 1 increased L to 1024. Revision 3 allows four (L, N) pairs: $(1024, 160)$, $(2048, 224)$, $(2048, 256)$, $(3072, 256)$.

DSA key generation

Given primes p and q of the right lengths such that $q \mid (p-1)$, here's how to generate a DSA key.

- Let $g = h^{(p-1)/q} \bmod p$ for any $h \in \mathbf{Z}_p^*$ for which $g > 1$. This ensures that $g \in \mathbf{Z}_p^*$ is a non-trivial q^{th} root of unity modulo p.
- Let $x \in \mathbf{Z}_q^*$.
- Let $a = g^x \bmod p$.

Private signing key: (p, q, g, x).

Public verification key: (p, q, g, a).

DSA signing and verification

Here's how signing and verification work:

To sign m:

1. *Choose random $y \in \mathbf{Z}_q^*$.*
2. *Compute $b = (g^y \bmod p) \bmod q$.*
3. *Compute $c = (m + xb)y^{-1} \bmod q$.*
4. *Output signature $s = (b, c)$.*

To verify (m, s), where $s = (b, c)$:

1. *Verify that $b, c \in \mathbf{Z}_q^*$; reject if not.*
2. *Compute $u_1 = mc^{-1} \bmod q$.*
3. *Compute $u_2 = bc^{-1} \bmod q$.*
4. *Compute $v = (g^{u_1} a^{u_2} \bmod p) \bmod q$.*
5. *Check $v = b$.*

Why DSA works

To see why this works, note that since $g^q \equiv 1 \pmod{p}$, then

$$r \equiv s \pmod{q} \quad \text{implies} \quad g^r \equiv g^s \pmod{p}.$$

This follows from the fact that g is a q^{th} root of unity modulo p, so $g^{r+uq} \equiv g^r(g^q)^u \equiv g^r \pmod{p}$ for any u.
Hence,

$$g^{u_1}a^{u_2} \equiv g^{mc^{-1}+xbc^{-1}} \equiv g^y \pmod{p}. \tag{1}$$

$$g^{u_1}a^{u_2} \bmod p = g^y \bmod p \tag{2}$$

$$v = (g^{u_1}a^{u_2} \bmod p) \bmod q = (g^y \bmod p) \bmod q = b$$

as desired. (Notice the shift between *equivalence* modulo p in equation 1 and *equality of remainders* modulo p in equation 2.)

Double remaindering

DSA uses the technique of computing a number modulo p and then modulo q.

Call this function $f_{p,q}(n) = (n \bmod p) \bmod q$.

$f_{p,q}(n)$ is not the same as $n \bmod r$ for any modulus r, nor is it the same as $f_{q,p}(n) = (n \bmod q) \bmod p$.

Example mod 29 mod 7

To understand better what's going on, let's look at an example. Take $p = 29$ and $q = 7$. Then $7|(29 - 1)$, so this is a valid DSA prime pair. The table below lists the first 29 values of $y = f_{29,7}(n)$:

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13
y	0	1	2	3	4	5	6	0	1	2	3	4	5	6

n	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
y	0	1	2	3	4	5	6	0	1	2	3	4	5	6	0

The sequence of function values repeats after this point with a period of length 29. Note that it both begins and ends with 0, so there is a double 0 every 29 values. That behavior cannot occur modulo any number r. That behavior is also different from $f_{7,29}(n)$, which is equal to n mod 7 and has period 7. (Why?)

Common Hash Functions

Common hash function

Many cryptographic hash functions are currently in use.

For example, the openssl library includes implementations of MD2, MD4, MD5, MDC2, RIPEMD, SHA, SHA–1, SHA–256, SHA–384, and SHA–512.

The SHA–xxx methods are recommended for new applications, but these other functions are also in widespread use.

SHA-1

The revised Secure Hash Algorithm (SHA–1) is one of five algorithms described in U. S. Federal Information Processing Standard FIPS PUB 180–3 (Secure Hash Standard).[3] It states,

> *"Secure hash algorithms are typically used with other cryptographic algorithms, such as digital signature algorithms and keyed-hash message authentication codes, or in the generation of random numbers (bits)."*

SHA–1 produces a 160-bit message digest. The other algorithms in the SHA–xxx family produce longer message digests.

[3] Available at http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf.

MD5 overview

MD5 is an older algorithm (1992) devised by Rivest. We present an overview of it here.

It generates a 128-bit message digest from an input message of any length. It is built from a basic block function

$$g : \text{128-bit} \times \text{512-bit} \rightarrow \text{128-bit}.$$

The MD5 hash function h is obtained as follows:

- The original message is padded to length a multiple of 512.
- The result m is split into a sequence of 512-bit blocks $m_1, m_2, \ldots, m_k$.
- h is computed by chaining g on the first argument.

We next look at these steps in greater detail.

MD5 padding

As with block encryption, it is important that the padding function be one-to-one, but for a different reason.

For encryption, the one-to-one property is what allows unique decryption.

For a hash function, it prevents there from being trivial colliding pairs.

For example, if the last partial block is simply padded with 0's, then all prefixes of the last message block will become the same after padding and will therefore collide with each other.

MD5 chaining

The function h can be regarded as a state machine, where the states are 128-bit strings and the inputs to the machine are 512-bit blocks.

The machine starts in state s_0, specified by an initialization vector IV.

Each input block m_i takes the machine from state s_{i-1} to new state $s_i = g(s_{i-1}, m_i)$.

The last state s_k is the output of h, that is,

$$h(m_1 m_2 \ldots m_{k-1} m_k) = g(g(\ldots g(g(IV, m_1), m_2) \ldots, m_{k-1}), m_k).$$

MD5 block function

The block function $g(s, b)$ is built from a scrambling function $g'(s, b)$ that regards s and b as sequences of 32-bit words and returns four 32-bit words as its result.

Suppose $s = s_1 s_2 s_3 s_4$ and $g'(s, b) = s'_1 s'_2 s'_3 s'_4$.

We define

$$g(s, b) = (s_1 + s'_1) \cdot (s_2 + s'_2) \cdot (s_3 + s'_3) \cdot (s_4 + s'_4),$$

where "$+$" means addition modulo 2^{32} and "$\cdot$" is concatenation of the representations of integers as 32-bit binary strings.

MD5 scrambling function

The computation of the scrambling function $g'(s, b)$ consists of 4 stages, each consisting of 16 substages.

We divide the 512-bit block b into 32-bit words $b_1 b_2 \dots b_{16}$.

Each of the 16 substages of stage i uses one of the 32-bit words of b, but the order they are used is defined by a permutation π_i that depends on i.

In particular, substage j of stage i uses word b_ℓ, where $\ell = \pi_i(j)$ to update the state vector s.

The new state is $f_{i,j}(s, b_\ell)$, where $f_{i,j}$ is a bit-scrambling function that depends on i and j.

Further remarks on MD5

We omit further details of the bit-scrambling functions $f_{i,j}$,

However, note that the state s can be represented by four 32-bit words, so the arguments to $f_{i,j}$ occupy only 5 machine words. These easily fit into the high-speed registers of modern processors.

The definitive specification for MD5 is RFC1321 and errata. A general discussion of MD5 along with links to recent work and security issues can be found on Wikipedia.

Although MD5 is widely used, recent attacks by Xiaoyun Wang and Hongbo Yu show that it is not collision resistant and hence no longer suitable for most cryptographic uses.