Lecture 1

6.857
NETWORK AND COMPUTER SECURITY
LECTURES 1-17

LECTURER: RONALD RIVEST
SCRIBE: JOHN WANG

1. Introduction to Cryptography

is computing or communicating in the presence of adversaries.

The presence of adversaries make security interesting, because you're working against the cleverness of other people. You are in the worst case scenario. Note that this is different than error correcting codes, for instance, where there is no adversary.

2. Security Policies

describes what is being protected and what activities or events should be protected.

If you don't have a policy, then you don't have security, because nothing is defined yet. Security policy is usually in terms of:

- Principals (actors or participants).
- Permissible (or inpermissible) actions or operations.
- Classes of objects.

2.1. **Examples.**

Security Policy: "Each registered voter may vote at most once." Principals are the voters and permissible actions are voting at most once. Security Policy: "Only an administrator can modify file x." Security Policy: "The recipient of an email should be able to authenticate the sender."

2.2. **Types of Policies.**

- C - confidentiality policies. Prevents unauthorized disclosure.
- I - integrity policies. Information should not be modifiable in an unauthorized manner.
- A - availability policies. Systems should remain available.

3. Security Mechanisms

are means for achieving security policies.

Examples: smart card for voters, password for sysadmin, digital signature for email, physical security.

Security mechanisms are usually one of two forms:

- Prevention: keeps policy from being violated.
- Detection: discovers if the policy has been violated.

If the detection mechanism goes off, then what? You must have a *recovery mechanism* for getting the system back to a good state. Notice that preventation and detection are not entirely unrelated. Detection system may involve deterrence, which helps prevents attacks.

4. Adversaries

The adversaries may be outsider or insider (ex: voter may want to be able to vote twice). Need to figure out who the adversary is. Note that there can be many adversaries.

What do the adversaries know? Usually, you assume that the adversary knows the engineering of the system and the security mechanisms. Security analysis is usually scenario based which is different depending on the assumptions one makes about the adversary and what he/she knows.

What resources does the adversary have? Does the adversary have a supercomputer or the ability to corrupt insiders or mathematical knowledge?

What are his motivations? Is the adversary economically motivated or is he just evil? Sometimes it is useful to assume that adversaries are rational economic players.

The best systems are those which are robust even in the worst-case scenarios. You want to have a system that is secure even when you have a perfect adversary.

5. Vulnerability

is a weakness in the system that can be exploited by the adversary.

Examples: poor password, buffer overflows, etc.

There is a distinction between the system as designed and the system as implemented. Implementations tend to have bugs, which could potentially introduce security vulnerabilities. Even if the design is perfect, the implemented system could weaken security.

There are *threats* to exploit vulnerability and *risk* that the vulnerability will be exploited.

Lecture 2

6. Introduction to Cryptography

6.1. Security Mechanisms.

- *Identification* of principals (username).
- *Authentication* of principals (password, biometric). Means by which one can prove identity.
- *Authorization.* Does the individual have permission to perform particular actions?
- *Physical Protection.*
- *Cryptography.* Using number theory for protection.
- *Economics.* Sometimes can assume that attackers are influenced by economic motivations.
- *Deception.* Try to deceive adversary. For example: honeypots are fake machines.
- *Randomness and Unpredictability.* Adversary is at a disadvantage because principals have done something that is unpredictable. Not having a good source of randomness has turned out to very bad.

6.2. Principles.

- *Be skeptical.* People make mistakes.
- *Be paranoid.* Things are likely not to be secure unless someone really convinces you.
- *Think adversarially.* What can an adversary do against this system? What could possibly go wrong with this system?
- *Think about user-supplied input.* User supplied input can be anything, so make sure to handle it properly.
- *Don't aim for perfection.* Bugs are a fact of life, so don't aim to try to be perfect.
- *Tradeoffs between cost and security.* To halve risk, does it double the cost?
- *Be prepared for losses.* Try to have the ability to rollback.
- *KISS.* Keep it simple. Complicated systems have more vulnerabilities and it is harder to tell what is going on.
- *Ease of use is important.* Don't have extremely long, convoluted process.
- *Separation of privilege.* For example: Two people need to sign off on a transaction.
- *Least privilege.* Don't give access to more than what someone needs.
- *Defense in depth.*
- *Complete mediation.*
- *Education.* Make sure people understand what is happening.
- *Transparency.* People should talk about exactly what they're doing.

7. Growth of Cryptography

7.1. Early Cryptography. Greeks: knew there were infinitely many primes, and also the GCD is easily computed by Euclid's algorithm. The Scytale wrapped paper around a rod of the same diameter for sender and receiver. If the attacker didn't have a rod of the correct diameter, he couldn't read it.

Fermat: Fermat's little theorem $a^{p-1} \equiv 1 \pmod{p}$ for any prime p and $1 \leq a < p$.

Euler: Generalize FLT to Euler's Theorem: if $gcd(a, n) = 1$ then $a^{\phi(n)} \equiv 1 \pmod{n}$. Where $\phi(n)$ is the Euler totient function.

7.2. World War I. The radio was a marvelous new communication technology – enabled instantaneous communication with remote ships and forces, but you also absolutely need to have encrpytion. Decipherment of the Zimmerman telegram from Germany to Mexico got America involved in World War I.

7.3. **Alan Turing.** Developed the theory of computability, but also worked a lot on cryptography. He and others deciphed the Axis Enigma machines, which had great impact on the war. Cryptanalytic effort involved development and use of early computers.

7.4. **Claude Shannon.** First post-war analysis of security systems.

7.5. **DES - U.S. Data Encryption Standard.** IBM designed DES, Horst Feistel was a key architect of the standard. NSA helped in return for keeping the size at 56 bits.

Scheme was in use for a long time (but it was also controversial).

7.6. **Computational Complexity.** People started asking question of how much time does it take to perform a certain operation. Key notions were polynomial time reductions, NP-completeness. For things that we know are computable in principle may still be infeasible because of the amount of time it takes to run.

8. Public Key Cryptography

Ralph Merkle, Marty Hellman, and Whit Diffie invented the notion of public-key cryptography. In 1976, Diffie and Hellman proclaim "We are at the brink of a revolution in cryptography."

Each party A has a public key PK_A others can use to encrypt messages to A: $C = PK_A(M)$. Each party A also has a secret key SK_A for decrypting a received ciphertext C: $M = SK_A(C)$.

It is easy to compute matching public/secret key pairs but publishing PK_A should not compromise SK_A. Idea: sign with SK_A and verify signature with PK_A. A produces signature σ for message M with $\sigma = SK_A(M)$. Given PK_A, M, and σ, anyone can verify validity of signature σ by checking $M = PK_A(\sigma)$.

However, Diffie and Hellman didn't know how to implement these ideas.

8.1. **RSA (Rivest, Shamir, Adleman 1977).** Security relies on the inability to factor product n and two large primes p, q. So we set $PK = (n, e)$ where $n = pq$ and $gcd(e, \phi(n)) = 1$.

$SK = d$ where $de = 1 \mod \phi(n)$.

8.2. **Digital Certificates.** Loren Kohnfelder's proposed notion of digital certificate, a digitally signed message attesting to another party's public key.

8.3. **RC4 Stream Cipher.** Most widely used software stream cipher. Not public-key, it exclusive-ors stream of pseudo-random bytes with plaintet to derive ciphertext. Extremely simple and fast, uses array $S[0..255]$ to keep a permutation of 0..255, initialized using secret key and two points into S. Created by Rivest.

8.4. **MD5 Hash.** Proposed as a pseudo random function mapping files to fingerprints. Collision resistance was design goal, it should be infeasible to find two files with the same fingerprint. Model for many later hash function designs.

8.5. **World Wide Web.** Just as radio did, this new communication medium drove demand for cryptography to new heights. Cemented transition of cryptography from primarily military to primarily commercial.

Lecture 3

9. Growth of Cryptography - Continued

9.1. **Zero Knowledge Proofs.** I can convince you that I know a solution to a hard problem while telling you nothing about my solution. I want to make sure you don't know anything about my solution.

Important for say, logging into a computer where you want to convince the computer that you know the secret key, but don't want to give the computer any information about the secret key.

9.2. **Micro Payments.** Micali and Rivest (2001). If you want to pay small amounts of money, how do you create a system for this?

Paying ten cents is equivalent to paying \$1 with probability 1/10. Uses pseudorandom digital signatures for verifiable fair dice. Neither the buyer or seller arranges the coin flips, there must be a cryptographically fair way of flipping the coin.

9.3. **Voting Systems.** There are new end-to-end cryptographic voting systems. All ballots are posted on the web and are encrypted, and the voters verify their votes are correct. Anyone can see the final tally.

Uses cryptography for increasing transparency and verifiability. You don't want people selling their own vote.

9.4. **Fully Homomorphic Encryption.** Can you compute on encrypted data, while keeping it encrypted? Given two encrypted data, are there ways to perform operations on them to create a new encrypted answer?

Craig Gentry in 2009 showed that you can do arbitrary computations on encrypted data based on the use of latticecs. Potential applications for cloud computing.

10. Encryption and One Time Pads

How do we get messages from Alice to Bob in a way that Eve can't listen in. There's some communication channel that Eve can listen to. Give Bob some key K, and Eve does not know K.

Alice encrypts message m and sends over $c \leftarrow enc(m)$ the encrpyted message over the channel.

Algorithms:

- Key generation $K \leftarrow gen(\mathbb{1}^{\lambda})$ where λ is key length. Here $\leftarrow$ denotes a randomized computation and $\mathbb{1}$ is a single bit.
- Encryption $c \leftarrow enc(K, m)$, where c is the ciphertext that is sent over the channel.
- Decryption $m = dec(K, c)$. Note that decryption is deterministic.

10.1. **Notion of Encryption.** Ideas of what encryption is:

- Eve should not be able to produce m from c.
- Can't identify any information about message and authenticate that the message is complete.
- Shouldn't see patterns in messages.

The encryption game: Eve can't distinguish $enc(K, m_1)$ from $enc(K, m_2)$ where $m_1 \neq m_2$ and $|m_1| = |m_2|$ if K is chosen randomly and Eve doesn't know K. Eve chooses m_1, m_2. Alice chooses one message m_b and encrypts that $c \leftarrow enc(k, m_b)$. Eve tries to guess b given ciphertext c.

10.2. **One Time Pad (Vernam 1917).** Message m, ciphertext c, key k are all λ bit quantities. The key k is also called the pad.

Generation: Flip coin λ times to get a pad k.

Encryption: Bitwise exclusive-or, $c = m \oplus k$.

Decryption: Exlusive-or, $m = c \oplus k$. This works because $m = m \oplus k \oplus k$.

10.3. **Proof of Security.** Thereom: One Time Pad is unconditionally/information-theoretically secure (Eve may have infinite computing power). Proof: Let $P(m)$ be Eve's a priori probability of message m, or Eve's state of knowledge. Need to show $P(m|c) = P(m)$.

We know that $|m| = |c| = |k| = \lambda$. We are assuming that k is chosen randomly and the probability of any particular key is $P(k) = 2^{-\lambda}$. We know that $P(c|m) = 2^{-\lambda}$ = probability of c given the message = probability that $k = m \oplus c = 2^{-\lambda}$.

$$P(c) = \sum_m P(c|m)P(m) = \sum_m 2^{-\lambda)} P(m) = 2^{-\lambda} \tag{1}$$

Where the last follows because $P(m) = 0$ unless m is the actual message sent, in which case $P(m) = 1$.

Now let us look at $P(m|c)$ which is the probability that Eve thinks message is m having seen c.

$$\begin{aligned} P(m|c) &= \frac{P(c|m)P(m)}{P(c)} && (2) \\ &= \frac{2^{-\lambda}P(m)}{2^{-\lambda}} && (3) \\ &= P(m). && (4) \end{aligned}$$

Thus, the ciphertext does not give Eve any information.

Lecture 4

11. One Time Pad

Surprising that it's not used more often. However, you need to have large secrets. Need to share them securely and keep them secret. These properties give some impact on usability. Most cryptography has small secrets that should be passed.

Another thing: you can't reuse the one time pad.

$$C_1 \oplus C_2 = (M_1 \oplus P) \oplus (M_2 \oplus P) \tag{5}$$
$$= M_1 \oplus M_2 \oplus (P \oplus P) \tag{6}$$
$$= M_1 \oplus M_2 \tag{7}$$

Famous attack from the NSA called Venona on the Russians.

Note that the One Time Pad (OTP) is malleable, which means it doesn't provide any message authentication. If the adversary attempts to change the bits over the channel, then the receiver of the message can't tell. The OTP provides superb confidentiality, but provides zero authentication. So we need to layer on top of it more techniques to achieve both of these.

Common mistake to think that just because something is encrypted, it can get to the receiver all hunky dorey.

12. Generating Randomness

- Flipping coins
- Environmental noise (microphone, keyboard, camera)
- Radioactive decay (bits in memory changing)
- Hard disk drive speed variations
- Intel new instructions (using the metastable state)
- Quantum randomness.

13. Cryptographic Hash Functions

Cryptographic Hash Function maps strings of arbitrary finite length to strings of fixed length (d bits). $h : \{0,1\}^* \to \{0,1\}^d$. We sometimes call $h(x)$ the hash of x or the digest of x.

These functinos should be efficiently computable, but it shouldn't sacrifice efficiency for security. There is no secret key, it should be a completely public function.

13.1. **Examples.** Rivest's functions: MD4, MD5.

NIST standards: SHA-1, SHA-2: (SHA-256, SHA-512), SHA-3 (Keccak).

13.2. **Random Oracle Model (ROM).** Specification of what the ideal cryptographic hash function would be. The user sends the oracle x, and the oracle sends the user back $h(x)$. If a second user sends the oracle x, the oracle will send back the same $h(x)$.

Oracle's Process: Receives x. If x is in the book, look up $h(x)$ and return it. Otherwise, flip a coin d times and call that $h(x)$. Write this $h(x)$ in the book and return it.

This is both consistent and random (the ideal of a cryptographic hash function). In the random oracle model, if $x \neq y$:

$$P[h(x) = h(y)] = \frac{1}{2^d} \tag{8}$$

13.3. **Properties.**

- One-wayness (OW) - preimage resistance. If $h(x) = y$ then x is the preimage of y and y is the image of x. It should be hard to go from y back to x.

  Infeasible for anyone given $y \in_r \{0,1\}^d$ (where $\in_r$ denotes randomly chosen) to find any x such that $h(x) = y$. Infeasible means that work is proportional to 2^d, which is just brute forcing every possible x and checking if it matches a y. Maybe take $d \geq 90$ to make this hard.
- Collision resistance (strong collision resistance).

  Infeasible for anyone to find x and x' such that $x \neq x'$ and $h(x) = h(x')$.

  In Random Orcale Model, difficulty is $\theta(2^{d/2})$ for finding any collision. The work should eceed 2^90 if $d > 180$. You lose a factor of 2 because of the birthday paradox.

$$x_1 \rightarrow y_1 \tag{9}$$
$$x_2 \rightarrow y_2 \tag{10}$$
$$\vdots \tag{11}$$
$$x_n \rightarrow y_n \tag{12}$$

$$E[\text{collisions}] = \sum_{i \neq j} Pr[h(x_i) = h(x_j)] \tag{13}$$
$$= \sum_{i \neq j} \frac{1}{2^d} \tag{14}$$
$$= \binom{n}{2} 2^{-d} \tag{15}$$
$$= \frac{n(n-1)}{2} \frac{1}{2^d} \tag{16}$$

  This is roughly $n^2 2^{-d}$, which means that if $n > 2^{-d/2}$, the expected number of collisions is greater than 1.
- Weak collision resistance (target collision resistance) (WCR).

  Infeasible given $x \in_r \{0,1\}^*$ to find $x' \neq x$ such that $h(x) = h(x')$. Like pairwise resistance, work is $\theta(2^d)$ in ROM.
- Pseudorandomness. Indistinguishable from a random oracle. Hard to define well.
- Non-malleability.

  Infeasible given $h(x)$ to produce $h(x')$ where x and x' are related.

13.4. **Applications.**

- Password storage: store $h(p)$ instead of string p. System compares $h(p)$ to $h(t)$ where t is the typed in password attempt. For a given user, this depends on the one-wayness property.
-

Lecture 5

14. Hash Function Applications

14.1. **Password Storage.** System stores $h(pw)$ rather than pw itself. System might also store username, salt, etc.

14.2. **File Modification Detector.** You want to monitor to detect when files have been changed. For each file, store $h(F)$ securely. You can check to see if the files have been modified by recomputing the hash. Provides detection (not prevention).

14.3. **Digital Signatures (hash and sign).**

- PK_A is Alice's public key (for signature verification).
- SK_A is Alice's secret key (for signing).
- Signing: $\sigma = sign(SK_A, m)$ and σ is Alice's signature on message.
- Verification: $verify(M, \sigma, PK_A) \in \{true, false\}$.

Idea: computing $h(m)$ is fast, so sign $h(m)$ instead of signing m. We do $sign(m, SK_A) = sign(m', SK_A)$ if $h(m) = h(m')$.

Problem is that if $h(m) = h(m')$, then asking Alice to sign m, her signature σ is also a signature for m'.

14.4. **Commitments.** Alice has value x which is her bid. She computes $C(x)$ and gives auctioneer $C(x)$, which is her sealed bid. When bidding is over, Alice should be able to open $C(x)$ to reveal x.

Want these properties:

- Binding: Alice should not be able to open $C(x)$ in more than one way.
- Secrecy: Anyone seeing $C(x)$ should have no information about x.
- Non-malleability: Anyone seeing $C(x)$ shouldn't be able to come up with a related bid, e.g. $C(x + 1)$.

Let's try $C(x) = h(\text{username}||x||r)$ where r is a random value which is secret to Alice. To make sure that the hash function satisfies all of the above properties, we need to have collision resistance (for binding), one-wayness (for secrecy, but we need a little more, we don't want to leak at any information on x), and non-malleability.

To open the bid, everyone sends in their bids once the auction is over, and you check to make sure the messages hash to their commitment value.

14.5. **Merkle Tree.** Authenticate a collection of objects $x_1, x_2, \ldots, x_n$. You go up the tree, computing hash values of two children below it. The hash value at the top of the tree is the root value. To check if some x is a member of the collection, then you must get x's brother, and you can compute up the tree if you are also given the other values of the nodes.

We have to have target collision resistance (to make sure you can't find another value that leads to the root hash), also collision resistance because maybe the guy making the tree made another tree as well.

15. Merkle-Damgard Construction

Start off the machine with a state of all zeros. Then you concatenate m_1 with the current state and hash this. This results in a new state c_1, and you concatenate m_2 with it and hash to result in c_2. Keep doing this for some number of iterations until you get $h(m)$ which is some partition of c_n.

Common design pattern for function f is $f(c_{i-1}, m) = C_{i-1} \oplus E(m_i, c_{i-1})$ where m_i is the key and c_{i-1} is the message.

16. Keccak

This is an iterative algorithm and there are two components to the state. The width is composed of $r + c$. You first take $r \oplus m_1$ and send that as well as c copied over to f, where f is a permutation of $\{0,1\}^{c+r} \to \{0,1\}^{c+r}$. It's a random looking object, and it's also public. One we've finished with all the message parts, then $h(m)$ is just r from the output state.

The compression is happening at the xor.

Lecture 6

17. The Web

The web works with http requests and responses.

17.1. **HTTP Request.** You have a method, path, and http version. The url specifies protocol, domain, port, etc using the DNS lookup system.

17.2. **HTTP Response.** Contains status code with reason text (200 is OK, 404 is not found, etc). You also have headers which contain server information, as well as possibly a cookie in the header. Then after the headers, there is the data content.

However, note that http server is stateless. Server and client don't know about each other (supposedly). You can maintain state using the cookies sent in the header of the HTTP response or a database on the server.

Cookies are files stored on the client. The database stores information about the users.

17.3. **Data Content.** The data content is the html file and references, where references are things like css, javascript, or flash plugins. The data content is structure in html in a tree structure.

18. Web security

18.1. **Authentication.** The goal of web security: safely browse the web. Users should be able to visit a variety of websites without incurring harm. Hackers shouldn't be able to steal, change, or read user's information.

Server authenticates and checks to make sure that a user U is indeed U. The most common method is passwords.

18.2. **Passwords.** The goal is to make guessing passwords the attacker's best strategy. User has a password and a username. In naive implementation the server has a database of usernames and passwords. The user sends the username and password. The server needs to check that these match. But stealing the database means all the username and passwords are given up.

Round 2: try storing the hash of the password. If the attacker steals the hashed password, you want one-wayness for the hash.

18.3. **Dictionary Attacks.** The attacker chooses a dictionary and picks password from the dictionary. Online attacks: the attacker doesn't have any way of figuring out if the password is correct other than sending a request to the server. To stop this, you can just rate limit. Offline attack, the attacker has the hash, so finding $h(pw)$ requires $O(|dict|)$.

But, you can try a batch offline dictionary attack. For every word in the dictionary, you computer the hash of the word. So now you have a list $L : \{w, h(w)\}$. Then you intersect this list with the database of passwords. Requires $O(|dict| + |T|)$ where T is the size of the intersection.

We can try to prevent these types of attacks by salting. Now, the server stores a username, password, and a random salt. The server then computes hash of the salt and the password. Therefore, the batch offline dictionary attack needs to compute passwords for every salt, so best attacker strategy is to just go through all possible passwords in the dictionary for each hash $O(|Dict||T|)$.

18.4. **Generating Multiple Client Passwords.** Problem with passwords is that people reuse passwords for each site. What we want is to create different passwords for different sites with some client software that can be accessed with a single password pw. We can hash $h(pw, server\_id)$ for each server. The hash needs to be one-way (given the hash don't want to be able to figure out the password) and non-malleability (given a hash of one website, I don't want to be able to figure out the hash of another website).

18.5. **Cookies.** Cookies are file stored by the server at the client. They help to maintain state, but they're also useful for authentication. Avoids sending the password over the network many times. Don't want to have to type in a password every time. You also don't want to send a password every single time you log in. However, if the adversary gets access to the cookie, then he has complete access to your account.

Cookies contain:

- name
- value (like user id, number of visits)
- domain (mit.edu)
- path (/courses/2013)
- expiration

Each browser contains a "cookie jar," these cookies are obtained in the following manner: user logs in with username and password. Server sends back an http response where the header contains a cookie. The next http request by the user uses the cookie that it obtained from the last time. The server now sends back an http response with an updated cookie.

These are dangerous, however, because of an adversary could just fake a cookie. We want the browser to not be able to come up with a fake cookie. To do this, we want a hash of username, expiration date, and a secret key from the server.

The server can now check that the cookie value is equal to the hash of username, expiration date, and secret key, this is how the server can check to make sure the cookies haven't been faked.

The hash needs to have one-wayness, non-malleability, but also unforgeability. Would suffice if hashes were random oracles.

19. Attacks on Web Applications

19.1. **SQL Injection.** Attacker sends malicious input to the server, but the server has bad input checking which leads to an actual sql query. For example, user sends the username and password. If the inputs come directly from the user without checking, you could send something like "or 1=1 –" which always evaluates to true then comments out the rest of the query. The authentication will always return true.

CardSystems had a SQL injection which allowed attackers to steal 263,000 credit card numbers. To fix these types of attacks, you need to sanitize inputs, make sure SQL arguments are properly escaped.

19.2. **CSRF: Cross Site Request Forgery.** Bad website sends a request to a good web site pretending to be the browser of an innoCent user, using the credentials of an innocent victim. Let's say the user has already authenticated with a good website. Then the user goes to a malicious webpage, and the malicious webpage returns something which asks the browser to use its cookie to authenticate with the good website.

Countermeasure: Good server needs to distinguish between good user and attacker. The user actually fetches a page, fills in the form for the request, and sends the request. The attacker never fetches a page, sends the request directly. Include some random token in the fetched page. When good user fetches a page, server embeds a random token in the forms and server stores it in the database.

When user sends the form, the token is sent to server along with the user cookie. The server checks to see that the token is correct. The attacker never knows the token because he never went to the page, he just tried to post a request using a url.

19.3. **XSS: Cross Site Scripting.** Attackers send data with script to server. Server stores it thinking it is data and serves it to other users. Now, when the website is visited by other users, the website pulls out the information from the database. However, when the browser renders the page, the browser executes the script. The script can steal all user cookies or other credentials to someone else or change the rest of the webpage to ask for credit card number.

Difficult to prevent, must employ a set of fixes. One way to fix is to escape special characters in any user-provided data before sending it to others.

Lecture 7

20. Buffer Overflow Overview

Buffer overflows are a code injection attack. Buffer overflows occur at a lower abstraction level than say, a SQL injection. They result in overwriting existing data in the memory. They exploit the layout of programs in memory. Also take advantage of how C deals with arrays.

20.1. Contents of Memory.

The contents of memory are as follows (from top to bottom of the memory stack):

(1) Program code
(2) Data segments for the global variables
(3) Heap. Grows downward. Dynamically allocated memory (e.g. malloc)
(4) Stack. Grows upward. Stores program execution context (e.g. function calls bool variables)

20.2. Stack Frames.

Here's an example C program:

```
void test(int a, int b) {
  int flag;
  char buffer[10];
}
```

On the stack, you have your return address, variables a and b, as well as a stack frame pointer. Then, the function dynamically allocates some memory using malloc and so you can make space for flag and buffer on the stack.

C string "hello world" is just an array of bytes that ends with a null byte. C functions like strcpy copies bytes from the string until it hits a null byte. If we strcpy "hello world" into the buffer, it will go and fill up the entire buffer, but it will still have more bytes so it will keep going and overwrite the first two flags of byte.

If you have too much user input that overwrites buffer, then you get a segmentation fault because you're trying to access memory locations which are invalid (because you've probably overwritten the return address to something that's invalid).

20.3. Consequences of Buffer Overflows.

Buffer overflows can:

- Crash programs
- Overwrite variables with new values
- Execute arbitrary code. You could change the return address to a new valid value. If you get the return address correct to point to a spot where you have put in arbitrary code, then that code will just start executing. One common location is the buffer itself.

20.4. Shell Code.

One of the most useful pieces of code to execute would be to open up a command shell. Shell code is code that opens up a command shell (*exec()*). However, its not so simple because you need to insert code which avoids null bytes (or else strcpy will stop copying to the buffer).

For x86, you can get shell code that is 31 bytes and which is composed of ASCII characters.

To get the attack to work, you can place a bunch of the same return addresses to the end of the buffer, and a bunch of NOPs (non-operation) to the beginning of the buffer before the shell code. Even if you don't know the exact location in memory, you can still get the program to go to the top to where the NOPs are placed, then run into the shell code. You don't need to know the exact location in memory.

20.5. **Return to libc Attack.** Pretty much every stack will have libc, which includes *printf()*, *exec()*. Set up stack to look like function calls to functions in libc.

21. Buffer Overflow Prevention

21.1. **Canary Values.** Insert a random value on the stack between the local variables and the control information. If that random value is changed to something else, then we probably have a buffer overflow. The value needs to be random so that the attacker can't predict it and just write it in on the buffer overflow.

You must check the canary value before returning. Typically, the value is stored in a register so that it will always be there. If it matches, then continue and return. However, if the value has been changed, you should just crash the program. Many major compilers use the canary idea.

21.2. **Safe Functions.** Have functions which make sure that everything you are handling is safe. For example *strncpy* which takes in a length as a parameter and copies until it hits a null byte or it reaches the length passed in.

If you only use safe functions, then you don't have to worry about buffer overflows. However, using safe functions require that you always use them and always use a valid, correct length.

21.3. **Non-executable Stack.** You should never be executing code in the stack. Execution context should never be in the stack, only at the text part of the program. This is often done in hardware, called NX. Also there are software emulations (ExecShield, $W\hat{X}$). These programs make sure that you don't have the ability to execute programs when you're in the stack, and you only have execution inside of the actual program code text.

21.4. **Address Space Layout Randomizaion (ASLR).** Make the guessing of the address space very difficult. The text, stack, global varaibles, and heap all live at a random point in address space. If you have a large address space, it probably doesn't matter that these are in a random place. Makes it more difficult for the attacker to supply a correct return address.

Lecture 8

22. Overview of Block Ciphers

Take in some plaintext p and a key k, then encrypt the plaintext to obtain ciphertext c. Each of p, c, k depend on the standard.

- Data Encryption Standard (DES): $|p| = |c| = 64$ bits, $|k| = 56$ bits.
- Advanced Encryption Standard (AES): $|p| = |c| = 128$ bits, $|k| = 128, 192, 256$ bits.

The above is the framework for a block cipher, which is usually the building block of more advanced crpytographic techniques.

23. Data Encryption Standard (DES)

Feistel Cipher: created a structure which you can undo. Feistel proposed an input block divided into two parts L_0 and R_0. Based on multiple rounds of encryption. In the first round, you take R_0 and call a function on $f(R_0, K_0)$, then xor the result with L_0. Once you are done, the new result becomes R_1 and the old R_0 becomes L_1. That is a single round of the algorithm, do this 16 times.

Notice that this is an invertible operation, even though f doesn't need to be invertible.

24. Types of Attacks

24.1. **Differential Analysis.**

Invented in the public domain by Shamir and Biham. Lets take a message M and run it through the encryption box, and it returns c. Now take $M \oplus \Delta$ where Δ is a small change in the message. This causes a result $c + \delta$. What happens when you change some bits of the message. You can track the changes of the bits down the entire structure.

It improves the number of things you need to run. Instead of 2^{56} brute force attacks, a differential attack could lead to only 2^{47} chosen messages. This is strictly better than what you could do if you didn't have the differential attack.

24.2. **Linear Attacks (Matsui).**

The idea is that maybe f is non-linear but can be approximated by something linear. Maybe most of the time it is linear. For example, suppose that the equation below was satisfied with probability $1/2 + \epsilon$:

$$M_3 \oplus M_{15} \oplus C_2 \oplus K_{14} = 0 \tag{17}$$

Run a bunch of message pairs, get the cipher text and solve for the key bit. Since this is true with probability $1/2 + \epsilon$, you can just take the average and it will give you the correct answer using the law of large numbers.

This actually works even better than differential attacks, only 2^{43} pairs needed.

25. Advanced Encryption Standard

Rijndael was the winner. It has a fixed length input and output block of 128 bits. They key size is either 128, 192, or 256 bit keys and you can have 10, 12, or 14 rounds.

Byte-oriented design. Can think of these as an element in $GF(2^8)$. You can think of 128 bits as a 4 by 4 array of bytes. Derive round keys which are each 128 bits (the keys are different for each round). Now each round has 4 steps:

(1) XOR the round-key into the message to derive the 4 by 4 table.
(2) Substitute bytes from a lookup table. Takes an 8 bit input and gives an 8 bit output. This is the only nonlinear operation in the standard.
(3) Rotate rows, each by different amounts. If bytes were in the same column, they are rotated so they are no longer in the same column.

(4) Mix each column to become $C \leftarrow AC$ where A is some matrix.

The above steps are run for each round and the final state is outputted from the algorithm.

26. Ideal Block Cipher

There are now two inputs, a key and a message. The ciphertext only needs to be invertible for each key. The ideal block cipher: *each key independently has a random permutation of the message space.* Thus, for each key, we have different random permutation of the message space. The information from one key doesn't give you any information about the other key.

27. Confidentiality

We want confidentiality, aside from the length of the message. Variable-length inputs from some message space $\mathrm{M} = \{0, 1\}^*$. How do we make sure we get confienitiality?

27.1. **Electronic Code Book (ECB).** We take our message and divide it into blocks $M_1, M_2, \dots, M_n$. Get ciphertext for each block using some key to get $C_1, C_2, \dots, C_n$. The concatenated result is the full ciphertext.

We need to make sure that the message ends up to be exactly 128 bits to make sure we can use block ciphers. To make sure this is the case, we can just pad our messages to make sure that this is true. We want the padding to be invertible (because we'll have to remove the padding). A typical approach is to always append a 1 and then as many 0s as necessary. If we always append a 1 no matter what, then we're fine no matter what message was inside.

This isn't very good, though, because repeated patterns in the message usually end up as repeated pattern in the cipher text, since the same key is being used the entire way through.

Because of this issue, ECB is pretty much deprecated except for very short messages or random data.

27.2. **CTR Mode.** We start a 128 bit counter at some bit i. We encrypt the counter i with some secret key K and get x_i. Then, we XOR this random looking quantity x_i with our first message block M_1 to get C_1. To get more ciphertext, add one to the counter, encrpyt it to be x_{i+1} and XOR M_2 with x_{i+1} to get C_2. Repeat until you get to the end.

The set of $x_i, x_{i+1}, \dots, x_{i+n-1}$ is equivalent to the pad. Once you have this, then you send $i, C_1, C_2, \dots, C_n$. The legitimate decrypter just needs to encrypt $i, i+1, \dots, i+n-1$ with K, which allows you to XOR x_i with C_1 to get M_1.

Lecture 9

28. Cipher Block Schemes

28.1. **Cipher Block Chaining (CBC).** There exists some initialization vector (IV) which is chosen at random by the sender. This is a starting point, but it's not a secret key. We do $IV \oplus M_1$ then encrypt the result to get ciphertext C_1. For the next block of message, we could XOR with IV like before, but that's not as good because we could give away the encryption. Therefore, we use the previous ciphertext and get $C_1 \oplus M_2$ and encrypt that to get C_2.

The basic pattern is then $C_{i-1} \oplus M_i$, which is encrypted to obtain C_i. Think of $C_0 = IV$. Thus, even if you change one bit, it changes the entire pattern down the line.

Ciphertext stealing is a cute trick for dealing with messages that are not exactly 128 bits in length. We transmit $IV, C_1, C_2, \ldots, C_n$. Decryption steps can be processed by decrypting C_1 and XORing it with IV to obtain M_1. Continue onwards until you have all M_i.

Interesting thing about CBC is that a single bit change will ripple down through the chain.

28.2. **Cipher Feedback Mode (CFB).** You have an initialization vector (IV) and a secret key K. You encrypt the IV using the secrety key and XOR the resulting value with M_1. This results in C_1. Then, encrypt C_1 and XOR the resulting value with M_2 to get C_2. Continue XORing the encrypted block of C_{i-1} with M_i to get C_i.

To decrypt we start off encrypting IV and XOR with ciphertext to get the message. Continue onwards in this manner.

Very similar to Cipher Block Chaining, and has many of the same properties like single bit rippling.

29. Ciphertext Indistinguishability

To prove that our cipher block schemes are secure, we need to define what it means to be secure. Let's define a game with the adversary. Our mode is secure if the adversary can't win the IND-CCA (Indistinguishability under Chosen Ciphertext Attack) game with probability significantly more than half the time.

Let's set up the game. The adversary is going to be asked, what is the message behind the ciphertext? The adversary will supply the plaintext messages to be encrypted, and he needs to guess which one is being encrypted. The key is chosen ahead of time in a random fashion.

Phase 1: ("Find"). Adversary makes up two messages $m_0 \neq m_1$ such that $|m_0| = |m_1|$. Adversary is given access to encryption oracle $E_k(.)$ and decryption oracle $D_k(.)$.

Phase 2: ("Guess"). Examiner picks $d \leftarrow \{0,1\}$ is a random bit (either 0 or 1). Then examiner encrypts message d and obtains $y = E_k(m_d)$. Adversary is given y and any state information s. Remember, adversary has access to $E_k(.), D_k(.)$ except on y. The adversary then gives $\hat{d}$ which is the adversary's guess for d.

The adversary's advantage is $|Pr[\hat{d} = d] - \frac{1}{2}|$. The scheme is secure against IND-CCA if adversary's advantage is negligible (should be done in probabilitistics polynomial time).

To be secure in IND-CCA, encryption must be randomized, otherwise the adversary could just pick two messages beforehand and compute the ciphertext. Notice that under IND-CCA, the CBC and CFB modes are not secure. You could just send the decrypter the first half of the cipher text, and you will get the first half of the message.

This is the strongest definition of security that we know about.

30. Unbalanced Fiestel Encryption Mode (UFE)

Let's say that we have a message M and we're going to use CTR mode for some key k_1. We use counter on r and XOR result with M. This results in ciphertex which is then sent to CBC MAC with IV = 0 and a new secret key k_2.

We have $k = (k_1, k_2, k_2)$ and $r \leftarrow \{0,1\}^b$ randomly chosen. The pad is $P = P_1, P_2, \ldots, P_n$ where $P_1 = E_{k_1}(r + i)$. The resulting ciphertext is $C = c_1, c_2, \ldots, c_n$ where $c_i = m_i \oplus P_i$. Then in CBC-MAC we have $x_0 = 0^b$ and set $x_1 = E_{k_2}(x_{i-1} \oplus c_i)$ and $x_n = E_{k_3}(x_{n-1} \oplus c_n)$.

Result is $\sigma = r \oplus X_n$ and $c_1, c_2, \ldots, c_n$.

Idea is that you send r implicitly through σ. The receiver can obtain r by sending C through the CBC-MAC and obtain X_n, so then you can XOR X_n with σ to obtain r. This allows you go to go ahead and compute P, which gives you the message by doing $P \oplus C$.

31. Message Authentication Codes (MAC)

Alice wants to send some message to Bob, and Bob wants to be able to know if Eve has intercepted the message and changed the message. Alice sends M and $MAC_k(M)$ which is a message authentication code. Bob can check if $MAC_k(M)$ is correct by recomputing it. It can't be recomputed by Eve because she doesn't know the secret key k. If Bob doesn't get the correct message authentication code, then he knows Eve has changed the message.

If $MAC_k(M)$ is correct, then Bob proceeds, otherwise Bob rejects. Remember that Eve can replay old messages and $MAC_k(M)$.

31.1. **MAC Game.** Alice and Bob share K. Eve wins if Bob accepts $(M', MAC_k(M'))$ where M' is different from anything that Eve has heard on the line. We give Eve oracle access to the MAC_k function, but these queries don't count as wins for Eve.

Eve wants to forge a new message authentication code. A good scheme should not allow an adversary to create valid MACs which have not already been seen.

Lecture 10

32. Authentication

Eve should not be able to fool Bob that Alice didn't actually create. This is the core of authentication. Eve can forge with probability $2^{-|T|}$ where T is the length of the MAC. Eve wins the game if she can forge with higher probability than that.o

32.1. CBC-MAC.

Start with message M_1 and a vector of zeros. Take $0 \oplus M_1$ and encrypt the result. The result of this encryption is sent to be XORed with M_2. Continue encrypting the result and sending this encrypted result to XOR with M_i until you get to the last block. The encryption in the first $n-1$ blocks all use the same key K. The last block, however, uses a different key K'. The resulting output of the last block is the MAC that you send over.

32.2. PRF-MAC Hash Function.

Use the following MAC: $MAC_k(M) = H(k||m)$ and truncate to τ bits. This is actually defective in practice because the hash functions are computed in a sequential manner so you might get a length extension attack. Sometimes people can get m' which is just an extension of m so that the entire m' isn't used in computing the hash, so that the MAC is defective.

Isn't necessarily used very often in practice because of this. Better idea is:

$$HMAC_k(M) = H(K_1||H(K_0||M)) \tag{18}$$

Where $K_0 \leftarrow K \oplus (pad)_0$ and $K_1 \leftarrow K \oplus (pad)_1$. HMAC is used much more frequently in practice.

32.3. Combining MAC and Encryption.

People really want something that does encryption and mac at the same time. We encrypt then MAC.

$$ENC_{k_1}(M) \rightarrow CMAC_{k_1}(C) \rightarrow T \tag{19}$$

Then, you send $\{C, T\}$. The receiver can check T and see if you want to decrypt the message. Make sure the ordering is that you encrypt, compute MAC, then on the otherside decrypt the MAC, and decrypt. You can get into difficulties if you don't do it the right way.

33. EAX Mode: Authenticated Encryption

Bellare, Rogaway, Wagner created EAX mode. *Nonce* is a value only used once, never repeats, could be a counter. $E_k(N)$ looks random. Associated data should be authenticated but doesn't need to be encrypted. We will call this the header H. Message M to be encrypted.

EAX uses one key and is online – no need to know length ahead of time.

33.1. Workings of EAX.

Takes as input variable τ, variable length nonce, arbitrary block cipher. Take the message M, and send it through counter mode with key K, results in ciphertext C. The starting value of the counter mode comes from the nonce N. The MAC of the nonce creates the starting counter value. The ciphertext is sent through a MAC of key K, as is the header, and the MACs of the nonce, message, and header are all XORed together and the result is T.

Use $MAC_k^i(m) \approx CBC - MAC(i||m)$ as the MAC, using $i = 0$ for the nonce and $i = 2$ for the message and header.

The following is transmitted: N, C, H, T.

34. Finite Fields

Set S and two binary operations $\times, +$.

- S is finite containing 0 and 1.
- $(S, +)$ is an abelian (commutative) group.
  - Identity $0 : a + 0 = a$
  - Associativity: $((a + b) + c) = (a + (b + c))$
  - Inverses: $\forall a \ \exists b$ s.t. $a + b = 0$
- $(S^*, \times)$ is an abelian group with identity 1. $a \times 1 = 1 \times a = a$, $a \times b = b \times a$. $\forall a \in S^*$ $\exists b \in S^*$ s.t. $a \times b = 1$.

Cryptography likes finite sets. Take $Z_p = GF(p)$, how do we solve $0 = ax + b \mod p$? We just do $a = a^{-1}(-b) \mod p$.

$GF(q)$ exists for $q = p^k$ for any $k \geq 1$. For example $GF(2^8)$. Polynomials in x with degree less than 8. Coefficient in $GF(2)$ is $1 + x + x^5 + x^7$ mod out by some fixed polynomial $f(x)$.

Repeated squaring to compute a^b where a is in the field and $b \geq 0$ is an integer.

$$a^b = \left\{ \begin{array}{cc} 1 & \text{if} b = 0 \\ (a^{b/2})^2 & \text{if} b > 0, b \pmod 2 = 0 \\ aa^{b-1} & \text{if} b > 0, b \pmod 2 = 1 \end{array} \right\} \tag{20}$$

Theorem: In $GF(q)$ for all $a \in GF(q)^*$, we have $a^{q-1} = 1$. Thus, $a^q = a$ for all $a \in GF(q)$. This means that $aa^{q-2} = 1$. This means that $a^{-1} = a^{q-2}$.

Lecture 11

35. Finite Fields

Finite fields $(S, +, \times)$ where S is a finite set. $GF(q)$ is the finite fields with q elements.

Theorem: $GF(q)$ exists if and only if $q = p^k$ for some prime p and integer $k > 0$.

Usually, if p is prime then $GF(p) = Z_p$ so we have addition modulo p and the field is $Z_p = \{0, 1, \ldots, p-1\}$. But consider $GF(2^2)$, we can't just take Z_4 because 2 does not have an inverse. Elements are polynomials in x of degree less than k with coefficients in $GF(p)$. So for $GF(2^2)$ we have: $\{0, 1, x, x+1\}$. Notice that coefficents are modulo 2. We just need to defined addition and multiplication for these. Addition is easy: take modulo 2 in each position, for example $x + (x+1) = 2x + 1 = 0x + 1 = 1$. Multiplication is a little trickier, but we do something very similar to $GF(p)$. Work modulo a polynomial of degree k which is irreducible. x^2, x^2+1, x^2+x are all reducible, so we should use x^2+x+1. All multiplication is done modulo this polynomial.

We can build up a multiplication table:

(21)

	0	1	x	$x+1$
0	0	0	0	0
1	0	1	x	$x+1$
x	0	x	$x+1$	1
$x+1$	0	$x+1$	1	x

What is $x^2 \mod x^2+x+1$? We have 1 remainder $-x-1$ but we know that $-x = x \mod 2$ and we know that $-1 = 1$. So we have $x+1$ as the remainder. So $x^2 = x+1$.

35.1. **Computing Powers.**

Given the ability to compute products, we can compute a^b in time $O(\log b)$. One of the reasons this is useful is because we have Euler's Theorem: in $GF(q)$, $forall a \in GF(q)^*$ then $a^{q-1} = 1$. Note that $GF(q)^*$ is all elements in $GF(q)$ which are non-zero. Want to work in $GF(p)$ but we need to find a large prime number.

Finding large prime numbers: turns out to be easy. This is because primes are relatively common.

35.2. **Generate and Test Primes.**

Generate a number p of the desired size at random and test it for primality. This is good as long as there are a sufficient number of primes and the test for primality is cheap.

Prime number theorem: Number of primes less than x is equal to $x/\ln(x)$. For us, $x = 2^k$, so we have $\approx 2^k/k$. There are 2^k total numbers of k bits, so you have $1/k$ probability of finding a prime. This is pretty good.

35.3. **Testing for Primality.**

If p is prime then $\forall a \in Z_p^*$ we have $a^{p-1} \equiv 1 \pmod p$. Going backwards the other way doesn't always work (Carmichael numbers) but they work with high probability. It almost suffices to test $2^{p-1} = 1 \pmod p$ for large p. In good implementations, you probably want to check to make sure these aren't Carmichael numbers.

36. One-time MAC

Recall the basic setup of Alice sending messages to Bob where both of them have a secret key K. Eve tries to listen in and has infinite computational power, but doesn't have the key. She tries to intercept the message and the MAC tag (M, T) and send a changed message (M', T'). Eve wins the game if Bob accepts the altered message with probability greater than chance.

$T = MAC_k(m) = aM + b \pmod{p}$ where p is a prime and the key is $K = (a, b)$ where $0 \leq a, b \leq p$. The message satisfies $0 \leq M \leq p$.

Theorem: Eve can only achieve success with probability $1/p$ for the One-Time MAC. *Proof:* Given M'. For each T' that she could guess, there exists a unique key $K = (a, b)$ such that both $T = aM + b \pmod{p}$ and $T' = aM' + b \pmod{p}$. Both of these conditions must hold in order for Eve to succeed. Need to find $a = (T - T')/(M - M')$, and we can do this because $M \neq M'$. Then we can solve for $b = T - aM$. There is exactly one key that makes this work. Therefore, Eve's chances of making M', T' work is the chance that Alice and Bob have picked the key. The probability of picking that particular key is $1/p$ because Alice and Bob already have the equation $T = aM + b$, so picking $T' = aM' + b$ requires probability $1/p$.

37. Number Theory

$d|a$ means that d divides a so that there exists a k such that $dk = a$. We know that d is a divisor of a if $d \geq 0$ and $d|a$. For all d, $d|0$, and for all a, $1|a$.

If d is a divisor of a, and d is a divisor of b, then d is a common divisor of a and b. The greatest common divisor is the largest of the common divisors. We specify $gcd(0, 0) = 0$ by definition and $gcd(0, 5) = 5$. We can compute $gcd(24, 30) = 6$, etc.

Definition: two numbers a and b are relatively prime if $gcd(a, b) = 1$.

37.1. **Euclid's Algorithm for GCDs.**

For $a, b \geq 0$, we have:

$$gcd(a,b) = \begin{cases} a & \text{if } b = 0 \\ gcd(a, a \mod b) & \text{else} \end{cases} \tag{22}$$

We see that $a \mod b$ is strictly less than b, so we know this will terminate.

One of the most important implications of this is to compute multiplicative inverses when two numbers are relatively prime. Suppose $a \in Z_p^*$ and we want to compute $a^{-1} \pmod{p} = a^{p-2} \pmod{p}$. This works when p is prime, but if p is not prime, and $a \in Z_n^*$, then we need to find $ax + ny = 1$ so that $ax = 1 \pmod{n}$ and $x = a^{-1} \pmod{n}$.

Lecture 12

38. Group Theory

38.1. **Orders of Elts.** The most common group we see in cryptography is $Z_p^* = \{1, 2, \ldots, p-1\}$. This is a pretty group, very simple, and nice to work with. We say that $order_n(a)$ is the order of a modulo n, where $a \in Z_n^*$. The order is the smallest $t > 0$ such that $a^t \equiv 1 \mod n$.

Fermat's Little Theorem: If p is prime, then $\forall a \in Z_p^*$ we have $a^{p-1} \equiv 1 \pmod p$.

Euler's Theorem: For all n and forall $a \in Z_n^*$, we have $a^{\phi(n)} \equiv 1 \pmod n$, where $\phi(n)$ is the cardinality of the group so $\phi(n) = |\{a; 1 \leq a \leq n; gcd(a, n) = 1\}|$.

For example: $Z_{10}^* = \{1, 3, 7, 9\}$. This is a multiplicative group, and $\phi(10) = 4$. Just to check, we have $3^4 \equiv 1 \pmod{10}$.

Notice that the order of a mod n is the length of the periodicity of a^i for $i = \{1, 2, 3, \ldots\}$. We notice that $order_n(a)||Z_n^*|$ so that the order always divides the size of the group.

38.2. **Generators.** Notation: define $< a >= \{a^i, i \geq 1\}$. We have $order_n(a) = | < a > |$.

Definition: If $order_n(g) = |Z_n^*|$, then g is called the generator of Z_n^*. In other words $< g >= Z_n^*$.

Theorem: Z_n^* has a generator (i.e. Z_n^* is cyclic) if and only $n = 2, 4, p^m, 2p^m$ for prime p and $m \geq 1$.

Theorem: If p is prime, then the number of generators mod p is $\phi(p-1)$. For example $p = 11$ then $|Z_p^*| = 10$, and we have $\phi(10) = 4$.

Theorem: If p is prime and g is a generator modulo p, then $g^x \equiv y \pmod p$ has a unique solution for $0 \leq x < p-1$ for each $y \in Z_p^*$. We call x the discrete logairthm of y, base g modulo p.

The discrete logarithm problem is the problem of computing x from y given some p and g. It is assumed that this is hard, no one has found an algorithm which solves this problem quickly.

38.3. **Generate and Test.** Randomly pick g in Z_p^* and test its order. If the order is $p-1$, then g is the generator. Fermat's Little Theorem implies that $g^{p-1} = 1 \pmod p$ and $g^d \not\equiv 1 \pmod p$ for d which is a divisor of $p-1$.

Assume we factored $p - 1 = q_1^{e_1} q_2^{e_2} \ldots q_k^{e_k}$. Check that $g^{(p-1)/q} \not\equiv 1 \pmod p$ for each $q|(p-1)$.

39. Public Keys

Idea: Pick a large "random" p such that we know the factorization of $p-1$. This is because factoring $p-1$ is hard.

If p and q are both primes and $p = 2q+1$, then p is called a safe prime and q is called a Sophie Germain prime. The factorization of $p-1$ is just $2q+1-1$ so that factorization is $p-1 = 2 \times q$. To get pairing, we find q first which is a prime, then let $p = 2q+1$ and we test p for primality.

Theorem: If p is a safe prime $p = 2q+1$ $(p-1 = 2-q)$ then for all $a \in Z_p^*$ we have $order(a) \in \{1, 2, q, 2q\}$.

To test of g is a generator we know that we check if $g^{p-1} \equiv 1 \pmod p$ and $g^2 \not\equiv 1 \pmod p$ and $g^q \not\equiv 1 \pmod p$, which implies that $order_p(g) = p-1$ and so g is a generator.

If $p = 2q+1$ and so p is a safe prime then the number of generators mod p is $\phi(p-1) = q-1$.

Find large random prime q. Let $p = 2q+1$ and test p for primality. If not, loop and pick new q, otherwise output p, q, g. We can find g by picking $g \in Z_p^*$ and test for g being a generator. Since there are a lot of generators, we've got high probability of getting a generator.

39.1. **Common Public Key Setup.** We have p prime and g generator of Z_p^* which are published. Alice chooses x where $0 \leq x < p-1$ as her secret key. Publishes $g^x \equiv y \pmod{p}$ as her public key. Secrecy of x depends on difficulty of computing $x = \log_{g,p}(y)$ (discrete logarithm problem).

Lecture 13

40. Group Theory Review

$(G, *)$ is finite abelian group of size t:

- Exists an identity 1 such that $\forall a \in G$ we have $a \cdot 1 = 1 \cdot a = a$.
- Inverses. For all $a \in G$ there exists a $b \in G$ such that $ab = 1$.
- Association. For all $a, b, c \in G$ we have $a(bc) = (ab)c$.
- Abelian. For all $a, b \in G$ we have $ab = ba$.

The order is defined $order(a)$ as the least $u > 0$ such that $a^u = 1$.

Theorem: In a finite group of size t, for all $a \in G$ we have $order(a)|t$.

Theorem: For all $a \in G$ we have $a^t = 1$. For example in Z_p^* we have $a^{p-1} \equiv 1 \pmod p$ since $|Z_p^*| = p - 1$.

Definition: $\langle a \rangle = \{a^i : i \geq 0\}$ is the subgroup generated by a.

Definition: If $\langle a \rangle = G$ then G is yclic and a is a generator of G. Note that $|\langle a \rangle| = order(a)$. As an exercise, if t is prime and G is finite abelian of order t, then for all $a \in G$ we have $[a \neq 1] = a$ is a generator of G. Fact: Z_p^* is always cyclic.

Fact: If G is a cyclic group of order t and g is a generator of G, then the mapping $x \to g^x$ is one-to-one between $[0, 1, \dots, t-1]$ and G.

40.1. **Programming with Groups.** API for a group:

- G ← create_group
- g ← G.generator() (could fail if the does not exist a generator)
- identity, G.identity()
- G.order_elt(a) (could be really hard)
- G.inverse(a)
- G.cyclic → true/false
- G.product(x,y)
- G.random_element
- G.order()
- G.exponentiation(a,k)
- G.elements()
- G.rep(M) (some element in the group representing message M).

41. Diffie-Hellman Key Exchange

Alice and Bob want to be able to talk openly about their new secret keys. They don't start off with any shared information at all. Suppose Alice generates secret key x and Bob generates secret key y from the group. Now Alice sends $a = g^x$ to Bob and Bob sends Alice $b = g^y$.

Alice can compute $b^x = (g^y)^x = g^{xy}$ and Bob can compute $a^y = (g^x)^y = g^{xy}$. The new secret key that is shared between Alice and Bob is $K = g^{xy}$.

Given $a = g^x$, Eve can't compute x because we're assuming the discrete log problem is hard. Same thing for $b = g^y$. Given a, b, can Eve compute $K = g^{xy}$. This is a new problem.

Computational Diffie-Hellman (CDH): Given a, b can you compute g^{xy} where $a = g^x$ and $b = g^y$. This is assumed to be hard.

Theorem: If CDH is hard, then Diffie-Hellman key exchange is secure.

42. Different Types of Groups

We've already learned about a number of groups:

- $Z_p^* = \{a : 1 \leq a \leq p\}$ where p is prime. We know that Z_p^* is always cyclic and if $p = 2q + 1$ then have of Z_p^* are generators.
- Q_p is the set of quadratic residues (squares) modulo p. So $Q_p = \{a^2 : 1 \leq a \leq p\}$. Notice that $Q_p \subset Z_p^*$. We also know that $|Q_p| = \frac{1}{2}|Z_p^*|$ because exactly two elements of Z_p^* make into a single element in Q_p. Also, Q_p is cyclic, so if $\langle g \rangle = Z_p^*$ then $\langle g^2 \rangle = Q_p$. If $p = 2q + 1$ then Q_p is cyclic and any element not the identity in Q_p is a generator.
- $Z_n^* = \{a : gcd(a, n) = 1, 1 \leq a < n\}$. We see that $|Z_n^*| = \phi(n)$ by definition. If $n = pq$ where p, q are distinct odd primes, then Z_n^* is not cyclic and $Z_n^* = Z_p^* \times Z_q^*$.
- $Q_n = \{a^2 : 1 \leq a \leq n, gcd(a, n) = 1\}$. If $n = pq$ and $p = 2r + 1, q = 2s + 1$ are safe primes, then $|Q_n| = rs$. $|Q_n|$ is cyclic.

42.1. **Elliptic Curves.** Working in Z_p and let a, b be elements such that $4a^3 + 27b^2 \not\equiv 0 \pmod{p}$. Consider $y^2 = x^3 + ax + b \pmod{p}$. Then E_{ab} is the elliptic curve and is the set of (x, y) satisfying the equation modulo p.

Lecture 14

43. Pedersen Commitments

43.1. **Commitment Scheme Overview.** Commitment Scheme Motivation: auctions. You want to commit to your bid, so you have to have the same bid as before. This can be thought of as a commitment. $\mathit{Commit}(x)$ is a commitment to x. $\mathit{Reveal}(x)$ opens the commitment and reveals x.

For commitment schemes, you want to have the following properties:

- Hiding: $\mathit{Commit}(x)$ shouldn't reveal anything about x.
- Binding: Can only open commitment in one way, i.e. you can't find x' such that $\mathit{Commit}(x') = \mathit{Commit}(x)$ where $x' \neq x$.
- Non-malleability: Can't create $\mathit{Commit}(x+1)$ from $\mathit{Commit}(x)$.

43.2. **Pedersen Commitment Scheme.** Our setup is as follows: we need two large primes p, q such that $q|p-1$ (e.g. p is a safe prime so that $p = 2q+1$). Let g be a generator of order q subgroup of Z_p^*. Squares modulo $p = Q_p$ has order q. We let $h = g^a$ where a is secret. It's hard to find a because the discrete log problem is hard.

We have $\mathit{Commit}(x)$ where $x \in Z_q$ so $0 \leq x < q$. Sender chooses some random $r \in Z_q$ and we have $\mathit{Commit}(x) = c = g^x h^r \pmod{p}$. To reveal, you give x, r and the receiver checks that $c = g^x h^r \pmod{p}$.

43.3. **Security of Pedersen Commitment Scheme.** Perfectly Hiding: $c = g^x h^r \pmod{p}$. c could be a commitment to any other x' as well. Suppose $g^x h^r \equiv g^{x'} h^{r'} \pmod{p}$. Can this hold? Since $h = g^a$ we have $g^{x+ar} = g^{x'+ar'} \pmod{p}$. Since g is a generator, exponents must be the same modulo the order, so we have $x + ar \equiv x' + ar' \pmod{q}$. This says that $r' \equiv (x - x')/a + r \pmod{q}$.

Note that we can divide modulo q because q is prime, so we can divide modulo q, since a is non-zero. We have just argued that c could be a commitment to any value x! This seems like maybe I'm at risk of not really being committed to anything.

We will show that we're actually computationally bound, even though it is theoretically possible for the committed value c to be opened in more than one way. In principle, can you open c as (x, r) and (x', r')?

Suppose $g^x h^r = g^{x'} h^{r'} \pmod{p}$. Then $x + ar = x' + ar' \pmod{q}$. We get $a = (x - x')/(r' - r) \pmod{q}$ because $r' \neq r$. Computing a is hard because it is the discrete log of h base g (mod p). Therefore, you're computationally bound because the sender can't compute a.

43.4. **Malleability.** However, this scheme is malleable. Consider $c = g^x h^r \pmod{p}$. But you can just get a commitment $c' = gc = g^{x+1} h^r \pmod{p}$ which is one higher than the other scheme. If the first person gives his x, r before you do, then you're home free.

44. Public Key Encryption

Let λ be a security parameter, which approximately gives you the key size. $1^\lambda = 11\ldots1$ is a set of ones of length λ. For key generation you have the following steps:

1. $\mathit{Keygen}(1^\lambda) \to (PK, SK)$ which runs in polynomial time $\mathit{poly}(\lambda)$ and it must be randomized.
2. $\mathit{Enc}(PK, m) \to c$. Takes some message $m \in \mathrm{M} \to c \in \mathrm{C}$ and is probably randomized.
3. $\mathit{Dec}(SK, c) \to m$. Deterministic operation which should work for all (PK, SK) pairs and all messages m.

44.1. **ElGamal Public Key Encryption.** Let $G = \langle g \rangle$ be a cyclic group with generator g. $G \in Z_p^*$ where $|p| = \lambda$ bits.

Keygeneration: We pick x at random from $[0, |G|-1]$ and let $SK = x$, $PK = g^x \pmod{p}$. We then output (PK, SK).

Encryption: Take $m \in G$ and pick k at random from $[0, |G|-1]$. Recipient's PK is $y = g^x \pmod{p}$. The shared secret key is going to be $y^k = (g^x)^k = g^{xk} \pmod{p}$. Output (g^k, my^k).

Decryption: We have $c = (a, b)$ be the received ciphertext. We can compute $y^k = g^{xk} = (g^k)^x$ because we get $g^k = a$. Therefore, we can get our message back by doing $m = b/a^x$.

Like DH key exchange, the receiver provides $y = g^x$ which is PK and the sender provides $a = g^k, mg^{xk}$.

44.2. **Security of ElGamal.** Let's analyze whether this is a secure scheme. Here are the phases of the game:

- Phase I ("Find"): Examiner generates (PK, SK) via $Keygen(1^\lambda)$. PK is then sent to the adversary. Adversary computes (in time polynomial in λ) and outputs two messages m_0, m_1 of the same length and state information.
- Phase II ("Guess"): Examiner picks $b \leftarrow \{0, 1\}$ which is randomly generated and computes $c_* = Enc(PK, m_b)$. The result c_* is given to the adversary. Adversaries compute for time (polynomial in λ) and outputs his guess $\hat{b}$ for b. The adversary wins if $\hat{b} = b$.

In this game it takes no skill to win with probability $\frac{1}{2}$. The scheme is *semantically secure* if the probability that the adversary wins is less than $\frac{1}{2} + \epsilon$ where $\epsilon = 2^{-\lambda}$ which is the probability of randomly choosing SK correctly.

Recall computational Diffie-Hellman (CDH) which is computing g^{ab} from g^a and g^b. We now have Decision Diffie-Hellman (DDH): distinguishing (g^a, g^b, g^{ab}) from (g^a, g^b, g^c) is hard. These problems are related and DDH implies CDH.

Theorem: ElGamal is semantically secure if and only if DDH holds in G.

Lecture 15

45. ElGamal: Malleability and Homomorphisms

Recall that ElGamal works modulo p. We have $SK = x$ and $PK = g^x = y$. We encrypt a message by $Enc(m) = (g^k, my^k)$ where k is randomized.

Unfortunately, ElGamal is malleable. If all you see is the ciphertext, is it hard for you to get an encryption of a related message? $Enc(2m) = (g^k, 2(my^k))$ so it's pretty easy to multiply by 2 and get a new ciphertext.

It's even stronger than that though, we have homomorphic encryption.

45.1. El Gamal is Homomorphic.

We have $c_1 = Enc(m_1) = (g^r, m_1y^r)$ and a second cipher text $c_2 = Enc(m_2) = (g^s, m_2y^s)$. We can obtain $c_1c_2 = (g^{r+s}, (m_1m_2)y^{r+s}) = Enc(m_1, m_2)$.

Homomorphism could be wonderful or bad depending on what you want. For example, in CryptDB, a homomorphic encryption scheme is awesome. Computations can be done on the ciphertext which actually do operations on the underlying plaintext.

To get a new encryption with a different secret key, pick some new secret key t and compute g^t and y^t. Then we can use $(g^tg^r, m_1y^ry^t) = (g^{r+t}, m_1y^{r+t})$ and this rerandomizes the ciphertet.

For many applications, homomorphism is not what you want because you don't want the adversary to be able to change your inputs.

46. IND-CCA2 Security

Phase I:

- Examiner generates (PK, SK) using $Keygen(1^\lambda)$.
- Give the PK to the adversary.
- Adversary gets time to compute in time polynomial in λ which access to $Dec(SK, x)$ oracle. (This is a new ability where he gets access to the decryption box).
- Adversary outputs m_0, m_1 of the same length and notes s which will be carried over to phase II.

Phase II:

- Examiner picks $b \leftarrow \{0, 1\}$ randomly and computes $c_* = Enc(PK, m_b)$ and sends c_* to adversary.
- Adversary computes for time polynomial in λ with access to $Dec(SK, *)$ except on c_*. Adversary outputs $\hat{b}$ (his guess for b) and wins if $\hat{b} = b$.

Definition: PK enryption method is *INDC-CCA2* secure (ACCA-secure) if $Prob(\text{Adv wins}) \leq \frac{1}{2} + \epsilon$ where $\epsilon = 2^{-\lambda}$.

46.1. El Gamal and IND-CCA2 Security.

Let's say you have $c_* = (g^r, m_by^r)$. You can make a new message $c'_* = (g^r, (2m_b)y^r)$ and decrypt to get $2m$ as the message. You can therefore trivially get m. Thus, ElGamal is not IND-CCA2 secure.

47. Cramer Shoup

We have a group G_q of size q which is prime (this can be done by finding a safe prime p and just deal with the group of quadratic residues modulo p). We'll have a keygen algorithm $g_1, g_2 \leftarrow G_q$ picked at random. We will have $SK = x_1, x_2, y_1, y_2, z \leftarrow Z_q$ picked at random. We set $c = g_1^{x_1}g_2^{x_2}$ and $d = g_1^{y_1}g_2^{y_2}$, as well as $h = g_1^z$.

We will now have a public key $PK = (g_1, g_2, c, d, h)$.

For encryption, we do $Enc(m)$: $r \leftarrow Z_q$ picked randomly, $u_1 = g_1^r$, $u_2 = g_2^r$, $e = h^r m$, $\alpha = H(u_1, u_2, e)$, $v = c^r d^{r\alpha}$. The ciphertext will be $c = (u_1, u_2, e, v)$.

For decryption, we have $Dec(u_1, u_2, e, v)$: $\alpha = H(u_1, u_2, e)$ and does a check $u_1^{x_1+y_1\alpha} u_2^{x_2+y_2\alpha} = v$. If this check does not work, then we fail. Otherwise, we output our decrypted message $m = e/u_1^z$.

Looking at the check: $u_1^{x_1} u_2^{x_2} = g_1^{rx_1} g_2^{rx_2} = c^r$ $u_1^{y_1} u_2^{y_2} = d^r$ $u_1^z = g_1^{rz} = h^r$.

Theorem: Cramer-Shoup is IND-CCA2 secure.

48. RSA

48.1. **Public Key Scheme.** The public key scheme is proposed as follows:

$Keygen(1^\lambda) \rightarrow (PK, SK, \mathrm{M}, \mathrm{C})$ where $|\mathrm{M}| = |\mathrm{C}|$. $Enc(PK, .)$ is efficiently computable, deterministic map from M to C. $c = Enc(PK, m)$ is a unique ciphertext for message m.

$Dec(SK, .)$ is efficiently computable inverse: $Dec(SK, c) = Dec(SK, Enc(PK, m)) = m$. It should be infeasible to decrypt knowing only PK.

48.2. **Keygen for RSA.** Keygen: two large primes p, q of length λ bits. $n = pq$ and have $\phi(n) = |Z_n^*| = (p-1)(q-1)$. Key insight is that you don't know the size of the group $|Z_n^*|$. Randomly select $e \leftarrow Z_{\phi(n)}^*$ where $gcd(e, \phi(n)) = 1$. Now we set $d = e^{-1} \pmod{\phi(n)}$ and we can run Euclid's algorithm to find d quickly.

$PK = (n, e)$ and $SK = (d, p, q)$. Now we have $\mathrm{M} = \mathrm{C} = Z_n$.

48.3. **Encryption and Decryption.** To encrypt, we just raise to a power. $Enc(PK, m) = c = m^e \pmod{n}$. You raise to the public exponent e.

Decryption is similar. $Dec(SK, c) = c^d \pmod{n}$, execept you raise to the private exponent d.

We assume the adversary cannot factor n, because if he could, he could repeat the key-generation operations and compute $\phi(n)$. This allows him to compute the private exponent $d = e^{-1} \pmod{\phi(n)}$. Once the adversary knows $\phi(n)$, adversary has access to SK.

48.4. **Proof of Correctness.** Lemma (Chinese Remainder Theorem): Let $n = pq$. We have $x = y \pmod{n}$ if and only if $x = y \pmod{p}$ and $x = y \pmod{q}$.

We know that $ed = 1 \pmod{\phi(n)}$ which means that $ed = 1 + t\phi(n) = 1 + t(p-1)(q-1)$ for some t. This implies that $ed = 1 \pmod{p-1}$.

Correctness of RSA is just $(m^e)^d = m \pmod{n}$. By CRT, we only need to show that $(m^e)^d = m \pmod{p}$ and $(m^e)^d = m \pmod{q}$. We'll just argue the case of p because the case of q follows similarly.

Case 1: $m = 0 \pmod{p}$. Then $(0^e)^d = 0 \pmod{p} = m \pmod{p}$ which works.

Case 2: $m \neq 0 \pmod{p}$ and $m \in Z_p^*$. Then we know that $m^{p-1} = 1 \pmod{p}$. Thus, let $u = t(q-1)$ then we have $m^{ed} = m^{1+u(p-1)} \pmod{p} = m(m^{p-1})^u = m1 = m \pmod{p}$. This works.

Therefore, we see that $m^{ed} = m \pmod{p}$ and $m^{ed} = m \pmod{q}$ for all m, which by CRT implies that for all m we have $m^{ed} = m \pmod{n}$.

Lecture 16

49. RSA and IND-CCA2 Security

Factoring: $exp(c - (\ln n)^{1/3}(\ln\ln n)^{2/3})$.

In practice, we've seen factoring of 768 bit numbers and 1024 bit numbers.

We need randomization in RSA and still have redundancy check.

49.1. The Scheme.

$|r| = k_1$ bits and $|m| = t$ bits. $G : \{0,1\}^{k_0} \to \{0,1\}^{t+k_1}$ is a random oracle function. We take r which is k_0 bits and send it to G and exclusive or the result with $m|0^{k_1}$.

Now we have $H : \{0,1\}^{t+k_1} \to \{0,1\}^{k_0}$. We can take the result of the exclusive or, and send it to H, take the result and exclusive or it with the r message.

The result is a string of $t + k_1 + k_0$ bits (just concatenate the two results), and now we feed that into RSA. We call the result x and do $x^e \pmod{n}$. This is called OAEP (Optimal Asymmetric Encryption Padding). This really fixes a lot of the RSA problems.

49.2. Decryption.

Set $x \leftarrow y^d \pmod{n}$. We invert OAEP and reject the output if 0^{k_1} is not present. Otherwise, if we don't reject, then we output m.

49.3. IND-CCA2 Security.

Theorem: RSA with OAEP is IND-CCA2 secure assuming a random oracle for the functions G and H and assuming RSA is hard to invert or random in paths.

49.4. Real World Attacks.

There are still attacks that can occur on imperfect implementations in the real world:

- Timing attacks. Figure out if different results have different times, then you can use the time to give you an idea of how to do things.
- Power attacks. If the power usage depends on the bits, then you can look at the power trace of the operation and try to figure out things which are happening inside of the computer.

Try interrupting power supply. If people are using CRT to get $y^d \pmod{n}$. Let mod p succeed and mod q fail. Then you have y^{d_1} but not y^{d_2} (say $n = pq$). We have something that is correct mod p (let's say the answer z), but incorrect mod q lets say z'. This means that $z = z' \pmod{p}$ and $z \neq z' \pmod{q}$. This means $z - z' = 0 \pmod{p}$. This gives $gcd(z - z', n) = p$.

It is ok for e to be short, but not ok for d to be short. For example $d < n^{1/4}$ is insecure. Recall that $\phi(n) = (p-1)(q-1)$ and $ed = q \pmod{\phi(n)}$.

50. Digital Signatures

Alice computes signature $\sigma_A(m) = sign(SK_A, m)$ and sends $m, \sigma_A(m)$ to Bob who uses Alice's public key. In digital signatures we have:

- $Keygen(1^\lambda) \to (PK, SK)$.
- $Sign(SK, m) \to \sigma_{SK}(m)$.
- $Verify(PK_A, m, \sigma) \to boolean$. We want for all m, we have $Verify(PK_A, m, Sign(SK_A, m)) = True$.

The adversary wants to get Bob to accept something that Alice did not sign. The adversary is interested in foregery. We can set this up as a game.

50.1. Definition: (Weak) existential unforgeability under adapative chosen message attack.

- Examiner gets (PK, SK) from $Keygen(1^\lambda)$ and PK is given to the adversary.
- Adversary obtains valid signature on messages $m_1, \dots, m_q$ of his choice for $q = poly(\lambda)$.
- Adversary outputs (m, σ_*) and adversary wins if $Verify(PK, m, \sigma_A) = True$ and $m \notin \{m_1, \dots, m_q\}$.

The scheme is secure if $Prob[Adversarywins]$ is negligible.

Lecture 17

51. Hash and Sign

If you have some very long message M, it's a pain to use all these public key operations on such a long message. What's typically done is that we don't sign the entire message M, we sign $m = h(M)$ which is some hash of M which is collission-resistant.

We've reduced the problem to sign short values (say 256 bit messages). This is a straightforward idea and is a nice application of hash functions. This is the model we will be using.

52. RSA Based Signing

52.1. PKCS. One of the first hash and sign methods was PKCS. It's just a padding approach. We're given a message M, so we compute its hash value $m = h(M)$. Now we want to pad it out to be the size of the RSA modulus. We define $pad(m) = 0x0001FF\dots FF00||hashname||m$. There are a lot of pad bytes (F in hexidecimal is just a series of 1s). The hashname is some identifier for the hash that you used.

We take this padded value $pad(m)$ to produce a signature $\sigma(M) = (pad(m))^d \pmod n$.

To verify: we take $x = \sigma(M)$ and we verify that $x^e \to pad(m)$ can be parsed with an acceptable hash function. We also check to make sure $m = h(M)$.

The question is whether or not an adversary can forge these messages. This is not provably secure, it's heuristic in design.

52.2. PSS (Probabilistic Signature Scheme). Created by Bellare and Rogaway in '96. It's a signature scheme based on RSA which uses randomization. We have a message M and a random value r as inputs. We want to combine these together and run them through RSA in some way. First, we hash the long message while putting randomization into it: $w = h(M, r)$. Now we take a new hash function g_1, and we compute $r \oplus g_1(w) = r^*$. Finally, we have a third hash function g_2 and we compute $g_2(w)$.

The result of these operations is that we have $w, r^*, g_2(w)$. We can concatenate these together to obtain $y = 0||w||r^*||g_2(w)$. Now we have a signature $\sigma(M) = y^d \pmod n$.

Writing it out more carefully, $r \leftarrow \{0,1\}^{k_0}$ is randomly chosen, $w \leftarrow h(M||r)$, $r^* \leftarrow g_1(w) \oplus r$, $y \leftarrow 0||w||r^*||g_2(w)$. Finally our signature is $\sigma(M) = y^d \pmod n$.

The verifitication is to compute $y = \sigma(M)^e \pmod n$. Now we parse y to get $b|w|r^*|\gamma$. We compute $r = r^* \oplus g_1(w)$ and we accept if $b = 0$ and $h(M||r) = w$ and $g_2(w) = \gamma$.

Theorem: PSS is (weakly) existentially unforgeable against a chosen message attack in the Random Oracle Model and RSA being not invertible on random inputs.

53. ElGamal Based Signing

53.1. ElGamal Digital Signatures. We have standard setup with large prime p and a generator g of Z_p^*. We have keygen $x \leftarrow \{0, 1, \dots, p-2\}$ being a randomly chosen element (we know $|Z_p^*| = p - 1|$). Now we set $y = g^x \pmod p$. So we have $SK = x$ and $PK = y$.

Signing procedure ($Sign(M)$): compute $m = h(M)$ from hash and sign, assuming h is collision resistance into Z_{p-1}. Pick a value $k \leftarrow Z_{p-1}^*$ at random such that $gcd(k, p-1) = 1$. Now take $r = g^k \pmod p$. This means that r is currently indpenedent of message M! Now set $s = \frac{m-rx}{k} \pmod{p-1}$. We know that k^{-1} exists modulo $p-1$. Now we have $\sigma(M) = (r, s)$.

Veritification procedure ($Verify(M, y, (r, s))$): Check that $0 < r < p$ so that r is in the correct range, otherwise reject. Now we check that $y^r r^s = g^m \pmod p$ where $m = h(M)$. We know that $y^r r^s = g^{rx} g^{ks} = g^{rx+ks} \pmod p$. We want to know if this is equal to $g^m \pmod p$. These are equal if and only if their discrete logs are equal. This means that the equality is equivalent to $rx + ks \equiv m \pmod{p-1}$. This is the same as saying $s = \frac{m-rx}{k} \pmod{p-1}$. But this was our definition for s, so this checks out.

Unfortunately this scheme isn't any good under our weak existential unforgeability. For example: lets choose $e \leftarrow Z_{p-1}$ which is chosen randomly. Now let $r = g^e y \pmod{p}$. We have $s = -r \pmod{p}$. Let $h(x) = x$ be the identity hash function (certainly collision resistant). Now we have $M = m = es \pmod{p-1}$. Now I claim that m has a valid signature (r, s). Let's check: $y^r r^s = g^m \pmod{p}$. This is just $g^{xr}(g^e y)^{-r} = g^{es} = g^m \pmod{p}$. We've forged a signature!

However, we can fix this pretty easily using the same concept we used in PSS. Instead of using $m = h(M)$, let's add some randomization. So let's take a random value r and compute $m = h(M||r)$.

Theorem: Modified ElGamal Signature Scheme is existentially unforgeable against a chosen message attack in Random Oracle Model, assuming discrete logairthm problem is hard.

53.2. DSS. Setup: q is a prime 160 bits long. $p = nq + 1$ where $|p| = 1024$ bits. Now take g_0 which is a generator of Z_p^* and $g = g_0^n$ which generates a subgroup of Z_p^*, Z_q, of order q.

Keygen: $x \leftarrow Z_q$ is the SK and $|x| = 160$ bits and take $y = g^x \pmod{p}$, where $|y| = 1024$ bits.

Signing ($Sign(M)$): We generate $k \leftarrow Z_q^*$ as a random value where $1 \leq k \leq q$. Now we have $r = (g^k \mod p) \pmod{q}$. Now compute $m = h(M)$ and $s = \frac{m+rx}{k} \pmod{q}$. Now s, r each have 160 bits. Redo everything if $s = 0$ or $r = 0$. Now we have a signature $\sigma(M) = (r, s)$.

Verification: Check that $0 < r < q$, $0 < s < q$. Now we check that $y^{r/s} g^{m/s} \pmod{p} \pmod{q} = r$ where $m = h(M)$. This is just $g^{(rx+m)/s} = g^k = r \pmod{p} \pmod{q}$.

Like ElGamal, it's not unforgeable in this form, you need to add r a random bit into $m = h(M||r)$ in order to get existential unforgeability.