CS 255 NOTES

ARUN DEBRAY
MARCH 13, 2013

These notes were taken in Stanford's CS 255 class in Winter 2013, taught by Dan Boneh. I live-TEXed them using `vim`, and as such there may be typos; please send questions, comments, complaints, and corrections to adebray@purdue.edu.

Contents

1. Overview and The One-Time Pad: 1/7/2013

The name of the one-time pad is apt: it should only be used once. Subsequent uses of the same one-time pad can be insecure, as seen in the zeroth homework assignment. In general, a little knowledge can be dangerous, as an improperly used encryption system appears to work fine but can actually be broken into. This actually happens a lot in the real world, and some systems with theoretically secure cryptography can be broken due to an incorrect implementation.

Crypto is everywhere: one common example is the HTTPS protocol, which is built around a system called TLS. This operates in the following manner:

(1) The session-setup step allows two machines who have never before interacted to generate a shared key (shared secret), using the methods of public-key cryptography (RSA, ELGamal, etc., which will be the second half of the class).

(2) Then, the shared key is used to transfer encrypted data, etc. This will be covered in the first half of the class.

Crypto is used in other ways, such as PGP for encoding email, EFS, TrueCrypt, etc., as well as user-authentication (passwords or challenge-response). Wireless systems also use crypto, including 802.11b, or WEP, (which is *very* insecure and shouldn't be used), WPA2, and so on. These applications will be discussed in the third... half of the class.

Crypto can also be used in elections, in a way that authenticates everyone's votes but keeps individual voters anonymous. Related protocols can be used to correctly run auctions, copyright protection (such as CSS for DVDs, another excellent example of what not do to; and AACS for Blu-Ray, which is a theoretically sound encryption which has a flawed implementation), and so on.

Crypto can also be used to implement digital cash in a manner that is anonymous and cannot be counterfeited, as with conventional cash.

An important thing to remember: avoid security by obscurity. The goal is to make systems in a way that is still secure even if all the implementation details are public (except, of course, the secret key). Many past systems (including CSS, mentioned above) assumed that the algorithm would be kept secret. This is not likely to be a valid assumption.

Ciphers are an example of a symmetric encryption system. There are two players, Alice and Bob, who share a secret key $K \in \{0,1\}^{128}$ (i.e. a key of 128 bits). There are also two algorithms E and D, respectively the encryption and decryption algorithms, such that for a plaintext m, the cipher must be *correct*: for any K, m, $D(E(K,m)) = m$.

Definition 1.1. A cipher (E, D) is defined over $(\mathcal{K}, \mathcal{M}, \mathcal{C})$, where $\mathcal{K}$ is the key space, $\mathcal{M}$ is the plaintext space, and $\mathcal{C}$ is the ciphertext space with $E : \mathcal{K} \times \mathcal{M} \to \mathcal{C}$ and $D : \mathcal{K} \times \mathcal{C} \to \mathcal{M}$.

This is called a symmetric encryption system because Alice and Bob share the same key.[1]

Example 1.2 (Substitution Cipher)**.** This is a cipher where the key is a permutation of $\{\mathsf{a}, \dots, \mathsf{z}\}$. Thus, there are $26! \approx 2^{88}$ possibilities.

For example, if $K = \{\mathsf{a} \to \mathsf{v}, \mathsf{b} \to \mathsf{l}, \mathsf{c} \to \mathsf{r}, \mathsf{z} \to \mathsf{b}, \dots\}$, then $E(K, \text{“bacz”}) = \text{“evrb”}$ and $D(K, \text{“evrb”}) = \text{“bacz”}$. ◁

This cipher is completely useless, because of frequency analysis:

(1) First check the most common letters: e appears 12.7% of the time, t 9.1%, a 8.1%, etc.
(2) Then, look at digraphs and common longer sequences of letters. The most common digraphs are th, he, and an.

This is a ciphertext-only attack, which is particularly problematic. Additionally, if $a \to b$ in one place, then $a \to b$ everywhere, so even if the whole ciphertext cannot be decrypted, much of it might be.

Here is a better example, which is much more secure. It was invented in 1917 by Vernam to encrypt telegrams.

Example 1.3 (One-Time Pad)**.** Here, $\mathcal{M} = \mathcal{C} = \mathcal{K} = \{0,1\}^n$. Then, letting $\oplus$ represent `xor`, $E(K, m) = K \oplus m$ and $D(K, c) = K \oplus c$. A key is a sequence of bits as long as the plaintext.

This is in fact correct: for all $m, K \in \{0,1\}^n$, $D(K, E(K, m)) = K \oplus K \oplus m = (K \oplus K) \oplus m) = 0 \oplus m = m$.[2] ◁

One can use information theory of security (as developed by Shannon in the 40s) to prove this is secure. Generally, a system is secure if the ciphertext reveals no information about the plaintext.

Definition 1.4. A cipher (E, D) over $(\mathcal{K}, \mathcal{M}, \mathcal{C})$ has perfect secrecy if for all $m_0, m \in \mathcal{M}$ where m and m_0 have the same length and for all $c \in \mathcal{C}$, then

$$\Pr_{K \in \mathcal{K}}[E(K, m_0) = c] = \Pr_{K \in \mathcal{K}}[E(K, m) = c],$$

where K is uniform in $\mathcal{K}$.

[1] In some asymmetric systems, the algorithms may be randomized; however, the decryption algorithm is always deterministic.

[2] $\oplus$ is just addition $\bmod 2$, so it is associative.

This means that knowing only the ciphertext, all possible keys and plaintexts have equal probabilities. In the language of probability theory, m and c are independent random variables. Even the most powerful adversary cannot learn anything about the plaintext from only the ciphertext. Note that this directly contradicts several Hollywood movies (e.g. Mercury Rising, Summer Wars).

Lemma 1.5. *The one-time pad has perfect secrecy (i.e. the* `xor` *of a random variable is also random).*

Proof. Because K is uniform in $\mathcal{K}$, then

$$\Pr_{K\in\mathcal{K}}[E(K,m_0)=c]=\frac{|\{K\in\mathcal{K}\mid E(K,m_0)=c\}|}{|\mathcal{K}|}.$$

Thus, $|\{K\in\mathcal{K}\mid E(K,m_0)=c\}|=|\{K\in\mathcal{K}\mid E(K,m)=c\}|$. For the one-time pad, only $K=m\oplus c$ maps $m\to c$, so each of these sets has size 1, so the probabilities are always equal to $1/|\mathcal{K}|$. ⊠

Note that this formulation of the one-time pad leaks the length of the post, which can lead to certain types of attacks (such as a notable recent attack on SSL). However, other methods can be used to hide the length.

However, there are costs associated with perfect secrecy:

Lemma 1.6 (Bad News Lemma)**.** *If a system has perfect secrecy, then* $|\mathcal{K}|\geq|\mathcal{M}|$.

Sketch of the proof. If there is a key K_0 that maps m_0 to c, then for any $m\in\mathcal{M}$ there exists a $K_1\neq K_0$ (so that decryption is possible) such that K_1 maps m to c.

Thus, the one-time pad is optimal with regard to this constraint: the size of the key space is the same as the size of the plaintext space.

There are various ways to make the one-time pad practical. One example is a stream cipher, in which the random key is replaced by a pseudorandom key.[3] Then, a one-time pad with h-bit keys ($h=128$ bits) is expanded with the pseudorandom generator, so $E(K,m)=m\oplus G(K)$ and $D(K,m)=c\oplus G(K)$.

Thanks to the Bad News Lemma, this lacks perfect secrecy, however. It's useful enough to be used a lot in practice (e.g. GMail's RC4).

One possible attack on the one-time pad is the two-time pad, in which one key is used multiple times. If $c_1=E(K,m_1)=K\oplus m_1$ and $c_2=E(K,m_2)=k\oplus m_2$, then the attacker can read c_1 and c_2 and computes:

$$c_1\oplus c_2=(K\oplus m_1)\oplus(K\oplus m_2)=m_1\oplus m_2.$$

English actually has enough redundancy (and even the ASCII character table for English text) that if m_1 and m_2 are English sentences, both of them can be retrieved from $m_1\oplus m_2$. This method allowed Americans to decrypt Soviet messages during the Cold War (Project Vanona).

2. Pseudorandom Generators: 1/9/2013

A pseudorandom generator (PRG) is a function $G:\{0,1\}^s\to\{0,1\}^n$, where $n\gg s$ (a 32-byte long seed can become a gigabyte-long string, for example). As seen in the previous lecture, this is used to implement a stream cipher, which isn't perfectly secure.

Even though the two-time pad attack makes one-time pads and stream ciphers vulnerable to attacks, one-time pads are still used incorrectly in practice.

Example 2.1 (MS-PPTP)**.** A laptop and a server share a secret key K, which is generated anew with every connection. Then, the laptop sends message m_1 to the server, which responds with message m_2 to the laptop, which responds with message m_3, ...

All of the messages $m_1,m_3,m_5,\ldots$ are concatenated and `xor`ed with $G(K)$ (in practice, the first message is encrypted with the first bits of $G(K)$, the next one with the next bits, etc.). The server does the same thing with $m_2,m_4,\ldots$

[3] A pseudorandom generator (PRG) is a function $G:\{0,1\}^n\to\{0,1\}^m$, where $m\gg n$. $\{0,1\}^n$ is called the seed. This is not random, because there isn't enough entropy, but it "looks" random in a way that will be formalized in the next lecture.

However, the laptop and the server share the same key, allowing for a two-time pad attack. To make this secure, the session key should contain two keys: $K = (K_{C\to S}, K_{S\to C})$, both of which are known to each party. ◁

Example 2.2 (802.11b)**.** This is a wireless protocol between a laptop and a wireless access point which share a (long-term) key K that is either 40 or 112 bits long. The laptop sends a message m to the access point and both compute a cyclic redundancy check $\texttt{CRC}(m)$. Then, the message is encrypted by running a stream cipher called RC4 on IV $\|\, M$, where IV is a 24-bit initialization vector and $\|$ represents concatenation. The initialization vector is sent in the clear, and allows the key to be different each time. It can be generated in several ways:

(1) Some systems use a counter for the IV. However, if the access point reboots, then it resets to zero, which makes for a standard two-time pad attack.
(2) The IVs can be randomly generated. However, thanks to the birthday paradox, a repeated value is likely after about $\sqrt{2^{24}} \approx 4000$ interactions, which allows for a two-time pad attack.
(3) System time.

No matter how these IVs are generated, there are only $2^{24} \approx 1.6 \cdot 10^7$ possible keys. This represents about two hours of traffic, which is not a lot, and allows for a two-time pad attack. And because the IV is sent unencrypted, then it is *very* obvious when a repetition occurs. ◁

Additionally, keys for a symmetric system should be uniform in the key space, but choosing keys such as $0 \,\|\, K$, $1 \,\|\, K$, etc. doesn't satisfy that. Related keys tend to make ciphers insecure, such as the Fluhrer-Mantin-Shamir attack in 2001. This attack demonstrated that after about a million frames[4] (or packets), the key can be deduced, irrespective of any two-time pads. This has been implemented in a system called Wepcrack, which you can try at home.

One way to generate a sequence of random, independent-looking keys is to take some key K, use RC4 to encrypt it into a much longer key, and then chop it up into a sequence $K_1, K_2, \ldots$, each of which is used as a key for exactly one message (using RC4 again to encrypt them). To the adversary, this pseudorandom sequence looks random. (It looks simpler to just treat the messages as streams, but this requires everything to arrive in order, which is not always feasible.) And of course, the two sides must use different keys, as discussed above.

There is another attack on the one-time pad. For example, if Bob sends Alice a message containing the string "From: Bob" using some key K then by `xor`ing the part of the ciphertext that corresponds to Bob with `07 19 07`, the message is changed to say 'From: Eve". The sequence of numbers was generated by $\texttt{Bob} \oplus \texttt{Eve}$. Of course, this only works if you know the structure of the message, but this can be done if, for example, an email client encrypts messages in a predictable way. And if the mail server reads that part of the message to determine whom to deliver it to, this can be used to redirect email to the wrong recipient. The difference between confidentiality and integrity is important, and many people in industry forget this.

This property is called malleability; the one-time pad is malleable, which means that it is possible to change the ciphertext to change the plaintext in predictable ways. Malleability implies vulnerability to active attacks.

RC4 is a pseudorandom generator invented in 1987, but is not completely secure; it was designed for an 8-bit processor, so it is not the most efficient or secure PRG. Nonetheless, it is used (incorrectly) in 802.11 WEP, as well as in TLS (Transport Layer Security, the protocol behind HTTPS, including use by Google), which is relatively secure.

One weakness of RC4 is that the probability that the 2^{nd} byte is zero is $2/256$, when it should be $1/256$. This bug means that current use of RFC recommends ignoring the first 256 bytes of output (even though TLS doesn't do that). Additionally, it becomes nonrandom relatively quickly. The probability of two consecutive zeros (16 bits) is $1/65536 + 1/256^3$, where it should be $1/65536$. This allows one to distinguish random output from pseudorandom output via RC4, and it in fact after $(256^3)^2$ bytes (about a gigabyte), one can reliably distinguish the two.

[4] This has since been improved to about $40,000$.

A project called eStream looked at a PRG $G : \{0,1\}^s \times R \to \{0,1\}^n$, where R is a nonce; that is, it is a non-repeating value. This doesn't have to be random (e.g. a counter), but the goal is that the same key can be used more than once. Then, $E(K, m; r) = m \oplus G(K; r)$, where the pair (K, r) should not be used more than once together.

Example 2.3 (Salsa 2012)**.** This protocol is designed for software environments (so that it runs quickly, taking advantage of the CPU; a hardware environment requires minimizing use of power, transistors, etc.). This uses a function $\text{Salsa2012} : \{0,1\}^{128} \times \{0,1\}^{64} \to \{0,1\}^n$, where the first part is the seed, the second part is the nonce, and n is at most 2^{73} (so that it can be used at most 2^{73} times). Specifically, the output is generated in 64-byte chunks, where K is some long-term key:

$$\text{Salsa2012}(K; r) = H(K, r, 0) \,\|\, H(K, r, 1) \,\|\, H(K, r, 2) \,\|\, \cdots$$

H is implemented using four constants $\tau_0, \ldots, \tau_3$ that are fixed in the protocol, and generate a 64-bit buffer $[\tau_0, K, \tau_1, r, i, \tau_2, K, \tau_3]$ (where commas denote concatenation) for $H(K, r, i)$. Then, there is a (hash) function $h : \{0,1\}^{64} \to \{0,1\}^{64}$ which is applied repeatedly. (Repeated application is a very good way to make things secure.) However, h is invertible, which means that only doing this would be insecure (not to mention would look decidedly nonrandom). Then, add one last `xor` of the first and last 64-bit sequences to make it look random.

h is particularly fast on the x86 architecture because of a command called `SSE2`, which was originally used for graphics processing.

Can you distinguish the output of Salsa 2012 from random? If so, you would get fame and glory, and it's not known to be impossible. ◂

Week 1 Videos: History of Cryptography and Discrete Probability

Similarly to what was said in the in-class overview, cryptography is used everywhere in today's world: secure communications (HTTPS, wireless traffic such as 802.11i WPA2), encrypting stored data (e.g. EFS and TrueCrypt), content protection (such as CSS on DVDs and AACS on Blu-ray discs), user authentication, etc.

Looking specifically at secure communication, a good model is of Alice, using a laptop, communicating with Bob, who is on a server. The goal of using a protocol such as SSL or TLS is to prevent eavesdropping or tampering. TLS has two parts: the first is a handshake protocol, which uses public-key cryptography to establish a secret key (techniques which appear in the second part of the course), and then using this secret key to transmit engrypted data.

When protecting files on disk, the goal is again to prevent eavesdropping and tampering if a disk is stolen. This can be though of as sending a message from Alice today to Alice tomorrow, so the two scenarios are not that different.

Note that it is extremely important that the algorithms E and D are publicy known (though the secret key is of course private). This is so that peer-review ensures that they are hard to break. A proprietary cipher is generally much more dangerous, and there many examples of proprietary ciphers which were reverse-engineered and broken.

There are two types of symmetric encryption: keys can be used only once (single-use key) or many times (multi-use key). The latter case is more complicated in terms of ensuring security.

It is important to recall what cryptography is not: it cannot work if implemented incorrectly (which sounds obvious, but happens surprisingly often), and it cannot prevent social-engineering type attacks. Relatedly, don't try to invent protocols, because they are less likely to be secure than peer-reviewed protocols.

One interesting application of cryptography is a zero-knowledge protocol. Suppose Alice has two large primes p and q. It is easy to multiply them together into $N = pq$. Factoring is harder than multiplication, but if Bob only knows N, then there is a proof that Alice can make to Bob without revealing the factors. This sort of proof exists for many types of puzzles.

Regarding the history of cryptography, there is a great book by David Kahn called *The Code-Breakers*. This lecture only contains a few examples, none of which are particularly secure.

A substitution cipher is probably the simplest example. The key is a substitution table (i.e. an element of S_{26} describing how letters map to each other). A related historical cipher is the Caesar cipher (though technically, since it doesn't have a key, then it is not a cipher). This is a fixed substitution that shifts each letter by 3: $K(a) = a + 3 \bmod 26$. This is not secure at all, and even the more general substitution cipher can be easily broken. Notice that the size of the key space is $|\mathcal{K}| = |S_{26}| = 26! \approx 2^{88}$. This is a large enough key space to be secure, but it can be broken with frequency analysis (e.g. the most common letter in the ciphertext is probably e in the plaintext, etc.). This can be aided by checking digraphs, trigraphs, and common words.

In the Renaissance, a man named Vigenère designed several related ciphers. For a message m and a key K, the ciphertext c is made by adding the n^{th} character of the key (mod the length of the key) and of $m \bmod 26$. For example, with the key "crypto," the message "what a nice day today" is encrypted into "zzzjucludtunwgcqs."

This is actually still very easy to break, assuming the length of the key $|K|$ is known. Then, take every $|K|^{\text{th}}$ letter and perform frequency analysis. This allows the key to be recovered and the ciphertext to be completely decrypted. Even if the length of the key is unknown, this procedure can be run for all possible key lengths until an intelligible message is recovered.

The basic idea of addition $\bmod 26$ is actually a great idea, but here it was implemented improperly.

In the 19^{th} Century, electrical machines for cryptography were invented. The Hebern machine is an example: it is a typewriter such that every time a key is pressed, a rotor encrypts it and then shifts the encryption so that it is not just a simple substitution cipher. Nonetheless, it is still vulnerable to frequency analysis and can be broken in a ciphertext-only attack.

Makers of rotor machines responded to these attacks by adding more complexity, including lots of rotors and dials. The best example of this is the Enigma, which is a rotor machine that uses three to five rotors. The secret key for this machine is the setting of the rotors, so the key space has 26^3 to 26^5 possibilities. This is relatively small and can be brute-forced easily with today's technology. Even during World War 2, the British successfully mounted ciphertext-only attacks.

After the war, computer-aided cryptography became more popular. The federal standard was a protocol called DES (Data Encryption Standard), with a (small for today, but not then) key space of 2^{56}. Thus, it is not secure anymore (though it is still used in some legacy systems). Newer ciphers exist today (e.g. AES, the Advanced Encryption Standard).

Another important ingredient in cryptography is discrete probability. This is defined over a universe U, which is some finite set (in this course, often $U = \{0,1\}^n$).

Definition 2.4. A probability distribution P over U is a function $U \to [0,1]$ such that $\sum_{x \in U} P(x) = 1$.

Example 2.5. If $U = \{0,1\}^2$, then a probability distribution on U is $P : (0,0) \to 1/2, (0,1) \to 1/8, (1,0) \to 1/4, (1,1) \to 1/8$. ◂

The uniform distribution is a common example of a probability distribution, given by a function $P(x) = 1/|U|$ for all $x \in U$. A uniform distribution on $\{0,1\}^2$ assigns the probability $1/4$ to each element.

Another example is the point distribution at x_0 for some $x_0 \in U$. This is a function P such that $P(x_0) = 1$ and $P(x) = 0$ if $x \neq x_0$.

Since the universe is finite, then it is possible to write down all of the probabilities as a vector in $\mathbb{R}^{|U|}$.

Definition 2.6. An event is a subset $A \subseteq U$. The probability of A is

$$\Pr[A] = \sum_{x \in A} P(x).$$

The probability of an event always satisfies $0 \leq \Pr[A] \leq 1$, and $\Pr[U] = 1$.

For example, on $U = \{0,1\}^8$ (the space of all possible bytes) with a uniform distribution and A is the set of bytes that end in 11, then $\Pr[A] = 1/4$.

Another property of discrete probability is the union bound: if A_1 and A_2 are two events on U, then

$$\Pr[A_1 \cup A_2] \leq \Pr[A_1] + \Pr[A_2].$$

Intuitively, the sum of the probabilities counts their intersection twice, so it cannot be less. Thus, if A_1 and A_2 are disjoint, then equality holds.

Definition 2.7. A random variable on U is a function $X : U \to V$. V is the set on which X takes its value.

For example, if $X : \{0,1\}^n \to \{0,1\}$, then X could just output the least significant bit of an $x \in U$. Then, $\Pr[X = 0] = \Pr[X = 1] = 1/2$.

A random variable $X : U \to V$ induces a probability distribution on V given by $P(v) = \Pr[X = v] = \Pr[X^{-1}(v)]$ for some $v \in V$.

A particularly important random variable is the uniform random variable, which is denoted $r \xleftarrow{R} U$. This is an identity function (i.e. $r(x) = x$ for all $x \in U$), so that $\Pr[r = a] = 1/|U|$ for all $a \in U$.

It is also worth discussing randomized algorithms. A determinstic algorithm can be considered as a function A that always returns the same output given the same input. A randomized algorithm does not; it can be thought of as a function $A(m; r)$, where $r \xleftarrow{R} \{0,1\}^n$. This defines a random variable which defines a distribution over all possible outputs of A for a given input m.

Definition 2.8. Two events A and B are independent if $\Pr[A \text{ and } B] = \Pr[A] \Pr[B]$. This means that the probability of B is not affacted by the presence or absence of A.

Similarly, two random variables $X, Y : U \to V$ are independent if $\Pr[X = a \text{ and } Y = b] = \Pr[X = a] \Pr[Y = b]$. For example, the random variables on $\{0,1\}^n$ that return the least and most significant bits are independent provided $n > 1$, since the least significant bit confers no information about the most significant one, and vice versa.

This is particularly helpful in the case of the `xor`, which comes up often in cryptography[5] because of the following theorem:

Theorem 2.9. *If X, Y are independent random variables over $\{0,1\}^n$, then $Z = X \oplus Y$ is a uniform random variable.*

The following is another important fact, and somewhat counterintuitive:

Theorem 2.10 (The Birthday Paradox)**.** *Suppose $r_1, \ldots, r_n$ be independent identically distributed random variables. Then, when $n \approx 1.2\sqrt{|U|}$, then $\Pr[r_i = r_j, i \neq j] \geq 1/2$.*

The name is given because after only about 24 people, it is likely that two of them will share a birthday, which is much smaller than expected. (Notice that birthdays aren't uniform, but it's close enough for this to work.)

As n increases, the probability of a collision increases very quickly to 1, which will be important later.

3. More Stream Ciphers : 1/14/2013

The most important aspect of any stream cipher is to never, ever, ever use the same key (or key-nonce pair) more than once. Like, ever.[6]

Since a stream cipher doesn't have perfect secrecy, when is it secure? Consider a stream cipher $m \oplus G(K) = c$. In some contexts (e.g. email or anywhere else where the message starts with a known header), the first part of the ciphertext can be identified with the first part of the plaintext, known to the adversary, then the first part of $G(K)$ can be determined.

If the PRG is predictable (such that a given $G(K)|_{0,\ldots,i}$ allows an efficient reconstruction of all of $G(K)$), then the stream cipher is clearly not secure. Thus, a PRG used in a stream cipher must be unpredictable, so that its output must be indistinguishable from random output. What this precisely means requires some theory.

Consider the PRG $G(k) = k \,\|\, k \,\|\, k \,\|\, \ldots$ This obviously isn't random, and can be broken by checking for the repetition; an algorithm A can check if $r[0] = r[s] = r[2s] \ldots$ More generally:

[5] It is often said (as a joke) that the only thing cryptographers know how to do is `xor` things.

[6] Seriously, though; this completely undermines the security, as discussed above.

Definition 3.1. The advantage of an algorithm A and a PRG G is

$$\mathrm{Adv}[A,G] = \left| \Pr_{r \xleftarrow{R} \{0,1\}^n} [A(r) \text{ “random”}] - \Pr_{s \xleftarrow{R} \{0,1\}^s} [A(G(s)) \text{ “random”}] \right|.$$

This is a number between 0 and 1 that indicates how effectively the algorithm A can distinguish random and pseudorandom output. If the advantage is very high ($\mathrm{Adv}[A,G] \approx 1$), then the PRG is not very secure. This indicates if a PRG is bad, and here's how to tell that it is good:

Definition 3.2. A PRG $G : \{0,1\}^n \to \{0,1\}^s$ is secure if for every efficient algorithm A the advantage $\mathrm{Adv}[A,G]$ is negligible.

The definition of negligible and efficient varies; in theory, one might want polynomial running time, but in practice the actual running time matters.

One of the embarrassments of computer science is that so many lower bounds for running times are unknown (e.g. P vs. NP), so many of the assumptions about running times are just guesses. In particular, if $\mathrm{P} = \mathrm{NP}$, then no PRG is secure and crypto couldn't work, since there are NP algorithms that can distinguish the output of a PRG from random output. In particular, proving the security of any particular cipher is equivaent to showing $\mathrm{P} \neq \mathrm{NP}$, which is, as you might imagine, is slightly impossible.

Lemma 3.3. *If a PRG is secure, then it is unpredictable.*

A concept called semantic security is a more practical analogue to perfect secrecy. In perfect security, the distributions on $E(k,m)$ and $E(k,m')$ for any two messages m, m' must be equal. However, for semantic security, this requirement is relaxed to indistinguishability of these two distributions by an efficient adversary. Additionally, this is specified for specific m and m'. Formally:

Definition 3.4. Consider an experiment between an adversary and some challenger. The challenger does one of two possible things (given by a $b \in \{0,1\}$): the adversary sends to messages $m_0, m_1 \in \mathcal{M}$, such that $|m_0| = |m_1|$ (and the lengths are known). Then, the challenger chooses a $K \xleftarrow{R} \mathcal{K}$ and returns $C = E(K, m_b)$. The attacker's goal is to determine whether C corresponds to m_0 or m_1 (and specifically, returns a $b' \in \{0,1\}$ corresponding to the adversary's guess).

The advantage of the adversary is defined similarly to before:

$$\mathrm{Adv}[A,E] = |\Pr[b' = 0 \text{ if } b = 0] - \Pr[b' = 0 \text{ when } b = 1]|.$$

Then, E is semantically secure if $\mathrm{Adv}[A,E]$ is negligible for all efficient adversaries.

Again, efficiency and negligibility are dependent on context. Additionally, this is only relevant for systems in which the key is used only once, a fact which is captured by the fact that the adversary only gets to see one ciphertext.

Theorem 3.5. *If a PRG is secure, then its induced stream cipher has semantic security.*

Here are some methods of generating entropy on a computer; do not invent your own, because they are unlikely to be very secure.

(1) RFC 4086, in which there is an entropy source which is fed into an RNG state, which outputs whenever enough entropy is available. This often includes measuring interrupt times (e.g. keypresses, packet reception times). One could also measure disc access times. Sometimes, there will actually be an RNG in the hardware.
(2) On Unix systems, one can use `/dev/random`, but this will halt the processor if there isn't enough entropy available. `/dev/urandom` is also available, but switches to a PRG instead of halting the processor, which could lead to a security flaw.
(3) On Intel processors, there is an `x86` instruction called `RdRng` that generates 3 Gb/sec of randomness (which is a lot) by reading thermal noise across transistors.

Consider a RNG that generates bits in a way such that the output bits are independent of one another but are biased (i.e. that for any i, $\Pr[b_i = 1] \neq 1/2$, so there are more `1`s than `0`s, or maybe vice versa).

This can be used to generate truly random output with something called an extractor that considers bits in pairs: an extractor ignores sequences of the form `00` or `11`, and outputs `0` on `01` and `1` on `11`. This produces `0`s and `1`s with the same probability. This actually comes up on some interview questions.

More powerful extractors exist, which can extract entropy from lots of different sequences with entropy; this is part of the subject of CS 355 (e.g. the Leftover Hash Lemma[7]).

The basic problem of single-use keys can be overcome by using a different type of cipher called a block cipher.

Definition 3.6. A block cipher is a pair of algorithms (E, D) which takes as input and creates as output fixed-size blocks called the plaintext block and the ciphertext block, repsectively.

While stream ciphers and PRGs are very nice, in practice the most common ciphers are blocks. Two examples are the 3DES cipher (pronounced "triple-DEZ"), which has a 64-bit block and a key space of $\mathcal{K} = \{0,1\}^{168}$, and AES, with a block size of 128 bits and a key space of either 128, 192, or 256 bits.

4. Block Ciphers : 1/16/2013

Before discussing block ciphers, one more thing should be mentioned about stream ciphers. One should never reuse a key (or key-nonce pair), so for systems which use stream ciphers, keys need to be generated frequently and never reused (as in NTTPS or file encryption). In the latter case, one could use a one-time stream cipher to encrypt the file, and then use a many-time encryption cipher to encrypt the keys. Even when modifying a file, it should be treated as an entirely new document. Taking some many-time cipher E_{sym}, the file is encrypted as $F \oplus G(K_F)$, and then encrypt everything as $E(K, F) = E_{\text{sym}}(K, K_F) \,\|\, F \oplus G(K_F)$.

A block cipher is a cipher that maps n bits to n bits. The canonical examples are 3DES and AES, as mentioned in the previous lecture. The block cipher is constructed by iteration: a 128-bit (for example) key is expanded into many keys $K_0, \ldots, K_d$ with a PRG. Then, the message is passed through a round function R (which is a function that is invertible) such that $C = R(K_d, R(K_{d-1}, R(K_{d-2}, \ldots (K_0, m))))$. Then, decryption just does the same with R^{-1}. There's a conjecture that some types of round functions, when iterated repeatedly, become secure block ciphers (though again, don't just invent one and assume that it will work). Notice that some functions will never be secure, such as $R(K, m) = K \oplus m$; linear functions in particular should be avoided. Most functions become secure eventually, but the goal (for performance reasons) is to choose functions that stabilize quickly.

Definition 4.1. A pseudorandom function (PRF) over a triple $(\mathcal{K}, X, Y)$ (where $\mathcal{K}$ is a key space) is just a function $F : \mathcal{K} \times X \to Y$ that is efficiently computable.

The definition of efficient depends on context; for a theoretician, polynomial time is necessary, but in implementation, one cares about absolute running times.

Definition 4.2. A pseudorandom permutation (PRP) is a pseudorandom function over $(\mathcal{K}, X)$ $E : \mathcal{K} \times X \to X$ that is efficiently computible and efficiently invertible; that is, there exists an efficiently computable $D : \mathcal{K} \times X \to X$ such that $D(K, E(K, X)) = X$.

Under this formalism, AES-128 : $\{0,1\}^{128} \times \{0,1\}^{128} \to \{0,1\}^{128}$ can be considered a PRP.

Consider a game between an encryptor and an adversary in which the encryptor chooses $K \xleftarrow{R} \mathcal{K}$, the attacker sends messages $X_1, X_2, \ldots$, and the encryptor sends back $F(K, X_1), F(K, X_2), \ldots$ In some ideal world, one would choose a truly random function $f : X \to Y$; there are $|Y|^{|X|}$ such functions, which is huge, and the adversary can query $X_1, \ldots$ and receive $f(X_2)$. A secure PRF is one in which no efficient adversary can distinguish these two scenarios.

For example, $F(K, X) = K \oplus X$ is not secure, since the attacker can just query x_0, x_1 and test if $f(x_0) \oplus f(x_1) = x_0 \oplus x_1$, reporting pseudorandom on success. This is extremely unlikely for a random function (since the size of the space of random functions is enormous), so they can be distinguished.

[7] The Leftover Hash Lemma joins the Sandwich Theorem and Ham Sandwich Theorem in the set of edible mathematical results.

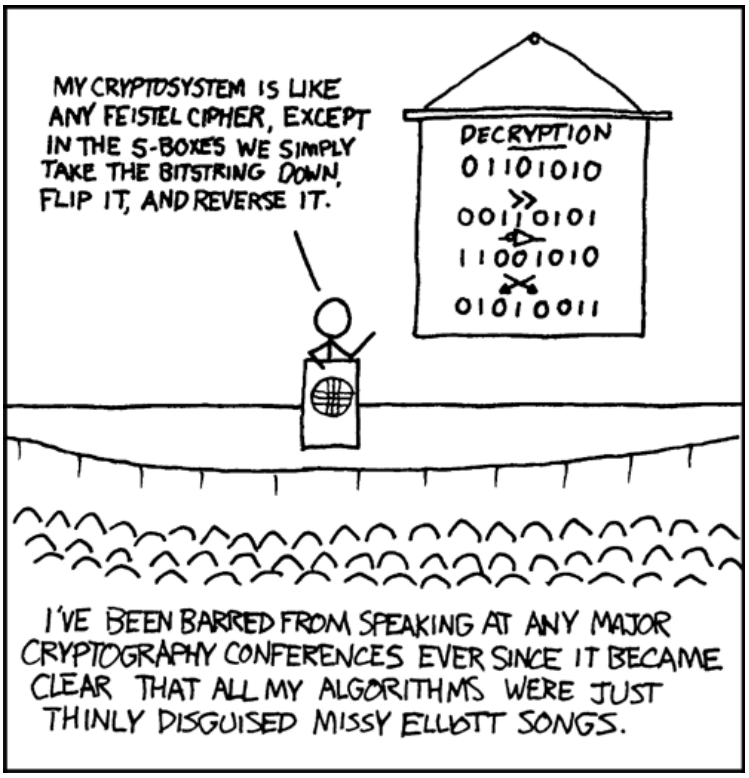

Figure 1. Relevant xkcd.

The Data Encryption Standard, or DES, was built at IBM in the 1970s. This was particularly noteworthy because cryptography in the civilian world was very primitive at that time. It was eventually adopted as a federal standard and eventually used in industry (though they key length was shortened from the original 128 bits to 56). However, the key space is small enough that exhaustive search became practical in 1997, so AES is now the standard.[8]

DES is built on the idea of a Fiestel network (though, oddly, AES isn't). This is built on a network of functions $f_1, \dots, f_n : \{0,1\}^d \to \{0,1\}^d$. These functions are secret, so they are derived in some way from the key. Then, the plaintext is divided into two halves L_0 and R_0. Then the output of the first round is $R_0 \| (L_0 \oplus F_1(R_0))$ (i.e. it is switched), and it is repeated n times for the remaining functions f_i. In other words, $L_i = R_{i-1}$ and $R_i = L_{i-1} \oplus F_i(R_{i-1})$.

Fiestel networks are nice because they're ivertible: $R_{i-1} = L_i$ and $L_{i-1} = R_i \oplus F_i(L_i)$. In particular, the inversion algorithm and diagram is no different from encryption: encrypting with $f_1, \dots, f_n$ is identical to decrypting with $f_n \dots, f_1$. Thus, from a hardware perspective, this is both easy and efficient. To design a cipher, it suffices to design functions $f_1, \dots, f_n$.

DES is a set of round functions $f_1, \dots, f_d : \{0,1\}^{32} \to \{0,1\}^{32}$ with a fixed function F such that $f_i(x) = F(k_i, x)$, where k_i is the i^{th} round key from a key schedule. The developers of DES tried to attack it, and created some attacks that weren't discovered until 20 years later.

Theorem 4.3 (Luby-Rackoff 1985)**.** *Suppose $F : \mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ is a secure PRF. Then, a 3-round Feistel network is a secure PRP (i.e. a secure block cipher).*

Proof. Take CS 355.

The easiest attack on DES is an exhaustive search, as in the RSA challenge. There was a message `msg = "The␣unknown␣message␣is␣"` (which is 3 plaintext blocks) and 7 ciphertext blocks (the latter

[8] You could think of this optimistically, though: if you ever lose your DES key, it can be recovered in a few days!

four of which are the secret message). The goal is to find the key to decrypt the remaining ciphertext. In 1997, the key was broken in 3 months (the rest of the message was something like "this is why you shouldn't use 56-bit keys"), which came with a $10,000 award. Then, another $10,000 was promised for doing the same in a quarter of the time. After building a $250,000 machine, the same group built it in 3 days. So DES is not terribly secure.

Since the best algorithm for breaking DES is generally brute-force, then it seems useful to try to strengthen it. This is also useful because it means that systems with hardware implementation of DES could be strengthened with a software patch.

The most obvious method is to lengthen the key. If $E : \mathcal{K} \times \mathcal{M} \to \mathcal{M}$ is a PRP, then there is a function $\texttt{3E} : \mathcal{K}^3 \times \mathcal{M} \to \mathcal{M}$ given by $\texttt{3E}((K_1, K_2, K_3), X) = E(K_3, E(K_2, E(K_1, X)))$ or sometimes $E(K_3, D(K_2, E(K_1, X)))$, so that regular DES can be implemented if for some terrible reason it were necessary; if it is necessary to use DES, only 3DES should be used. Its parameters are a block size of 64 bits and a key size of 3×56 bits.

Why triple and not double? It turns out 2DES is insecure: suppose $\texttt{2E}((K_1, K_2), m) = E(K_1, E(K_2, m))$. This can be attacked with a so-called meet-in-the-middle attack,[9] which does the following:

(1) First, it builds a table of all 2^{56} keys K_2 and $E(K_2, m)$ for each of these possible keys (which takes a while, though is technically $O(1)$).
(2) Then, for all 2^{56} keys K, test if $D(K, c)$ is in the right column of the table (which would yield both keys, and can be found relatively quickly).

This takes time 2^{57}, since two things of time 2^{56} are done. However, there could be an accidental collision, so one should use several blocks to minimize this collision probability; with 10 blocks, it's basically impossible.

An exhaustive search in AES would take about 2^{73} days if breaking DES took 3 days, which is a long time.

5. Section 1: Getting Started With Javascript, taught by Lucas Garron: 1/18/2013

For the first assignment (encrypted Facebook groups), it is worth noting that groups are hidden on Facebook's homepage. Additionally, it is necessary for your account to friend at least one other account to create a group (since Facebook apparently thinks the trivial group isn't a group).

A Chrome script is installed by going to the Window menu, clicking on Extensions, and then dragging and dropping the `.js` file to enable it. to enable it. Additionally, using a version control system if you are working with a partner will be useful.

In addition to using the keys to encrypt and decrypt Facebook messages, it will be necessary to encrypt the key list on the local storage. The password (or the key generated from it) is stored in a session cookie. Functions that do this have been provided.

Everything that one would need to edit is at the top of the file, though asserts and an RNG are provided. In particular, for random number generation, do not use the `Math.random()` function; it isn't cryptographically secure! The provided function `GetRandomValues` is better.

Addtionally, if it seems unwieldy to load Chrome each time, use phantomjs to run everything in a terminal. However, this requires installing it.

Javascript is unrelated to Java, and only has marginally similar syntax. It is actually relatively unsafe, since it's very weakly typed (prototyped, technically) and very lenient. Variables are declared with the `var` keyword (e.g. `var f = 3;`) and otherwise are global.

In order to print to stdout, write to `Console.log` (e.g. `Console.log("Hi!");`). Additionally, one can use the `debugger;` command somewhere to trigger a breakpoint there (and yield lots of cool information, such as local variables).

Javascript looks fairly similar to other language, in it use of numbers and string constants. It has objects, but no classes. After writing `var a = "str"`, one can call things such as `a.substr()` and

[9] This is not, as I had originally thought, the meat-in-the-middle attack. Sadly.

`a.indexOf("substr")`. However, every number is a 64-bit float, so try not to use integers larger than 2^{32} (accessed as `Math.pow(2, 32)`.

Arrays are similar to Python, such as `arr = [3,4]`. However, sometimes weird things happen; calling `arr.join(arr)` returns the string `"33,44"`. Functions are objects, defined in a reasonable way, such as `function foo(x,y) {var z = x+1; return z}` (though this may not work well in the console). There is support for nice things like `switch` and `if else`. There is no semantic difference between a function that returns something and one that returns nothing, and so a function can be created that returns something only sometimes. (Then, the return value is `NULL` if a return value is requested when there isn't one.) Additionally, if an argument to a function is unspecified, it defaults to `NULL` rather than throwing an exception. Functions can also be bound to variables.

One final important thing is the idea of a bit array. These are arrays which can be convered to and from strings.

6. AES: 1/23/2013

When the US encryption standard was updated in the late 1990s to make AES, the cipher chosen was the Rijndael cipher (a Belgian cipher). The key sizes can be increased from 128 to 256 to guard against weaknesses. The block size is 128 bits.

AES is an iterated EM cipher (i.e. an IEM cipher). There are d functions $f_1, \dots, f_d : \{0,1\}^n \to \{0,1\}^n$ which are fixed invertible functions. This is very different from the Feistel network, in which the functions are secret and not necessarily invertible. The cipher is a key schedule, which takes a 128-bit key K and expands it into several keys $K_0, \dots, K_d \in \{0,1\}^n$ (note that there is one more key than function). The keys are to look pseudorandom. Then, the ciper is made by taking $X \in \{0,1\}^n$ and computing $X \oplus K_0$, then taking f_1, then xor with K_2, etc. so that after the last xor (with K_d) one has an encrypted n-bit cipher.

The following theorem is the theoretical foundation of this cipher. Note that it doesn't apply to AES, since the functions in AES are fixed. It just applies some good intuition; there is no proof that AES is secure.

Theorem 6.1. *If $f_1, \dots, f_d, R\{0,1\}^n \to \{0,1\}^n$ are random functions, then no attacker can distinguish $(f_1, \dots, f_d, E(K_j))$ and $(f_1, \dots, f_d, R)$ with q queries with probability greater than $\Omega(q^{d+1}/2^{nd})$ (i.e. the larger d is, the smaller the probability).*

For AES, in fact, $f_1 = f_2 = \cdots = f_{d-1}$, and f_d is slightly different.

Recently, Intel and others added hardware support (called NI, native implementation) with the commands `aesenc` and `aesenclast`, which do one round of AES. This makes them go much faster, since they're so much lower-level.

A mistake in the design of the key schedule leads to a related-key attack on AES-256. Given 2^{99} plaintext-ciphertext pairs (this is basically a science-fiction number, and will never happen; thus, the cipher has 2^{99} bits of security, not 2^{256}) from four related keys (related by two bit-flips), the keys (specifically, the master key) can be recovered in time 2^{99}. This doesn't apply to AES-128, which oddly implies that the latter is more secure than AES-256. This attack isn't too bad, though; people still use AES-256 all the time, and it's not rendered completely insecure. But it illustrates how difficult designing ciphers can be.

Let $F : \mathcal{K} \times X \to X$ be a PRF and define $\mathrm{Funs}[X, Y]$ to be the set of all functions $X \to Y$ (which is an incredibly huge set, of size $|Y|^{|X|}$) and the much smaller $S_F = \{g(x) : X \to Y \mid g(x) = F(K, x) \text{ for some } k \in K\} \subseteq \mathrm{Funs}[X, Y]$. This has size $|\mathcal{K}|$. Then, a PRF is secure if a random function in $\mathrm{Funs}[X, Y]$ is indistinguishable from a random function in S_F; more precisely, consider an experiment with an adversary in which a key K is chosen exactly once. For $b \in \{0,1\}$, define an experiment

$$\mathrm{EXP}(b) = \begin{cases} b = 0 : & k \leftarrow \mathcal{K}, f \leftarrow S_F \\ b = 1 : & f \leftarrow \mathrm{Funs}[X, Y]. \end{cases}$$

If the attacker cannot efficiently determine the value of b, then the PRF is secure.

Definition 6.2. F is a secure PRF if for all efficient A,

$$\mathrm{PRF\ Adv}[A, F] = |\Pr[\mathrm{EXP}(0) = 1] - \Pr[\mathrm{EXP}(1) = 1]|$$

is negligible.

Under this definition, it is believed that AES is secure; in general, it isn't really possible to prove that a function is secure, but after making a single assumption, a huge amount can be built and learned. A secure PRP is no different, though instead of $\mathrm{Funs}[X, Y]$ one considers the set $\mathrm{Perms}[X]$, the set of all permutations on X.

The specific AES-PRP assumption is that all 2^{80}-time algorithms A have $\mathrm{PRF\ Adv}[A, \mathrm{AES}] < 2^{-40}$. This is a reasonable assumption, but not known with certainty.

Lemma 6.3 (PRF Switching Lemma)**.** *Any secure PRP is also a secure PRF: let E he a PRP over $(\mathcal{K}, X)$. Then, for any q-query adversary A,*

$$|\mathrm{PRF\ Adv}[A, E] - \mathrm{PRP\ Adv}[A, E]| < \frac{q^2}{2|X|}.$$

Thus, if $|X|$ is large enough such that $q^2/2|X|$ is negligible, then $\mathrm{PRP\ Adv}[A, E]$ is negligible, so $\mathrm{PRF\ Adv}[A, E]$ is as well. This is related to the birthday paradox: if one makes less than $\sqrt{|X|}$ queries, these functions are indistinguishable. Thus, AES can also be used as a secure PRF.

Now, how should one build a secure encryption system from a PRP? Security is defined with 2 parameers: the power that the adverary has (which differs depending on whether the key is used once or with many plaintext-ciphertext pairs) and what goal the attacker is trying to achieve.

A common mistake is to use Electronic Code Book (ECB) mode, a bock cipher in which $c_i = E(K, m_i)$ for some function fixed function E. This is a problem because if $m_1 = m_2$, then $c_1 = c_2$, which is already information the adversary shouldn't know. This is particularly problematic for pictures, since the human eye can interpret such badly encoded data reasonably well.

For a one-time key (e.g. encrypted email, in which there is one key used per message), the notion of semantic security is used. The one-time pad is semantically secure for *all* adversaries, no matter how efficient or smart (or even if they are in movies).

A stream cipher built from a PRF also has semantic security in deterministic counter mode: here, $c_i = m_i \oplus F(K, i)$, so that c_1 and c_2 aren't necessarily equal if $m_1 = m_2$.

Theorem 6.4. *For any $L > 0$, if F is a secure PRF over $(\mathcal{K}, X, X)$, then the above encryption scheme (sometimes called E_{DETCTR}) is semantically secure over $(\mathcal{K}, X^L, X^L)$. In particular, for any adversary A attacking this, there exists a PRF adversry B such that* $\mathrm{SS\,Adv}[A, E_{\mathrm{DETCTR}}] = 2\,\mathrm{PRF\ Adv}[B, F]$.

For a many-time key (as in encrypted file systems or packets in a network), the above options don't apply, and some things must be redefined:

Definition 6.5. Consider a many-time key system in which an adversary can query q times with 2 messages $m_0, m_1 \in \mathcal{M}$, and consistently either the first or the second message is encrypted and returned. Then, the system is semantically secure if the adversary cannot tell which side is encrypted.

Note that this simulates real-world conditions reasonably well, and that if the adversary wants the ciphertext for a given plaintext m, then it just needs to query (m, m). Thus, this attack is called a choesn plaintext attack (CPA), and some types of ciphers (e.g. stream ciphers) are insecure under this attack. In general, if $E(k, m)$ always produces the same ciphertext, then it is insecure (since one can try (m, m) and then (m, m') to determine which of the pair is encoded). In the real world, of course, many systems have this folly.

7. More PRFs and PRPs: 1/28/2013

In some sense, a PRF is a "random-access" PRG; from some seed K, it yields a sequence $F(K, 0)$, $F(K, 1)$, ..., each of which looks random. Importantly, in order to get any particular element of the sequence, one doesn't need to compute all of the previous blocks (which is unlike some other ciphers,

such as RC4). This PRF can be used to generate keys for a stream cipher, which is referred to as counter mode. It can also be used for key derivation.

If the same key is used many times (e.g. packets in IPsec or multiple files in a file system), then the notion of security must be generalized.

Definition 7.1. Suppose a cipher $E = (E, D)$ is defined over $(\mathcal{K}, \mathcal{M}, \mathcal{C})$. Choose a $b \in \{0, 1\}$ and allow the adversary to repeat queries: the adversary sends two messages $m_0, m_1 \in \mathcal{M}$ and sees $x_i = E(k, m_b)$. Then, the system is CPA secure (chosen plaintext attack) if the attacker cannot determine the value of b in the advantage formalism as before.

Notice that the adversary can encrypt any given plaintext by setting $m_0 = m_1$ in any particular trial.

As discussed previously, stream ciphers are insecure under a CPA, and more generally, if $E(k, m)$ always produces the same ciphertext, then it cannot be secure: first, the attacker sends m_0 and m_0 to get the ciphertext c_0, and then sends m_0 and some other message m_1 and compare the result to c_0 to determine the value of b. Thus, if one uses a many-time key, the encryption algorithm can't be deterministic!

The solution to this is a nonce-based encryption system, so that the key is instead a key-nonce pair (or alternatively, the encryption and decryption algorithms accept a nonce $E(k, m, n)$ and $D(k, c, n)$ that is shared between the encryptor and the decryptor). The important property of the nonce is that each key-nonce pair can never be used more than once, though otherwise the nonce can be anything. (In the real world, many systems don't implement this correctly and end up reusing nonces.) Thus, one can choose a nonce randomly from a large enough space (in which the birthday paradox makes life difficult) $n \leftarrow \mathcal{N}$. A 128-bit nonce requires 2^{64} tries on average before a collision, which is almost certainly fine. But the nonce can even be a counter (which is particularly helpful in packet protocols) — it doesn't matter if the nonce is known, or even if it is secret (as long as the key is). However, counters don't always work perfectly; if the nonce is deterministic, then the states of the two communicators must be the same (and in particular cannot be reset).

One example of this is CBC (cipher-block chaining) with a random nonce (also called an IV). Then, $c_0 = E(k, \text{IV} \oplus m_0)$ and $c_j = E(k, c_{j-1} \oplus m_j)$. Then, the ciphertext is $\text{IV} \,\|\, c_0 \,\|\, c_1 \,\|\, \ldots$. However, this isn't necessarily secure.

Theorem 7.2 (CBC)**.** *For any $L > 0$, if E is a secure PRP over $(\mathcal{K}, X)$, then E_{CBC} is semantically secure under CPA over $(\mathcal{K}, X^L, X^{L+1})$.*

In particular, for a q-query adversary A attacking E_{CBC} there exists a PRP adversary B such that

$$\text{SS}_{\text{CPA}}\,\text{Adv}[A, E_{\text{CBC}}] \leq 2\,\text{PRP}\,\text{Adv}[B, E] + \frac{2q^2L^2}{|X|}.$$

Note that the CBC is only secure if $q^2L^2 \ll |X|$ (which is a birthday-like bound to prevent collisions).

In this case, if the IV is a counter, the message is insecure. However, encrypting the IV with something such as the key used for the rest of the message is also insecure. In order to properly encrypt with a randomized nonce, an entirely independent key must be used to encrypt the nonce. This is tricky, because the API for many crypto libraries doesn't make this extremely obvious, which leads to many of the errors in practice.

There's a technicality here; if the length of the message doesn't divide the block size, there must be some padding that can be unambiguously removed from the decrypted ciphertext. One standard (e.g. in TLS) for an $(n + 1)$-byte pad is to write n to each byte (so that the decryptor can tell exactly how many bytes to remove). However, if all of the bytes are message bytes, this is ambiguous, so the solution is to add a dummy block. This is fine in principle, but sometimes means you have to send lots of excess data as padding (e.g. a large number of 16-bit messages). Once again, the takeaway is to not use CBC.

But there's one more issue yet with CBC: it's sequential. This means that it can't be done quickly, even if one has many CBC processors.

Another alternative is randomized counter mode. Here, the IV is chosen at random and then incremented for each block. This, unlike CBC, is parallelizable, making lots of people happy. However, there are many browsers (where by "many" we mean IE6) that might not accept this mode.

To ensure that the particular $F(k, x)$ (where $x = IV + j$ for some j) is never reused, then the IV is generated in 2 64-bit sections: the first is the nonce and the second is a counter that starts at 0 for every message. Another advantage of random counter mode is that there's no need for padding, so it's even more awesome.

However, CPA security is not sufficient for real-world encryption, so much of this lecture wasn't completely useful. In particular, neither CBC or its randomized version are useful on their own; they must be combined with an integrity mechanism, which will be discussed next lecture.

Consider an exhaustive search attack on a PRP (e.g. AES). Given some number of pairs $(m_1, F(k, m_1))$, is it possible to obtain the key k? Typically, the time is $O(|\mathcal{K}|)$, which is 2^{128} for 128-bit AES (i.e. billions of years). However, quantum computers make life more interesting. With quantum physics, an electron's spin is in a superposition of two states (top and bottom), so n electrons represent 2^n possible states. This is a very efficient way to encode information, so one might be able to execute programs on all 2^n possibilities very quickly (so, basically, magic).

A quantum computer could make almost all of the topics in this class invalid or easily broken, but the only issue is that nobody knows how to build one. But ignoring this trivial implementation problem, there exists an algorithm called Grover's algorithm for an exhaustive search. If $f : \mathcal{K} \to \{0, 1\}$, the goal is to find a $k \in \mathcal{K}$ such that $F(k) = 1$. On a quantum computer, there is a method to implement this in time $O(\sqrt{|\mathcal{K}|})$, so a search on AES-128 takes time 2^{64}, which is not that much longer than that for classical DES. Thus, on a quantum computer that operated about as fast as a classical computer, it would only take about 256 days to break AES. This is actually one of the reasons AES-256 exists: even on a quantum computer, breaking it would take a very long time (billions of years again). There might yet be more sophisticated quantum attacks on AES-256, but nobody has discovered them yet, and information-theoretically, it's not really possible to do much better.

8. Message Authentication Codes: 1/30/2013

The general goal here is to preserve integrity. Unlike in previous examples, the information transmitted could be freely public (e.g. a software patch), but the goal is to prove that the message has integrity, or hasn't been tampered with.

If Alice sends a message to Bob, they share a secret key K. Alice has an algorithm $S(K, m)$ that generates a tag that is appended to the end of a message. Then, Bob has a data verification algorithm $V(K, m)$ that returns whether the data has been changed or not.

Two important points are that the confidentiality isn't really the point, but they need to share a secret key. Networking people tend to be more concerned with random errors, so message authentication is as easy as calculating a checksum. However, if the errors are adversarial rather than random, an attacker can change the data and calculate the new checksum to make it look legit. This leads to mistakes: the first version of SSH used CRC, for example.

Definition 8.1. An MAC is a pair $I = (S, V)$ over $(\mathcal{K}, \mathcal{M}, \mathcal{T})$ (the latter being the tag space), such that for all $K \in \mathcal{K}$, $m \in \mathcal{M}$, and $t \in \mathcal{T}$, $V(K, m, S(k, m))$ returns `True`.

For security, the attacker has the ability to get the tag of any message (since this does happen in the real world) $t_i \leftarrow S(K, m_i)$ for $1 \le i \le q$. The attacker's goal is something called existential forgery, which is to produce a valid pair $(m, t) \notin \{(m_1, t_1), \ldots, (m_q, t_q)\}$. m may be gibberish, but it is still a reasonably conservative definition of security that leads to a very secure MAC.

Definition 8.2. An $I = (S, V)$ is secure if for all efficient adversaries A,

$$\mathrm{MAC}\,\mathrm{Adv}[A, I] = \Pr[\text{the adversary succeeds}]$$

is negligible.

The formal game is, given some $K \in \mathcal{K}$, the attacker may query some $m_i \in \mathcal{M}$ and see the tags $t_i = S(K, m_i)$. Then, he wins if he sends a (or one successful one of several) (m, t) pair such that $V(K, m, t)$ returns `True`.

It turns out that any secure PRF gives a secure MAC (through a compression algorithm for the tag) $F : \mathcal{K} \times X \to Y$, where $X = \{0,1\}^m$ and $Y = \{0,1\}^n$, with $m \gg n$.

Definition 8.3. If F is a PRF, then define $I_F = (S, V)$, where $S(K, x) = F(K, x)$ and $V(K, x, t) = [F(K, x) = t]$.

Though this seems easy in practice, care must be taken in checking equality; if the verifier takes different amount of time to reject more or less correct false tags, then the MAC can be broken. This is an excellent reason to use libraries rather than implementing one's own primitives. In particular, if your implementation of some protocol is significantly faster than the standard, then you're going to have a bad time.

Lemma 8.4. *If $F : \mathcal{K} \times X \to Y$ is a secure PRF, then I_F is a secure MAC.*

Proof. Suppose an algorithm A can break I_F. Then, there exists a PRF attacker B with the same running time as A such that

$$\text{MAC Adv}[A, I_F] \leq \text{PRF Adv}[B, M] + \frac{1}{|Y|},$$

as long as this error term is non-negligbile. ⊠

For a (bad) example, consider a secure PRF $F : \mathcal{K} \times X \to \{0,1\}^{10}$. Then, the MAC I_F has a very small tag space, so one could just guess keys and be right 0.1% of the time (which is a lot).

So as a better example, consider the friendly neighborhood cipher AES, which is much more secure, but only worls for 16-byte messages. Thus, one important question is: if given a small MAC, is it possible to build a Big MAC out of it?

One method, called CBC, takes a PRF $F : \mathcal{K} \times X \to X$ (where $X = \{0,1\}^n$, and $n = 128$ in the real world) and builds an L-block MAC

$$F_{\text{CBC}} : \mathcal{K}^2 \times X^\ell \to X$$

(where $\ell \leq L$) given by a raw CBC

$$R = F(K, m[L] \oplus F(K, m[L-1] \oplus \dots F(K, m[0]) \dots))$$

and a final product $F(K_2, R)$ (which is necessary for security). Sometimes, one also adds an IV, but this has no security effect, and is there just to make implementation easier, so it can be ignored.

Theorem 8.5. *For any $L > 0$, if F is a secure PRF, then F_{CBC} is a secure PRF and therefore a secure MAC. In particular, for every PRF attacker A on F_{CBC}, there exist a PRF attacker B on F such that*

$$\text{PRF Adv}[A, F_{\text{CBC}}] \leq \text{PRF Adv}[B, F] + \frac{q^2 L}{|X|},$$

so that if the right-hand side is negligible, then the left-hand side is as well. Here, q represents the number of messages one can MAC without changing the key.

It is very important that the raw CBC is not the final step (though of course some people forget this in the real world): the attacker can pick a random $m \in X$ and request the tag t of m. Then, the attacker outputs t as the tag on the message $m' = (m, t \oplus m) \in X^2$. This is an existential forgery, since

$$R(K, (m, t \oplus m)) = F(K, F(K, m) \oplus (t \oplus m)) = F(k, m) = t.$$

In general, though, one needs to pad the message to get the message to the right length for the algorithm. One common solution (even in an ANSI standard) is to pad with zeroes, but this is insecure: $\text{mac}(K, m) = \text{mac}(K, m \,\|\, 0)$. This seems benign, but what if the message is "Withdraw \$1000" and a few extra zeros can be very dangerous? Maybe it would be better to prepend the padding. This is called prefix stream encoding, which works because no message is the prefix of another message, but this doesn't work well with libraries (since they typically accept plaintext only one block at a time).

The padding needs to be invertible for security, so one can append `100...000` if the message length is a multiple of the block length, and otherwise append a dummy block of `100...000`.

CBC MAC is a relatively old standard, and is very commonly used. An alternative is PMAC, a parallel MAC. This has a simple, easy-to-compute function (a couple shifts and multiplications) $p(K, n)$ such that the MAC is

$$F\left(K_2, \bigoplus_{i=0}^{n-1} F(K, p(K, i), m[i])\right),$$

with K_2 needed to implement security as in the previous example. p is used to make sure that the order of the blocks matters, and should be much faster than F. The error term for security is slightly worse than for CBC MACs.

For an example consider a sequential MAC called NMAC.[10] Let $F : \mathcal{K} \times X \to X$ be a PRF. Then, the MAC is

$$F(K_2, F(m[n], F(m[n-1], \dots, F(m[0], K)\dots))),$$

where $K, K_2 \in \mathcal{K}$ are unrelated (otherwise this isn't secure, just like above).

Theorem 8.6. *If F is a secure PRF, then NMAC is also a secure PRF.*

Definition 8.7. A hash function is a function $H : \mathcal{M} \to \mathcal{T}$ such that $|\mathcal{M}| \gg |\mathcal{T}|$. A collision for H is a pair $m_0, m_1 \in \mathcal{M}$ such that $H(m_0) = H(m_1)$.

A hash function is collision-resistant if it is hard to find collisions for it: for all uniform (i.e. explicit) efficient algorithms A,

$$\mathrm{CR\,Adv}[A, H] = \Pr[A \text{ outputs a collision for } H]$$

is negligible.

Collision-resistant hash algorithms are pretty magical, since they skirt arount the pidgeonhole principle. Thus, they are very good for going from a small MAC to a big MAC: if $I = (S, V)$ on $(\mathcal{K}, \mathcal{H}, \mathcal{T})$, then one can make a new MAC $I' = (S', V')$ on $(\mathcal{K}, \mathcal{M}, \mathcal{T})$ given by $S'(K, m) = S(K, H(m))$. Collision-resistance is pretty necessary for security, since if it were easy to determine collisions, then one could replace a message m with a given MAC with something that collides with it.

9. Section 2: More on the First Project, taught by Lucas Garron: 2/1/2013

Some details of the project have changed in the last two weeks; see Piazza for the whole list.

(1) The session cookie code had some issues, so now you should use the function `sessionStorage.setItem("key","value")` and `sessionStorage.getItem("key")`. This is secure and temporary storage per tab. Notice that everything has to be passed as a string.
(2) Facebook seems to clear local and session storage when one logs out, which makes having a key database difficult. This means that there is now a `manifest.json` that needs to exist (so the name of the script in this file should be changed). Everything is run as an extension. Facebook also clears everything after 20 to 30 minutes of inactivity, so try not to let that happen. Fixing this would require enough elbow grease that it's not worth worrying about.
(3) Since password security is somewhat complicated, there is now a `sjcl.misc.pbkdf2("password-string",salt,null,128,null)`, where the `salt` is a randomly generated nonce that is stored somewhere distinct from the message (i.e. in local storage, just like for the database, and it can even be stored in plaintext) and 128 is a number of iterations.

The starter code has been updated correspondingly.

The Javascript crypto library, `sjcl`, stores data in bit arrays (e.g. `GetRandomValues(4)`, which returns an array of 4 random ints). A string can be converted to a bit array, but it's important to not confuse strings, binary data, and string representation of binary data. Going from a string to a bit array is given by `sjcl.codec.utf8String.toBits(str)`. Another useful function is `sjcl.bitArray.bitLength()`, but in order to use AES, the array has to be padded to a multiple of 128, as `arr1.concat(arr2)`. Then, bitwise `xor` is as easy as:

[10] This is the basis of a very famous MAC called HMAC, which uses a collision-resistant hash function. Thus, one obtains hash tags, which are (again) used to compress information.

```
for(i = 0;i < 8;i++) {
   out.push(arr1[i] ^ arr2[i]);
}
```

Then, one can do encryption:

```
a = new sjcl.cipher.aes(GetRandomValues(4))
var encrypted = a.encrypt(arr1)
```

Similarly, `a.decrypt(encrypted)` returns the original bit array.

10. Collision Resistance : 2/4/2013

Suppose $H : \mathcal{M} \to \mathcal{T}$ is a hash function (so that $|\mathcal{M}| \gg |\mathcal{T}|$). A collision is a pair $X_0 \neq X_1$ such that $H(X_0) = H(X_1)$. Because the hash function goes from a large space to a smaller one, then collisions are inevitable, but a collision-resistant hash function is one for which no efficient algorithm can find one with greater than negligibl probability.

Consider a secure MAC (S, V) with message space $\mathcal{T}$. Then, a hash function can be used to implement a much larger MAC (S', V'), such that $S'(K, m) = S(K, H(m))$ and $V'(K, m, t) = V(k, H(m), t)$. As demonstrated in the previous lecture, this big MAC is secure if H is collision-resistant.

This has applications in file integrity: if there exist files F_1, $F_2, \ldots$ and some public read-only space, one can post the hashes of the files $H(F_1)$, etc., in the public read-only space. If H is collision-resistant, then an attacker modifying the files cannot do so undetected.[11] This is nice because it requires no key, but it does require the public, read-only space.

Here is a method of attacking collision-resistance: choose $O(\sqrt{|\mathcal{T}|})$ messages m_i and hash them as $t_i = H(m_i)$. Then, search for collisions; if none are found, then repeat.

Theorem 10.1. *Suppose that $t_1, \ldots, t_n \in \mathcal{T}$ are identically and independently distributed.[12] Then, when $n = 1.2|\mathcal{T}|^{1/2}$, then $\Pr[t_i = t_j$ for some $i \neq j] \geq 1/2$.*

Proof.

$$\begin{aligned}
\Pr[t_i = t_j \text{ for some } i \neq j] &= 1 - \Pr[t_i \neq t_j \text{ for all } i \neq j] \\
&= 1 - \prod_{k=1}^{n-1} \left(1 - \frac{k}{|\mathcal{T}|}\right) \\
&\geq 1 - \prod_{k=1}^{n-1} e^{-\frac{k}{|\mathcal{T}|}} = 1 - e^{\sum_{k=1}^{n-1} \frac{1}{|\mathcal{T}|}} \\
&\geq 1 - e^{-\frac{n^2}{2|\mathcal{T}|}} \geq 1 - e^{-\frac{1.2^2}{2}} > \frac{1}{2}
\end{aligned}$$

⊠

This algorithm has both time and space of $O(\sqrt{|\mathcal{T}|})$, which is fairly large.[13] Interestingly, there is a

Name	$\log_2(\lvert\mathcal{T}\rvert)$	Speed (MB/sec)	generic attack time
Sha-1	160	153	2^{80}
Sha-256	256	111	2^{128}
Sha-512	512	99	2^{256}
Whirlpool	512	57	2^{256}

Table 1. A comparison of various hash functions.

(again theoretical) quantum attack on hash functions that runs in time $O(|\mathcal{T}|^{1/3})$.

[11] If the attacker is able to intervene in the read-only data, this is not secure, however.

[12] This theorem holds for a general distribution, but everything is nicer in the uniform distribution, so the proof will consider that case.

[13] The algorithm can be optimized to use constant space, but that is beyond the scope of this course.

Building a hash function relies on the Merkel-Damgard Paradigm, which takes a small hash function h, which relies on constant-sized inputs, and builds a larger hash function H that operates on arbitrarily-sized inputs. H operates as follows: if m is a message broken up into k blocks of size appropriate for h (plus a padding block p), then $H(m) = h(m[k] \,\|\, p, h(m[k-1], h(\dots(h(m[1], \mathrm{IV}))\dots)))$, for some fixed initial value, and with the padding block equal to $100000\dots$ followed by the message length.

Theorem 10.2. *If h is collision-resistant, then so is H. In particular, a collision of H induces a collision of h.*

Proof. Suppose the inputs to the last (outside) call to h are the same in some collision of H; if not, then there is a collision found for h. Then, using the same logic on the previous call to h and winding back, either a collision for h is found somewhere or the two messages hashed with H are the same. ⊠

Thus, the only ingredient left in H is the small function h. This can be done by everyone's favorite tool: the block cipher. One's first idea might be to take $h : \mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$ given by $h(H, m) = E(H, m)$, but this is completely insecure: given m, m', H, one can find H' by computing $D(m', E(m, H)) = D(m', E(m', H')) = H'$.

The better way to do this is $h(H, m) = E(m, H) \oplus H$, called the Davies-Meyer protocol.

Theorem 10.3. *If E is an ideal cipher (i.e. it can be modelled as $|\mathcal{K}|$ random permutations),*[14] *then finding a collision requires $O(2^{n/2})$ evaluations of E (which is ideal).*

SHA-256 uses both Merkel-Damgard and Davies-Meyer, and its underlying block cipher is a 256-bit block, 512-bit key cipher called SHACAL-2. Thus, the output size of SHA-256 is in fact 256 bits.

HMAC is built from a hash function, and the construction is as follows:

$$S(K, m) = H(K \oplus \mathrm{opad} \,\|\, H(K \oplus \mathrm{ipad} \,\|\, m)),$$

where opad and ipad are 2 strings (the inner and outer pads) and H is some hash function (which can just be SHA-256). This can't be proven to be secure, but it's reasonable enough to assume so. NMAC is fairly similar, and has similar security.

MACs are vulnerable to a generic timing attack: since verifying a MAC is as easy as comparing two strings, generically comparing strings takes different amounts of time depending on how different they are, so the attacker can learn to construct a message by guessing and keeping the correct parts.

Defending against this attack requires constructing a separate routine to compare the strings, but can be fooled by optimizing compilers. One can also apply the MAC again rather than comparing against the precomputed answer, which obscures the timing a bit but not completely. Once again the upshot is to not implement the crypto yuorself.

11. Cryptography and Security at Facebook: 2/6/2013

Today's lecture was given by Scott Renfro, an employee of Facebook who works in cryptography. Previously, he worked at Yahoo! and several small startups implementing cryptography and security. All of these stories are true, though names may have been changed. It seems that bad crypto is everywhere, written by people who haven't had a lot of experience in being careful.

Specifically, Scott work in the Security Infrastructure team, which is a sort of misnomer for a group that works on cryptography and web security. One major issue is making security happen at scale, which comes with its host of problems:

- The goal is to build "systems to remain dependable in the face of malice, error, or mischance." - Ross Anderson. Security has also been described as programming Satan's computer.
- At scale, another problem is one's coworkers: not everybody is a security pro, but they may all have to interact with security.

[14] Like most things named ideal, such a cipher doesn't actually exist, but some ciphers come closer than others.

Security is generally improved by discovering new problems. Typically, these lead to instructions such as: when doing X, be sure to also do Y and Z. This is hard to remember, so it's often smarter to make a secure library for doing X that automatically also calls Y and Z. Designing this library (or API) sometimes should have an unsafe design: if you make an ugly, annoying backdoor, people who need to use it can, but you can monitor how it is used, and the ugliness repels some users who shouldn't use it. However, keep in mind that no name is so repulsive that nobody will use it. Even a method called `__do_not_access_me_or_you_will_be_fired` was misused! Thus, it helps to make these names greppable, so that it's clear when they're being used.

When shoud cryptography even be used? The correct answer is only when nothing else works: it's hard to get right and if there's a better solution in practice, it is far more likely to be secure, no matter how theoretically interesting crypto is. Thus: one shoud prefer a stored value to an encrypted value, and if encryption is necessary, use a per-user secret. For example, if the user forgets their password, one can generate a random string per user that can be used a passsword reset, which avoids a single master key. Ideally, one should avoid bugs in these programs, though this can be wishful thinking.

Here are some examples of bad crypto:

(1) Once, Facebook had an option in which someone could share a link to a photo album to someone not on Facebook with a link such as `http://www.facebook.com/album.php?user=1234&album=4&hash=5a3ff`. The red text is a hash, but since there are only 1 million possibilities, this can be brute-forced.

(2)
```
function photo_code($user_id, $album_id) {
    return substr(
        md5('m0r3-sekret'.$user_id.$album_id),
    );
}
```
(In PHP, the dot means concatenation). This has lots of problems: 5 bits isn't sufficient, the secret key is hardcoded in and can't be changed, there is a single global key, the MAC is a bare hash, MD5 is old and broken, etc.

(3) Try a Google search on the order of `http://www.google.com/search?q=secret_token+site=github.com`. This reveals lots and lots of backdoors...

Facebook's current crypto library is better, with project-specific keys and key versioning, and with a simple API.

Project specific keys prevent oracle attacks (also known as sticking tab A into slot C); if the same key is used for multiple features, one might leak information about the other. For example, Facebook can implement a social captcha that checks if you are who you say you are by identifying your friends (e.g. when logging in from a foreign location). But the report abuse form might include encrypted data called with the same decrypt function, so the two can be chained to compromise the social captcha.

One important function is `isValidMac()`, whose importance is a bit less obvious. There are lots of reasons for it to exist: it defends against timing attacks, for example, and the concept of key rotation means that valid messages might not be caught otherwise. This is also because of some stupid PHP tricks, such as this implementation of MD5:

```
<?php
    $x = '00e298837774667068706870210040432782';
    echo (int)(md5($x) == $x), "\n"
```

PHP will interpret things with an `e` as floating point, so this code is equivalent to comparing whether `0.0 == 0.0`. Since this would be the most obvious solution to many programmers, various tricks have to be used to get around it.

However, making these algorithms interoperate (i.e. work correctly between different protocols)is tricky. This is particularly hard because different languages treat strings very differently, etc. Additionally, in order to have meaningful communication, you can't just specify a cipher, but also a mode, a MAC, key derivation, etc.

One tempting method of encrypting is CFB mode (which this class also calls CFC mode), which is nice because there isn't too much padding involved. But libraries are also inconsistent: one of OpenSSL

and Microsoft CryptoAPI shifts per iteration per bit, and the other shifts per byte, so they can't talk to another.

On top of that, AES means a lot of things: some implementations of AES are actually Rijndael and vice versa. AES-128 might mean AES with 128-bit keys, AES with 128-bit blocks, AES eith 128-bit keys and 128-bit blocks, and even AES with 128-bit keys and 256-bit blocks! Thus, using a library is nicer, but you'll probably have to write your own.

Key versioning and rotation is also important, since it can either be done ahead of time or in an emergency. There should be two keys, a current and a next key, such that there's a nice protocol for updating all of the data. However, make sure people choose good keys! `password` is not a good key. This is funny until it happens in the real world.

Another application of libraries is to encrypt and then MAC, which avoids a lot of padding attacks (look one of these up: they're quite interesting). For some public-key operations, this isn't necessarily secure, though. A variety of the Anderson-Needham attack shows that a key that's the product of three large primes has two methods of factorization, leading to two secret keys (which can cause issues).

Another useful library feature is the option to choose several different algorithms: a fast but insecure algorithm has its uses, for example. One could also bind data to the user ID, so that only the same user can decrypt it. Mistakes can be caught with Lint (e.g. looking for MD5, etc.) and using greppable alternatives to make switching easier.

Two libraries worth checking out are Google's KeyCzar and Dan Bernstein's NaCl. Both of these are open-source.

One can use a library feature to sign URIs, or to generate tokens (encrypted payloads that are only viewable for a little while). Then, a timestream is embedded before the token is encrypted.

There are lots of issues with session identification. A cookie is used to do this, but how is it specifically implemented? The most common is `user=$user`, which is completely insecure (local storage can be edited). It is helpful to MAC the user or encrypt some per-user secret that is stored in a database (or maybe its hash is stored in the database). Similarly, for password storage, some sites still store plaintext passwords, and others use unsalted hashes. Rainbow tables or even a Google search can be used to get past this. Thus, a salted hash is nice, but it can even be brute-forced, and says nothing about security of stupid passwords.

Key to this is keeping the secret key within only the scope it needs. In many situations, security is more important than latency, so the key shouldn't be distributed to other servers (in case they are compromised). Generally, one server serves as a keychain (which may be more or less secure).

When scaling everything up, anything can happen! The bug is never in the compiler (network, CPU; particularly in embedded systems) until it is. One-bit errors (such as domain squatting on sites such as `woogle.com`) are disturbingly common, and come with bad session cookies, which also happen fairly frequently. These can be little network issues in which errors cancel out, etc. Sometimes also a migration from an old system to a new system fails for some reason. This can be mitigated by double-reading and -writing.

For example, one example of actual code has a broken shift line that occasionally (1 time in 5) inverts a bit and causes an error. Some of these bugs can be very pernicious: they might only be on one machine, or an error that only happens on SHA-1 but not SHA-256, when byte 12 is nonzero on the first block on an unaligned read (three days after the winter solstice, etc.).

12. Authenticated Encryption: 2/11/2013

Authenticated encryption seeks to combine both secrecy and integrity. In some sense, this is the most important aspect of symmetric encryption. Thus far, this class has considered

- integrity, with MACs such as CBC-MAC and HMAC, which has integrity but not secrecy, and
- confidentiality with block ciphers, which doesn't have integrity.

If the attacker can mount active attacks, CPA security is basically useless without some sort of integrity. Generally, stream ciphers are malleable, so an attacker can, for example, change the recipient field in an encrypted email, and then read the email. Even in CBC, you can take the IV and xor it by a fixed amount,

changing the plaintext in the first block by that same amount. The second or third block requires more ingenuity, but the same issue stands. The point is, encryption without integrity is akin to not encrypting at all, which is a common mistake.

The way to fix this is to use a mode called authenticated encryption, which combines confidentiality and integrity. The goal is to have CPA security (including semantic security when using a multi-use key), as well as a new property called ciphertext integrity. This means that the decrypter should reject modified ciphertexts, which is modelled as a game:

The challenger chooses a random key $K \xleftarrow{R} \mathcal{K}$, and the attacker can send some number of pairs of messages $m_{i,1}$ and $m_{i,2}$, and receives the encryption $E(K, m_{i,b})$ for some constant $b \in \{0, 1\}$. Then, the attacker wins if it can provide a forgery: $D(K, c)$ is not rejected, but $c \notin c_1, \ldots, c_n$ (the ciphertexts seen by the attacker).

Definition 12.1. An encryption scheme (E, D) has ciphertext integrity if for all efficient attackers A, the probability of such an attack succeding is negligible.

This has some implications: if (E, D) has ciphertext integrity, then any ciphertext c^* which passes decryption must have come from someone with the secret key, since no other keys can be generated.

The history of authenticated encryption is kind of sad, because until relatively recently, there were lots of ways to try it, most of which wouldn't work. Here were three common examples; common to all three are an encryption key K_E and a MAC key K_I.

(1) TLS MACed, then encrypted: $m \mapsto m \,\|\, S(K_I, m)$, and the whole thing is encrypted as a block with K_E: $c = E(K_E, m \,\|\, S(K_I, m))$. Since there need to be different keys in different directions, this means there are lots of keys floating around; however, one could start with a single key and then use a PRF to obtain others. This is called (unsurprisingly) MAC-then-encrypt.
(2) IPSec encrypts first: the ciphertext is $c = E(K_E, m)$, and then the transmitted message is $c^* = c \,\|\, S(K_I, c)$. Thus, the MAC is computed on the ciphertext and appended. This method is called encrypt-then-MAC.
(3) SSH encrypts, but computes the MAC based on the plaintext: $c = E(K_E, m) \,\|\, S(K_i, m)$.

Can you guess which of these is secure? The moral is to not implement or invent crypto on your own.

Clearly, SSH[15] is not secure; the ciphertext leaks some information about the plaintext. Option 2 prevents this, because, informally, encrypting the MAC prevents someone from tampering with the ciphertext. This can be formalized:

Theorem 12.2 (Authenticated Encryption #1)**.** *Method 2, or encrypt-then-MAC, provides authenticated encryption for any primitives that are CPA secure.*

Note that method 1 is not secure, even if the primitives are secure. However, it can be used with randomized counter mode and a secure MAC to create a secure encryption. Additionally, if the MAC has some sort of flaw, MAC-then-encrypt provides authenticated encryption even if the MAC is only one-time secure (i.e. secure only when the attacker can make only one query). There are very fast and efficient MACs that provide one-time security, so this method could be implemented nicely, but it's extremely difficult to get it to work right (e.g. very vulnerable to timing attacks). Thus, the overall recommendation is to encrypt, then MAC.

There are several standards for encrypt-then-MAC:

(1) GCM (Galois Counter Mode): the encryption scheme is randomized counter mode followed by an odd MAC called CW-MAC.
(2) CCM (e.g. 802.11i): CBC-MAC, then randomized counter mode encryption. This doesn't follow the above recommendation, but it is secure.
(3) EAX: the encryption scheme is randomized counter mode, and then CMAC.

Thus, since integrity is necessary, it is necessary to use one of these modes to be secure. Crypto libraries tend to provide APIs for some of these modes. The x86 hardware even has instructions that make CW-MAC work especially quickly.

[15] Disclaimer: the current version of SSH uses a more secure protocol than this.

All of these standards provide authenticated encryption plus associated data, in which someone might want to encrypt some data but leave the rest as plaintext (for example, when sending packets, you might not want to encrypt the target address). Thus, the goal is to provide message integrity to the entire message, but encryption for only part of it. This works with encrypt-then-MAC by taking the plaintext header and prepending it to the ciphertext before calculating the MAC.

Here is an attack on MAC-then-encrypt. After calculating the tag t for a message m, the block size might not be right, so one has to encrypt $m \| t \| p$ for some padding p. Then, decryption involves removing the pad, then checking the MAC. In early TLS, decryption consists of first decrypting the ciphertext and checking the pad (if it's invalid, then the message failed). Then, the MAC is checked; if valid, the message is valid. Note that when the decryption fails, the attacker knows why... thus, if the last byte in the ciphertext is `0`, there is a bad MAC, and if the last two bytes are `01`, then it's a bad pad, so it gets rejected with a different reason! Thus, one can compute what the last byte is, and then this can be repeated until the bad MAC report is hit, and so on, and so forth. Thus, the message is decrypted from right to left, one byte at a time (much like in Hollywood).

The standard fixed this very quickly by changing both messages to say the same thing. This is still vulnerable: the two errors take different times to respond, so there is a timing attack that does essentially the same thing. This was fixed, which worked up until about two weeks ago.[16] The standard tried to ignore the pad, pretending it is zero bytes long, and then check the whole message. The Lucky 13 attack depends on the fact that, in the case of a valid pad, the number of blocks is enough that they need to call SHA-1 twice, but if the pad is invalid, a byte drops off, and this means that only one call of SHA-1 is necessary, which induces a small timing difference. This is in fact small enough that it is only statistically meaningful over a local network, but it is a problem.

The end result of this long chain of attacks is tht TLS isn't really trusted; people use stream ciphers or encrypt-then-MAC.

For a case study, consider an encrypt-then-MAC scheme in which TLS uses the encryption scheme 3DES wuth HMAC and SHA1. The keys are unidirectional, so $K = (K_{b\to s}, K_{s\to b})$, and TLS uses stateful encryption: each side has two 64-bit counters, incremented every time each side sends a packet, and initialized to zero. These counters prevent the replaying of packets in an active attack.

The message, consisting of some initial data (a header: the type, version, and length) sent as plaintext, then the payload and the MAC (with padding), which are encrypted. Note that the MAC depends on some things which are encrypted and some things which are decrypted. The message is encrypted in a way that requires the counter, which means that this must be done over a reliable network (since if the packets arrive in the wrong order, decryption fails).

There is yet another hole in this implementation: the IV for one record can be derived from the IV for the previous record. Thus, in TLS 1.1 and up, the IV is random and sent along with the packet. A stopgap measure was to inject a dummy packet in between two meaningful packets, but this isn't exactly secure either.

One striking theoretical development is a mode for authenticated encrytption called OCB. This requires half as many calls to a MAC as in previous developments. There is a padding function p and a PRP E: $c[k] = p(N, K, i) \oplus E(p(N, K, i) \oplus m[i])$, similarly to the parallel MAC. Then, integrity is just a checksum that is the xor of all of the ciphertexts; this checksum is xored with p, then encrypted, then xored with p again.

This is nice because the block cipher is just applied once per tag, making it extremely efficient. However, it's not used all that often in practice, since bits and pieces of it are under three different patents.

13. Key Management: 2/13/2013

So far, all cryptography assumed that Alice and Bob can share a key K. But where does this key come from? If there are n users, each user has to manage n keys in order to communicate, which is unweildy.

[16] Talk about bad timing.

The first idea was that of a key distribution center (KDC). Then, each party only shares a key with this center. Then, if two users want to talk to each other, they use the KDC to establish a shared key. For example, users 1 and 2 with keys K_1 and K_2 can communicate by asking the KDC for a key K_{12}. Then, it sends to user 1 the encrypted key $c_1 = E(K_1, \text{“1,2”} \,\|\, K_{12})$ and to user 2 the encrypted key $c_2 = E(K_1, \text{“1,2”} \,\|\, K_{12})$. This is CPA secure (though may be vulnerable to active attacks). This was used by a system called Kerberos (with some modifications).

The pressure point of the KDC is problematic: if it goes offline or is broken into, bad things happen, and even when things are working normally, the KDC can listen in on messages. Thus, people began wondering if secure key exchange could be done without a KBC.

For secret key exchange, Alice and Bob should have never met, and exchange messages over the network such that they come away with a shared key K, but an eavesdropper cannot deduce it.

This can be done with block ciphers (due to Merkle in 1976, when he was still an undergrad at Berkeley). However, it's impractical, requiring about 100 terabytes for the exchange of a single key. Merkle puzzles are interesting, however, and theoretically elegant.

Another, more practical idea was Diffie-Hellman key exchange (unlike the previous method, this was invented at Stanford): fix a finite cyclic group G of order n.[17]. Then, fix a generator $g \in G$.

Then, Alice and Bob choose numbers $a, b \in G$; Alice sends $A = g^a$ to Bob, and Bob sends $B = g^b$ to Alice. Then, the secret key is $K = g^{ab} = B^a = A^b$. Both parties can thus communicate, but can an attacker compute g^{ab} given g, g^a, and g^b? (Note that today, people use more interesting methods such as elliptic curve cryptography.)

Definition 13.1. The Diffie-Hellman function is $\mathrm{DH}_g(g^a, g^b) = g^{ab}$.

This leads to the computational Diffie-Hellman assumption in the group G (CDA): for all efficient algorithms A, the probability $\Pr[A(g, X, Y) = \mathrm{DH}_g(X, Y)]$ is negligible when $X, Y \xleftarrow{R} G$.

Note that the group G is completely public. If the CDA is hard (e.g. the set of points on an elliptic curve or numbers modulo a prime), then the system is still secure.

For prime numbers, $G = \mathbb{Z}_p^*$, and the protocol is instantiated with $A = g^a \bmod p$, $B = b^b \bmod p$, and $K = g^{ab} \bmod p$. Then, $|G| = p - 1$ (which isn't prime).

A related problem to CDH is the discrete-log problem. Let $g \in G$ be a generator. Then, the discrete logarithm of an $x \in G$ is $\mathrm{Dlog}_g(x) = \min\{a \in G \mid x = g^a\}$. For example, if $G = \mathbb{Z}_7^*$, then, $\mathrm{Dlog}_3(2) = 2$, since $3^2 = 2 \bmod 7$, and $\mathrm{Dlog}_3(5) = 5$, because $3^5 = 243 = 5 \bmod 7$. It's been known for a long time that this is hard for large (about 300-digit) primes.

Lemma 13.2. *If* Dlog_g *is easy to compute, then* $\mathrm{DH}_g(X, Y)$ *is as well.*

Proof. $\mathrm{DH}_g(X, Y) = g^{\mathrm{Dlog}_g(Y)}$. ⊠

What we want is the converse (that if Diffie-Hellman can be computed efficiently, then the discrete log is as well), which is an open problem. Then, all that would be necessary is to find groups in which the discrete log is difficult.

In $\mathbb{Z}_p^*$, the discrete log can be calculated by a general number field sieve (GNFS), which has running time $O(e^{\sqrt[3]{\ln p}})$. This is subexponential time, which means that doubling the size of p doesn't make the algorithm take much more work. If p has 128 bits, then this is actually very easy to solve, so to be secure p should have as many as 2048 bits. This makes its running time 2^{128}, which is considered sufficiently hard (comparable to the security of the symmetric key earlier in the class). These large primes make the algorithm slow, so people tend to use things such as elliptic curves, where the best discrete log algorithm has time $O(\sqrt{p})$, which is exponential in the size of p. Thus, one should use a group based on a 256-bit prime, which makes computation faster (except for adversaries).

A different approach to key exchange uses asmmetric cryptography, in which the encryptor encrypts a message using a public key and the decryptor uses a secret key to decrypt.

[17] If you don't know what a cyclic group is, think of this as the set of numbers $\{0, \ldots, n-1\}$, where multiplication is done modulo n.

Definition 13.3. A public-key encryption scheme is a triple (G, E, D), where G outputs a public key pK and a secret key sK, and such that for all (pK, sK) output by G and for all $m \in \mathcal{M}$, $D(sK, E(pK, m)) = m$.

This can be used for key exchange: Alice generates $(pK, sK) \xleftarrow{R} G$ and sends pK to Bob. Then, Bob generates a $K \xleftarrow{R} \mathcal{K}$ and sends the ciphertext $c = E(pK, K)$. Then, Alice applies the decryption algorithm D to obtain the key. An attacker only has access to pK and c.

This can be proven to be secure if the underlying public-key system is semantically secure.

Definition 13.4. A public-key system (G, E, D) is semantically secure if for all efficient algorithms A, the probability

$$|\Pr[\text{EXP}(0) = 1] - \Pr[\text{EXP}(1) = 1]|$$

is negligible, where the adversary sends messages m_0 and m_1 and receives $c = E(pK, m_b)$. Since E is nondeterministic, it is nontrivial to determine the value of b.

Thus, the key-exchange system is secure, since the key K can be replaced with 0 without the attacker telling the difference; thus, the attacker cannot possibly obtain K with non-negligible probability. Note that this is not secure against active attacks (particularly a man-in-the-middle attack); the attacker can intercept the public key pK from Alice and send his own public key pK^* to Bob. Then, Bob encrypts something with pK^*, which the attacker can decrypt (and then reencrypt it with pK so that Alice doesn't notice anything is wrong).

Similarly, an active attacker can break Diffie-Hellman: it could receive g^a and g^b, and then send instead $g^{a'}$ and $g^{b'}$. Thus, Alice and Bob encrypt data with keys K_1 and K_2 that the attacker can read and translate back and forth before passing back on.

There are two famous encryption schemes for public-key encryption. The first is called Elgamal encryption, named after Taher Elgamal (who was Hellman's student, also here at Stanford). Let G be a finite cyclic group of order n, and fix a generator $g \in G$. Let $H : G^2 \to \{0,1\}^k$ be a hash function (e.g. SHA-1). Finally, let (E_s, D_s) be a semantically secure symmetric encryption system.

The algorithm G chooses an $\alpha \xleftarrow{R} \mathbb{Z}_n$ and let $h = g^\alpha$. Then, the public key is (g, h) and the secret key is α. Finding the secret key from the public key is equivalent to solving the discrete log problem. Then, $E(pK, m)$ chooses some $\beta \xleftarrow{R} \mathbb{Z}_n$ and let $u = g^\beta$ and $v = h^\beta = g^{\alpha\beta}$.[18] Then, the key is $K = H(u, v)$ and the ciphertext is $c = E_s(K, m)$.

Then, $D(sK = \alpha, (u, c))$. The decryptor can calculate $K = H(u, u^\alpha)$ and output $D_s(K, c)$.

Theorem 13.5. *Elgamal is semantically secure assuming H is a random function, CDH is hard in G, and (E_s, D_s) is semantically secure.*

Sometimes, this method of key exchange is called non-interactive, since people can post their public keys on some bulletin board and then Alice and Bob just read each other's keys to obtain their shared secret key.

Can this be generalized to more than two-party encryption? For users A, B, C, and D, one user given a, g^b, g^c, and g^d cannot compute g^{abcd} without breaking Diffie-Hellman, so more ingenuity is required. Thus, Diffie-Hellman is a two-party key exchange. There is a complicated n-party key exchange, but it's not very elegant, and coming up with a better one is an open problem.

14. Section 3: Number Theory: 2/15/2013

Cryptography tends to be concerned with very large numbers (often 300 digits, or 1024 bits), since anything interesting with small numbers can be done by brute-force.

Definition 14.1. If p is a prime, then $\mathbb{Z}_p = \{0, 1, 2, \ldots, p-1\}$.

This is a group additively and $\mathbb{Z}_p \setminus 0$ is a group multiplicatively, so this is a finite field.

Theorem 14.2 (Fermat)**.** *For any $g \neq 0$, $g^{p-1} \equiv 1 \pmod{p}$, where p is any prime number.*

[18] Though u looks unnecessary, it is helpful in the security proof, so in fact we couldn't do it without u.

Example 14.3. Since 5 is prime, we can consider $\mathbb{Z}_5$. Then, $3^4 = 81 \equiv 1 \pmod 5$. ◁

Definition 14.4. The inverse of an $x \in \mathbb{Z}_p$, denoted x^{-1}, is an element of $\mathbb{Z}_p$ such that $(x)(x^{-1}) = 1$.

For example, $3^{-1} \bmod 5 = 2$, and $2^{-1} \bmod p = (p+1)/2$ if p is any odd prime, since $2(p+1)/2 = p+1 \equiv 1$. All elements of $\mathbb{Z}_p$ are invertible except 0, and there is a simple algorithm for computing x^{-1}: $x^{-1} = x^{p-2} \pmod p$, since $(x)(x^{p-2}) = x^{p-1} \equiv 1$.

The name $\mathbb{Z}_p^*$ is used to refer to the set of invertible elements in $\mathbb{Z}_p$, so $\mathbb{Z}_p^* = \mathbb{Z}_p \setminus 0$. In rings that aren't fields, this becomes a bit more complicated, however.

There is also a simple algorithm for solving linear equations over finite fields: if $ax = b \pmod p$, then $x = a^{-1}b \pmod p$. For quadratic equations $ax^2 + bx + c$, one can use the quadratic formula $x = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$, which is fine unless the square root doesn't exist (in which case the equation has no solutions).[19]

$\mathbb{Z}_p^*$ is also a cyclic group: there exists a $g \in \mathbb{Z}_p^*$ such that $\mathbb{Z}_p^* = \{1, g, g^2, \ldots, g^{p-2}\}$. Such a g is called a generator, and one writes $\mathbb{Z}_p^* = \langle g \rangle$. For example, $\mathbb{Z}_7^* = \langle 3 \rangle = \{1, 3, 3^2, \ldots, 3^5\} = \{1, 3, 2, 6, 4, 5\}$. Not every element is a generator; for example, $\langle 2 \rangle = \{1, 2, 4\} \neq \mathbb{Z}_7^*$, so 2 doesn't generate $\mathbb{Z}_7^*$.

Definition 14.5. The order of an element g is the smallest positive a such that $g^a = 1$.

The order of an element g in $\mathbb{Z}_p^*$ is sometimes denoted $\mathrm{ord}_p(g)$. Thus, in $\mathbb{Z}_7^*$, the order of 3 is 6, but the order of 2 is 3.

Theorem 14.6 (Lagrange)**.** *The order of any element in G divides the order of the group, so in particular* $\mathrm{ord}_p(g) \mid p-1$.

Thus, if the factorization of $p-1$ is known, then orders in $\mathbb{Z}_p^*$ can be computed efficently, so this should not be used as a hard problem.

The square root of an $x \in \mathbb{Z}_p$ is any $y \in \mathbb{Z}_p$ such that $y^2 = x$. There might be zero or two such square roots: $\sqrt{2} \bmod 7 = 3$, since $3^2 = 9 \equiv 2 \bmod 7$, but $\sqrt{3} \bmod 7$ doesn't exist.

Definition 14.7. In general, an element $x \in \mathbb{Z}_p^*$ is called a quadratic residue if it has a square root.

Thus, there is a solution to a quadratic equation iff the discriminant is a quadratic residue.

If $x = y^2 = z^2 \bmod p$, then $y^2 - z^2 = 0$, so $(y+z)(y-z) = 0$ and thus $y = \pm z$. This means there are either no square roots or two of them, since any two square roots must satisfy this relation.

Theorem 14.8 (Euler)**.** *$x \in \mathbb{Z}_p$ is a quadratic residue iff* $x^{(p-1)/2} \equiv 1 \pmod p$.

For example, $2^{(7-1)/2} = 2^3 = 8 \equiv 1 \bmod 7$, so 2 is a quadratic residue. However, $3^3 = 6 \equiv 1 \bmod 7$, so 3 is not a quadratic residue.

If $a = g^{(p-1)/2}$, then $a^2 = (g^{(p-1)/2})^2 = g^{p-1} = 1$, so $a = \pm 1$.

Definition 14.9. This a is called the Legendre symbol of x, defined formally as

$$\left(\frac{x}{p}\right) = x^{\frac{p-1}{2}} \bmod p = \begin{cases} 1 & \text{if } x \text{ is a quadratic residue in } \mathbb{Z}_p^*, \\ -1 & \text{if } x \text{ isn't a quadratic residue in } \mathbb{Z}_p^*, \text{ and} \\ 0 & x = 0. \end{cases}$$

This can be efficiently computed. In particular, if $\mathbb{Z}_p^* = \langle g \rangle$, then all of the even powers of g are residues, and all of the odd powers are non-residues, so it is clear whether any element is an even power or an odd power of the generator.

Square roots can be computed efficiently: if $p \equiv 3 \pmod 4$, then let $a = x^{(p+1)/4}$, so that $a^2 = x^{(p+1)/2} = (x)(x^{(p-1)/2} = x$, assuming x is a quadratic residue. The modular condition on p is needed so that the power is an integer. If $p \equiv 1 \bmod 4$, there is an efficient randomized algorithm for computing square roots, which is a bit more complicated.[20] Thus, the square root and such aren't difficult problems, so one would need to work in rings which aren't fields to find cryptographically difficult problems.

[19] Of course, this doesn't work in $\mathbb{F}_2$, since you would end up dividing by zero, but 2 is of no use to any cryptographer, and it is possible to assume that all primes are odd.

[20] In fact, there is a randomized algorithm for solving any d^{th}-degree equation.

If p is a large prime, it doesn't fit in a register (since $1024 \gg 64$), so the representation of p is stored in a continuous array of bits. Then:

- Addition takes linear time in the length n of p, or $O(\log p)$.
- Multiplication can be done in quadratic time in the obvious way, but there are more clever algorithms: Koratsuba gives an algorithm in $O(n^{\lg 3}) \approx O(n^{1.6})$; fast Fourier transforms yield a similarly fast solution.
- Inversion takes $O(n^2)$ time, which can be seen from the computation of inverses given above.
- For exponentiation, finding $x^r \pmod p$, the brute-force solution is *very* bad. One can do it in time $\log r O(n^2)$, though, using repeated squaring until something repeats.

All of these were easy; what are some hard problems? Let g generate $\mathbb{Z}_p^*$.

- The discrete logarithm is believed to be hard: given an $x \in \mathbb{Z}_p$, find an r such that $x = g^r \bmod p$.
- The Diffie-Hellman problem: given an $x, y \in \mathbb{Z}_p^*$, such that $x = g^{r_1}$ and $y = g^{r_2}$, find $z = g^{r_1 r_2}$ (such that r_1 and r_2 aren't known as any precondition). This is also believed to be hard, and if discrete-log is easy, then this problem follows as well.
- Finding roots of sparse polynomials is also belived to be hard. An example of a sparse polynomial is $x^{(2^{500})} + 7x^{(2^{301})} + 11x^{(2^{157})} + x + 17 = 0 \bmod p$.

If N is composite, then $\mathbb{Z}_N$ is a ring that is not a field. Then, $\mathbb{Z}_N^*$, the multiplicative group of $\mathbb{Z}_N$, is not the same thing as $\mathbb{Z}_n \setminus 0$: for example, $2 \notin \mathbb{Z}_6^*$. The number of invertible elements is $|\mathbb{Z}_N^*| = \varphi(N)$. This is easy to compute if the factorization of N is known, but is believed to be hard otherwise; many results in cryptography will depend on something such as this, in which one party knows a factorization but the adversary doesn't.

Square roots seem to be hard to find in composite rings (the Rabin problem): given an $m = x^2 \pmod N$, how does one find x? Similarly, the RSA problem is finding x given $f_{\text{RSA}} = x^e \bmod N$. Both of these are easy if the factorization of N is known.

15. Trapdoor and One-Way Functions: 2/20/2013

Let's begin with a quick review of the Diffie-Hellman and Elgamal protocols discussed last week.

For Diffie-Hellman, let G be a cyclic group of (prime) order q, and g be a generator of G. Then, Alice computes $a \xleftarrow{R} \{0, \dots, q-1\}$ and Bob computes $b \xleftarrow{R} \{0, \dots, q-1\}$; Alice sends g^a to Bob, and he sends g^b to Alice, and the secret key is g^{ab}, which is hard to calculate without one of a or b (as far as we know). Note that the attacker cannot compute g^{ab} from g^a, g^b, and g, but the computational Diffie-Hellman assumption doesn't means that the attackercan't compute any of g^{ab}. Perhaps an attacker can compute a fraction of it.

Thus, one also has the hash Diffie-Hellman assumption (HDH): let $H : G^2 \to \{0,1\}^n$ be a hash function (e.g. SHA-256). The assumption is that for all efficient algorithms A, the advantage

$$\left|\Pr[A(g, g^a, g^b, H(g^a, g^b)) = 1] - \Pr[A(g, g^a, g^b, R) = 1]\right|$$

is negligible, where $a, b \xleftarrow{R} \{0, \dots, q-1\}$ and $R \xleftarrow{R} \{0,1\}^n$.

If R is completely random, the adversary can't determine anything about R, and if the HDH assumption holds, then the adversary can't predict any bits about the hash of g^{ab}; thus, the hash of g^{ab} looks like a random string and can be used as a key in a symmetric encryption system.

There is a stronger assumption, called the decision Diffie-Hellman assumption, which states that g^{ab} itself has enough entropy to be a key, but that's slightly beyond this scope.

So far, the best way to approach this problem is the discrete log, for which a good algorithm might exist.

Elgamal encryption takes some group G with a generator g, and chooses a random $x \in G$. Then, the secret key is x and the public key is g^x. Then, $E(pk, m)$ involves computing $c = (g^r, c_1 = m \oplus H(g^{xr}, g^r))$ and $D(x, (c_0, c_1)) = H(c_0^x, c_0) \oplus c_1 = m$.

Theorem 15.1 (Semantic Security)**.** *If (G, H) satisfy the HDH assumption, then Elgamal is semantically secure.*

In practice, such a system uses hybrid encryption, combining a public-key system (G, E, D) with a symmetric system (E_s, D_s) (e.g. AES-GCM). Then, to encrypt, let $K \xleftarrow{R} \mathcal{K}$. Then, $c_1 = E_s(K, m)$ and $c_0 = E(pK, K)$, so that the ciphertext is $c = (c_0, c_1)$. Thus, the data is encrypted with a symmetric cipher (which is fast) but the header (for key management) is done with Elgamal, which would take much longer for much larger messages.

Definition 15.2. A trapdoor function (TDF) $X \to Y$ is a triple of efficient algorithms (G, F, F^{-1}) such that:

- G is a randomized algorithms that outputs a pair of a public key and a secret key (pk, sk).
- $F(pk, \cdot)$ is a deterministic algorithm that gives a function $X \to Y$.
- $F^{-1}(sk, \cdot)$ defines a function $Y \to X$ that inverts F.

The goal is for F to be a "one-way function:" it can be evaluated, but computing F^{-1} is hard without the secret key. To be precise, consider a security game in which the challenger chooses a pk and an sk from G and chooses an $x \xleftarrow{R} X$. The adversary's goal is to output an x' given the ciphertext that is a guess of x.

Definition 15.3. (G, F, F^{-1}) is a secure TDF if for all efficient A, $\mathrm{Adv}_{\mathrm{OW}[A,F]} = \Pr[x = x']$ is negligible.

Example 15.4. One example of a one-way function is $F_{\mathrm{AES}}(x) = \mathrm{AES}(x, 0)$. This is a one-way function, because if not, then given $(0, \mathrm{AES}(K, 0))$ one could obtain K, which is bad. However, this is not a trapdoor function, because there's no easy way to invert this given some secret value.[21]

Hash functions can be considered one-way, but they aren't trapdoor functions. ◁

A TDF induces a public-key encryption system: let (G, F, F^{-1}) be a secure TDF $X \to Y$ and (E_s, D_s) be a symmetric authenticated encryption system defined over $(\mathcal{K}, \mathcal{M}, \mathcal{C})$ and let $H : X \to \mathcal{K}$ be a hash function. Then, to encrypt with pk, let $x \xleftarrow{R} X$, $Y = F(pk, x)$, $k = H(x)$, and $c = E_S(k, m)$; then, the ciphertext is (y, c). Decryption is straightforward: use F^{-1} to get x, run the hash, and get the key.

This is an ISO standard and a fine way of doing things. Like the previous hybrid encryption, most of the message is via the symmetric system.

Theorem 15.5. *If (G, F, F^{-1}) is a secure TDF, (E_s, D_s) provides authenticated encryption, and H is a random oracle, then this encryption method (G, E, D) is chosen-ciphertext (CCA^{ro}) secure.*

There is a flaw in many crypto textbooks and in practice: *never* encrypt by applying F directly to the plaintext. This is deterministic, which we know to be insecure.

An example requires a quick digression into arithmetic modulo composites: let $N = pq$, where p and q are independent large primes. Then, $\mathbb{Z}_N = \{0, 1, 2, \ldots, N-1\}$ and $\mathbb{Z}_N^*$ represents the invertible elements in $\mathbb{Z}_N$ as before. $x \in \mathbb{Z}_N$ is invertible iff $\gcd(x, N) = 1$, so the number of invertible elements in $\mathbb{Z}_N^*$ is $\varphi(N) = (p-1)(q-1) = N - p - q + 1$ (since if p is prime, then $\varphi(p) = p - 1$). Note that for large N, almost all elements are invertible, and a random one can be considered to be.

Theorem 15.6 (Euler)**.** *For any $x \in \mathbb{Z}_N^*$, $x^{\varphi(N)} = 1$.*

RSA is a trapdoor permutation (i.e. a trapdoor function in which the domain and range are the same). Note that after 40 years of research, RSA is still the only trapdoor permutation we have. Keep in mind that it is not an emcryption scheme!

- For G, choose two random, 1024-bit primes p and q, and let $N = pq$. Then, choose $e, d \in \mathbb{Z}$ such that $ed \equiv 1 \pmod{\varphi(n)}$. Then, output as the public key $pk = (N, e)$ and for the secret key $sk = (N, d)$.
- $F(pk, x) : \mathbb{Z}_N^* \to \mathbb{Z}_N^*$ is given by $\mathrm{RSA}(x) = x^e$ (in $\mathbb{Z}_N$).
- $F^{-1}(sk, y) = y^d$. Then, if $y = \mathrm{RSA}(x)$, then

$$y^d = \mathrm{RSA}(x) = x^{ed} = x^{k\varphi(N)+1} = (x^{\varphi(N)})^k \cdot x = x.$$

[21] This is yet another open problem. AES could be a trapdoor function. Building trapdoor functions is very hard; only two examples are really known.

The RSA assumption is that RSA is a one-way permutation: for all efficient algorithms A, the probability $\Pr[A(N,e,y)] = y^{1/e}$ is negligible, where p and q are randomly chosen n-bit primes, $N = pq$, and $y \xleftarrow{R} \mathbb{Z}_N^*$. Like all of the other things in class, we don't know if it's true, but it seems reasonable.

RSA leads to an encryption system as indicated above, but with $\mathbb{Z}_N$ playing the role of G. However, the "textbook" implementation of RSA is insecure, in which encryption is $c = m^e \bmod N$ and decryption is $m = c^d$. This leads to a very simple attack: suppose the key k is 64 bits, and Eve sees $c = k^e \in \mathbb{Z}_N$. If $k = k_1 k_2$ with k_1, k_2 on the order of 2^{34} (which is true about 20% of the time), then $c/k_1^e = k_2^e$ in $\mathbb{Z}_N$. This is exactly the settings of a meet-in-the-middle attack: Eve builds a table of all $c/1^e, \dots, c/2^{34e}$ in time 2^{34}, and then for $k_2 = 0, \dots, 2^{34}$ test if k_2^e is in the table. This has total time of $2^{40} \ll 2^{64}$, which breaks much faster than anticipated. Use the ISO standard!

. . . well, in practice, textbook RSA is used after some preprocessing. For example, PKCS1 takes a message m (e.g. the key) and then append two bits of `00`. Then, a random pad is appended (that doesn't contains zero bits), and then two bits `02`, which gives an input of the right size. This is used by TLS, HTTPS, etc. Decrpytion is straightforward: remove things until the zero bits are found.

This is completely insecure, since if there isn't a `02`, an abort message is sent. This allows the Bleichenbacher attack: the attacker has a chosen-ciphertext attack by choosing an $r \in \mathbb{Z}_N$ and computing $c' = r^e c = (r \cdot \mathrm{PKCS1}(m))^e$ and gets the response of the server. This is useful because, for example, multiplying by 2^k is equivalent to a bit shift, so all of the bits are revealed.

This is a problem, and an *ad hoc* fix was implemented. If the `02` is not present, then generate a string R of 46 random bytes, set it as the plaintext, and continue. This fake plaintext is going to fail at some point, but not in a way that an attacker can see. This appears to work, even today.

The second version, OAEP, is a different preprocessing function, which runs the message and padding through two hashes H and G with randomness. The plaintext is $H((m \,\|\, p) \oplus r) \,\|\, G((m \,\|\, p) \oplus r)$.

Theorem 15.7. *If RSA is a trapdoor permutation, then RSA-OAEP is secure when H and G are random oracles.*

In practice, of course, nobody really uses this, because standards are hard to change. But you could use SHA-256, etc. There are several other ways of implementing a secure pad (including one discovered by the professor). There are subtleties to this, as some implementations of OAEP signal an error if the pad looks wrong. This leads to a standard timing attack. Oops. It's very difficult to do this correctly, especially because optimizing compilers sometimes undo one's work. This is an excellent reason to use crypto libraries; they might be slow, but they are secure.

16. More RSA and Public-Key Signatures: 2/25/2013

As a recap, recall that a public-key encryption scheme has a public key pk and a secret key sk such that the encryption of a message depends only on the public key, but decryption depends on the secret key.

One example of this was based on Diffie-Hellman, in which $pk = (g, g^x)$ and $sk = x$. Then, $E(pk, m) = (g^r, H(g^r, g^{xr}))$. This is a very common method of cryptography, though it tends be used with elliptic curves. Alternatively, one can use a trapdoor function F such that anybody can calulate $F(pk, \cdot) : X \to X$, but calculating F^{-1} requires knowing the secret key. The only known example of a TDP is RSA: one takes $N = pq$ and chooses e and d such that $ed = 1 \bmod \varphi(N)$. Then, $pk = (N, e)$, and $sk = (N, d)$, so $F(pk, x) = x^e \bmod N$ and $F^{-1}(sk, y) = y^d \bmod N$.

The first way of doing RSA is to let p and q be very large primes; for example,

$$p = 5 \text{ and } q = 11.$$

Then, $\varphi(N) = (p-1)(q-1) = 40$. If $e = 3$, then $d = 3^{-1} \bmod 40 = 27$. This doesn't always trigger the modular property, so using something like $e = 65537$ is a little more common. Then, $F(pk, q) = q^3 \bmod 55 = 14$, and $F^{-1}(sk, 14) = 14^{27} \bmod 55 = 9$.

RSA is assumed to be a one-way function, and computing e^{th} roots modulo N is so hard that the best algorithms factor N as the first step! Attempting to find an algorithm that finds these roots without factoring N is an open problem, and one of the oldest in cryptography.

Exercise 16.1. Computing square roots is equivalent to factoring N; show that if an algorithm generates square roots, then it can be used to efficiently factor N.

One way to make RSA run faster is to make d small (such as $d \approx 2^{128}$). However, it was shown that if $d < N^{0.292}$, then RSA is completely insecure. This is too weird of a number for it to be the right answer, and the belief is that RSA is insecure if $d < \sqrt{N}$, but this is an open problem. Do not optimize RSA by choosing a small D!

Here is Wiener's attack (for $d < N^{0.25}$): since $ed = 1 \pmod{\varphi(N)}$, so there exists a $k \in \mathbb{Z}$ such that $ed = k\varphi(N) + 1$. Thus,

$$\left| \frac{e}{\varphi(N)} - \frac{k}{d} \right| = \frac{1}{d\varphi(N)} \leq \frac{1}{\sqrt{N}}.$$

k and d are unknown, but e is known, and we have a really good approximation of $\varphi(N) \approx N$ (since $|N - \varphi(N)| \leq p + q \leq 3\sqrt{N}$). If $d \leq N^{0.25}/3$, this becomes

$$\left| \frac{e}{N} - \frac{k}{d} \right| \leq \left| \frac{e}{N} - \frac{e}{\varphi(N)} \right| + \left| \frac{e}{\varphi(N)} - \frac{k}{d} \right| \leq \frac{1}{kd^2}.$$

Number-theoretically, one can create the continued-fraction expansion of e/N. There are few of these, so one of them is k/d. Then, since $\gcd(k, d) = 1$ (since $ed = 1 \bmod k$), then one can obtain d from k/d.

This algorithm is actually faster than the normal key generation, even if the factorization of N is known...

Thus, one might wish to optimize by making e small. The minimm possible value is $e = 3$ ($e = 2$ is bad because e must be relatively prime to $\varphi(N)$, which is even). The recommended value is $65537 = 2^{16} + 1$, which makes only 17 multiplications necessary. This is known as the asymmetry of RSA (by a factor of 10): encryption is very fast, but decryption is slow. Note that Elgamal doesn't have this problem; it is slow in both directions.

Thanks to subexponential factorins algorithms if the key length is 128 bits, the RSA modulus size must be 3072 bits... and for 256-bit keys, one needs 15000-bit moduli. This is one of the reasons for moving past RSA (and into elliptic curves).

RSA is very vulnerable to being implemented incorrectly:

- There is a timing attack (due to Kocher et al, 1997), in which the time taken to compute $c^d \pmod{N}$ can expose d. This can be fixed by making everything take the same time in the code, but this is hard to accomplish, particularly in the context of the frequently changing standards.[22] Another option is to add a random delay, but over the long term these random delays will converge to their mean, making the timing attack still possible.
- Kocher et al also showed that there is an attack based on power consumption, as the power consumption done by a smart card that computes $c^d \pmod{N}$ can expose d as well. This involves charging a capacitor to drive the decryption algorithm, which masks how much power is used. Accessing the smart card is as easy as breaking into a bus or wallet. This isn't just theoretical: some European credit cards use public-key encryption, and the credit card can be used in a power attack.[23]
- Finally, a faults attack exists; if a computer error occurs during the computation, d may be exposed. This is meaningful because faults can be introduced by bombarding it with radiation or such. A common defense against this is to check whether the output is correct, at the cost of a 10% slowdown. This was adapted into OpenSSL, which caused the Internet in general to slow down by 10% (and the professor was part of the reason).

[22] As has been said before, this is one reason someone could implement RSA faster than the standard — but this is guaranteed to be insecure.

[23] Of course, here in the USA, it's even easier, since the credit card number is all that is needed.

- It is difficult to generate keys, and the entropy issue was only discovered last summer. Generally, the algorithm seeds a PRG and then computes p, then adds some randomness and generates q. If there is poor entropy at the startup (e.g. factory settings), then the same p will be generated by different devices with different q. Then, if N_1 and N_2 are RSA keys for different devices, then $\gcd(N_1, N_2) = p$. Researchers collected millions of RSA keys and noticed that 0.4% of the keys factored, which is a large number of keys!
- Another vulnerability was a nondescript posting on the Ruby on Rails forum about a year ago in which a bug set $e = 1$. This works, since it's relatively prime to $\varphi(N)$. Thus, for three months, keys were created with this, and the certification authority had rejected these keys.

Definition 16.2. A one-way function is a function $F : X \to Y$ such that:

(1) There exists an efficient algorithm to compute F, and
(2) For all efficient algorithms A, the probability $\Pr[F(A(F(x))) = F(x)]$ is negligible.

Example 16.3. We have already seen many one-way functions:

(1) The general one-way function $F^E(x) = E(k, 0) \,\|\, E(k, 1)$, where E is a secure cipher.
(2) Suppose p is prime and $\mathbb{Z}_p = \langle g \rangle$. Then, $F^{\mathrm{Dlog}(x)} = g^x \in \mathbb{Z}_p$ is a one-way function.[24]
(3) RSA is a one-way function: $F^{\mathrm{RSA}}(x) = x^e \bmod N$. ◂

Message integrity in the public-key world is accomplished with signatures:

Definition 16.4. A signature scheme is a triple (G, S, V), where:

- G is a key generator that outputs sk (signing key) and vk (verification key),
- $S(sk, m) = \sigma$ generates a signature, and
- $V(pk, m, \sigma)$ returns `0` or `1` indicating whether the message is valid, such that

for all $m \in \mathcal{M}$, if (sk, vk) are outputs of G, then $V(pk, m, S(sk, m)) = 1$ (i.e. yes).

Security requires defending against a chosen plaintext attack, in which the attacker's goal is an existential forgery. Formally, the challenger can make queries $m_1, \ldots, m_q$ and receives valid signatures corresponding to these messages. Then, the attacker wins if it can produce a $m \notin \{m_1, \ldots, m_1\}$ and a signature σ such that $V(pk, m, \sigma)$ returns 'yes.'

This should look very similar to the symmetric case.

Definition 16.5. (G, S, V) is a secure signature if for all efficient algorithms A, the probability that A wins this game is negligible.

It turns out to not be too difficult to construct one of these, using the hash-and-sign paradigm. If (G, S, V) is a signature scheme for short messages, then one can construct a signature scheme for longer messages (G', S', V') given by $S'(sk, m) = S(sk, H(m))$, where $H : \mathcal{M} \to M$ is a collision-resistant hash function (and M is the message space of (G, S, V)). It is possible to generate these short signatures from a generic one-way function, and in particular discrete-log (e.g. DSS, the digital signature standard).

One can also make a one-way function from RSA, which is in fact very simple. Here, the public key includes $NN = pq$ and a hash $H : \mathcal{M} \to \mathbb{Z}_N$, Then, $sk = d$ and $vk = e$. This gives the functions $S(sk, m) : \sigma = F^{-1}(sk, H(m)) = H(m)^d \bmod N$ and $V(pk, m, \sigma)$ returns whether $H(m)$ equals $F(k, \sigma)$, or whether $H(m) = \sigma^e \bmod N$. The fact that e is small means that verification time is about 10 times faster than signing time, which is nice for a signature scheme (in which one signature is verified by many people).

17. More Digital Signatures: 2/27/2013

As a quick review, a digital signature scheme is a triple (G, S, V), where G generates a public key pk and a secret key sk, $S(sk, m)$ generates a signature σ, and $V(pk, m, \sigma)$ returns whether a signature is valid. Such a scheme is secure if it is unforgeable under a chosen message attack.

There are several ways to construct a signature scheme:

[24] This depends on $F(x+y) = F(x)F(y)$; given a plainext a and $F(x)$, one can compute $F(ax)$, which ,makes the Diffie-Hellman protocol possible.

(1) A generic one-way function can be used to obtain a signature, as will be shown later in this lecture. For example, one might have $F(x) = \text{AES}(x, 0)$.
(2) Discrete-log signatures are built from functions such as $F(x) = g^x \pmod p$ (e.g. DSS).
(3) Signatures can be obtained from trapdoor functions such as RSA. This is known as a full domain hash: G is the same as for the trapdoor permutation, and $pk = (N, e)$ and $sk = (N, d)$. Then, $S(sk, m) = F^{-1}(sk, H(m)) = H(m)^d \bmod N$, where H is a hash $\{0,1\}^* \to \mathbb{Z}_N$.
Then, $V(pk, m, \sigma)$ tests if $F(pk, \sigma) = H(m)$.

Theorem 17.1 (BR94[25])**.** *If $H : \{0,1\}^* \to \mathbb{Z}_N$ is a random oracle (which is to say that it behaves like a random function, so that the adversary cannot implement it on its own), then RSA-FDH is secure assuming RSA is a one-way function.*

The proof will seem kind of science-fictionish, and will illustrate why this class doesn't contain all that many proofs.

Proof of Theorem 17.1. Assume this scheme isn't secure, so that there is an efficient forger $\mathcal{A}$. Then, it will be shown that $\mathcal{A}$ can be used to break RSA: given some input N, e, and y, it will be possible to compute $y^{1/e} \pmod n$.

The attacker is given the verification key $vk = (N, e)$ amd can make signature queries of $m_i \in \mathcal{M}$ to obtain their signatures, and hash queries of x_i to obtain $H(x_i)$. Let q be the number of signature queries that $\mathcal{A}$ issues, and let the hash query of x_i be given by the following:

- Flip a weighted coin with $\Pr[0] = 1/q$.
- If the result of the toss is 0, let $H(x_i) = y \cdot r_i^3 \pmod N$, where $r_i \xleftarrow{R} \mathbb{Z}_N$. This is a random value in $\mathbb{Z}_N$.
- If the result of the toss is 1, then $H(x_i) = r_i^e \pmod N$, where r_i is as before.

For the signature query: if $H(m_i)$ has been queried, return the signature corresponding to m_i.

- In case 0, $H(m_i) = y \cdot r_i^e$. Then, we don't know how to compute $y^{1/e}$, so abort the whole algorithm. However, this is unlikely to happen (probability $1/q$).
- In case 1, where $H(m_i) = r_i^e$, respond with $\sigma(m_i) = r_i$.

Then, the adversary produces a forgery (m^*, σ^*) such that $\sigma^* = H(m^*)^{1/e}$. In case 0, $H(m^*) = y \cdot r_i^e$, so $\sigma^* = y^{1/e} r_i$, so our algorithm outputs $\sigma^*/r_i \bmod N$. In case 1, he fails to break RSA. The algorithm cannot return an m^* that hasn't been queried before, because $H(m^*)$ is random and something it hasn't seen before. Thus, this isn't going to happen.

Then, the probability of success is equal to the probability that there is no failure, which is the probability that all q queries are in case 1, but the forgery is in case 0. This is $\left(1 - \frac{1}{q}\right)^q \times \frac{1}{q} \approx \frac{1}{qe}$, which is non-negligible.[26] Thus, this algorithm breaks RSA with non-negligible probability. ⊠

Philosophically, people debate how useful things such as random oracles are, but this proof can be adapted to real-world hashes (which is more useful in practice).

The hash is very necessary: suppose $\sigma = m^{1/d} \pmod N$. Then, there are two attacks:

(1) One existential forgery is the message-signature pair $(1, 1)$, or in more generality choose a signature σ and return (σ^e, σ) (i.e. the message is σ^e).
(2) However, existential forgeries are a theoretician's concern, and there is a much worse forgery that might actually be practical, called a blinding attack:
   (a) The adversary chooses an $r \xleftarrow{R} \mathbb{Z}_N$, called the blinding value.
   (b) It sets $m' = m \cdot r^e \pmod N$, where m is the message the attacker wants to forge for. m' looks like random garbage, so in theory it's fine to sign it.
   (c) The attacker queries m', getting $\sigma' = (m')^{1/e} \pmod N$.

[25] Interestingly, in mathematics, "classical" means 300 years old. In computer science, theorems such as this one can be considered classical!

[26] This seems kind of sketchy, but it can be made more formal: if $F : \mathbb{N} \to \mathbb{R}$, then F is polynomial if $F(x) < x^d$ for sufficiently large x, or $F(x) = O(x^d)$. F is negligible if $F(x) < 1/x^d$ for some d and sufficiently large x.

(d) The attacker outputs $\sigma = \sigma'/r \pmod N$.

Claim 17.2. Then, σ is a valid signature on m.

Proof.

$$\sigma^e = \left(\frac{\sigma'}{r}\right)^e = \left(\frac{(m')^{1/e}}{r}\right)^e = \frac{m'}{r^e} = \frac{m \cdot r^e}{r^e} = m \pmod N. \qquad \boxtimes$$

This latter concept is crucial to things called blind signature systems, which sign things without knowing much information about them. This is the basis of anonymous electronic cash, etc.

In practice, RSA is implemented as a partial domain hash, or PKCS1 (mode 1) signatures. This is implemented as

(1) $D = h(m)$, where h is just SHA-256, so $m \in \{0,1\}^{256}$.
(2) Then, let $EB = [\texttt{00 01 11 11 11} \cdots \texttt{11 11 00} \,\|\, D]$, and
(3) $\sigma = EB^{1/e} \bmod N$.

Proving this scheme is secure is also an open problem, if you happen to be suffering from insomnia. The best proof so far is when the size of the domain is $N^{1/3}$, as shown in 2002. Why hasn't anyone made progress in the intervening decade? Because this stuff is hard.

Now, suppose $h : \{0,1\}^n \to \{0,1\}^n$ is a generic one-way function. Lamport's one-time scheme is a signature scheme as long as each key is single-use. The key generator function generates elements ($2n$ random values) $x_1^{(0)}, \ldots, x_n^{(0)}, x_1^{(1)}, \ldots, x_n^{(1)}$, which becomes the secret key, and the public key is their hashes $h(x_1^{(0)}), \ldots, h(x_n^{(0)}), h(x_1^{(1)}), \ldots, h(x_n^{(1)})$. Then, if $m \in \{0,1\}^n$ has bit representation $m_1 \ldots m_n$, then $S(sk, m) = \sigma = (x_1^{(m_1)}, \ldots, x_1^{(m_n)})$. Since h is one-way, then given m and $S(sk, m)$, it isn't possible to produce a signature for an $m' \neq m$, since at least one of their bits differs, so it isn't possible to create the new message.

Of course, if the same key is used for multiple signatures, bad things happen. But this has one-time chosen-message security.

18. Section 4: The Chinese Remainder Theorem and SSL: 3/1/2013

Suppose (as a motivating example) we know $x \equiv 3 \bmod 5$ and $x \equiv 2 \bmod 7$. Then, $x = 3 + 5k$, so $3 + 5k \equiv 2 \bmod 7$, so $k \equiv 4 \bmod 7$. Thus, $k = 4 + 7\ell$ and thus $x = 23 + 35\ell$, or $x \equiv 23 \bmod 35$.

This is a special case of the Chinese Remainder Theorem: if $r \equiv x \bmod p$ and $s \equiv x \bmod q$, then $t \equiv x \bmod pq$ is given by $t = r(q^{-1} \bmod p)q + s(p^{-1} \bmod q)p$. The other direction also works, but is much easier to see.

This is important because of RSA encryption: suppose $N = pq$; then, to compute $x^d \bmod N$, one can let $d_0 = d \bmod (p-1)$ and $d_1 = d \bmod (q-1)$. Then, $x^{d_0} \equiv x^d \bmod p$, and $x^{d_1} \bmod q = x^d \bmod q$. Thus, the Chinese Remainder Theorem gives $x^d \bmod N$. But more significantly, this is significantly faster than the naive repeated-squaring algorithm, on the order of $O((\log N)^3)$, which is a fourfold increase in performance. Thus, RSA in practice always implements this.

Turning to the project, the TA demonstrated how one might run the project, but wasn't particularly illuminating.

The project requires forging certificates for a man-in-the-middle attack, but also implementing the administration client, which allows dumping some statistics about the running of the server. This requires authenticating the client. Oddly enough, it looks like the password is given on the command line as
`java mitm.MITMAdminClient -password foobar -cmd stats`, which seems pretty insecure.

19. One-time Signatures: 3/4/2013

One-time signatures were discussed in the previous lecture, defined to be secure if each key is used only once. The security game gives a public key pk to an adversary, which is allowed to make exactly one query m before producing a message-signature pair (m^*, σ^*), with $m^* \neq m$.

One application of such a signature scheme is signing streamed data, where a sender sends lots of packets to lots of different receivers. Using a many-time signature scheme to sign every packet (such as RSA) is computationally intensive. Thus, one can instead use a one-time scheme: packet p_1 is received by the signer, and a key pair pk_1, sk_1 is generated. $[p_1 \,\|\, pk_1]$ is signed with RSA and the sender sends $[p_1 \,\|\, pk_1 \,\|\, \sigma]$ (where σ is the signature). Then, the next message p_2 is signed with sk_1 (or technically $[p_2 \,\|\, pk_2]$ is signed), and the message is sent with the signature. Thus, a chain of packets is created, with each packet p_i signed with sk_{i-1}, and the first one signed with RSA. This is nice because both verification and signing are really fast (except for the head of the chain), and this scheme only uses each key once, so it is secure.

Signatures are used instead of MACs because it eliminates a forgery attack on one-to-many MACs; see one of the previous homeworks.

A one-time signature scheme can be used to make a many-time signature scheme using a collision-resistant hash function (sometimes called Merkel signatures). Merkel implemented this using a tree: using the key generation function G, generate and publish a public key pk^* and use it to sign a message containing two more public keys pk_1 and pk_2. Then, pk_1 is used to sign two more public keys pk_3 and pk_4, etc.

Then, if $pk_{i_1}, \ldots, pk_{i_n}$ is the path from the root to the leaf pk, the signature is $\sigma(pk) \,\|\, \sigma(pk_{i_1}), \ldots, \sigma(pk_{i_n})$. This is very fast, but has losts of storage (which can be simplified somewhat to a very small amount of memory) and the signatures are relatively long. However, this allows a huge number of distinct signatures.

One of the most important applications is for certificates, which are used for public key management, allowing, for example, Alice to obtain the public key for Bob, There is a single-domain certificate authority (CA), which verifies that Alice is who she claims to be and then sends her Bob's public key.

For example, in the protocol x509v3, the certificate contains: an issuer ID contains lots of information, such as name, country, etc.; the subject ID, or whom it was issued to; a public key; and various control fields (e.g. expiration dates). This whole certificate is signed with the CA's secret key, and the signature is embedded in the certificate.

Alice obtains the certificate offline, and since certificates expire annually, this is more secure as well as a good business model. Additionally, notice tht Bob must have the CA's public key (which is usually shipped with the browser). This means that if the secret key is leaked, then the only way to fix it is through a software update — but if you sign your software updates with the same certificate authority, bad things will happen; they should be entirely independent signature chains.

Certificate generation can be accomplished in the following way:

(1) Alice generates a pair pk, sk, and keeps sk to herself.
(2) Alice somehow has to convince a CA that she is Alice.[27] There are several ways this is done (e.g. corporate letters, manual review, or sometimes just proving you own an email address). This validation does come with a cost. She also has to prove that she knows the secret key (so that it's not possible to get certificates for random strangers' public keys). This can't cause that much damage, but it's still good to be sure. Then, Alice generates the CSR (certificate signing request).
(3) Alice sends the CSR to the CA (with some money), and the certificate signs it.

It is important that the CA doesn't know Alice's secret key: consider Gmail and a CA (DigiNotar or TurkTrust) that acts as a gateway could mount a man-in-the-middle attack if it knew the secret key of Gmail. However, Chrome contains something called certificate pinning, in which the Gmail public key was hardwired into the browser. Chrome noticed that a Gmail certificate was issued with the wrong public key (which is BAD) and lots of things happened (DigiNotar went out of business and had its authority revoked). The world responded with software patches that didn't trust DigiNotar certificates, but some old phones don't have the necessary upates.

[27] There is a very obvious problem with convincing someone you're someone else. For example, people have claimed to be Gmail with surprising degrees of success three times in the past two years.

Outside of Chrome's foresight, discussing the ways to deal with this sort of thing is an active field, and four main proposals have been advanced.

Without public-key crypto, one would have to use a key distribution center, which is fast, but if it is compromised all past and future keys are compromised. This is at least somewhat worse than the CA issues, which are offline. In particular, VeriSign keys are mostly offline, and signing is accomplished via intermediate keys in a certificate heirarchy (which is important; if VeriSign failed, the Internet might stop working).

Another method is called cross-certification, in which there are no root CAs (unlike certification heirarchies), but each CA certifies each other CA. This is fine, but if there are a lot of CAs they all have to verify each other, and the complete graph is kind of crazy.

A third solution is called the web of trust, in which a reduced graph is used: lots of nodes, some of which are connected. If $C_1, \ldots, C_n$ is the path from Alice to Bob, then the certificate is the certification of Alice with C_1, then C_1 with C_2, etc. If you want multiple such certificates, it actually becomes an interesting graph algorithm, to find two node-disjoint paths from one node to another.

One of the hardest problems in certificates is that of certificate revocation: once Alice is issued a certificate, how should it be revoked? This should be done if Alice's secret key is stolen, or possibly if Alice herself is not to be trusted. There are three answers:

(1) Using the certificate expiration field, certificates decay after a year or so. This isn't a good answer, since things might be compromised for the intervening time.
(2) Alternatively, one could use a certificate revocation list (CRL), in which every certificate contains a URL to a certificate revocation list which is signed by the CA (for several reasons which should be clear). Then, every day or so, all certificates that are revoked are published at this site, and if a certificate is revoked, then it isn't trusted. However, this requires downloading lots and lots of information in order to check one small item.
(3) OCSP: Online Certificate Status Protocol. If Alice received a certificate from Bob, then she sends the CA the serial number corresponding to this certificate, and the CA returns a yes/no answer signed by the CA as to whether or not it can be trusted. This has lots of problems: in particular, this is online, which is problematic. Also, OCSB responses are cacheable, so they live for about a week. Thus, if a key is stolen, you might not know for several days, unless you go online. This is a choice that can't really be avoided.

    Additionally, this implies that the CA knows everywhere you go. This is... not ideal. OCSB stapling, which is not very widely implemented, is a proposed solution to the tracking problem, and most websites that use OCSB inherently expose their clients to tracking anyways.

    Finally, what if the OSCB responder is offline? What should the browser do in this case? Sometimes, this can't be avoided, such as paying for Internet access with a credit card (and failure is bad). But of course this makes security more difficult too.

20. SSL, TLS, and More About PKIs: 3/6/2013

SSL[28] is a very well-known and very successful security protocol. Though its earlier iterations were extremely insecure, it went through standardization and is now safer. SSL is transparent (so it isn't extremely burdensome to the end user), and is used in a lot of situations (the Internet, some emails, etc.). It is designed to be robust against active attacks, and to run in two modes: one-sided authentication, in which the browser knows who the server is, but not vice versa; and mutual authentication, where each knows the other. The latter protocol isn't used very often.

Here is an excellent way *not* to do things:

(1) Use Diffie-Hellman or some other secure key exchange mechanism to create a shared key.
(2) Then, using this key, the browser sends its identity to the server and the server sends its identity to the browser (e.g. password-based identification).

[28] SSL, TLS, and HTTPS are essentially different words for the same thing. SSL will be used for consistency in this lecture.

This can be broken (of course) with a man-in-the-middle attack. The adversary sets up two key exchanges, one with the browser and one with the server, and then the adversary can retrieve passwords, hijack the session, etc. Since anonymous key exchange with authentication isn't secure, how might authenticated key exchange be accomplished?

SSL operates as follows:

(1) The browser sends a client-hello message, which contains a session-ID, a random 28-byte value (client randomness) c_r, and the cipher-specs, a list of cryptographic algorithms that the browser supports.
(2) The server sends a server-hello message, containing (amidst other things) a session-ID, server randomness (28 bits again) s_r, and the specific cipher-spec to use.
(3) Key exchange is performed, yielding something called the pre-master secret (unfortunately also sometimes called PMS).
(4) Then, a message key $mk = [K_{b\to s}, K_{s\to b}]$ is generated as $mk = \mathrm{KDF}(\mathrm{PMS}, c_r, s_r)$.

Thus, the client and server have communicated and agreed on the key. The randomness is necessary to prevent replay attacks, since they would end up with the same master key each time otherwise.

The key exchange mechanism can be RSA, in which case the browser already has the public key (N, e) from the certificate. Then, the browser picks its client randomness, and encrypts it with RSA-PKCS1 to obtain a ciphertext c. The server obtains the PMS with the secret key. Then, the attacker only sees c and the public key. This is a fine approach, and is even pretty fast. However, it has no forward secrecy: if the server's secret key is leaked a month later, all past secrets are also compromised.

An alternative solution is a variant on Diffie-Hellman called EDH. The public key is (N, e), and when the server sends $(p, g \in \mathbb{Z}_p^*, z = g^a \pmod p, \sigma)$ (where a is a random element of $\mathbb{Z}_p^*$), σ is the signature using the RSA secret key. Then, the client verifies the signature, chooses a random $b \in \{1, \dots, p-1\}$, and computes $z_2 = g^b \pmod p$. Then, z_2 is anonymous, and the pre-master secret is $g^{ab} \bmod p$, which both sides can compute as in standard Diffie-Hellman. This has forward secrecy, and the RSA signature prevents a man-in-the-middle attack. However, the server has to perform a lot more work (an RSA signature is comparable to the RSA decryption), but it also has to do two more exponentiations (to compute z and the pre-master secret). Thus, it is about three times slower. Thus, in practice, many websites still use RSA, but as more websites move to elliptic-curve crypto, this difference is less important, since the exponentiations are much cheaper. In fact, many protocols use RSA for the signature and an elliptic curve group for Diffie-Hellman, and eventually the signatures will use elliptic curves as well.

One can use the session-ID to reduce the number of key exchanges: if the browser has connected with the server before, it sends a session-ID corresponding to that session, and if not, it sends 0. Then, the server looks to see if it has that ID in its cache; if so, it reuses that session key, and if not, the full key exchange is performed.

Here, a log from an SSL session was shown. Interestingly, ads are also sent over SSL, even though they don't really need to be secure, because the alternative is a mixed protocol error. This sometimes causes situations where an advertisement is encrypted more securely than the actual content!

In the real world, there are plenty of issues that TLS has to deal with:

(1) Lots of traffic is routed through proxies, which can cause things to become tangled. HTTPS was aware of this, and send a connect request to the proxy that deals with this problem.
(2) A virutal hosting site may host two different domains on one IP address if it runs out, which causes issues if the client-hello is based only on IP. Thus, virtual hosting inherently cannot use SSL, which is sad. One of the client-hello extensions for TLS solves this problem by including the hostname in the hello message. All modern browsers (i.e. everything that is not IE6) do this, but virtual hosting can't work because of the 5% of users that still use IE6.

SSL can be used to perform mutual authentication without a password, but this isn't used all that often in practice. Both the client and the server (the former of which is a problem) have a secret key and a certificate; then, after the hello messages, the server sends a certificate request, prompting the user to

select a client certificate.[29] Then, the client sends its certificate plus the signature on all handshake messages (and standard key exchange).

The client certificate and secret key can be bought from VeriSign (though if nobody's using them, why would you?); specifically, it is a Javascript object called `crypto.generateCRMFRequest()` that generates certificate requests to send to a CA and a `crypto.importUserCert()` that imports a certificate into the certificate store.

This protocol is mostly unused because nobody buys client certificates.

Returning to PKIs, there are lots of issues with certificate authorities:

(1) There are about sixty root CAs, which have generated certificates for about 1200 other intermediate CAs. If any of these are compromised, bad things happen, since browsers are programmed to trust them. Indeed, CAs such as Comodo, DigiNotar, and TurkTrust have been caught writing bad certificates.
(2) Additionally, there are man-in-the-middle attacks, as in the second programming assignment.

There are a number of proposed solutions to this, of varying degrees of quality:

(1) DANE places a site's public key in DNS, a distributed database. Instead of just linking a name to a webserver, there is another field for the public key, which is used to set up a secure session with the bank. This is actually being put into practice, but the DNS is unauthenticated, so there is no way to know that the key is correct or was transmitted correctly. This solution only makes sense in a secure version of DNS, called DNS-SEC, in which these records are signed by something which becomes a sort of master CA. Additionally, 5% of the Internet is behind old DS servers, so they can't work with this protocol.
(2) Certificate Authority Authorization (CAA): the DNS registry also contains an entry that indicates which CA is allowed to issue certificates for a given website. Thus, if another CA is asked to issue a certificate, it declines. There are lots of issues with this:
    - This doesn't actually solve the problem. DigiNotar was hacked so badly that this defense wouldn't have made a difference.
    - This is an all-or-nothing protocol, which makes it very difficult to implement; if just one CA ignores this protocol, that is the one that will be targeted.

    People still support it, since it can't hurt.
(3) Public-key pinning: Chrome has a built-in check that knows the public key of Google, and raised an alarm when someone served a certificate with a forged key. Several other websites have done the same, so if someone changes their public key, there has to be a software update. This is called static public-key pinning. This is not scalable, so two related proposals have been advanced:
    (a) TACK is a TLS extension.
    (b) Public-key pinning, which is part of the HTTP header. A public-key pin is added, which includes the expiration date of the certificate and the hashes of the certificate chain. On the next connection, the browser accepts the certificate chain if there is a nonzero intersection with the list of pins of that site from the previous connection.

    This is a good method, but first-use is a fairly obvious problem. This is particularly concerning given that some corporations issue their employees clean laptops for use in certain countries. Initializing pin requests with a clean laptop is concerning. Additionally, if one wants to remove private data, the pinning database has to be removed as well, since the pin database provides a nice set of tracks for the user.

21. User Authentication: ID Protocols: 3/11/2013

The setup for this is that the user P is a prover, who has a key that opens a server V, the verifier. There is an algorithm G that generates a signing key sk and a verification key vk, such that V gives a yes/no

[29] Even if there is only one option, this is still done; if it were done automatically, the user would be authenticating itself to anyone without consent, which is a great way to be tracked without knowing it.

answer that detects whether the user is verified. Notice that there is no key exchange. Additionally, vk can be either public or secret, though the latter is a problem if the server is broken into.

There are several examples of such security:

- Physical locks (e.g. car locks), which might be used for opening cars, garage doors, etc. One of these systems for cars, Keeloq, is not quite secure, which means cryptographers can drive a lot of fancy cars.
- One might login at a bank ATM or a website, which is a somewhat different problem.
- Finally, there is authentication over SSL.

Here is an example that should indicate how *not* to use ID protocols: they do not establish a secure session between two users! Even when combined with anonymous key exchange, this is vulnerable to a man-in-the-middle attack, as mentioned in the previous lecture. Anonymous key exchange by itself doesn't add anything.

There are three security models:

(1) The weakest model is a direct attacker, who impersonates the prover with no information other than vk (i.e. a door lock).
(2) An eavesdropping attacker impersonates the prover after listening to conversations between the prover and the verifier, as in a wireless car entry system. One cool application of this is to allow a car to open a garage door by essentially performing a man-in-the-middle attack, since the garage door is only protected against a direct attack.
(3) An active attacker can change the information sent between the prover and the verifier. For example, one might set up a fake ATM, which can collect and modify data from the prover before sending it to the verifier.

If all we care about is a direct attack, we use a password system. This is the weakest form of security, but also one that is very commonly used. A basic password protocol (which actually isn't quite correct) considers the set PWD of all passwords. This set is smaller than one might think.

This incorrect algorithm generates a password $pw \in \mathrm{PWD}$ and lets $sk = vk = pw$. Then, the user authenticates by checking if the password received is identical to the stored password. This is a problem if the server is broken into; such an adversary can impersonate the user without ever knowing anything else about it. This happens sadly often in the real world, and the takeaway lesson is to never store passwords in the clear![30]

A better solution is to use a one-way hash function $H : \mathrm{PWD} \to X$. Then, $sk = pw$, but $vk = H(pw)$. The user sends sk to V to authenticate, and the server computes $H(vk)$ and returns whether they are equal. If the server is compromised, the passwords aren't revealed.

This isn't without problems: people tend to choose passwords from a small set. For example, `123456` appeared almost 1% of the time in a password sample. Six different passwords (including such great examples as `12345` and `Password`) accounted for 5%, and 23% of users chose passwords in a dictionary for which hashes could be computed in a couple seconds. People used to do online dictionary attacks, entering passwords one after another, but a good defense is to double response times after each incorrect failure. Locking an account would be a solution, but it can be used to lock legitimate users out of their accounts due to an adversary making lots of requests. Online attackers have adapted to this slowed response time by taking a common password and trying lots of usernames until one works, rather than breaking one specific user with lots of passwords; this sort of attack is a bit harder to fix, and requires tracking IP addresses. Attackers use botnets to get around this.

So why haven't hackers done this all day long? It's actually easier to get passwords in other ways, such as phishing attacks. This will be discussed more in CS 155.

There is yet another attack called an offline dictionary attack. If an attacker obtains lots of hashed passwords $vk = H(pw)$ from a server. Then, the attacker hashes all words in a dictionary D until a word $w \in D$ is found such that $H(w) = vk$. The time is $O(|D|)$ per passwords, which equates to 360,000,000 guesses in a few minutes, which is enough to recover a quarter of the passwords. There are lots of

[30] It turns out that most of the studies that analyze common passwords use lists of passwords published by such hackers...

tools for this (e.g. John the Ripper, Cain and Abel). Using a batch offline dictionary attack, there is a much better algorithm that computes a list $L = \{w, H(w) \mid w \in D\}$ and computes its intersection with a password file F. This takes time $O(|D| + |F|)$, which is much less than ideal (for security). Hashing by itself doesn't provide any security.

This is why a salt is used. When setting a password, pick a random $S_A \in \{0,1\}^n$ and store it in the password file. Thn, when verifying the password, test if $H(pw, S_A) = h_A$ (the stored value). This makes the dictionary attack much less possible, particularly since different users have different salts even if they have the same password.

One could alternatively use a slow hash function. The burden on the user or server is small, but an attcker doing a brute-force dictionary attack is hindered. One good example is 4000 iterations of SHA-1, which might induce an 0.1-second delay, which is enough. Another example is encrypting a `.dmg` file on a Mac, which requires 250000 iterations and takes about half a second. It happens that there might be a way to bypass this without computing it naively, but we don't know; lower bounds in cryptography are open problems. However, there are definitely one-way functions for which a shortcut exists: for example, in RSA, $H(x) = x^e \bmod N$, so $H(H(x)) = (x^e)^e = x^{(e^2)}$. Then, $H^{1000}(x) = x^{e^{1000}} \ (N)$. If $\varphi(N)$ is known, this can be found as $x^{e^{1000} \bmod \varphi(N)}$, which is a nice shortcut. This has useful applications, such as testing the processing power of a machine. One can compute something like this via brute-force if they don't know the factorization of N, and then the solution can be checked once the factorization is known. Slow hash functions are a very important way to protect against dictionary attacks.

Another solution is to use secret salts, in which an 8-bit salt is chosen for each user, but that is never stored. The hashed value with the salt is stored, but the salt is thrown out. Then, when testing the user, compute all possible values of the hash given any salt, which slows down any attacker but doesn't do too much against the user. This is basically a slow hash function.

Note that 4000 iterations are necessary for a slow machine (e.g. a linksys router), and 250000 for a faster machine. At least today; who knows for next year?

Unix uses a 12-bit public salt (which isn't really enough) and a hash function that iterates DES 25 times. Windows NT and later use MD4 without public and secret salts. Today, one must use a slow hash function and a much larger salt.

Sometimes people use biometrics (e.g. fingerprints, retina scanners). These are hard to forget, which is nice. However, they shouldn't be used to replace passwords, but along with them. This is because biometrics tend not to be secret, and they cannot be changed. For example, if one leaves fingerprints on a wall or something, they can be put onto a "gummy finger" and used to break the security.

A final concern about password systems is that people use the same password at many different sites (common password problem). If one uses a banking password at a high school reunion site that stores passwords insecurely, and the latter site is broken into, the bank can't do anything about it.[31]

The solution would be to implement some client-side software that converts a common password pw into a unique per-site password $H(pw, \text{user\_id}, \text{server\_id})$. However, this isn't how the Internet works, and it would have been nice if it were designed in this way.

Moving to the eavesdropping security model, the adversary is given vk and the transcript of several interactions between an honest prover and a verifier. A protocol is secure against eavesdropping if no efficient adversary can win this game.

The password protocol is clearly insecure against this. There are seveal solutions: the SecurID system is a stateful system with a secret key. At setup, an algorithm G generates a secret key sk, which is given to the verifier and the prover. Strictly speaking $sk = vk = (k, 0)$ for some $k \in \mathcal{K}$. Then, the prover has some PRF F and sends $r_0 = F(k, 0)$. The verifier can check this. Then, the counter changes, so the next authentication computes $F(k, 1)$. In practice, the user is given a dongle, so that it doesn't have to keep track of a secret key.

The most obvious issue, that one might press the dongle button lots of times without the server getting the message, is solved by having the server walk the chain $F(k, 0)$, $F(k, 1)$, etc. to confirm. The counter is no secret, but k is.

[31] Relevant xkcd.

It is a "theorem" that if F is a secure PRF then this protocol is secure from eavesdropping. Thus, this protocol tends to just use AES. However, it depends on the server being secure — the RSA server was broken into last year, which caused their dongles to be compromised.

A better solution is the S/Key algorithm. When setting up, the algorithm chooses a random $k \in \mathcal{K}$, and outputs $sk = (k, n)$ and $vk = H^{(n+1)}(k)$ for some large n. Then, the user sends $H^{(n)}(k)$ and the verifier can easily check. Then, the second verification is $H^{(n-1)}(k)$, etc. This is secure against breaking into the server since the hash function is one-way. The downside is that evetuntually n will dwindle to zero, though for very large n it's not that big of a deal. One can actually store $\log n$ keys at unequal points among the chain, which makes the amortized work for the hash computation take constant time.[32] S/Key is often implemented with pencil and paper.

Finally, it is necessary to protect against active attacks. Here, the attacker sees q authentications, then attempts to impersonate without the prover's help. If the prover's help were necessary, this would be a man-in-the-middle attack, which requires authenticated key exchange. The protocols discussed above are vulnerable, causing something called a preplay attack: the user enters some passwords and the attacker passes those on to the verifier, gaining access. This happens in fake ATMs and online phishing.

The way to prevent these attacks is called challenge-response authentication. Here, the setup generates a secret key $sk = vk = k$. Then, the server sends a message $m \in \mathcal{M}$ and the prover returns $t = S_{\text{MAC}}(k, m)$. The verifier then verifies the MAC for the given key and message. Another "theorem" shows this is secure assuming the MAC is, though if the server is broken into, all bets are off. Additionally, this is vulnerable to a dictionary attack: if k is a dictionary word, all such dictionary words can be tried. However, the benefit is that both m and t are very short (Google's CryptoCards use 8 character- long messages).

One solution that makes the verification key public is to replace the MAC with a signature scheme. This is far less common because the message can be very short, but the signature must be longer. The user would have to type in 20 bytes, which is enough for error to be meaningful, but it is worthwhile for inter-computer communications.

Some advanced topics: one of these will be discussed next lecture:

- Digital cash is a protocol that allows spending money such that no money can be spent by a user twice, and the money is spent anonymously (e.g. Bitcoin).
- Zero-knowledge protocols that underlie discrete-log signatures.
- Quantum computing can be used to break stuff, in theory.
- Elliptic curves
- Factoring algorithms
- Advanced public-key techniques (identity-based encryption, so one's public key is just their email address, or functional encryption).

22. Advanced Topics: Elliptic Curves: 3/13/2013

First, some late-breaking news. Just last night (!), a fairly cute attack on RC4 came out. RC4 shouldn't be used anymore, as we already knew, and Salsa should be used instead (though many people still use RC4), since RC4 has a bias in some of the earlier bytes. This has been known for a while, but someone actually calculated the bias of all possible values for the first 256 bytes (it decreases, but gradually). Thus, they can retrieve the plaintext with a non-negligible probability: the plaintext can be recovered by assuming the ciphertext is encoded with the most likely value. After 2^{24} connections, the 2^{nd}, 8^{th}, and 32^{nd} bytes are leaked with probability 1, and many other bytes are recovered with healthy probability. If you have a billion ciphertexts, or 2^{32}, the entire plaintext can be recovered with near-certain probability. Because of this vulnerability, the recommendation is to ignore the first 256 bytes of output of RC4, but of course IE6 doesn't support that, so nobody can use it.

[32] Another relevant xkcd about why we can't really protect against the user's computer being compromised: Dr. Boneh pointed out "the best way around an authentication system is to tie the user to a tree and hit him with a wrench until he gives you the password."

There are two potential solutions: you could have really long URLs, which makes the bias less visible (since the first 256 bytes are public). It might be better to change the session cookie every few hundred requests, to prevent this sort of attack from working. However, don't go back to AES-CBC, since it was worse, and in general don't assume the first bytes of an HTTPS request are completely secure.

Before explaining elliptic-curve cryptography, which would seem like a black art, it is worthwhile discussing the history. Mathematicians have been interested in elliptic curves because they unite the three branches of math: algebra, analysis, and geometry (and topology).

Suppose one is asked to compute the arc length of the segment of an ellipse. This is a standard single-variable calculus calculation, and gives you an integral

$$F(x) = \int_0^x \frac{\mathrm{d}t}{\sqrt{t^3 + at + b}}. \tag{22.1}$$

Lagrange studied this integral for much of his life, but missed something important. The integral $\int \mathrm{d}t/\sqrt{t^2-1} = \arcsin(t)$ is inverse periodic, so Abel and Jacobi guessed that (22.1) would have a similar property. In fact, it is doubly periodic: the function $F^{-1} : \mathbb{C} \to \mathbb{C}$ is such that there exist ω_1, ω_2 such that for all $z \in \mathbb{C}$, $F^{-1}(z) = F^{-1}(z + \omega_1) = F^{-1}(z + \omega_2)$. Thus, the function is defined on a fundamental parallelogram and then can be extended to the entire complex plane.

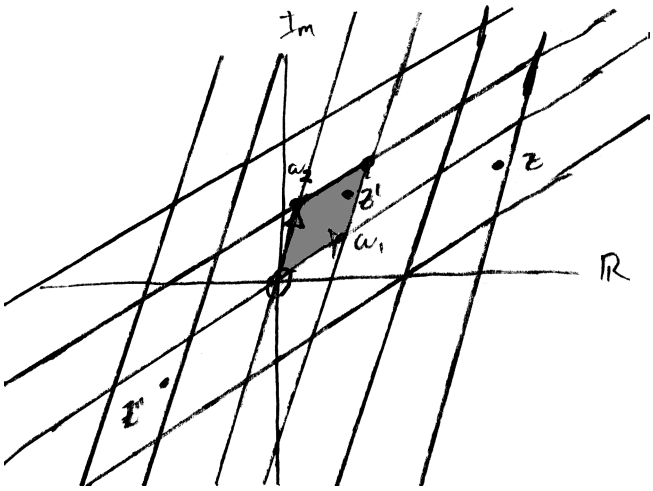

Figure 2. If $f : \mathbb{C} \to \mathbb{C}$ is a doubly periodic function given by periods ω_1 and ω_2, then the shaded region is its fundamental domain and $f(z) = f(z') = f(z'')$.

There is a lot to say about doubly periodic functions. Another example is the pay function $\mathcal{P} : \mathbb{C} \to \mathbb{C}$, which is a well-studied doubly-periodic function in complex analysis. All doubly periodic functions are polynomials in the pay function. Then, consider the dfferential equation $\mathcal{P}'(z)^2 = 4\mathcal{P}(z)^3 - g_2\mathcal{P}(Z) - g_3$ for some $g_2, g_3 \in \mathbb{C}$ and define $E(\mathbb{C}) = \{(x, y) \in \mathbb{C} \mid y^2 = 4x^3 - g_2x - g_3\}$. This means that for any $z \in \mathbb{C}$, $(\mathcal{P}(z), \mathcal{P}'(z)) \in E(\mathbb{C})$, so we obtain a function $\mathbb{C} \xrightarrow{f} E(\mathbb{C})$ which is one-to-one from the fundamental domain (or fundamental parallelogram) to the curve, which motivates studying $E(\mathbb{C})$ in more detail.

$E(\mathbb{C})$ is called an elliptic curve, since it comes from an elliptic integral, which was named because it was used to calculate an ellipse. Defining addition on $\mathbb{C}$ defines addition on $E(\mathbb{C})$: if $z_1, z_2 \in \mathbb{C}$, then define $f(z_1) + f(z_2) = f(z_1 + z_2)$, which gives a map $+ : E(\mathbb{C}) \times E(\mathbb{C}) \to E(\mathbb{C})$. This looks unpleasant to compute, but Abel found a way to do this more quickly.

Geometrically, one can identify the edges of the fundamental domain, gluing opposite edges of the parallelogram together to obtain a torus. A torus is called a genus 1 surface, since it has one hole.

The addition described has an extremely geometric meaning called the cord-and-tangent method. Consider the curve $\{(x, y) \mid y^2 = 4x^3 + ax + b\}$, and for the picture restrict it to $\mathbb{R}$, since we don't have four-dimensional paper. If $p_1, p_2 \in E(\mathbb{C})|_{\mathbb{R}}$, then the line through p_1 and p_2 intersects $E(\mathbb{C})$ in at most three points, since $E(\mathbb{C})$ is a cubic. Then, take the third point $-p_3$ and flip it across the x-axis to obtain

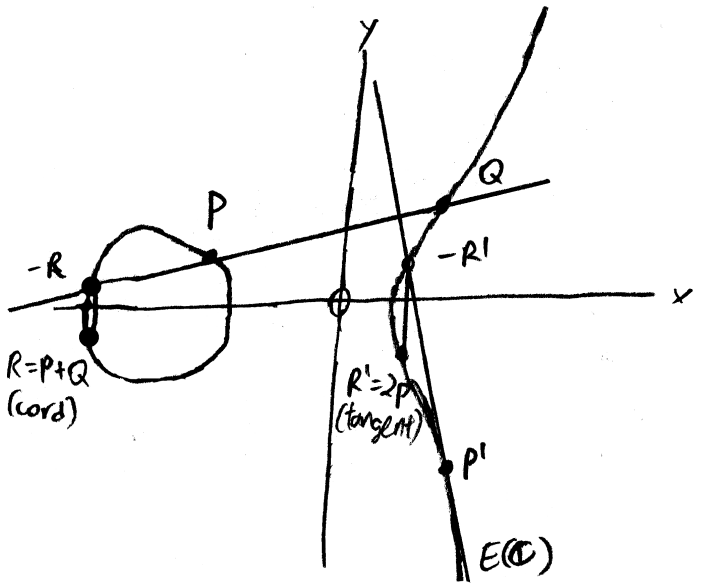

Figure 3. The cord method, demonstrated by $R = P + Q$, and the tangent method, demonstrated by $R' = 2P'$.

their sum $p_3 = p_1 + p_2$. Note that this geometric operation fails in fields other than $\mathbb{R}$, and the algebraic definition has to be used.

If $p_1 = p_2 = p$, then draw the tangent to p, which can be thought of as intersectiong twice at that point. Then, that line intersects another point $-2p$, which is flipped to $2p$.

In general, the geometric interpretation doesn't help very much. The algebraic definition of addition, which is more helpful for computation, is: if $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$, then if $p_3 = (x_3, y_3) = p_1 + p_2$, then let $s = (y_2 - y_1)/(x_2 - x_1)$, giving $x_3 = s^2 - x_1 - x_2$ and $y_3 = y_2 + s(x_3 - x_1)$ (cord method; the tangent method is similar). This is just the restatement of the geometric rule, but makes sense in $\mathbb{C}$ or in $\mathbb{F}_p$.

Over a finite field, choose soem $a, b \in \mathbb{F}_p$ and define $E_{a,b}(\mathbb{F}_p) = \{(x, y) \mid y^2 \equiv x^3 + ax + b \bmod p\}$. Here, the geometry doesn't work as much, so the equations given above have to be used to compute addition. There are also inverses and an identity, so $E(\mathbb{F}_p)$ is a group! In fact, it's an abelian group. In 1942, Weil showed that for all $a, b \in \mathbb{F}_p$, $|E(\mathbb{F}_p)| = p + 1 - t$, where $t < 2\sqrt{p}$ is called the trace.[33]

Now, cryptographers see a group and ask, how hard is the discrete log problem? In $\mathbb{F}_p^*$, the best discrete-log algorithms run in time approximately $e^{\sqrt[3]{\log p}}$, which means primes on the order of 2^{2048} bits are needed for 128-bit security (i.e. time 2^{128}). In an elliptic curve group $E(\mathbb{F}_p)$, the best discrete-log problem runs in time $O(\sqrt{p}) = e^{(\log p)/2}$, so the same amount of security can be found in primes of size 2^{256}. And for 256-bit security, operating over $\mathbb{F}_p^*$ requires primes $p \approx 2^{15000}$, but elliptic curves only require 2^{512}. Of course, the day when someone builds a quantum computer is the day when all of this dies.

These elliptic curves are newer than discrete-log over finite fields by about 20 years, which suggests that it's only more secure because less time has been spent on it. However, any mathematician interested in crypto has probably taken a shot or two at this, which suggests this problem is somewhat hard. Nonetheless, people keep making breakthroughs, and for modular discrete-log, a breakthrough was made for finite fields of low characteristic that takes $e^{\sqrt[4]{\log p}}$ relatively recently, and might generalize to all finite fields.

Now, all of the methods developed for modular crypto translate over to elliptic curves. For example, Diffie-Hellman translates to EC-EDH. One fixes a curve $E(\mathbb{F}_p)$ and chooses a $P \in \mathbb{F}_p$. Then, the broswer

[33] Weil actually proved this while he was in prison for dodging the French World War II draft.

chooses an $a \xleftarrow{R} \{1, \ldots, |E(\mathbb{F}_p)| \approx 2^{256}\}$ and the server chooses a random b in the same way. Then, the broswer calculates $a \cdot P = (x_b, y_b)$ using repeated squaring and sends it to the server, and the server sends $b \cdot P = (x_s, y_s)$.

Elliptic curves have rank at most 2, so they are almost cyclic, and one can choose curves such that the elliptic curve group is cyclic. However, there is no security risk to reusing curves, and there are actually specificed standard curves that are resistant to known attacks.

All of this is old crypto, but there is another amazing fact. Some elliptic curves have additional structure known as a pairing. Let $y^2 = x^3 + 1 \pmod p$, where $p = 2 \pmod 3$. Then, there is a computable, bilinear map $e : E(\mathbb{F}_p) \times E(\mathbb{F}_p) \to \mathbb{F}_{p^2}{}^*$ called a Weil pairing such that $e(a \cdot P, b \cdot P) = e(P, P)^{ab}$. The runtime for e is about the same as a modular exponentiation in $\mathbb{F}_p$.

This has positive and negative applications. Positively, one obtains DLS signatures, which are very useful in practice. Fix a curve $E(\mathbb{F}_p)$, a public $P \in E(\mathbb{F}_p)$, and a hash function $H : \{0,1\}^* \to E(\mathbb{F}_p)$. Then, the key generation function chooses a random $\alpha \in \{1, \ldots, |E(\mathbb{F}_p)|\}$. The secret key is α and the public key is $\alpha \cdot P$ (since this is just repeated addition, it is hard to invert). Then, the signature of a message m is $\sigma = \alpha \cdot H(m) \in E(\mathbb{F}_p)$. This seems like it would take 64 bytes to represent, but only one of (x, y) needs to be given, since the other can be easily calculated (since taking cube roots modulo a prime is easy), so the signature is 32 bytes, but gives 128-bit security. This is the shortest signature that cryptographers currently have, so it is a good way to send sighatures over low-bandwidth connections.

Verification is done as follows: $V(pk = \alpha \cdot P, m, \sigma = \alpha \cdot H(m))$ is equivalent to calculating whether $e(pk, H(m)) = e(P, \sigma)$, because $e(\alpha \cdot P, H(m)) = e(P, H(m))^\alpha$, and $e(P, \sigma) = e(P, \alpha \cdot H(m)) = e(P, H(m))^\alpha$. This is also simple because it's relatively easy to explain. Its security is due to the following theorem:

Theorem 22.2. *If the computatonal Diffie-Hellman problem is hard in $E(\mathbb{F}_p)$ and H is a random oracle, then this signature scheme is unforgeable under a chosen message attack.*

Pairings also solve an ancient problem in crypto called identity-based encryption (IBE). Consider Alice and Bob; in this scheme, Bob's public key is his email address, `bob@gmail.com`. Bob proves to a certificate authority that he is the owner of that email address, and the CA gives him a secret key that allows him to decrypt things. This is much cleaner than the RSA public keys.

This can be implemented in practice with elliptic curves. Fix a curve $E(\mathbb{F}_p)$, a $P \in E(\mathbb{F}_p)$, and a hash function $H : \{0,1\}^* \to E(\mathbb{F}_p)$. Then, one picks an $\alpha \in \{1, \ldots, |E(\mathbb{F}_p)|\}$, with $mk = \alpha$ and $pp = \alpha \cdot P$. Then, Bob's secret key is $sk_{\text{Bob}} = \alpha \cdot H(\text{id}_{\text{Bob}})$. To encrypt a message m, $E(pp, \text{id}_{\text{Bob}}, m)$ involves choosing an $r \in \{1, \ldots, E(\mathbb{F}_p)\}$ and send as a ciphertext $c = (r \cdot pp = r\alpha \cdot p, m \cdot e(pp, H(\text{id}_{\text{Bob}}))^r = e(P, H(\text{id}_{\text{Bob}})^{\alpha r}))$.

Then, decryption is $D(sk_{\text{Bob}}, (c_0, c_1))$ involves computing $e(c_0, sk_{\text{Bob}}) = e(r \cdot P, \alpha \cdot H(\text{id}_{\text{Bob}})) = e(P, H(\text{id}_{\text{Bob}}))^{r\alpha}$. The key of this system is that it uses the bilinearity to equate exponents.

Of course, this is insecure if the CA is broken into, though that isn't much different than what happens in the world today.

This elliptic-curve crypto means that real, theoretical, 20^{th}-Century mathematics has these amazing applications. Of course, Weil was a pure mathematician, and would turn over in his grave to learn of all these terrible applications — another mathematician once expressed a desire for cryptographers to find a fast algorithm for discrete-log so that it could become pure math again!

One disadvantage of pairings is that they allow for a much faster computation of the discrete log, so for curves which admit a pairing, much larger primes have to be used.