Modern Cryptography

600.442

Lecture #16

Dr. Christopher Pappacena

Fall 2013

Public-Key Encryption

A *public-key encryption scheme* $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ is given by:

- $\mathsf{Gen}(n)$ outputs a *public key* $\mathtt{pk}$ and a *secret key* $\mathtt{sk}$.
- Enc takes as input the public key and a message m and returns a ciphertext c. We write $c \leftarrow \mathsf{Enc}_{\mathtt{pk}}(m)$.
- Dec takes as input the secret key and a ciphertext c and returns either a message m or a decryption failure $\perp$. We write $m = \mathsf{Dec}_{\mathtt{sk}}(c)$.

We require that $\Pr[\mathsf{Dec}_{\mathtt{sk}}(\mathsf{Enc}_{\mathtt{pk}}(m)) \neq m] = \mathtt{negl}(n)$ for some negligble function of n.

Remarks

- The public key is made available so that *anyone* can encrypt messages.

- We allow a negligible probability of decryption failure.

- We generally need a method for encoding messages as elements of a group. For RSA, we can encode a nonzero message of $\leq n-1$ bits by viewing it as an integer in $\{1, \ldots, N-1\}$.

- For El Gamal encryption this encoding is more complicated.

The Indistinguishability Experiment

- Gen is run to produce $\mathtt{pk}$ and $\mathtt{sk}$.

- $\mathcal{A}$ is given $\mathtt{pk}$ and produces two messages m_0, m_1 of the same length.

- $b \leftarrow \{0, 1\}$ is chosen at random and $\mathcal{A}$ is given $c \leftarrow \mathsf{Enc}_{\mathtt{pk}}(m_b)$.

- $\mathcal{A}$ outputs a bit b'. We write $\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1$ if $b = b'$ and say that $\mathcal{A}$ *succeeds*; otherwise $\mathcal{A}$ *fails*.

Definition: A public-key encryption scheme Π has indistinguishable encryptions in the presence of an eavesdropper if, for all PPT adversaries $\mathcal{A}$, we have

$$\Pr[\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1] \leq \frac{1}{2} + \mathtt{negl}(n)$$

for some negligible function $\mathtt{negl}$.

Remark: There is *no* notion of “perfect secrecy” for a public-key encryption schems. A computationally unbounded adversary can recover the message m from c with probability 1 (HW Problem).

Public-Key Encryption and CPA Security

In the public-key setting, the adversary is given access to the public key $\mathtt{pk}$. This means two things:

- Enc *must* be randomized. Otherwise, $\mathcal{A}$ just computes $c_b = \mathsf{Enc}_{\mathtt{pk}}(m_b)$ for $b = 0, 1$ and easily succeeds.

- $\mathcal{A}$ has access to an encryption oracle and so is able to mount a chosen-plaintext attack.

Proposition: If a public-key encryption scheme Π has indistinguishable encryptions in the presence of an eavesdropper, then Π is also CPA secure.

Multiple Encryptions

We can define an experiment $\mathsf{PubK}^{\mathsf{mult}}_{\mathcal{A},\Pi}(n)$, where the adversary $\mathcal{A}$ produces a pair of message *vectors* $M_0 = (m_0^1, \ldots, m_0^t)$ and $M_1 = (m_1^1, \ldots, m_1^t)$ for some t.

The game is played the same way, with $\mathcal{A}$ receiving a ciphertext vector C_b. $\mathcal{A}$ returns b' and succeeds if $b = b'$.

We define security in the presence of multiple encryptions in the obvious way.

One For All and All For One

Theorem: If a public key scheme Π has indistinguishable encryptions in the presence of an eavesdropper, then it has indistinguishable multiple encryptions in the presence of an eavesdropper.

The proof uses a hybrid argument.

Proof of Theorem

In the experiment $\mathsf{PubK}^{\mathsf{mult}}_{\mathcal{A},\Pi}(n)$, $\mathcal{A}$ selects two plaintext vectors M_0 and M_1 of length t.

For $0 \leq i \leq t$, define the ciphertext vector $C^{(i)}$ by

$$C^{(i)} = (\mathsf{Enc}_{\mathtt{pk}}(m_0^1), \ldots, \mathsf{Enc}_{\mathtt{pk}}(m_0^i), \mathsf{Enc}_{\mathtt{pk}}(m_1^{i+1}), \ldots, \mathsf{Enc}_{\mathtt{pk}}(m_1^t)).$$

In words, $C^{(i)}$ is an encryption of the first i plaintexts of M_0 followed by the last $t-i$ plaintexts of M_1.

As we range over all possible randomizations of Enc, each $C^{(i)}$ defines a *distribution* on vectors with t ciphertexts.

Proof of Theorem, II

Now let $\mathcal{A}'$ be an adversary which uses $\mathcal{A}$ as a subroutine:

- $\mathcal{A}'$ gives $\mathtt{pk}$ to $\mathcal{A}$ and receives M_0 and M_1.
- $\mathcal{A}'$ chooses $i \leftarrow \{1, \ldots, t\}$, outputs m_0^i, m_1^i, and receives c^i.
- $\mathcal{A}'$ computes $c^j \leftarrow \mathsf{Enc}_{\mathtt{pk}}(m_0^j)$ for $j < i$ and $c^j \leftarrow \mathsf{Enc}_{\mathtt{pk}}(m_1^j)$ for $j > i$.
- $\mathcal{A}'$ gives $(c^1, \ldots, c^t)$ to $\mathcal{A}$ and returns the bit b' returned by $\mathcal{A}$.

Proof of Theorem, III

If $b = 0$, then the ciphertext that $\mathcal{A}'$ gives $\mathcal{A}$ is $C^{(i)}$. We have:

$$\begin{aligned}\Pr[\mathcal{A}'(n) = 0 | b = 0] &= \sum_{j=1}^{t} \Pr[\mathcal{A}'(n) = 0 | b = 0 \wedge i = j] \times \Pr[i = j] \\ &= \frac{1}{t} \sum_{j=1}^{t} \Pr[\mathcal{A}(C^{(j)}) = 0].\end{aligned}$$

If $b = 1$, then the ciphertext is $C^{(i-1)}$ and

$$\Pr[\mathcal{A}'(n) = 1 | b = 1] = \frac{1}{t} \sum_{j=0}^{t-1} \Pr[\mathcal{A}(C^{(j)}) = 1].$$

Proof of Theorem, IV

Combining these gives

$$\begin{aligned}\Pr[\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A}',\Pi}(n) = 1] &= \frac{1}{2}\Pr[\mathcal{A}'(n) = 0 | b = 0] + \frac{1}{2}\Pr[\mathcal{A}'(n) = 1 | b = 1] \\ &= \frac{1}{2t}\left(\sum_{j=1}^{t} \Pr[\mathcal{A}(C^{(j)}) = 0] + \sum_{j=0}^{t-1} \Pr[\mathcal{A}(C^{(j)}) = 1]\right) \\ &= \frac{t-1}{2t} + \frac{1}{2t}\Pr[\mathsf{PubK}^{\mathsf{mult}}_{\mathcal{A},\Pi}(n) = 1].\end{aligned}$$

If the advantages of $\mathcal{A}$ and $\mathcal{A}'$ are $\epsilon(n)$ and $\epsilon'(n)$ then this gives

$$\epsilon(n) = t \cdot \epsilon'(n).$$

Recap

- Indistinguishable encryptions in the presence of an eavesdropper implies indistinguishable *multiple* encryptions in the presence of an eavesdropper.

- We can bootstrap a fixed-length public-key system into one for arbitrary-length messages.

- Indistinguishability implies CPA-security for public-key encryption. As a result, any secure public-key encryption scheme must have *randomized* encryptions.

- “Textbook” RSA doesn’t use randomization, so is insecure!

Padded RSA

Let $\ell(n)$ be a function with $\ell(n) \leq 2n - 2$. To encrypt $m \in \{0,1\}^{\ell(n)}$, choose a random $r \leftarrow \{0,1\}^{\|N\|-\ell(n)-1}$ and set

$$c = (r||m)^e \pmod{N}.$$

To decrypt, let $\tilde{m} = c^d \pmod{N}$ and set m equal to the low $\ell(n)$ bits of $\tilde{m}$.

Theorem: If $\ell(n) = O(\log n)$ and the RSA problem is hard relative to GenRSA, then this gives a CPA-secure public-key encryption scheme.

The proof uses the fact that the low-order bits give hard-core predicates for RSA.

PKCS #1 v1.5

Let k be the size of N in bytes. Messages m range from 1 to $k-11$ bytes long.

The encryption of an s-byte message m is

$$(00000000||00000010||r||00000000||m)^e \pmod{N},$$

where r is a random string of $k-s-3$ nonzero bytes.

This is believed to be CPA-secure but no proof is known based on the RSA assumption. It is known to *not* be CCA-secure.

El Gamal Encryption

In 1985, El Gamal introduced a public-key encryption scheme whose security is based on the DDH problem.

Let $\mathcal{G}$ be a group generation algorithm for which the DDH problem is hard.

- Run $\mathcal{G}(n)$ to obtain (G, q, g). Choose $x \leftarrow \mathbb{Z}_q$ and set $h = g^x$. Set $\mathtt{pk} = (G, q, g, h)$ and $\mathtt{sk} = (G, q, g, x)$.

- Given a message $m \in G$, choose $y \leftarrow \mathbb{Z}_q$ and set $c = (g^y, h^y m)$.

- Given $c = (c_1, c_2)$, decrypt by setting $m = c_2/c_1^x$ $(= c_2 \circ c_1^{-x})$.

Theorem: If the DDH problem is hard relative to $\mathcal{G}$, then El Gamal encryption scheme Π has indistinguishable encryptions in the presence of an eavesdropper and is CPA-secure.

Proof: Let $\mathcal{A}$ be an adversary who can break Π with advantage $\epsilon(n)$.

Define $\widetilde{\mathsf{Enc}}_{\mathrm{pk}}(m)$ to be $(g^y, g^z m)$ for random $y, z \leftarrow \mathbb{Z}_q$.

If $\widetilde{\mathsf{Enc}}_{\mathrm{pk}}$ is used in place of $\mathsf{Enc}_{\mathrm{pk}}$ then $\mathcal{A}$ will succeed with probability $1/2$ since no information about m is revealed.

Proof, Continued

We design a distinguisher D for the DDH problem. Recall that D is given (G, q, g, g^x, g^y, h) and needs to decide if $h = g^{xy}$.

- D gives $\texttt{pk} = (G, q, g, g^x)$ to $\mathcal{A}$ and receives m_0, m_1.
- D chooses b, gives $c = (g^y, hm_b)$ to $\mathcal{A}$, and receives b'.
- If $b' = b$ then D returns 1, otherwise D returns 0.

Proof, Continued

If $h \neq g^{xy}$, then $\mathcal{A}$ returns 0 and 1 with probability 1/2 each. So D returns 1 with probability 1/2 in this case.

If $h = g^{xy}$, then $\mathcal{A}$ returns b with probability $1/2 + \epsilon(n)$ and so D returns 1 with probability $1/2 + \epsilon(n)$ in this case.

Hence,

$$|\Pr[D(h = g^{xy}) = 1] - \Pr[D(h \neq g^{xy}) = 1]| = \epsilon(n).$$

Since the DDH problem is hard for $\mathcal{G}$, $\epsilon(n)$ is negligible.

Factoring and One-Way Functions

Let $t(n)$ be the maximum number of random bits needed by GenMod(n) to produce (p, q, N).

We define a function $f : \{0, 1\}^{t(n)} \to \{0, 1\}^{2n}$ as follows:

- Run GenMod(n) using input $x \in \{0, 1\}^{t(n)}$ in place of the random bits.
- Given the output (p, q, N) of GenMod, let $f(x) = N$.

If f can be inverted, then the input to f can be used to find the factors of N. So if factoring is hard relative to GenMod, then f is a one-way function.

The RSA Problem and One-Way Permutations

Let GenRSA(n) $= (N, e, d)$. Then the map

$$f_{e,N} : x \mapsto x^e \pmod{N}$$

is a *permutation* of the set $\mathbb{Z}_N^*$, with inverse $f_{d,N}$.

The problem of computing x given $y = f_{e,N}(x)$ is exactly the RSA problem with input y.

So if the RSA problem is hard relative to GenRSA, then $f_{e,N}$ is (morally) a one-way permutation.

In order to make this idea fit the general framework of one-way functions we have to be a bit fussy. Details are in the book.

Hash Functions

Define a fixed-length hash function H as follows:

- Run $\mathcal{G}(n)$ to obtain (G, q, g), select $h \leftarrow G$, and set $s = (G, q, g, h)$.
- Given input $x = (x_1, x_2) \in \mathbb{Z}_q \times \mathbb{Z}_q$, ouptut $H^s(x) = g^{x_1} h^{x_2}$.

H looks like a hash function since it reduces two elements of a cyclic group of order q to a single element of a cyclic group of order q.

For it to actually compress its output we need elements of G to be encoded by no more than $2n - 2$ bits, or use a *randomness extractor*.

The Discrete Logarithm Problem and Hash Functions

Theorem: If the discrete logarithm problem is hard relative to $\mathcal{G}$ and H compresses its input, then H is a collision-resistant hash function.

Proof: Suppose $\mathcal{A}$ can invert H. Given $s = (G, q, g, h)$, call $\mathcal{A}$ as a subroutine with seed s and receive x, x'.

Write $x = (x_1, x_2)$ and $x' = (x'_1, x'_2)$ and set $y = (x_1 - x'_1)/(x_2 - x'_2)$ (mod q).

If $H^s(x) = H^s(x')$, then $y = \log_g h$.

The above can be turned into a formal security proof – see Theorem 7.73 of Katz and Lindell.

Modern Cryptography

600.442

Lecture #17

Dr. Christopher Pappacena

Fall 2013

Announcements

- Homework 8 due on Tuesday, November 19.
- Exam 2 on Thursday, November 21.
- Exam will cover Chapters 6, 7, 9, and 10.

Last Time

- RSA and El Gamal Encryption
- Multi-Message and CPA Security of Public-Key Cryptography
- Connections between factoring and the discrete log problem to one way functions, one way permutations, and secure hash functions.

Hybrid Encryption

In general, public-key encryption is more computationally expensive than private-key encryption.

According to the book, a reasonable estimate for the relative efficiency of private-key to public-key encryption is a factor of 10^6!

We would like to combine the functionality of public-key encryption with the efficiency of private-key encryption, particularly for long messages.

Hybrid Encryption

Let $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a public-key encryption scheme and let $\Pi' = (\mathsf{Gen}', \mathsf{Enc}', \mathsf{Dec}')$ be a private-key encryption scheme.

We define a *hybrid* scheme $\Pi^{\mathsf{hyb}} = (\mathsf{Gen}^{\mathsf{hyb}}, \mathsf{Enc}^{\mathsf{hyb}}, \mathsf{Dec}^{\mathsf{hyb}})$ as follows.

- $\mathsf{Gen}^{\mathsf{hyb}}(n)$ runs $\mathsf{Gen}(n)$ to obtain keys $\mathtt{pk}$ and $\mathtt{sk}$.
- Run $\mathsf{Gen}'(n)$ to obtain a key $k \leftarrow \{0,1\}^n$. Set $c_1 \leftarrow \mathsf{Enc}_{\mathtt{pk}}(k)$, $c_2 \leftarrow \mathsf{Enc}'_k(m)$, and $(c_1, c_2) \leftarrow \mathsf{Enc}^{\mathsf{hyb}}_{\mathtt{pk}}(m)$.
- To decrypt $c = (c_1, c_2)$, set $k = \mathsf{Dec}_{\mathtt{sk}}(c_1)$ and $m = \mathsf{Dec}'_k(c_2)$.

Note that Π^{hyb} is a *public-key scheme*, even though it uses a private-key scheme to do “most” of the encryption.

Theorem: If Π is a CPA-secure public-key encryption scheme and Π' has indistinguishable encryptions in the presence of an eavesdropper, then Π^{hyb} is a CPA-secure public-key encryption scheme.

Outline of Proof

An adversary $\mathcal{A}$ cannot distinguish:

- $(\mathtt{pk}, \mathsf{Enc}_{\mathtt{pk}}(k), \mathsf{Enc}'_k(m_0))$ from $(\mathtt{pk}, \mathsf{Enc}_{\mathtt{pk}}(0^n), \mathsf{Enc}'_k(m_0))$.

- $(\mathtt{pk}, \mathsf{Enc}_{\mathtt{pk}}(0^n), \mathsf{Enc}'_k(m_0))$ from $(\mathtt{pk}, \mathsf{Enc}_{\mathtt{pk}}(0^n), \mathsf{Enc}'_k(m_1))$.

- $(\mathtt{pk}, \mathsf{Enc}_{\mathtt{pk}}(0^n), \mathsf{Enc}'_k(m_1))$ from $(\mathtt{pk}, \mathsf{Enc}_{\mathtt{pk}}(k), \mathsf{Enc}'_k(m_1))$.

By "transitivity of indistinguishability", Π^{hyb} has indistinguishable encryptions and hence is CPA-secure.

Adversary $\mathcal{A}_1$

Let $\mathcal{A}^{\mathsf{hyb}}$ be an adversary that can break Π^{hyb} with advantage $\epsilon(n)$.

$\mathcal{A}_1$ will be an adversary which calls $\mathcal{A}^{\mathsf{hyb}}$ as a subroutine:

- $\mathcal{A}_1$ is given $\mathtt{pk}$, runs $\mathsf{Gen}'(n)$ to produce k, gives the messages $k, 0^n$, and receives c.

- $\mathcal{A}_1$ calls $\mathcal{A}^{\mathsf{hyb}}$ as a subroutine and receives m_0, m_1.

- $\mathcal{A}_1$ gives $(c, \mathsf{Enc}'_k(m_0))$ to $\mathcal{A}^{\mathsf{hyb}}$, and returns the bit b' returned by $\mathcal{A}^{\mathsf{hyb}}$.

Success Probability for $\mathcal{A}_1$

We have

$$\begin{aligned}\Pr[\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A}_1,\Pi}(n) = 1] = \frac{1}{2}\Pr[\mathcal{A}^{\mathsf{hyb}}(\mathsf{Enc}_{\mathtt{pk}}(k), \mathsf{Enc}'_k(m_0)) = 0] \\ + \frac{1}{2}\Pr[\mathcal{A}^{\mathsf{hyb}}(\mathsf{Enc}_{\mathtt{pk}}(0^n), \mathsf{Enc}'_k(m_0)) = 1].\end{aligned}$$

Since Π is CPA-secure, this success probability is $\leq 1/2 + \mathtt{negl}_1(n)$ for some negligible function $\mathtt{negl}_1$.

Adversary $\mathcal{A}'$

Next consider an adversary $\mathcal{A}'$ which uses $\mathcal{A}^{\mathsf{hyb}}$ as a subroutine:

- $\mathcal{A}'$ runs $\mathsf{Gen}(n)$ to produce keys $\mathtt{pk}, \mathtt{sk}$.
- $\mathcal{A}'$ runs $\mathcal{A}^{\mathsf{hyb}}$ with public key $\mathtt{pk}$ and receives m_0, m_1.
- $\mathcal{A}'$ outputs m_0, m_1 and receives c.
- $\mathcal{A}'$ gives $(\mathsf{Enc}_{\mathtt{pk}}(0^n), c)$ to $\mathcal{A}^{\mathsf{hyb}}$ and returns the bit b' output by $\mathcal{A}^{\mathsf{hyb}}$.

Success Probability for $\mathcal{A}'$

We have

$$\begin{aligned}\Pr[\mathsf{PrivK}^{\mathsf{eav}}_{\mathcal{A}',\Pi'}(n) = 1] = \frac{1}{2}&\Pr[\mathcal{A}^{\mathsf{hyb}}(\mathsf{Enc}_{\mathtt{pk}}(0^n), \mathsf{Enc}'_k(m_0)) = 0] \\ &+ \frac{1}{2}\Pr[\mathcal{A}^{\mathsf{hyb}}(\mathsf{Enc}_{\mathtt{pk}}(0^n), \mathsf{Enc}'_k(m_1) = 1)].\end{aligned}$$

Since Π' has indistinguishable encryptions in the presence of an eavesdropper, this success probability is $\leq 1/2 + \mathtt{negl}'(n)$ for some negligible function $\mathtt{negl}'$.

Adversary $\mathcal{A}_2$

Finally, consider the adversary $\mathcal{A}_2$ which uses $\mathcal{A}^{\mathsf{hyb}}$ as a subroutine as follows:

- $\mathcal{A}_2$ is given $\mathtt{pk}$, runs $\mathsf{Gen}'(n)$ to produce k, gives the messages $0^n, k$, and receives c.
- $\mathcal{A}_2$ calls $\mathcal{A}^{\mathsf{hyb}}$ and receives m_0, m_1.
- $\mathcal{A}_2$ gives $(c, \mathsf{Enc}'_k(m_1))$ to $\mathcal{A}^{\mathsf{hyb}}$ and returns the bit b' returned by $\mathcal{A}^{\mathsf{hyb}}$.

Success Probability for $\mathcal{A}_2$

We have

$$\begin{aligned}\Pr[\mathsf{PubK}^{\mathsf{eav}}_{\mathcal{A}_2,\Pi}(n) = 1] = \frac{1}{2}\Pr[\mathcal{A}^{\mathsf{hyb}}(\mathsf{Enc}_{\mathtt{pk}}(0^n), \mathsf{Enc}'_k(m_1)) = 0] \\ + \frac{1}{2}\Pr[\mathcal{A}^{\mathsf{hyb}}(\mathsf{Enc}_{\mathtt{pk}}(k), \mathsf{Enc}'_k(m_1) = 1)].\end{aligned}$$

Since Π is CPA-secure, this success probability is $\leq 1/2 + \mathtt{negl}_2(n)$ for some negligible function $\mathtt{negl}_2$.

Putting it Together

Summing our three inequalities gives

$$\begin{aligned}\frac{3}{2} + \mathtt{negl}(n) \geq \frac{1}{2}(&\Pr[\mathcal{A}^{\mathsf{hyb}}(\mathsf{Enc}_{\mathtt{pk}}(k), \mathsf{Enc}'_k(m_0)) = 0] \\ &+ \Pr[\mathcal{A}^{\mathsf{hyb}}(\mathsf{Enc}_{\mathtt{pk}}(0^n), \mathsf{Enc}'_k(m_0)) = 1] \\ &+ \Pr[\mathcal{A}^{\mathsf{hyb}}(\mathsf{Enc}_{\mathtt{pk}}(0^n), \mathsf{Enc}'_k(m_0)) = 0] \\ &+ \Pr[\mathcal{A}^{\mathsf{hyb}}(\mathsf{Enc}_{\mathtt{pk}}(0^n), \mathsf{Enc}'_k(m_1)) = 1] \\ &+ \Pr[\mathcal{A}^{\mathsf{hyb}}(\mathsf{Enc}_{\mathtt{pk}}(0^n), \mathsf{Enc}'_k(m_1)) = 0] \\ &+ \Pr[\mathcal{A}^{\mathsf{hyb}}(\mathsf{Enc}_{\mathtt{pk}}(k), \mathsf{Enc}'_k(m_1)) = 1]).\end{aligned}$$

Here $\mathtt{negl}(n) = \mathtt{negl}_1(n) + \mathtt{negl}'(n) + \mathtt{negl}_2(n)$ is negligible.

Cleaning Up

Simplifying the right hand side in the previous slide gives

$$\frac{3}{2} + \mathtt{negl}(n) \geq 1 + \frac{1}{2}(\mathsf{Pr}[\mathcal{A}^{\mathsf{hyb}}(\mathsf{Enc}_{\mathtt{pk}}(k), \mathsf{Enc}'_k(m_0)) = 0] \\ + \mathsf{Pr}[\mathcal{A}^{\mathsf{hyb}}(\mathsf{Enc}_{\mathtt{pk}}(k), \mathsf{Enc}'_k(m_1)) = 1])$$

So,

$$\frac{1}{2} + \mathtt{negl}(n) \geq \frac{1}{2} + \epsilon(n)$$

and $\epsilon(n)$ is negligible.

CCA Security

CCA indistinguishability for public-key encryption is defined analogously to the private-key setting: After receiving the challenge ciphertext, $\mathcal{A}$ has access to a decryption oracle for any ciphertext *except* c.

The formal definition is in Katz and Lindell, p. 371.

Definition: We say that Π is *CCA-secure* if

$$\Pr[\mathsf{PubK}^{\mathsf{cca}}_{\mathcal{A},\Pi}(n) = 1] \leq \frac{1}{2} + \mathtt{negl}(n)$$

for some negligible function $\mathtt{negl}$.

Homomorphic Encryption and Malleability

"Textbook RSA" and El Gamal encryption are both *homomorphic*, which means that if $c_1 = \mathsf{Enc}_{\mathtt{pk}}(m_1)$ and $c_2 = \mathsf{Enc}_{\mathtt{pk}}(m_1)$, then $c_1 c_2 = \mathsf{Enc}_{\mathtt{pk}}(m_1 m_2)$.

For textbook RSA, this says

$$c_1 c_2 = m_1^e m_2^e = (m_1 m_2)^e \pmod{N}.$$

For El Gamal, we mean coordinate-wise multiplication:

$$(g^{y_1}, h^{y_1} m_1) \cdot (g^{y_2}, h^{y_2} m_2) = (g^{y_1+y_2}, h^{y_1+y_2} m_1 m_2).$$

This *ciphertext malleability* leaves both systems open to chosen ciphertext attacks.

Example: El Gamal

- Suppose $\mathcal{A}$ generates $m_0, m_1 \in G$ and receives back $c_b = (g^y, h^y m_b)$.

- $\mathcal{A}$ chooses a random $m \in G$ and encrypts it to make $c = (g^z, h^z m)$.

- Now $\mathcal{A}$ creates the ciphertext $c' = (g^{y+z}, h^{y+z} m_b m)$ and queries the decryption oracle.

- $\mathcal{A}$ computes $m_b = (m_b m)/m$ and succeeds with probability 1.

Padded RSA

Padding RSA such as in PKCS #1 v1.5 eliminates the homomorphic property – the product of two valid messages is not likely to be a valid message.

However, an adversary can deduce information about the plaintext by observing if chosen ciphertexts decrypt succeessfully or return $\perp$.

A different padding scheme called OAEP (Optimal Asymmetric Encryption Padding) is CCA-secure in the *random oracle model* (Chapter 13).

Trapdoor Permutations

Roughly, a *trapdoor permutation* is a permutation f together with a "trapdoor" value `td`:

- Without knowledge of `td`, f is difficult to invert; i.e. f is a one-way permutation.

- Given `td`, it is possible to invert f in polynomial time.

If f is a trapdoor permutation with hardcore predicate hc, then we can construct a secure public-key system from f.

Public-Key Encryption from Trapdoor Permutations

Define Π as follows.

- We set $\mathtt{pk} = (f, \mathsf{hc})$ and $\mathtt{sk} = \mathtt{td}$.

- To encrypt a bit b, choose $x \leftarrow \{0,1\}^n$ and set

$$\mathsf{Enc}_{\mathtt{sk}}(b) = (f(x), \mathsf{hc}(x) \oplus b).$$

- To decrypt (y, c), use $\mathtt{td}$ to invert $y = f(x)$ and set $b = c \oplus \mathsf{hc}(x)$.

Theorem: If f is a trapdoor permutation then Π is CPA-secure.

The idea of the proof is to note that an adversary who can break the system is computing hc(x) from $f(x)$ with some advantage. Since f is one-way and hc is a hardcore predicate, this advantage is negligible.

The hardest part is giving a definition of "trapdoor permutation" that varies with the security parameter.

Definition: A *trapdoor permutation* is a tuple $(\mathsf{Gen}, \mathsf{Samp}, f, \mathsf{Inv})$

- $\mathsf{Gen}(n)$ outputs $(I, \mathtt{td})$, where I specifies the doman D_I and range $R_I = D_I$ of the function f_I.

- $\mathsf{Samp}(n)$ samples uniformly from D_I.

- $f_I : D_I \to D_I$ is a one-way permutation.

- $\mathsf{Inv}_{\mathtt{td}}(y)$ computes the inverse of y, given the trapdoor secret: $\mathsf{Inv}_{\mathtt{td}}(f_I(x)) = x$ for all $(I, \mathtt{td})$ and $x \in D_I$.

Even though the construction only encrypts a single bit, we know that CPA-security implies indistinguishable multiple encryptions. So we can iterate the construction to encrypt arbitrary length messages.

Chapter 12: Digital Signatures

In their seminal paper, Diffie and Hellman note that a public-key encryption system can be used to provide a *digitial signature* – a message tag that only one person can produce, but anyone can verify:

If m is a message, it is signed by computing $t = \mathsf{Dec}_{\mathtt{sk}}(m)$, where Dec denotes the decryption of the "ciphertext" m with the user's secret key.

Now, anyone can verify the validity of the message with the user's *public* key: $m = \mathsf{Enc}_{\mathtt{pk}}(t)$.

For this to work, encryption must be a "two-sided" inverse to decryption. This is true of both RSA and El Gamal encryption.

While this intuition is still used to popularly describe digital signatures, it predates modern cryptography.

The formal definition of digital signature will show that it is not quite as simple as “reversing” public-key encryption.

Nevertheless, it is closely related and once again illustrates Diffie and Hellman’s insight.

Digital Signatures Defined

A *signature scheme* is a triple of PPT algorithms $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$.

- $\mathsf{Gen}(n)$ produces public and secret keys $\mathtt{pk}$ and $\mathtt{sk}$.
- Given a message $m \in \{0,1\}^*$, $\sigma \leftarrow \mathsf{Sign}_{\mathtt{sk}}(m)$ is the signature of m.
- Given a pair (m, σ), $\mathsf{Verify}_{\mathtt{pk}}(m, \sigma)$ returns a bit b. If $b = 1$ then σ is *valid*, otherwise it is *invalid*.

We require that $\mathsf{Verify}_{\mathtt{pk}}(m, \mathsf{Sign}_{\mathtt{sk}}(m)) = 1$, except possibly with negligible probability.

Uses of Digital Signatures

- Non-repudiation: like a physical signature, a party can attest to a digital document by digitally signing it.

- Message integrity: By signing a message, a party can prove that the message originated from him or her.
  - Frequently-used instance: Signing software updates.

- Digital signatures are an important part of *public-key infrastructure*, where trusted parties attest to the validity of a public key by signing it.

Digital Signatures vs. MACs

Digital signatures offer similar functionality to MACs. The main difference is that *anyone* can verify a digital signature but only the *intended recipient* can verify a MAC.

This has its benefits but has drawbacks as well:

- Digital signatures are more expensive to compute and verify than MACs.

- In some cases, the sender might only want the intended recipient to be able to authenticate the message.

In short, there is need for both primitives in different cryptographic applications.

The Signature Experiment

- Gen(n) is run to obtain signing keys $\mathtt{pk}, \mathtt{sk}$.
- $\mathcal{A}$ is given oracle access to $\mathsf{Sign}_{\mathtt{sk}}(\cdot)$ and outputs (m, σ).
- We say that $\mathsf{SigForge}_{\mathcal{A},\Pi}(n) = 1$ if $\mathsf{Verify}_{\mathtt{pk}}(m, \sigma) = 1$ and $\mathcal{A}$ did not query the oracle with m. We also say that $\mathcal{A}$ *suceeds* or *forges* the signature.

Definition: A signature scheme $\Pi = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ is *existentially unforgeable under an adaptive chosen-message attack* if, for every PPT adversary $\mathcal{A}$, there is a negligible function $\mathtt{negl}$ such that

$$\Pr[\mathsf{SigForge}_{\mathcal{A},\Pi}(n) = 1] \leq \mathtt{negl}(n).$$

Insecurity of “Textbook” RSA

Just as textbook RSA is insecure as a public-key algorithm (relative to modern notions of security), it is also insecure as a digital signature algorithm.

Given a message m, an adversary $\mathcal{A}$ can request the signature of m_1 and $m_2 = m_1^{-1}m$. If the signatures are σ_1 and σ_2, then

$$\sigma_1\sigma_2 = m_1^d m_2^d = (m_1 m_2)^d = m^d \pmod{N}$$

so $\sigma = \sigma_1\sigma_2$ is a valid signature for m.

There is even a “no message” attack: $\mathcal{A}$ chooses σ and sets $m = \sigma^e \pmod{N}$. Then σ is a valid signature for m!

Hashed RSA

Fix a collision-resistant hash function $H : \{0,1\}^* \to \mathbb{Z}_N^*$. To sign a message m, compute

$$\sigma = H(m)^d \pmod{N}.$$

To verify (m, σ), compute $H(m)$ and check if $\sigma^e = H(m) \pmod{N}$.

There is no known function H for which hashed RSA can be proven to be secure. However, if we model H as a *random* function, then it is possible to prove security of hashed RSA.

This *random oracle* model is discussed in Chapter 13.

Hash and Sign

The hashed RSA construction doesn't just make finding collisions harder – it also allows the user to sign messages of arbitary length.

Let Π be a fixed-length signature scheme and let H be a hash function. Construct Π′ as follows.

- Gen′(n) runs Gen(n) to generate keys $\mathtt{pk}, \mathtt{sk}$ and also generates a seed s for H. Set $\mathtt{pk}' = (\mathtt{pk}, s)$ and $\mathtt{sk}' = (\mathtt{sk}, s)$.
- $\mathsf{Sign}'_{\mathtt{sk}'}(m) = \mathsf{Sign}_{\mathtt{sk}}(H^s(m))$.
- $\mathsf{Verify}'_{\mathtt{pk}'}(m, \sigma) = \mathsf{Verify}_{\mathtt{pk}}(H^s(m), \sigma)$.

Theorem: If Π is existentially unforgeable under a chosen-message attack and H is collision-resistant, then Π' is existentially unforgeable under a chosen-message attack.

Proof Outline: Suppose an adversary $\mathcal{A}$ forges the message (m, σ) for Π'. Then he has forged the message $(H^s(m), \sigma)$ for Π. One of two things has happened:

- $\mathcal{A}$ has forged the message $H^s(m)$, which he has never seen before. This violates security of Π.

- $\mathcal{A}$ has seen $H^s(m) = H^s(m')$ before. This means that $\mathcal{A}$ has found a collision in H.

Since both of these are hard, Π is secure.

Modern Cryptography

600.442

Lecture #18

Dr. Christopher Pappacena

Fall 2013

The Digital Signature Standard (DSS)

- Published by NIST in 1991.
- Security is based on the difficulty of the discrete logarithm problem.
- No known security proof, even in the random oracle model.

DSS

Let $\mathcal{G}$ be a group generation algorithm that, on input n, returns (p, q, g) where p and q are primes with $\|q\| = n$, q exactly divides $p - 1$, and g is a generator for the subgroup of $\mathbb{Z}_p^*$ of order q.

- Gen(n) calls $\mathcal{G}(n)$ to obtain (p, q, g), chooses $x \leftarrow \mathbb{Z}_q$ at random, and sets $y = g^x \pmod{p}$. The public key is (H, p, q, g, y) and the private key is (H, p, q, g, x), where $H : \{0, 1\}^* \rightarrow \mathbb{Z}_q$ is a hash function.

- To sign m, choose $k \leftarrow \mathbb{Z}_q$, set $r = [g^k \pmod{p}] \pmod{q}$, compute $s = (H(m) + xr)k^{-1} \pmod{q}$, and output (r, s).

- To verify (m, r, s), compute $u_1 = H(m)s^{-1} \pmod{q}$ and $u_2 = rs^{-1} \pmod{q}$ and verify that $r = [g^{u_1}y^{u_2} \pmod{p}] \pmod{q}$.

So far we haven't given any actual signature schemes that we can prove secure.

In fact, it is hard to produce such schemes. To get there, we will begin with a weaker type of scheme called a *one-time signature scheme*.

In this setup, the adversary $\mathcal{A}$ is allowed to ask for the signature of a *single* message m' before producing his putative forgery (m, σ).

We write $\mathsf{SigForge}^{\mathsf{1time}}_{\mathcal{A},\Pi}(n) = 1$ if $\mathcal{A}$ succeeds.

Lamport's One-Time Signature Scheme

Let f be a one-way function.

- Gen(n): Choose random $x_{i,0}, x_{i,1} \in \{0,1\}^n$ for $1 \leq i \leq \ell$ and compute $y_{i,b} = f(x_{i,b})$. Set $\mathtt{pk} = \vec{y}$ and $\mathtt{sk} = \vec{x}$.

- To sign the message $m = (m_1, \ldots, m_\ell)$, output
$$\sigma = (x_{1,m_1}, x_{2,m_2}, \ldots, x_{\ell,m_\ell}).$$

- To verity (m, σ), check that $f(\sigma_i) = y_{i,m_i}$ for all i.

Theorem: If f is a one-way function then Π is a one-time signature scheme for messages of length ℓ.

Proof: Let $\mathcal{A}$ be an adversary which succeeds with probability $\epsilon(n)$. We define an algorithm $\mathcal{I}$ for inverting f which calls $\mathcal{A}$ as a subroutine.

- Given y, $\mathcal{I}$ chooses i, b at random and set $y_{i,b} = y$. Otherwise, $\mathcal{I}$ chooses $x_{i,b}, y_{i,b}$ as above.

- $\mathcal{I}$ calls $\mathcal{A}$ as a subroutine with input $\vec{y}$. If $\mathcal{A}$ asks for the signature of m' with $m'_i = b$, stop. Otherwise, return correct σ.

- If $\mathcal{A}$ outputs (m, σ) with $m_i = b$, then $\mathcal{I}$ returns $\sigma_{i,b}$. Otherwise $\mathcal{I}$ returns a random value.

Success Probability for $\mathcal{I}$

Note that $\mathcal{I}$ succeeds whenever $\mathcal{A}$ outputs a successful forgery with $m_i = b$.

The probability that $m_i = b$, *conditioned on the forgery being successful*, is at least $1/2\ell$. So,

$$\Pr[\text{Invert}_{\mathcal{I},f}(n) = 1] \geq \frac{\epsilon(n)}{2\ell(n)}.$$

Since $\ell(n)$ is a polynomial in n and f is one-way, $\epsilon(n)$ is negligible.

Stateful Signature Schemes

A *stateful signature scheme* outputs an initial state s_0 in addition to $\mathtt{pk}, \mathtt{sk}$.

Given a message m and state s_{i-1}, Sign$_{\mathtt{sk},s_{i-1}}(m)$ updates the state to s_i and outputs signature σ_i.

We require Verify$_{\mathtt{pk}}$(Sign$_{\mathtt{sk},s_{i-1}}(m)) = 1$, except possibly with negligible probability.

Note that s_i is not needed to verify the signature and in some cases it must be kept secret.

Example: ℓ-time Signatures

- Generate one-time signature schemes $(\mathtt{pk}_i, \mathtt{sk}_i)$ for $i \leq i \leq \ell$. The initial state is $s_0 = 1$.

- To sign message m with state i, use $\mathtt{sk}_i$ and update the state to $i + 1$.

- To verify (m, σ), check if σ is a valid signature for m with respect to *some* $\mathtt{pk}_i$.

Note that we need to choose ℓ ahead of time.

Example: Signature Chains

Initially, generate $(\mathtt{pk}_1, \mathtt{sk}_1)$ for a one-time signature scheme. The initial state is empty.

When a message m_1 needs to be signed, generate new one-time signature keys $(\mathtt{pk}_2, \mathtt{sk}_2)$, sign the message $m_1||\mathtt{pk}_2$, and publish $(m_1, \sigma_1, \mathtt{pk}_2)$. The internal state is updated to $(m_1, \sigma_1, \mathtt{pk}_2, \mathtt{sk}_2)$.

To sign message m_i, generate new keys $(\mathtt{pk}_{i+1}, \mathtt{sk}_{i+1})$, set

$$\sigma_i = \mathsf{Sign}_{\mathtt{pk}_i}(m_i||\mathtt{pk}_{i+1})$$

and output $(m_i, \sigma_i, \mathtt{pk}_{i+1}, \{m_j, \sigma_j, \mathtt{pk}_{j+1}\}_{j\leq i-1})$. The internal state is $(\{m_j, \sigma_j, \mathtt{pk}_{j+1}, \mathtt{sk}_{j+1}\}_{j\leq i})$.

Validating the Chain

To validate a signature chain, use the fact that $\sigma_j = \text{Sign}_{\texttt{sk}_j}(m_j||\texttt{pk}_{j+1})$. All the information needed to validate these steps is included in the i^{th} signature.

Intuitively, the signature chain is existentially unforgeable: Forging a signature requires forging links in the chain, and each link is signed using a different secure one-time signature scheme.

We won't prove this formally.

Issues

- We must be able to sign messages that are strictly longer than the key. So Lamport's one-time signature scheme doesn't work unless we use "hash and sign".

- The signature σ_i for message m_i contains all messages m_j for $j < i$.

- The size of the signature is linear in the number of previous signatures issued.

Tree-Based Signatures

Fix a binary tree of depth n, with leaf nodes corresponding to all possible n-bit messages. Fix a one-time signature scheme Π and a pair of keys $\mathtt{pk}_\epsilon, \mathtt{sk}_\epsilon$ corresponding to the root.

To sign a message m:

- Generate keys $\mathtt{pk}_i, \mathtt{sk}_i$ for each node on the path from the root to the leaf labeled m.
- Certify the path to m by signing $\mathtt{pk}_{w0}||pk_{w1}$ with key $\mathtt{sk}_w$ for each prefix w of m.
- Finally, sign m with private key $\mathtt{sk}_m$.

Details

For an n-bit message $m = (m_1, \ldots, m_n)$, let $m_{\leq j}$ denote the prefix $(m_1, \ldots, m_j)$.

At level j of the tree, the signature is $\sigma_{m_{\leq j}} = \mathsf{Sign}_{\mathtt{sk}_{m_{\leq j}}}(\mathtt{pk}_{m_{\leq j}0} || \mathtt{pk}_{m_{\leq j}1})$.

The final signature is $(\{\sigma_{m_{\leq j}}, \mathtt{pk}_{m_{\leq j}0} || \mathtt{pk}_{m_{\leq j}1}\}_{j \leq n-1}, \sigma_m)$.

Remarks

- The state of the signature is the full list of keys generated for all messages that have been signed so far.

- If a new message passes through a node which already has keys associated to it, the keys are reused.

- The total number of messages signed is polynomial in n, so even though the size of the tree is exponential in n only polynomially-many nodes are ever visited.

Finally, a Digital Signature!

Theorem: If Π is a secure one-time signature scheme, then the tree-based signature is existentially unforgeable under an adaptive chosen-message attack.

Proof: Denote the tree-based signature by Π^*, let $\mathcal{A}^*$ be an adversary who can break Π^* with advantage $\epsilon(n)$, and let $\ell^*(n)$ be an upper bound on the number of oracle calls made by $\mathcal{A}^*$.

Set $\ell(n) = n\ell^*(n) + 1$, an upper bound on the number of public keys needed to answer $\mathcal{A}^*$'s oracle queries.

Proof, II

Let $\mathcal{A}$ be an adversary attacking Π which uses $\mathcal{A}^*$ as a subroutine:

- $\mathcal{A}$ chooses a random index $i \leftarrow \{1, \ldots, \ell\}$, sets $\mathtt{pk}^i = \mathtt{pk}$, and computes $(\mathtt{pk}^j, \mathtt{sk}^j)$ via Gen(n) for all $j \neq i$.

- $\mathcal{A}$ calls $\mathcal{A}^*$ as a subroutine with input $\mathtt{pk}_\epsilon = \mathtt{pk}^1$. When $\mathcal{A}^*$'s query passes through a prefix w, $\mathcal{A}$ uses the next two unused values $\mathtt{pk}^j, \mathtt{pk}^{j+1}$ for $\mathtt{pk}_{w0}, \mathtt{pk}_{w1}$.

- If $\mathcal{A}$ needs to use the secret key $\mathtt{sk}^i$ corresponding to $\mathtt{pk}^i$, it requests the appropriate signature from its oracle.

Proof, III

$\mathcal{A}^*$ outputs the signature $\sigma = (\{\sigma'_{m_{\leq j}}, \mathtt{pk}'_{m_{\leq j}0} || \mathtt{pk}'_{m_{\leq j}1}\}_{j \leq n-1}, \sigma'_m)$ for a message m.

If σ is valid, let j be the smallest index for which $\mathtt{pk}'_{m_{\leq j},0} \neq \mathtt{pk}_{m_{\leq j},0}$ or $\mathtt{pk}'_{m_{\leq j},1} \neq \mathtt{pk}_{m_{\leq j},1}$. Let i' be the index such that $\mathtt{pk}^{i'} = \mathtt{pk}_{m_{\leq j}} = \mathtt{pk}'_{m_{\leq j}}$.

If no such j exists, then $\mathtt{pk}_m = \mathtt{pk}'_m$. In this case let i' be the index with $\mathtt{pk}^{i'} = \mathtt{pk}_m$.

Proof, IV

In the first case, if $i = i'$, then $\mathcal{A}$ outputs $(\mathtt{pk}'_{m_{\leq j},0} || \mathtt{pk}'_{m_{\leq j},1}, \sigma'_{m_{\leq j}})$. Otherwise $\mathcal{A}$ outputs a random message and signature.

In the second case, if $i = i'$ then $\mathcal{A}^*$ never requested a signature with respect to $\mathtt{pk}^i = \mathtt{pk}_m$, yet σ'_m is a valid signature. So $\mathcal{A}$ outputs (m, σ'_m).

Conditioned on $\mathcal{A}^*$ outputting a forgery, $\mathcal{A}$ outputs a forgery with probability $1/\ell(n)$. So the probability that $\mathcal{A}$ succeeds is $\epsilon(n)/\ell(n)$. Since Π is secure, $\epsilon(n)$ is negligible.

Making it Stateless

The tree-based scheme requires maintaining state. If we fix a pseudorandom function F with keys k, k', we can make it stateless.

If we need to generate a pair of keys at prefix w, we use the pseudorandom bits $F_k(w)$ as the source of random for generating $\mathtt{pk}_w, \mathtt{sk}_w$..

Similarly, we use the pseudorandom bits $F_{k'}(w)$ for the random bits needed to construct the signature $\sigma_w = \mathsf{Sign}_{\mathtt{sk}_w}(\mathtt{pk}_{w0}||\mathtt{pk}_{w1})$.

This way, the intermediate states can be constructed on the fly and do not need to be stored. The deterministic nature of the pseudorandom function ensures we always construct the same values at a given node.

The 400 Pound Gorilla

Although we have not made it explicit, we have assumed some amount of *authentication* on the communication channels used for public-key encryption and key exchange.

- For key exchange, how does Alice know that the person she is communicating with is Bob?

- For public-key encryption, how does Bob know that Alice's public key actually belongs to Alice?

Man in the Middle

Suppose that Alice and Bob want to agree on a shared secret key for a private key algorithm Π using a key exchange protocol.

An eavesropper Eve can potentially insert herself *between* Alice and Bob during the exchange:

$$\texttt{Alice} \xrightarrow{A} \texttt{Eve} \xrightarrow{A'} \texttt{Bob}$$

$$\texttt{Alice} \xleftarrow{B'} \texttt{Eve} \xleftarrow{B} \texttt{Bob}$$

- Alice shares key $k = f(a, B')$ with Eve, thinking she is Bob.

- Bob shares key $k' = f(A', b)$ with Eve, thinking she is Alice.

Man in the Middle

What happens when Alice sends message m to Bob?

- She sends $c \leftarrow \mathsf{Enc}_k(m)$ to Eve, thinking its Bob.
- Eve computes $m = \mathsf{Dec}_k(c)$ and sends $c' \leftarrow \mathsf{Enc}_{k'}(m)$ to Bob.
- Bob receives c' and computes $m = \mathsf{Dec}_{k'}(c')$.

Alice and Bob are able to communicate successfully and, they believe, securely, but Eve gets a copy of every message.

Even worse, Eve can *modify* the message if she wants!

Remarks

- While we have described man-in-the-middle attacks against key exchange protocols, they also apply to public-key encryption schemes: Eve simply swaps her public key for Alice's.
- To defeat these attacks, Alice and Bob need a way to authenticate themselves.
- Digital signatures provide authentication but lead to a chicken and egg problem: How do you trust the identity of the signer?

Public Key Infrastructure

The basic idea is simple. Suppose that Bob trusts Charlie, and Charlie attests to the validity of Alice's public key. Then Bob can trust that Alice's public key really belongs to Alice.

How does Charlie do this? He signs Alice's public key with his own signing key!

If Charlie's signing keys are $\mathtt{pk}_C, \mathtt{sk}_C$, then Charlie publishes the signed message

$$\mathsf{Cert}_C(A) = \mathsf{Sign}_{\mathtt{sk}_C}(\texttt{"Alice's public key is pk}_A\texttt{"}).$$

Alice then gives $(\mathtt{pk}_A, \mathsf{Cert}_C(A))$ to Bob. Bob verifies $\mathsf{Cert}_C(A)$ with $\mathtt{pk}_C$ and then knows he can use $\mathtt{pk}_A$ to communicate with Alice.

Certificates and Certificate Authorities

Charlie's digital signature is called a *certificate*. Certificates serve to bind electronic data such as public keys to entities (people, companies, IP addresses, etc.).

Note that in reality, the certificate needs to do a much better job of uniquely identifying Alice.

Issuers of certificates are sometimes called *certificate authorities*, or CAs.

We're back to the chicken and the egg: How does Bob come to trust Charlie, and why does Charlie trust Alice?

Building Trust

Charlie trusts Alice because she authenticates herself to him using a trusted communication channel. For example, Alice can *physically* present herself to Charlie (with 2 forms of ID) along with her public key.

Building Trust

Bob trusts Charlie for one of several reasons:

- Bob has no choice.
- Everybody else trusts Charlie.
- Bob knows Charlie to be trustworthy.

Single CA

In some settings, e.g. a company's internal network, there is a *single* CA who validates everyone's keys.

Employees are authenticated in person when issued their keys and the CA maintains a list of signed keys.

Similar to a KDC in that the CA is a single point of failure – if the CA's secret key is compromised then it is possible to subvert the system.

One big advantage over the KDC: The CA does not participate in each communication – once it signs a key, it is not needed.

Multiple CAs

In other applications, there is no single CA. Instead there are multiple CAs and Alice may get her key signed by one, or several, of these authorities.

Alice sends $(\texttt{pk}_A, \text{Cert}_{C_1}(A), \ldots, \text{Cert}_{C_t}(A))$ to Bob. If Bob trusts some C_i, he verifies the signature and trusts that $\texttt{pk}_A$ belongs to Alice.

In this model, Charlie can *delegate* signing privileges to another CA, Derek:

$$\text{Del}_C(D) = \text{Sign}_{\texttt{sk}_C}(\texttt{"Derek is trustworthy and has key pk}_D\texttt{"}).$$

If Alice obtains a certificate $\text{Cert}_D(A)$ from Derek, she sends the triple $(\texttt{pk}_A, \text{Cert}_D(A), \text{Del}_C(D))$ to Bob.

Multiple CAs in Practice

The multiple CA model is used for SSL/TLS. Your web browser comes pre-loaded with public signing keys for CAs that it trusts.

Some CAs are root certificate authorities, others have delegated signing privileges.

(Demo)

Web of Trust

In this *peer to peer model*, users of a system attest to the validity of *each other's* keys. There are no "authorities", and individuals decide whether to trust certificates issued by other users.

The more "independent" signatures Alice obtains, the more likely Bob is to trust her public key.

Maybe Alice can get a handful of people to collude with her, but the more signatures she gets, the less likely it is that all signers are malicious.

Web of Trust in Practice

The web of trust model is used by the encryption program PGP.

People hold “key signing parties” where people sign each other’s keys.

Keys, along with certificates, can be uploaded and downloaded from sites, e.g. `pgp.mit.edu`.

Webs of trust work well for non-commercial uses.

Revocation

At times it is necessary to *revoke* a public key associated to an individual.

- The individual leaves the company.
- The private key is comprimised.
- The individual is found to be untrustworthy.

Revocation is also handled by certificate authorities.

Expiration Dates and Revocation Lists

For starters, public keys often come with *expiration dates*. When the expiration date passes, a new key needs to be issued and certified.

For example, a company may reissue keys every year.

To revoke keys, a CA signs a *revocation* which is publicly posted:

$$\mathsf{Rev}_C(A) = \mathsf{Sign}_{\mathtt{sk}_C}(\texttt{"Key pk}_A\texttt{ is revoked"}).$$

Bob must check if Alice's key has been revoked before using it.

Public Key Infrastructure

The details of how to issue, retrieve, and verify public keys, including the role of certificate authorities, is known as *public key infrastructure* or PKI.

Establishing a secure, robust, and efficient PKI is no easy task. For details, consult references on protocols and/or network security.

Modern Cryptography

600.442

Lecture #10

Dr. Christopher Pappacena

Fall 2013

Chapter 6: One-Way Functions

Inverting a Function

Let $f : \{0,1\}^* \to \{0,1\}^*$ be a function. The *inverting experiment* for f is defined as follows:

- Choose n and $x \leftarrow \{0,1\}^n$, and compute $y = f(x)$.
- $\mathcal{A}$ is given n and y and returns x'
- $\mathcal{A}$ *succeeds* if $f(x') = y$; we write $\text{Invert}_{\mathcal{A},f}(n) = 1$ if $\mathcal{A}$ succeeds.

Note that the returned value x' need not equal x.

One-Way Functions

Definition: A function $f : \{0,1\}^* \to \{0,1\}^*$ is called a *one-way function* if:

- $f(x)$ can be computed in polynomial time in $|x|$.
- For all PPT algorithms $\mathcal{A}$, $\Pr[\text{Invert}_{\mathcal{A},f}(n) = 1] \leq \mathtt{negl}(n)$.

NB: A function f that is not one-way is not necessarily *always* easy to invert - it is just *non-negligibly* invertible infinitely often.

Connection with $P \neq NP$

The existence of a one-way function implies that $P \neq NP$, since confirming that $f(x') = y$ can be done in polynomial time.

The converse is not true – $P \neq NP$ is a statement about *worst-case* complexity, while one-way functions are *almost always* hard to invert.

So the existence of a one-way function is *stronger* than $P \neq NP$!

Nevertheless, we believe that one-way functions exist.

Example: Define $f(x)$ for $x \in \{0,1\}^n$ as follows:

- Write $x = r||s$ with r and s each $n/2$ bits.
- Set $p = \text{NextPrime}(1||r)$ and $q = \text{NextPrime}(1||s)$, and $f(x) = pq$.
- Inverting f amounts to factoring $N = pq$, which is believed to be hard (though believed to *not* be NP-complete).

Example: Given n-bit strings $x_1, \ldots, x_n$ and J, define f by

$$f(x_1, \ldots, x_n, J) = (x_1, \ldots, x_n, sum_{j \in J} x_j)$$

where we identify J with the subset $\{i : J_i = 1\}$ of $\{1, \ldots, n\}$.

Inverting f means, given $(x_1, \ldots, x_n, y)$, find J with $\sum_{j \in J} x_j = y$.

This is known as the *subset sum* problem and is known to be NP-complete.

One-Way Functions and Cryptography

The existence of one-way functions implies the existence of:

- Pseudorandom generators
- Pseudorandom functions
- Strong pseudorandom permutations
- CCA-secure encryption schemes

Conversely, the existence of an encryption scheme with indistinguishable encryptions in the presence of an eavesdropper implies that one-way functions exist.

Hiding Inputs

A one-way function is not necessarily good at hiding its input, despite being hard to invert:

Example: If f is one-way and we define $g(x, y) = (x, f(y))$ for $|x| = |y|$, then g is one-way, even though g “gives away” half of its input.

Intuitively, there must be *something* hard to figure out about x from $f(x)$, otherwise we should be able to invert f.

This leads to the notion of a *hard-core predicate* (also called a *hard-core bit*).

Hard-Core Predicates

Definition: Let f be a one-way function. A function $\mathsf{hc} : \{0,1\}^* \to \{0,1\}$ is called a *hard-core predicate* for f if $\mathsf{hc}(x)$ can be computed in polynomial time in $|x|$ and, for all PPT algorithms $\mathcal{A}$,

$$\Pr[\mathcal{A}(f(x)) = \mathsf{hc}(x)] \leq \frac{1}{2} + \mathtt{negl}(n).$$

In other words, given $f(x)$, a PPT algorithm $\mathcal{A}$ can determine $\mathsf{hc}(x)$ with only negligible improvement over a random guess.

Hard-Core Predicates Exist

Theorem: Let f be a one-way function and define g by $g(x,r) = (f(x), r)$ for $|r| = |x|$. Define $\mathsf{gl}(x,r) = x \cdot r = \oplus_{i=1}^{n} x_i r_r$. Then $\mathsf{gl}(x,r)$ is a hard-core predicate for g.

The book writes $\mathsf{gl}(x)$ for the function $\oplus_{i=1}^{n} x_i r_i$, after Goldreich and Levin, who first introduced the concept and proved the theorem.

The idea of the proof is to show that if $\mathcal{A}$ can guess $\mathsf{gl}(x,r)$ successfully, then this knowledge can be used to invert f.

A Simple Idea Which Doesn't Quite Work

We can see the value r from $g(x, r)$, so once we know $g(x, r)$ we can generate $g(x, r')$ for other $r' \in \{0, 1\}^n$.

If we can determine $\mathsf{gl}(x, r)$ and $\mathsf{gl}(x, r \oplus e_j)$ for some r and j, then we know x_j:

$$\mathsf{gl}(x, r) \oplus \mathsf{gl}(x, r \oplus e_j) = x \cdot r \oplus x \cdot (r \oplus e_j) = x \cdot e_j = x_j.$$

If we can do this for every j then we have inverted f!

What Goes Wrong?

Two main problems:

- The elements r and $r \oplus e_j$ are not independent, so it is hard to get good bounds for the probability of successfully recovering x_j from the probability of correctly determining gl(x, r).

- If the probability that we succeed in recovering x_j is $\epsilon(n)$, then the probability of recovering x is $\epsilon(n)^n$. This is negligible even if $\epsilon(n) = 0.9999999$.

Let $\mathcal{A}$ be an algorithm that can compute $\mathsf{gl}(x,r)$ from $g(x,r)$ with advantage $\epsilon(n)$. This says that

$$\Pr_{(x,r)\leftarrow\{0,1\}^{2n}}[\mathcal{A}(f(x),r) = \mathsf{gl}(x,r)] \geq \epsilon(n).$$

We first show that we can remove the dependence on x by identifying a suitable subset of $\{0,1\}^n$:

Lemma: There exists a set $S_n \subseteq \{0,1\}^n$, of size at least $\epsilon(n)2^n/2$, such that

$$\Pr_{r\leftarrow\{0,1\}^{n}}[\mathcal{A}(f(x),r) = \mathsf{gl}(x,r)] \geq \epsilon(n)/2$$

for all $x \in S_n$.

Proof of Lemma

Let S_n be the set of all x for which $\mathrm{Pr}_r[\mathcal{A}(f(x), r) = \mathrm{gl}(x, r)] \geq \epsilon(n)/2$. Then write

$$\Pr_{(x,r)}[\mathcal{A}(f(x), r) = \mathrm{gl}(x, r)] \leq \Pr_x[x \in S_n] + \Pr_{(x,r)}[\mathcal{A}(f(x), r) = \mathrm{gl}(x, r) | x \notin S_n].$$

Then

$$\begin{aligned}\Pr_x[x \in S_n] &\geq \Pr_{(x,r)}[\mathcal{A}(f(x), r) = \mathrm{gl}(x, r)] - \Pr_{(x,r)}[\mathcal{A}(f(x), r) = \mathrm{gl}(x, r) | x \notin S_n] \\ &\geq \frac{1}{2} + \epsilon(n) - \left(\frac{1}{2} + \frac{\epsilon(n)}{2}\right) = \epsilon(n)/2.\end{aligned}$$

This says that the size of S_n is at least $\epsilon(n)2^n/2$.

Making Independent Choices

Lemma: For any ℓ, let $s_1, \ldots, s_\ell$ be chosen uniformly at random from $\{0,1\}^\ell$ and, for each nonempty subset $I \subseteq \{1, \ldots, \ell\}$, let $r_I = \oplus_{i \in I} s_i$. Then the $2^\ell - 1$ values $\{r_I\}$ are pairwise independent and uniformly distributed.

Proof: Board.

If we guess the ℓ bits $\{\mathsf{gl}(x, s_1), \ldots, \mathsf{gl}(x, s_\ell)\}$ correctly, then we also know the correct values of the $2^\ell - 1$ bits $\{\mathsf{gl}(x, r_I)\}$.

Inverting f

Given $g(x, r)$, run the following algorithm $\mathcal{A}'$.

- Set $n = |x|$ and $\ell = \lceil \log(2n/\epsilon(n)^2 + 1) \rceil$.
- Choose values $s_1, \ldots, s_\ell \in \{0, 1\}^n$ and guess the bits $\sigma_i = \mathsf{gl}(x, s_i)$.
- For every j and every nonempty subset I of $\{1, \ldots, \ell\}$, use $\mathcal{A}$ to produce a guess for $\mathsf{gl}(x, r_I \oplus e_j)$.
- Set x_j equal to the majority vote of $\{\sigma_j \oplus \mathsf{gl}(x, r_I \oplus e_j)\}$.
- Return $x = x_1 \ldots x_n$.

When Does $\mathcal{A}'$ Succeed?

The algorithm will succeed when the following events happen:

- The value x belongs to S_n.

- $\mathcal{A}'$ correctly guesses the ℓ values $\sigma_i = \mathsf{gl}(x, s_i)$.

- $\mathcal{A}$ computes the correct value of $\mathsf{gl}(x, r_I \oplus e_j)$ for a majority of the indices I.

Note that $\mathcal{A}'$ can also succeed if some of these events don't happen: Mod 2, two wrongs make a right!

Modern Cryptography

600.442

Lecture #11

Dr. Christopher Pappacena

Fall 2013

Last Time

- One-way functions
- Hardcore predicates
- Started to prove the Goldreich-Levin Theorem.

Goldreich-Levin Theorem

Let f be a one-way function and define g by $g(x, r) = (f(x), r)$ for $|r| = |x|$. Define gl$(x, r) = x \cdot r = \oplus_{i=1}^{n} x_i r_r$. Then gl$(x, r)$ is a hardcore predicate for g.

Proof

Given $\mathcal{A}$ which can guess $\mathsf{gl}(x, r)$ with advantage $\epsilon(n)$, run the following algorithm $\mathcal{A}'$:

- Set $\ell = \lceil \log(2n/\epsilon(n)^2 + 1) \rceil$.
- Choose $s_1, \ldots, s_\ell \in \{0,1\}^n$ and guess the bits $\sigma_i = \mathsf{gl}(x, s_i)$.
- For every j and every nonempty subset I of $\{1, \ldots, \ell\}$, use $\mathcal{A}$ to produce a guess for $\mathsf{gl}(x, r_I \oplus e_j)$.
- Set x'_j equal to the majority vote of $\{\sigma_j \oplus \mathsf{gl}(x, r_I \oplus e_j)\}$ and return x'.

When Does $\mathcal{A}'$ Succeed?

The algorithm will succeed when the following events happen:

- The value x belongs to S_n.
- $\mathcal{A}'$ correctly guesses the ℓ values $\sigma_i = \mathsf{gl}(x, s_i)$.
- $\mathcal{A}$ computes the correct value of $\mathsf{gl}(x, r_I \oplus e_j)$ for a majority of the indices I.

Note that $\mathcal{A}'$ can also succeed if some of these events don't happen: Mod 2, two wrongs make a right!

The probability of each of the first two events is easy to estimate:

- x is in S_n with probability $\geq \epsilon(n)/2$.
- Since $\ell \leq \log(2n/\epsilon(n)^2 + 1) + 1$, $\mathcal{A}'$ guesses the ℓ values correctly with probability

$$2^{-\ell} \geq \frac{1}{2\left(2n/\epsilon(n)^2 + 1\right)} \geq \frac{\epsilon(n)^2}{5n}.$$

Note that these events are independent.

A Leap of Faith

To estimate the probability that $\mathcal{A}$ computes the correct value of x_j we need to use a result which we will not prove:

Lemma: Suppose that $\{X_1, \ldots, X_t\}$ are pairwise independent binary random variables such that $\Pr[X_i = 1] \geq \frac{1}{2} + \epsilon$ for some ϵ. Let X be the majority vote of $\{X_1, \ldots, X_t\}$. Then

$$\Pr[X = 0] \leq \frac{1}{4\epsilon^2 t}.$$

The proof can be found in Katz and Lindell, pp. 208–210.

We can apply this result to bound the probability that $\mathcal{A}$ correctly determines the bit x_j, conditioned on x being in S_n and $\mathcal{A}'$ guessing $\{\mathsf{gl}(x, s_i)\}$ correctly.

For each subset I we let X_I be the random variable which is 1 if the estimate for x_j is correct, 0 otherwise. Then $\Pr[X_I = 1] \geq \frac{1}{2} + \frac{\epsilon(n)}{2}$.

So the probability that the majority vote gets x_j wrong is

$$\Pr[X = 0] \leq \frac{1}{4(\epsilon(n)/2)^2(2^\ell - 1)} \leq \frac{1}{2n}.$$

Hence the conditional probability that $x' = x$ is $\geq \frac{1}{2}$.

Putting it all Together

In total, the probability that $\mathcal{A}'$ correctly inverts f is at least

$$\frac{\epsilon(n)}{2} \times \frac{\epsilon(n)^2}{5n} \times \frac{1}{2} = \frac{\epsilon(n)^3}{20n}.$$

So, if $\epsilon(n)$ is non-negligible, then we can invert f with non-negligble probability. This shows that gl is a hardcore predicate for g.

Remarks

- We have *not* shown that f itself has a hard-core predicate. In fact, it is *unknown* whether a general one-way function has a hardcore predicate.

- Many cryptography references (including the textbook and the original Goldreich-Levin paper) are sloppy on this point.

- If $f(x)$ is a permutation, then so is $g(x, r) = (f(x), r)$, a fact we will use later.

- We have taken full advantage of the asymptotic method by not being too careful with our probability estimates.

Whew!

With that hard work behind us we can use one-way functions and hard-core predicates to construct pseudorandom generators, pseudorandom functions, and strong pseudorandom permutations.

Actually, we'll base our constructions on one-way *permutations* with hardcore predicates.

Pseudorandom Generators

As a warm-up, we will construct a pseudorandom generator G with an expansion factor of $\ell(n) = n + 1$.

Theorem: Let f be a one-way permutation with hardcore predicate hc. Then the function $G(s) = (f(s), \mathsf{hc}(s))$ is a pseudorandom generator.

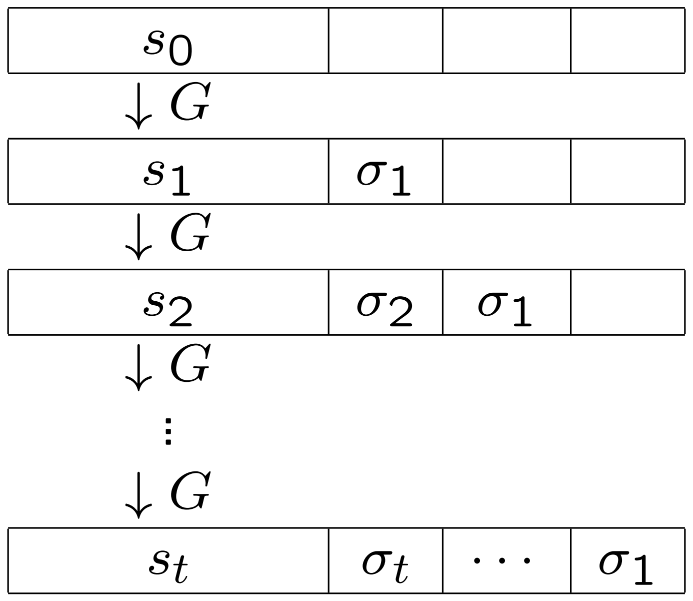
Proof: Intuitively, $f(s)$ is uniformly distributed and hc(s) looks random when all we can see is $f(s)$. So $G(s)$ looks random.

More detailed proof on board.

Bootstrapping

We can "bootstrap" a pseudorandom generator with $\ell(n) = n + 1$ to one with $\ell(n)$ any polynomial in n.

The following picture explains how we do this:



Theorem: The bootstrapping construction applied to G $p(n)$ times gives a pseudorandom generator $\tilde{G}$ with expansion factor $\ell(n) = n + p(n)$.

The proof of this theorem uses a technique called a *hybrid argument*:

- We define $p(n)+1$ different probability distributions $\{H_n^i\}$ on $\{0,1\}^{\ell(n)}$.
- H_n^0 is the distribution induced by $\tilde{G}$ and $H_n^{p(n)}$ is the uniform distribution.
- Distinguishing H_n^i from H_n^{i+1} amounts to distinguishing the output of G from random.

The Hybrid Construction

We define H_n^i as follows:

- Choose $s_i \leftarrow \{0,1\}^{n+i}$ uniformly at random.
- Run $\tilde{G}$ from iteration $i+1$ and output $s_{p(n)}$.

We see that H_n^0 is the distribution induced by $\tilde{G}$ and $H_n^{p(n)}$ is the uniform distribution on $\{0,1\}^{\ell(n)}$.

Given a distinguisher D, let $\epsilon(n)$ denote its advantage in telling the output of $\tilde{G}$ apart from random:

$$\begin{aligned}\epsilon(n) &= |\Pr_{s\leftarrow\{0,1\}^n}[D(\tilde{G}(s) = 1] - \Pr_{r\leftarrow\{0,1\}^{\ell(n)}}[D(r) = 1]| \\ &= |\Pr_{s_{p(n)}\leftarrow H_n^0}[D(s_{p(n)}) = 1] - \Pr_{s_{p(n)}\leftarrow H_n^{p(n)}}[D(s_{p(n)}) = 1]|\end{aligned}$$

Given D, we define a new distinguisher D' as follows. Given an input $w \in \{0,1\}^{n+1}$, do the following:

- Choose $i \leftarrow \{1, \ldots, p(n)\}$ uniformly.
- Choose $\sigma_i \leftarrow \{0,1\}^{i-1}$ uniformly.
- Set $s_i = (w, \sigma_i)$, run $\tilde{G}$ starting at iteration $i+1$ to compute $s_{p(n)}$, and output $D(s_{p(n)})$.

D' is a Distinguisher for G:

We have

$$\Pr_{w \leftarrow \{0,1\}^{n+1}}[D'(w) = 1] = \frac{1}{p(n)} \sum_{i=1}^{p(n)} \Pr_{s_{p(n)} \leftarrow H_n^i}[D(s_{p(n)}) = 1]$$

and

$$\Pr_{s \leftarrow \{0,1\}^n}[D'(G(s)) = 1] = \frac{1}{p(n)} \sum_{i=0}^{p(n)-1} \Pr_{s_{p(n)} \leftarrow H_n^i}[D(s_{p(n)}) = 1].$$

Details on board.

Putting these together gives:

$$|\,\mathsf{Pr}[D'(G(s)) = 1] - \mathsf{Pr}[D'(w) = 1]| = \frac{\epsilon(n)}{p(n)}.$$

Details on board.

Since G is pseudorandom, $\epsilon(n)$ must be negligible and $\tilde{G}$ is pseudorandom.

Pseudorandom Functions

With pseudorandom *generators* in hand, we can now construct pseudorandom *functions*.

Fix a pseudorandom generator $G : \{0,1\}^n \to \{0,1\}^{2n}$ and write $G(s) = (G_0(s), G_1(s))$ where G_0 and G_1 are each n bits long.

From G, we define a keyed function $F^{(1)} : \{0,1\}^n \times \{0,1\} \to \{0,1\}^n$ by $F_k^{(1)}(b) = G_b(k)$.

$F^{(1)}$ is pseudorandom, since a distinguisher which can tell the output of $F_k^{(1)}$ from random can tell the output of G from random.

Bootstrapping

We can bootstrap this basic construction to get a keyed function $F^{(m)}$: $\{0,1\}^n \times \{0,1\}^m \to \{0,1\}^n$ by setting

$$F_k^{(m)}(x_1 \ldots x_m) = G_{x_m}(F_k^{(m-1)}(x_1 \ldots x_{m-1}).$$

For example, with $m = 3$ we have

$$F_k^{(3)}(011) = G_1(G_1(G_0(k))).$$

The function $F_k^{(m)}$ can be viewed as a binary tree of depth m (picture on board).

Theorem: The keyed function $F^{(n)}$ is a pseudorandom function.

Proof: The proof uses a hybrid argument. Define a distribution H_n^i on binary trees of depth n as follows:

- For nodes at level $j \leq i$, the values are chosen uniformly at random from $\{0,1\}^n$.
- For nodes at level $j > i$, look at the value k' of the node's parent and assign $G_0(k')$ if it is a left child and $G_1(k')$ if it is a right child.

Note that H_n^n corresponds to a random function and H_n^0 corresponds to $F^{(n)}$ with a uniformly-chosen key.

Modern Cryptography

600.442

Lecture #12

Dr. Christopher Pappacena

Fall 2013

Last Time

- Fire Alarm
- Proved Goldreich-Levin Theorem
- One-way functions produce pseudorandom generators
- Pseudorandom generators produce pseudorandom functions

Constructing Pseudorandom Functions

- Fix a pseudorandom generator $G : \{0,1\}^n \to \{0,1\}^{2n}$ and write $G(s) = (G_0(s), G_1(s))$.

- Define a keyed function $F^{(1)} : \{0,1\}^n \times \{0,1\} \to \{0,1\}^n$ by $F_k^{(1)}(b) = G_b(k)$.

- Extend to $m > 1$ by setting
$$F_k^{(m)}(x_1 \ldots x_m) = G_{x_m}(F_k^{(m-1)}(x_1 \ldots x_{m-1}).$$

- $F_k^{(m)}$ can be viewed as a binary tree of depth m.

Theorem: The keyed function $F^{(n)}$ is a pseudorandom function.

Proof: The proof uses a hybrid argument. Define a distribution H_n^i on binary trees of depth n as follows:

- For nodes at level $j \leq i$, the values are chosen uniformly at random from $\{0,1\}^n$.

- For nodes at level $j > i$, look at the value k' of the node's parent and assign $G_0(k')$ if it is a left child and $G_1(k')$ if it is a right child.

Note that H_n^n corresponds to a random function and H_n^0 corresponds to $F^{(n)}$ with a uniformly-chosen key.

A Tale of Two Distinguishers

Let D be a distinguisher that can tell $F_k^{(n)}$ from a random function with advantage $\epsilon(n)$ and let $t(n)$ be the number of oracle queries that D makes.

We design a distinguisher D' which distinguishes outputs of G from random.

D' takes as input $t(n)$ strings of length $2n$ and chooses $i \to \{0, \ldots, n-1\}$ at random.

D' will run D as a subroutine and answer all oracle queries that D makes.

D' as Oracle

D' answers D's oracle queries as follows:

- On input $x_1 \ldots x_n$, D' uses $x_1 \ldots x_i$ to reach a node at level i in a binary tree.

- D' labels the left and right child of the node using one of its strings of length $2n$ and applies $G_{x_{i+1}}, \ldots, G_{x_n}$ to traverse down to the root.

- D' remembers the values of the tree it has filled in and answers consistently.

- D' only has to generate and store polynomially-many entries in its tree.

D's Advantage Becomes D''s Advantage

If D' is given $t(n)$ random strings of length $2n$, then it answers D's oracle queries with a function sampled from the distribution H_n^{i+1}.

If D' is given $t(n)$ outputs of the pseudorandom generator G, then it answers D's oracle queries with a function sampled from the distribution H_n^i.

As with the previous hybrid argument, D' succeeds in distinguishing the output of G from random with advantage $\epsilon(n)/n$.

This shows that $\epsilon(n)$ is negligible so that $F^{(n)}$ is a pseudorandom function.

Pseudorandom Permutations

We can use pseudorandom functions to build pseudorandom permutations and strong pseudorandom permutations via a Feistel network.

Let $F : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ be a pseudorandom function.

Given a sequence $k = (k_1, \ldots, k_r)$ of r keys in $\{0,1\}^n$ and $x \in \{0,1\}^{2n}$, we can apply an r-round Feistel network with round functions F_{k_i} to x.

This construction yields a keyed function with key length rn and input and output length $2n$.

Feistel Networks Work

Theorem: If F is pseuodrandom, then a three-round Feistel network is a pseudorandom permutation on $2n$-bit strings with $3n$-bit keys.

Theorem: If F is pseudorandom, then a four-round Feistel network is a strong pseudorandom permutation on $2n$-bit strings with $4n$-bit keys.

Proofs are omitted. The fact that 2 rounds don't work and that 3 rounds don't give strong pseudorandom permutations are exercises in Katz and Lindell.

Recap So Far

So far we have demonstrated:

- One-way permutations imply pseudorandom generators.
- Pseudorandom generators imply pseudorandom functions.
- Pseudorandom functions imply strong pseudorandom permutations.

These are all that we need for secure private-key cryptography!

Theorem: If there exists a one-way permutation, then there exists a CCA-secure encryption scheme and a MAC that is existentially unforgeable under a chosen message attack.

One Way Functions Imply Private-Key Cryptography

In fact, it is possible to construct a pseudorandom generator from a one-way *function*.

Thus the existence of one-way functions implies the existence of secure cryptography.

Remarkably, the converse is true!

Private-Key Cryptography Implies One Way Functions

Theorem: If there exists a private-key encryption scheme with indistinguishable encryptions in the presence of an eavesdropper, then there exists a one-way function.

The proof will use the fact that messages longer than the key are allowed.

So this result does not contradict the unconditional security of the one-time pad.

Proof of the Theorem

Let $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a private-key encryption scheme with indistinguishable encryptions in the presence of an eavesdropper.

To allow for Enc to be a randomized algorithm we write $c = \mathsf{Enc}_k(m, r)$ where r is a random value.

Assume that when an n bit key is used to encrypt a $2n$-bit message, then r has length $\ell(n)$.

The One Way Function

Define $f(m, k, r) = (\mathsf{Enc}_k(m, r), m)$.

- Note that f is efficiently computable beacuse $\mathsf{Enc}_k(m, r)$ is.

- We need to show that it is hard to invert.

- Intuitively, inverting f should require breaking Π. We will formalize this intuition.

The Adversary

Let $\mathcal{A}$ be a PPT algorithm that inverts f with success probability $\epsilon(n)$. We will construct an adversary $\mathcal{A}'$ for Π which uses $\mathcal{A}$ as a subroutine.

- $\mathcal{A}'$ chooses m_0 and m_1 of length $2n$ and receives the challenge ciphertext c from Π.

- $\mathcal{A}'$ gives (c, m_0) to $\mathcal{A}$ and receives back (m, k, r).

- $\mathcal{A}'$ guesses $b = 0$ if $f(m, k, r) = (c, m_0)$; otherwise $\mathcal{A}'$ outputs a random bit.

Success Probability

When $b = 0$, $\mathcal{A}$ inverts (c, m_0) with probability $\epsilon(n)$. In this case $\mathcal{A}'$ returns the correct answer. If $\mathcal{A}$ fails to invert (c, m_0), $\mathcal{A}'$ still gets the correct answer with probability 1/2. So

$$\Pr[\mathsf{PrivK}^{\mathsf{eav}}_{\Pi,\mathcal{A}'}(n) = 1 | b = 0] = \epsilon(n) + \frac{1}{2}(1 - \epsilon(n)) = \frac{1}{2} + \frac{\epsilon(n)}{2}.$$

When $b = 1$, $\mathcal{A}$ can still successfully invert (c, m_0). This means that for some different key k' and random string r', we have $c = \mathsf{Enc}_k(m_1, r) = \mathsf{Enc}_{k'}(m_0, r')$.

How likely is this to happen?

There are only 2^n messages which encrypt to c, namely the decryptions of c with all 2^n possible keys.

Since there are 2^{2n} messages of length $2n$, c will be a valid ciphertext for m_0 with probability 2^{-n}. So,

$$\Pr[\mathsf{PrivK}^{\mathsf{eav}}_{\Pi,\mathcal{A}'}(n) = 1 | b = 1] \geq \frac{1}{2}(1 - 2^{-n}) = \frac{1}{2} - \frac{1}{2^{n+1}}.$$

Putting this all together gives

$$\Pr[\mathsf{PrivK}^{\mathsf{eav}}_{\Pi,\mathcal{A}'}(n) = 1] \geq \frac{1}{2} + \frac{\epsilon(n)}{4} - \frac{1}{2^{n+2}}.$$

Since Π has indistinguishable encryptions in the presence of an eavesdropper, $\epsilon(n)$ is negligible and f is a one-way function.

Practical Considerations

So why do we use AES to instantiate a pseudorandom permutation, or RC4 to instantiate a pseudorandom generator?

Why not create cryptography from some NP-complete problem like the subset sum problem, or use another hard problem like factoring?

Because these would be *horribly* inefficient – running orders of magnitude more slowly!

Closing Observations

- It is also possible to prove the existence of one-way functions from secure MACs.

- We have not constructed collision-resistant hash functions from one-way functions.

- No construction is known and there is evidence that no such construction exists.

- Similarly, one-way functions do not appear to be enough to accomplish *public-key cryptography*, the topic of the second half of the course.

We have concluded the private-key portion of the course.

Next time, we will discuss public-key cryptography, which:

- Revolutionized cryptography in the 1970s.
- Gave birth to the science of modern cryptography.
- Pervades all aspects of digital life, for example enabling e-commerce.

Modern Cryptography

600.442

Lecture #13

Dr. Christopher Pappacena

Fall 2013

Private Key Cryptography

- Private key cryptography can be used to provide secure, authenticated communication.
- Its use depends on the communcants being able to agree on a shared secret key.
- How do the communicants decide on their shared key?

Three Challenges

- Key Distribution: How do communicants establish their private keys?
- Key Management: How do individuals communicate with many other people?
- Transient Communcations: How do individuals with no prior relationship communicate securely?

Key Distribution

In a *closed* environment (e.g. a company), key distribution can be handled *physically*:

- Each new person, when hired, is issued keys to communicate with other employees.
- Unwieldy, since a new employee needs n keys if the company employs n people.
- Current employees also need to update their key lists as new employees are enrolled.
- These keys must all be stored securely.

Key Distribution Centers

In a closed environment, a *key distribution center* (KDC) can eliminate lots of these problems.

- Each employee, when hired, is issued private keys (encryption and authentication) to communicate securely with the KDC.

- When employees want to communicate securely, they contact the KDC to receive a *session key*.

Basic KDC Protocol

- When Alice wants to communicate securely with Bob, she contacts the KDC.
- The KDC generates a session key k and sends Alice $(\mathsf{Enc}_A(k), \mathsf{Enc}_B(k))$, where A and B are Alice and Bob's keys.
- Alice decrypts $\mathsf{Enc}_A(k)$ to obtain k and uses it to encrypt her message to Bob.
- Alice sends Bob $\mathsf{Enc}_B(k)$ and $\mathsf{Enc}_k(m)$ where m is her message.
- Authentication can be accomplished in the same fashion.

KDCs – Pros and Cons

- Each employee only needs to securely store a single key, making key distribution and management easier for individuals.

- The KDC has access to everyone's key, making it a target for adversaries.

- The KDC is a single point of failure for secure communication throughout the company – if it goes down, secure communication becomes impossible.

- KDCs can be replicated to increase resiliency, but this introduces more targets for adversaries.

In practice, KDCs are used to provide enterprise-level secure communications.

But they do not address the problem of enabling *transient* secure communication:

> *How can two people communicate securely over a public channel without having previously established a secret key and without access to a private channel to establish this secret?*

Until the mid 1970's, accomplishing this was considered *impossible*.

Diffie-Hellman and the Public Key Revolution

In 1976, Diffie and Hellman published a paper titled "New Directions in Cryptography".

In it, they introduced three public-key primitives:

- Public-Key Encryption
- Digital Signatures
- Interactive Key Exchange

Diffie and Hellman presented a solution to the last of these, now known as the *Diffie-Hellman Key Exchange*.

Original Diffie-Hellman Key Exchange

- Alice chooses a large prime p and a *generator** g.
- Alice chooses $x \leftarrow \{1, \ldots, p-1\}$ and computes $A = g^x \pmod{p}$.
- Alice sends p, g, and A to Bob.
- Bob chooses $y \leftarrow \{1, \ldots, p-1\}$, computes $B = g^y \pmod{p}$, and sends B to Alice.
- Alice and Bob both know $g^{xy} = A^y = B^x \pmod{p}$.

*We'll define this later.

Diffie and Hellman's work preceded modern notions of cryptographic security so they did not provide a formal proof of security.

They did observe that, for the key exchange to be secure, *at a minimum* the following problem must be hard:

The Discrete Logarithm Problem: Given p, g, and $A = g^x \pmod{p}$, determine x.

This is because an eavesdropper sees both $A = g^x$ and $B = g^y$ during the exchange.

We'll discuss this problem in more generality later.

Rivest-Shamir-Adleman and ElGamal

- In 1978, Rivest, Shamir, and Adleman presented solutions to public-key encryption and digital signatures.

- RSA encryption remains the most widely-used public-key algorithm in the world.

- In 1985, ElGamal created public-key encryption and digital signature algorithms based on the same underlying problem as the Diffie-Hellman key exchange.

In short, Diffie and Hellman revolutionized cryptography, created the field of public-key cryptography, and came very close to providing solutions to all of the basic problems that they introduced.

Public-key algorithms are built using operations that are easy to implement, but hard to undo:

- Given integers g and x and a prime p it is easy to compute g^x (mod p), but given g^x (mod p) it is difficult to determine x.

  This dichotomy is the heart of the Diffie-Hellman key exchange and El Gamal encryption and digital signatures.

- It is easy to multiply two primes togther, but difficult to factor an integer which is the product of two large primes.

  This dichotomy is the heart of RSA encryption and digitial signatures.

To understand these problems, we need to do some number theory.

Elementary Number Theory

Division Algorithm: Let a be an integer and let b be a positive integer. Then there exist integers q, r, with $0 \leq r < b$, such that $a = bq + r$.

This just says we can divide a by b and get a remainder which is less than b.

If a and b are nonnegative integers, then their *greatest common divisor* (GCD) is the largest integer c which divides both a and b. We write $c = (a, b)$ for the GCD of a and b.

Example: $(12, 16) = 4$.

Lemma: Let $c = (a, b)$. Then there exist integers x, y such that

$$ax + by = c,$$

and c is the smallest positive integer which can be written in this form.

Example: $12 \cdot (-1) + 16 \cdot 1 = 4$, so $x = -1$ and $y = 1$ works. Taking $x = 15$ and $y = -11$ also works.

We write $\|a\|$ for the number of bits in a, which is the relevant quantity for algorithmic complexity.

Given integers a and b, we can compute $c = (a, b)$ and integers x, y such that $c = ax + by$ in time polynomial in $\max(\|a\|, \|b\|)$ using the *extended Euclidean algorithm*

Modular Arithmetic

Let N be a positive integer. We write $a = b \pmod{N}$, or $a \equiv b \pmod{N}$, if $a - b$ is divisible by N.

This is equivalent to saying that a and b have the same remainder when divided by N.

Example: $345678722566563473 = 544574598645 29373 \pmod{100}$.

Reduction modulo N respects addition, subtraction, and multiplication: if $a = a' \pmod{N}$ and $b = b' \pmod{N}$, then $a \pm b = a' \pm b' \pmod{N}$ and $ab = a'b' \pmod{N}$.

In general, we *cannot* divide modulo N: $3 \cdot 2 = 15 \cdot 2 \pmod{24}$ but $15 \neq 3 \pmod{24}$.

If a is an integer, then there exists a *unique* integer r with $0 \leq r \leq N-1$ such that $a = r \pmod{N}$ – namely, the remainder when we divide a by N.

We write $\mathbb{Z}_N$ for $\{0, \ldots, N-1\}$ with arithmetic defined modulo N. The fact that we can add, subtract, and multiply essentially says that $\mathbb{Z}_N$ is a *ring*.

An important computational fact is that we can reduce modulo N "as we go" when doing computations.

Addition and multiplication in $\mathbb{Z}_N$ can be computed in polynomial time in $\|N\|$.

Units in $\mathbb{Z}_N$

An element u of $\mathbb{Z}_N$ is called a *unit* if it has a multiplicative inverse v, so that $uv = 1 \pmod{N}$. The set of units in $\mathbb{Z}_N$ is written $\mathbb{Z}_N^*$.

Example: 7 is a unit modulo 10 since $7 \cdot 3 = 1 \pmod{10}$. In fact, $\mathbb{Z}_{10}^* = \{1, 3, 7, 9\}$.

Proposition: u is a unit modulo N if and only if $(u, N) = 1$.

Proof: If $(u, N) = 1$, write $ux + Ny = 1$. Then $x = u^{-1} \pmod{N}$.

Groups

A *group* G is a set with a binary operation $\circ$ satisfying three axioms:

- Associativity: $g \circ (h \circ k) = (g \circ h) \circ k$.
- Identity: There exists an element e such that $e \circ g = g \circ e = g$ for all g.
- Inverse: For every g, there exists h such that $g \circ h = h \circ g = e$. We write $h = g^{-1}$.

If $g \circ h = h \circ g$ for all $g, h \in G$, we call G an *abelian group*. We will *only* work with finite abelian groups.

Examples

- The set of integers modulo N, $\mathbb{Z}_N$, with operation addition, is an abelian group.

  The identity element is 0 and the inverse of a is $N - a$:

  $$a + (N - a) = N = 0 \pmod{N}.$$

- The set of units modulo N, $\mathbb{Z}_N^*$, with operation multiplication, is an abelian group.

  The identity element is 1 and the fact that every element has an inverse follows from the definition of “unit”.

Modern Cryptography

600.442

Lecture #13

Dr. Christopher Pappacena

Fall 2013

Last Time

- The limitations of private-key cryptography
- Diffie-Hellman key exchange, informally
- The discrete log problem
- Number theory

Group Exponentiation

If G is a group with operation $\circ$, we define g^m to be the result of applying the group operation to g a total of m times:

$$g^m = g \circ \cdots \circ g \text{ (m times).}$$

If the group operation in G is addition we write mg in place of g^m.

All the usual rules of exponentiation hold: $g^0 = e$, $g^m \circ g^n = g^{m+n}$, and $g^{-m} = (g^{-1})^m$.

Proposition: g^m can be computed in $O(\|m\|)$ group operations.

This is known as “double and add” or “square and multiply”.

Lagrange's Theorem

The following result from group theory is very important to cryptography:

Theorem: Let G be a finite group, say $|G| = n$. Then $g^n = e$ for every $g \in G$.

Proof: (G abelian) Multiplication by g is permutation of the elements of G. If we write $G = \{g_1, \ldots, g_n\}$ then

$$g_1 \circ \cdots \circ g_n = (g \circ g_1) \circ \cdots \circ (g \circ g_n) = g^n \circ g_1 \circ \cdots \circ g_n$$

Canceling $g_1 \circ \cdots \circ g_n$ gives $g^n = e$.

Cyclic Groups

A group G is called *cyclic* if there exists an element g, called a *generator* for G, such that every element of G is a power of g. We write $G = \langle g \rangle$.

Example: The additive group $\mathbb{Z}_N$ is cyclic with generator 1.

Example: The additive group $\mathbb{Z}_{10}$ is cyclic with generator 3 (board).

If G is a group and $g \in G$, then the set of powers of g forms a subgroup of G called the *cyclic subgroup generated by* g. The *order* of g is the smallest positive m with $g^m = e$.

Example: The element 2 has order 5 in $\mathbb{Z}_{10}$ since $\langle 2 \rangle = \{0, 2, 4, 6, 8\}$.

Important Facts About Cyclic Groups

- If g has order m then $g^x = g^y$ if and only if $x = y \pmod{m}$.

- If G has order n and $g \in G$ has order m, then m divides n.

- If G has prime order p then G is cyclic.

- If p is a prime integer then $\mathbb{Z}_p^*$ is cyclic of order $p - 1$.

Example: $\mathbb{Z}_7^*$ is generated by 3 (board).

The Discrete Logarithm Problem

If $G = \langle g \rangle$ is a cyclic group of order n, then every $h \in G$ is a power of g: $h = g^x$ for some $x \in \{0, \ldots, n-1\}$.

The *discrete logarithm problem* is to find x, given h.

The difficulty of the discrete logarithm problem does not depend on G as an *abstract* group; rather, it depends on how G is represented.

Example: The group $\mathbb{Z}_p^*$ is a cyclic group with generator 3 for $p = 2147483647$. The group $\mathbb{Z}_{p-1}$ under addition is also a cyclic group of order $p-1$, with generator 1.

The discrete logarithm of 2041619674 in the first group is 62 and in the second group it is 2041619674.

The Discrete Logarithm Experiment

Let $\mathcal{G}$ be an algorithm which, on input n, returns a cyclic group G of order q (with $\|q\| = n$) and a generator g. Consider the following experiment:

- Run $\mathcal{G}(n)$ to obtain (G, q, g).
- Choose $h \leftarrow G$ uniformly at random. This can be done by choosing $y \leftarrow \{0, \ldots, q-1\}$ uniformly at random and setting $h = g^y$.
- $\mathcal{A}$ is given (G, q, g, h) and outputs $x \in \{0, \ldots, q-1\}$.
- We say $\mathcal{A}$ *succeeds* if $h = g^x$, otherwise $\mathcal{A}$ *fails*.

We write $\mathsf{DLog}_{\mathcal{A},\mathcal{G}}(n) = 1$ if $\mathcal{A}$ succeeds, and 0 if $\mathcal{A}$ fails.

Definition: The *discrete logarithm problem is hard relative to* $\mathcal{G}$ if, for all PPT adversaries $\mathcal{A}$, we have

$$\Pr[\mathsf{DLog}_{\mathcal{A},\mathcal{G}}(n) = 1] \leq \mathtt{negl}(n)$$

for some negligible function `negl` of n.

Example: On input n, $\mathcal{G}$ selects a prime p with $\|p\| = n$ such that $(p-1)/2$ prime, finds a generator g for $\mathbb{Z}_p^*$, and returns $(\mathbb{Z}_p^*, p-1, g)$.

The discrete logarithm problem is widely believed to be hard relative to $\mathcal{G}$.

The Diffie-Hellman Problem

In the original Diffie-Hellman protocol, the shared secret is g^{xy} and the eavesdropper observes g^x and g^y. The difficulty of recovering the former from the latter is known as the *Diffie-Hellman problem*.

Formally, there are two variants:

- Computational Diffie-Hellman (CDH): Given g^x and g^y, compute g^{xy}.

- Decisional Diffie-Hellman (DDH): Given a triple (g^x, g^y, h), decide whether or not $h = g^{xy}$.

If we can solve the discrete log problem in G then we can solve CDH and DDH, and if we can solve CDH then we can solve DDH.

(These are both problems on HW 7.)

So DDH is the strongest of the three problems.

The DDH Problem, Formally

Definition: We say that the *DDH problem is hard realative to $\mathcal{G}$* if for all PPT algorithms $\mathcal{A}$, there exists a negligible function $\texttt{negl}(n)$ such that

$$|\Pr[\mathcal{A}(\mathcal{G}, g, g^x, g^y, g^{xy}) = 1] - \Pr[\mathcal{A}(\mathcal{G}, g, g^x, g^y, h)]| \leq \texttt{negl}(n)$$

where $x, y \leftarrow \{0, \ldots, q-1\}$ and $h \leftarrow \mathcal{G}$.

Key Exchange, Formally

Consider the following *key exchange experiment* $\mathsf{KE}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n)$:

- Two parties execute a key-exchange protocol Π with input n. The protocol produces a transcript of all messages sent between the two parties as well as a secret key k.

- A random bit $b \leftarrow \{0,1\}$ is chosen. If $b = 0$ set $\tilde{k} \leftarrow \{0,1\}^n$, otherwise set $\tilde{k} = k$.

- $\mathcal{A}$ is given $\tilde{k}$ and the transcript and outputs a bit b'.

- We set $\mathsf{KE}^{\mathsf{eav}}_{\mathcal{A},\Pi}(n) = 1$ if $b = b'$ and say that $\mathcal{A}$ *succeeds*; otherwise $\mathcal{A}$ *fails*.

Diffie-Hellman, Formally

- Given n, Alice runs $\mathcal{G}(n)$ to obtain (G, q, g).

- Alice chooses $x \leftarrow \{0, \ldots q-1\}$, sets $A = g^x$, and sends (G, q, g, A) to Bob.

- Bob chooses $y \leftarrow \{0, \ldots, q-1\}$, sets $B = g^y$, and sends B to Alice.

- The shared key is $k = B^x = A^y$ and the transcript is (G, q, g, A, B).

A Technical Detail

The definition of a key exchange protocol produces a shared secret key which is a bit string. The Diffie-Hellman protocol produces a shared secret element of the group G.

We need a way to extract bit strings from group elements.

In practice this is not too hard, since the group elements are represented by bit strings on a computer. Sometimes we have to be a bit careful because some of the individual bits which represent group elements may be biased.

Theorem: If the decisional Diffie-Hellman problem is hard relative to $\mathcal{G}$, then the Diffie-Hellman Key Exchange protocol is secure in the presence of an eavesdropper.

Proof: Board.

DDH being hard relative to $\mathcal{G}$ is exactly what is needed to make the proof work out.

Note that this is a strictly stronger assumption than the discrete logarithm problem or the computational Diffie-Hellman problem, originally identified by Diffie and Hellman.

Surprisingly, the DDH problem is actually *easy* relative to $\mathcal{G} = \mathbb{Z}_p^*$, the group we introduced earlier for which the discrete log problem is believed to be hard!

This suggests that the DDH problem is *strictly* stronger than the discrete log problem.

Quadratic Residues

Let p be a prime and a an element of $\{1, \ldots, p-1\}$. We say that a is a *quadratic residue* if there exists x such that $x^2 = a \pmod{p}$. Otherwise a is a *quadratic nonresidue*.

Example: For $p = 11$ the quadratic residues are $\{1, 3, 4, 5, 9\}$ and the quadratic nonresidues are $\{2, 6, 7, 8, 10\}$.

Theorem: There are exactly $(p-1)/2$ quadratic residues and exactly $(p-1)/2$ quadratic nonresidues modulo p.

Define the *Legendre Symbol* $\left(\frac{a}{p}\right)$ by $\left(\frac{a}{p}\right) = 1$ if a is a quadratic residue modulo p and $\left(\frac{a}{p}\right) = -1$ if a is a quadratic nonresidue.

Quadratic Residues and DDH

Basic facts about Legendre Symbols:

- $\left(\frac{a}{p}\right)$ can be computed in polynomial time in $\|p\|$.
- $\left(\frac{ab}{p}\right) = \left(\frac{a}{p}\right)\left(\frac{b}{p}\right)$.

If g is a generator for $\mathbb{Z}_p^*$ then $\left(\frac{g^x}{p}\right) = 1$ if and only if x is even. Using this fact we can determine $\left(\frac{g^{xy}}{p}\right)$ from g^x and g^y.

This means we can use Legendre Symbols to solve the DDH problem in $\mathbb{Z}_p^*$ with probability 3/4 – details on board.

There is a simple fix for this - the set of all quadratic residues forms a cyclic subgroup of $\mathbb{Z}_p^*$ of order $(p-1)/2$, so we modify $\mathcal{G}$ to return a generator for this group:

On input n, $\mathcal{G}$ selects a prime p with $\|p\| = n$ such that $(p-1)/2$ is prime. It selects a generator h for $\mathbb{Z}_p^*$, sets $g = h^2$, sets G equal to the group of quadratic residues modulo p, and returns $(G, (p-1)/2, g)$.

The DDH problem, CDH problem, and discrete log problem are all believed to be hard relative to this $\mathcal{G}$.

Modern Cryptography

600.442

Lecture #15

Dr. Christopher Pappacena

Fall 2013

Last Time

- Diffie-Hellman Key Exchange
- Discrete Logarithm, CDH, and DDH Problems
- We gave a group generation $\mathcal{G}$ algorithm for which the DDH problem is believed to be hard:

RSA Encryption

The first, and most widely used, public-key encryption scheme is RSA encryption, named after Rivest, Shamir, and Adelman.

We'll talk about RSA before giving a formal definition of public-key encryption.

The construction builds on the number theory we used for Diffie-Hellman.

More Number Theory

RSA encryption uses the group $\mathbb{Z}_N^*$ where $N = pq$ is the product of two primes.

The first question we can ask about $\mathbb{Z}_N^*$, for *any* integer N, is: What is its order?

Since $a \in \mathbb{Z}_N^*$ if and only if $(a, N) = 1$, we know that $|\mathbb{Z}_N^*|$ is equal to the number of integers a in $\{1, \ldots, N-1\}$ with $(a, N) = 1$.

Euler's Phi Function

The integer $|\mathbb{Z}_N^*|$ is denoted by $\phi(N)$ and ϕ is called *Euler's phi function*. For example, if p is prime then $\phi(p) = p - 1$.

Proposition: If $N = pq$ then $\phi(N) = (p-1)(q-1)$.

Proof: An integer is relatively prime to N provided it is not a multiple of p or a multiple of q. There are $q-1$ nonzero multiples of p in $\mathbb{Z}_N$: $p, 2p, \ldots, (q-1)p$. Similarly, there are $p-1$ nonzero multiples of q, namely $q, 2q, \ldots, (p-1)q$. So

$$\phi(N) = (N-1) - (p-1) - (q-1) = pq - (p+q) + 1 = (p-1)(q-1).$$

A General Formula

Theorem: If $N = p_1^{r_1} \times \cdots \times p_n^{r_n}$ is the prime factorization of N, then

$$\phi(N) = \prod_{i=1}^{n} p_i^{r_i - 1}(p_i - 1).$$

An important thing to note is that computing $\phi(N)$ is easy, *provided we know the factorization of* N.

In general, there is no known way to compute $\phi(N)$ which does *not* require knowing the factorization of N.

This fact is critical for RSA encryption as well as several generalizations, including Paillier encryption.

How Do We Find Primes?

The Diffie-Hellman and RSA generation schemes require us to find n-bit primes for some security parameter n. How do we do this?

Prime Number Theorem: Let $\pi(x)$ equal the number of primes $\leq x$. Then $\pi(x) = O(x/\ln x)$.

In fact, much sharper statements about the distribution of primes are known.

For us, we simply note that if you write down an n-bit integer *at random*, it will be prime with probability about $1.44/n$.

So the expected run time for the "random guess" algorithm is $O(n)$ primality tests.

Primality Testing

Given an integer N, how can we tell if it is prime?

Here are a couple of "easy out" tests:

- If N is divisible by a small prime, then N is composite. So simple *trial division* can quickly rule out most numbers.

- If N is prime then we know that $a^{N-1} = 1 \pmod{N}$. So we can choose a random a and check if this identity holds. If not, then N is composite. The integer a is called a *witness* for N.

There are infinitely many composite integers N for which $a^{N-1} = 1 \pmod{N}$ (*Carmichael numbers*). These numbers do not have *any* witnesses.

Strong Witnesses

Write $N - 1 = 2^r u$ for some $r \geq 1$ and odd u. Given $a \in \mathbb{Z}_N^*$, consider the sequence

$$(a^u, a^{2u}, \ldots, a^{2^r u}).$$

Three possibilities for this sequence modulo N:

- $(\pm 1, 1, \ldots, 1)$.
- $(*, \ldots, *, -1, 1, \ldots, 1)$
- $a^u \neq \pm 1 \pmod{N}$ and $a^{2^i u} \neq -1 \pmod{N}$ for $1 \leq i \leq r - 1$. In this case a is called a *strong witness* for N.

Strong Witnesses

As the name suggests, if a is a witness then a is a strong witness.

Also, if N is prime then N does not have any strong witnesses: one of the first two cases *must* happen.

Theorem: If N is an odd composite which is not a power of a prime, then at least half of the elements in $\mathbb{Z}_N^*$ are strong witnesses for N.

If we choose t elements of $\mathbb{Z}_N^*$ at random and none of them are strong witnesses for N, then the probability that N is composite is at most 2^{-t}.

The Miller-Rabin Primality Test

Given input N and t:

- If N is even or a perfect power, output “composite”.
- Write $N - 1 = 2^r u$ with u odd and $r \geq 1$.
- For $j = 1, \ldots, t$, do:
  - Choose $a \leftarrow \{1, \ldots, N-1\}$. If $(a, N) \neq 1$, return “composite”.
  - If a is a strong witness, return “composite”.
- Return “prime”.

Theorem: The Miller-Rabin test runs in time polynomial in $n = \|N\|$ and t. If N is prime, it returns "prime" with probability 1. If N is composite, it returns "prime" with probability at most 2^{-t}.

To generate a random n-bit prime, we select a random n-bit integer and test it for primality. This algorithm runs in probabilistic polynomial time since we expect to try $O(n)$ integers and each trial is polynomial in n.

In fact, there is a *deterministic* polynomial-time primality testing algorithm (Agrawal, Kayal, and Saxena, 2004).

In practice, the Miller-Rabin test runs faster and produces primes with a high degree of confidence.

Alternative Approach

The advantage to choosing n-bit integers uniformly and testing for primality is that it samples *uniformly* from n-bit primes.

It is easier in practice to select an n-bit integer k and find the *next prime*. We can use the Sieve of Eratosthenes to eliminate nearby composites and reduce the number of Miller-Rabin tests.

This method runs faster and requires fewer random bits.

We write $p = \text{NextPrime}(k)$ to denote the first prime which is $\geq k$.

Generating RSA Moduli

Define a PPT algorithm GenMod(n) as follows:

- On input n, GenMod(n) finds two random n-bit probable primes p and q using the Miller-Rabin test.
- The parameter t in the Miller-Rabin test is chosen so that 2^{-t} is a negligible function of n.
- GenMod(n) returns (p, q, N).

We can also define GenMod by calling the NextPrime function if we prefer.

The Factoring Experiment

- GenMod(n) is run to produce (p, q, N).
- The adversary $\mathcal{A}$ is given N and returns p', q'.
- $\mathcal{A}$ *succeeds* if $N = p'q'$ and *fails* otherwise. We write $\text{Fac}_{\mathcal{A}}(n) = 1$ if $\mathcal{A}$ succeds.

We say that *factoring is hard* relative to GenMod if $\Pr[\text{Fac}_{\mathcal{A}}(n) = 1]$ is negligible for all PPT adversaries $\mathcal{A}$.

Factoring Is Hard (We Think)

- It is widely believed that factoring is hard relative to GenMod(n).
- As with the hardness of DDH for $\mathcal{G}(n)$, we do not have a *proof* that factoring is hard.
- Unconditional proofs of either of these facts would imply $P \neq NP$.
- These problems (especially factoring) have been studied for a long time, which adds to our confidence that they are hard.

How Hard Is Factoring?

How hard is it to factor $N = pq$, with p and q both $O(\sqrt{N})$?

Selected Methods

Method	Year	Complexity
Trial Division	$-\infty$	$O(\sqrt{N})$
Fermat's Method	1600s	$O(\sqrt[3]{N})$
Pollard Rho	1975	$O(\sqrt[4]{N})$
Continued Fractions	1931, 1975	$L_N(1/2, \sqrt{2})$
Quadratic Sieve	1981	$L_N(1/2, 1)$
Number Field Sieve	1991	$L_N(1/3, \sqrt[3]{64/9})$

RSA Encryption

- Run GenMod(n) to produce (p, q, N).

- Choose e (the *encryption exponent*) relatively prime to $\phi(N)$ and publish (N, e) as the public key.

- Find d (the *decryption exponent*) satisfying $ed = 1 \pmod{\phi(N)}$ using the Euclidean algorithm.

- The private key (also called the secret key) is (p, q, d).

RSA Encryption and Decryption

- To encrypt a message $m \in \mathbb{Z}_N^*$, set $c = m^e \pmod{N}$.
- To decrypt a ciphertext c, set $m = c^d \pmod{N}$.
- This works because

$$(m^e)^d = m^{ed} = m^{ed \bmod \phi(N)} = m^1 = m \pmod{N}.$$

- We can also use the *Chinese Remainder Theorem* to compute $c^d \pmod{p}$ and $c^d \pmod{q}$ and use them to reconstruct $c^d \pmod{N}$. This offers no cryptographic advantage but is cheaper to compute.

Security of RSA

- For RSA Encryption to be secure, it must be hard for an adversary $\mathcal{A}$ to determine m from $m^e \pmod N$.
- If $\mathcal{A}$ can determine d, then he can read the message.
- Knowing d allows $\mathcal{A}$ to compute $\phi(N)$ (board).
- Given $\phi(N)$, it is possible to factor N (HW problem).
- On the other hand, factoring N allows $\mathcal{A}$ to compute $\phi(N)$, and then d.

Security of RSA, II

- We conclude that factoring N is equivalent to determining d.

- But, is it possible to recover m from $m^e \pmod{N}$ *without* determining d? Nobody knows.

- For RSA to be secure, it must be difficult to recover m from $m^e \pmod{N}$, regardless of how it is done.

- This is called the *RSA Problem*.

The RSA Experiment

Let GenRSA be a PPT algorithm which, on input n, returns (N, e, d) where $N = pq$ is an RSA modulus.

- Run GenRSA(n) to obtain (N, e, d) and choose $y \leftarrow \mathbb{Z}_N^*$ uniformly.

- $\mathcal{A}$ is given N, e, y, and outputs $x \in \mathbb{Z}_N^*$.

- We say $\mathsf{RSAInv}_{\mathcal{A}}(n) = 1$ if $x^e = y \pmod{N}$ and say that $\mathcal{A}$ *succeeds*. Otherwise $\mathcal{A}$ *fails*.

Definition: We say that *the RSA problem is hard relative to* GenRSA if, for all PPT adversaries $\mathcal{A}$, we have

$$\Pr[\mathsf{RSAInv}_{\mathcal{A}}(n) = 1] = \mathtt{negl}(n)$$

for some negligibe function $\mathtt{negl}$.

As mentioned above, it is widely believed that if GenRSA(n) is instantiated by calling GenMod(n), then the RSA problem is hard for (almost) any choice of e with $(e, \phi(N)) = 1$.

A Cautionary Example

- Generate an RSA modulus N with encryption exponent $e = 3$.
- Let m be a short message. Here “short” means $m < N^{1/3}$.
- Then $c = m^3$; that is, no modular reduction takes place!
- An adversary $\mathcal{A}$ can easily find m by taking a real cube root.

This does not violate the assertion that the RSA problem is hard, even for $e = 3$. (Why not?) It does show that “textbook RSA” can have unintended weaknesses.