Introduction to Cryptography

Public Key Cryptography

Who Has the Key?

- For two parties—in cryptography, they're traditionally called *Alice and Bob*—to communicate securely, they both need to know the same key
- How do they get it?
- How do they handle it?

Codebooks

- Bob to Alice: here's a codebook; go spy and send me lots of messages
- Alice loses her codebook; she can't communicate securely
- Alice is caught and her codeebook is seized: she can't communicate securely, and the other side can read all her old messages
- Alice hangs onto her codebook and isn't caught, but sends so many messages that the other side breaks the code

Ciphers

- Bob to Alice: here's a cipher key; go spy and send me lots of messages
- Alice loses her key; she can't communicate securely
- Alice is caught and her key is seized: she can't communicate securely, and the other side can read all her old messages
- Alice hangs onto her key and isn't caught, but sends so many messages that the other side breaks the cipher

One-Time Pad

- Bob to Alice: here's a one-time pad; go spy and send me lots of messages
- Alice loses her one-time pad; she can't communicate securely
- Alice is caught and her one-time pad is seized: she can't communicate securely, and the other side can read all her old messages
- Alice hangs onto her one-time pad and isn't caught, but sends so many messages that she uses up the one-time pad and can't communicate

This was a Real Problem!

- Codebooks were captured, e.g., the *SMS Magdeburg*, 1914
- Cipher keys were captured, e.g., the *U-110*, 1941
- Cipher keys were compromised after agent capture, e.g., Englandspiel, 1941–43
- Soviet spies—and their one-time pads—were captured
- Etc.
- The problem seemed unsolvable—until it was solved in the 1970s

The Inventors and Their Impact

1970–1974	James Ellis, Clifford Cocks, Malcom Williamson
1975	Ralph Merkle
1976	Whit Diffie and Martin Hellman
1978	Ron Rivest, Adi Shamir, Len Adleman

Let’s start with Diffie and Hellman

A Remarkable Paper

- Diffie, then a grad student, (somehow!) understood the key distribution problem
- He conceived of a radically different idea: a *public* encryption key that is separate but derived from a *private* decryption key
- He conceived of *digital signatures*: a way to use a private key to sign a document in a way verifiable by anyone who knows the public key
- He and his advisor, Hellman, wrote a paper that changed the world

Requirements

- Two functions, E_k and D_k, such that E_k is the inverse of D_k and vice-versa
- From a seed key k, it must be easy to compute E_k and D_k
- It is infeasible to derive D_k from E_k
- The encryption and decryption keys are different, so this is sometimes called *asymmetric cryptography*

Use

- Bob publishes his public key E_{k_B}
- To send him a message m, Alice computes $E_{k_B}(m)$
- Bob decrypts that using his private key D_{k_B}
- Alice can publish her own public key, E_{k_A}; Bob can use that to send her replies

Digital Signatures

- D_k is the inverse of E_k—but what about the reverse?
- If those functions commute, E_k is the inverse of D_k
- To sign a message, *decrypt* it with your private key
- Anyone can verify the signature using your public key
- What Alice really sends Bob is $D_{k_A}(E_{k_B}(m))$
- Bob thus receives an authenticated, secure message
- Only one problem: Diffie and Hellman couldn't find suitable functions E and D. . .
- But their partial solution is itself useful

Diffie-Hellman Key Exchange

- Both parties agree on a prime p and a base g, where g is a generator of the group $\mathbb{Z}_p^*$, the positive integers modulo p under multiplication
- (If $p = 2q + 1$, where q is prime, half of the elements of $\mathbb{Z}_p^*$ are generators)
- Alice picks a random number r_A and sends Bob $g^{r_A} \bmod p$. Similarly, Bob picks a random number r_B and sends $g^{r_B} \bmod p$.
- Alice now knows r_A and $g^{r_B} \bmod p$; Bob knows r_B and $g^{r_A} \bmod p$
- Alice calculates $(g^{r_B})^{r_A} = g^{r_A r_B} \bmod p$; Bob calculates $(g^{r_A})^{r_B} = g^{r_A r_B} \bmod p$
- $g^{r_A r_B} \bmod p$ is now a shared secret
- This is called a *public key distribution system*

Security of Diffie-Hellman Key Exchange

- Given x and x^y, finding y is easy: it's $\log_x$
- But $x^y \bmod p$ requires solving the *discrete logarithm* problem, and that's believed to be very hard
- We don't know if there's any other way to crack this—but it doesn't seem likely

Authenticating Diffie-Hellman Key Exchange

- Alice and Bob have another problem: how do they know that the received exponentials are genuine?
- Maybe Bob has really received $g^{r_E} \bmod p$, which belongs to Eve, the eavesdropper
- (Is that a real threat? Yes!)
- Either we need some way to authenticate it—Alice and Bob could publish their long-term exponentials as their public keys—or we just accept that this is an unauthenticated key exhange

Network Threat Models

- Standard assumption: the enemy controls the network
- Mental model: you hand your packets to the enemy to be delivered
- More formally: network messages can be created, deleted, modified, replicated, duplicated, etc.
- Note the resemblance to the CIA model
- (How do we authenticate published keys? An interesting question; stay tuned. . .)

Rivest, Shamir, and Adleman

- Rivest, Shamir, and Adleman were MIT professors who saw the Diffie and Hellman paper
- They tried and discarded many schemes, before finally finding one that worked
- Their solution is still in use today

The RSA Algorithm

- Pick two large primes, p and q (today, about 1024 bits long); let $n = pq$
- Pick a public encryption key e, typically 65537 ($2^{16} + 1$)
- Calculate $ed \equiv 1 \bmod (p-1)(q-1)$
- Encryption: $c \equiv m^e \bmod n$
- Decryption: $m \equiv (m^e)^d \bmod n \equiv m^{ed} \bmod n \equiv m$
- The public key is $\langle e, n \rangle$; the private key is $\langle d, n \rangle$ or $\langle d, p, q \rangle$
- The equations are symmetric; we can thus achieve digital signatures as well

Implementation and Security

- Probabilistic primality testing of large numbers is easy and efficient; we can thus generate p and q easily enough
- However, factoring a large number n into p and q is believed to be extremely hard
- There is no known way to calculate d without knowledge of p and q; n and e alone will not suffice

Is RSA Secure?

- There is no known way to calculate *d* without knowing *p* and *q*, i.e., factoring *n*
- Factoring has been studed for hundreds of years and is believed to be very hard
- In other words: we do not *know* if RSA is equivalent to factoring, nor do we *know* that factoring is intrinsically hard—but both are believed to be the case

A Toy Example

Let $p = 13, q = 29$
Thus, $n = 377$ and $(p-1)(q-1) = 336$
Let $d = 131$; thus, $e = 59$
$ed = 7729 \equiv 1 \bmod 336$

Let $m = 42$
$42^{59} \equiv 22 \bmod 377$
$22^{131} \equiv 42 \bmod 377$

```
$ bc -q
scale=0
p = 13; q = 29; n=p*q
n
377
d = 131; e = 59
(e*d) % ((p-1)*(q-1))
1
m=42
(m^e)%n
22
(22^d)%n
42
```

The Real Inventors. . .

- Diffie and Hellman were brilliant, but they were amateurs—the pros got there first
- At GCHQ, James Ellis was asked to look into the key distribution problem—and in 1970, he published an internal memo saying that “non-secret encryption” was at least conceputally possible
- However, he wasn’t a mathematician and did not propose a solution
- A few years later, Clifford Cocks invented the algorithm now known as RSA
- Malcom Williamson invented what is now known as Diffie-Hellman key exchange
- And we never heard about this—because it was classified until 1997!
- But—it’s interesting that the academic sector was only a very few years behind the professionals

Ralph Merkle’s Puzzles

- In 1975, Merkle suggested that Alice create N “puzzles”—encrypted messages that could be solved, but only in $O(N)$ time
- These messages all contain a random key
- She sends all N to Bob
- Bob picks one puzzle at random, spends $O(N)$ time solving it, and recovers the key to start using
- An eavesdropper will have to spend $O(N^2)$ time
- This is also a public key distribution function

What Happened?

- Technically, it isn't clear that N^2 is costly enough
- N could be very large, but then there are bandwidth issues and creation time for Alice
- More seriously: Merkle had a much harder time getting traction for his idea

Digital Signatures?

- A former director of the NSA claimed that the US invented public key cryptography a decade before Diffie and Hellman—which would also be before GCHQ
- I have heard a claim that the motivation was technical mechanisms for control of nuclear weapons
- It may be true—but objective evidence is lacking
- My research suggests that what they would have needed is digital signatures—which GCHQ did not invent
- Verdict: possible but unproven

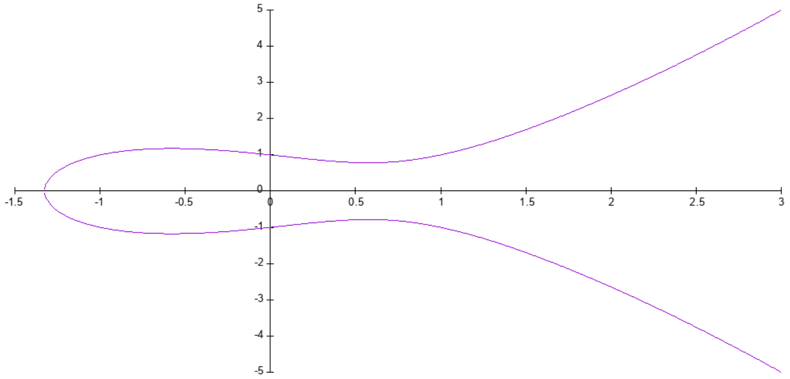
Elliptic Curves

- RSA is secure, but it's slow and its keys are large
- We want something faster, especially for low-end devices
- (Besides, mathematicians are making some progress on factoring)
- The answer: *elliptic curves*
- With elliptic curves, we can do signatures and Diffie-Hellman exchanges

What is an Elliptic Curve?

A curve in two variables where one is of degree 2 and the other is of degree 3.

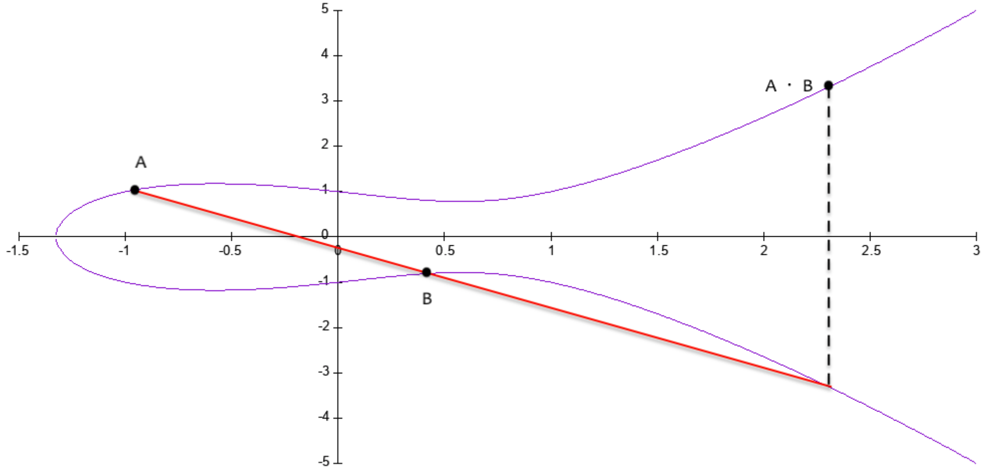
Example: $y^2 = x^3 - x + 1$



Using Elliptic Curves for Cryptography

- We use only integer points
- Elements of the field are (x, y) points; composing two points involves finding where a line between them next intersects the curve and projecting up or down
- Arithmetic is modulo some large number—but much smaller than for RSA
- For RSA and Diffie-Hellman, use at least 2048-bit moduli; for elliptic curves, we might use 256-bit moduli
- Computations are 20–30$\times$ faster

Composition



However. . .

- There may be some patent issues
- Many people distrust NIST's standardized curves—did the NSA do something nasty?
- Digital signatures using elliptic curves require a good source of randomness for each signature—often hard on, e.g., smart cards

Encrypting with RSA

- We could try encrypting a file with RSA, but that’s unpleasant
  - The blocksize, with a 2048-bit modulus, is 255 bytes, so we’d have to chop up the file into smaller pieces
  - RSA encryption and decryption are *slow*
- Besides, RSA is just simple mathematical operations, so there may be a mathematical attack
- Example: “yes”^{3} is only 69 bits, and won’t be reduced by the modulus operation; finding $\sqrt[3]{503565527901556194283}$ is easy.

A (More) Realistic Scenario

- Bob generates a random key k for a conventional cipher.
- Bob encrypts the message: $c = E(k, m)$
- Bob pads k with a known amount of padding, to make it at least 1024 bits (half the modulus size) long; call this k'.
- k' is encrypted with Alice's public key $\langle e, n \rangle$.
- Bob transmits $\{c, (k')^e \bmod n\}$ to Alice.
- Alice uses $\langle d, n \rangle$ to recover k', removes the padding, and uses k to decrypt ciphertext c.
- In reality, it's even more complex than that. . .

What About Digital Signatures?

- The same issues apply to digital signatures—how do we sign a file?
- We need some sort of analog to conventional (symmetric) encryption
- Answer: we sign a *hash* of the file

Signing a File

- Alice wants to sign a file m
- She calculates $h = H(m)$, where H is a *cryptographic hash function*
- She signs the hash, which (of course) involves padding h to make h'
- She then calculates $s = (h')^d \bmod n$ and sends $\langle m, s\rangle$ to Bob
- Naturally, she could also encrypt m, adding even more complexity...

Requirements for Cryptographic Hash Functions

Cryptographic hash functions must have some special properties

Size	Produce relatively-short, fixed-length output string from arbitrarily long input
Preimage resistance	Given y, it is infeasible to find x such that $H(x) = y$
Second preimage resistance	Given x and $H(x)$, it is infeasible to find $y, y \neq x$ such that $H(x) = H(y)$
Collision resistance	It is infeasible to find two strings, x and y, $x \neq y$, such that $H(x) = H(y)$

Common Hash Functions

- Best-known cryptographic hash functions: MD5 (128 bits), SHA-1 (160 bits), SHA2-256/384/512 (256/384/512 bits), SHA3-224/256/384/512 (224/256/384/512 bits)
- Wang et al. found practical collision attacks against MD5 and SHA-1

☞ They're insecure; *never* use them

- SHA2-256/384/512 have the same basic structure as MD5 and SHA-1—but NIST believes they're secure despite Wang's attack
- NIST held a design competition for a new hash SHA-3 function; the winner (Keccak) has a completely different internal structure

Hash Function Strength

- Hash functions can be cryptographically weak, e.g., vulnerable to differential cryptanalysis and the like
- (Hash functions generally use iterated rounds, too, and have a diffusion property)
- Just as with ciphers, though, they have a maximum strength, dictated by the output size
- For preimage and second preimage attacks, that strength is $2^{blocksize}$—each random input change changes the output randomly, so enough tries will probabilistically find the answer eventually
- For collision attacks, though, it's half the blocksize: $2^{blocksize/2}$, because of the birthday paradox

The Birthday Paradox

- How many people need to be in a room for the probability that two will have the same birthday to be $> .5$?
- Naive answer: 183
- Correct answer: 23
- The question is not "who has the same birthday as Alice?"; it's "who has the same birthday as Alice or Bob or Carol or . . . " assuming that none of them have the same birthday as any of the others

The Birthday Attack

- Alice can prepare lots of variant contracts, looking for any two that have the same hash
- More precisely, she generates many trivial variants on m and m', looking for a match between the two sets
- This is much easier than finding a contract that has the same hash as a given other contract
- As a consequence, the strength of a hash function against brute force attacks is approximately half the output block size: 64 bits for MD5, 80 bits for SHA-1, etc.
- Hash function collisions have been used in real-world attacks: some intelligence agency used a novel attack in the “Flame” malware

(The Birthday Paradox and Block Ciphers)

- Suppose that we're using a block cipher like DES, with a 64-bit blocksize
- Each encryption of a different plaintext block generates a random-seeming 64-bit block of ciphertext
- What are the odds that two blocks are identical? That would leak information about the plaintext!
- From the birthday paradox, at 2^{32} blocks—2^{35} bytes, or about 34GB—the probability is $> .5$
- Conclusion: never encrypt that much with a single DES or 3DES key—which is why AES has 128-bit blocks
- Historical note: in 1974, a large disk held 200 MB, well below that limit, and a 1.5Mb/s data link was very fast

Hash Functions are Powerful

- Hash functions are used for far more than digital signatures
- In fact, they're among the most flexible cryptographic primitives around
- Other uses: random number generation, message integrity, sophisticated tricks like coin-flipping, cryptocurrencies

Building a Message Authentication Code (MAC)

- We need a way to prevent tampering with messages
- Best-known construct is HMAC—provably secure under minimal assumptions
- $\mathrm{HMAC}(m, k) = H(\mathrm{opad} \oplus k, H(\mathrm{ipad} \oplus k, m))$ where H is a cryptographic hash function and m is the message
- Note: if the message is encrypted, do the HMAC over the *ciphertext*, not the plaintext
- Note: the authentication key *must* be distinct from the confidentiality key
- Frequently, the output of HMAC is truncated

Using Hash Functions for Random Number Generation

Cryptographic hash functions can be used for secure random number generation—but you have to do it carefully.

Bad $s = H(s);$ `return` s

Using Hash Functions for Random Number Generation

Cryptographic hash functions can be used for secure random number generation—but you have to do it carefully.

Bad $s = H(s)$; `return` s
Very bad—seeing one random number lets the attacker know all future ones

Using Hash Functions for Random Number Generation

Cryptographic hash functions can be used for secure random number generation—but you have to do it carefully.

Bad $s = H(s)$; `returns` s
Very bad—seeing one random number lets the attacker know all future ones

OK `return` H`(ctr++);`

Using Hash Functions for Random Number Generation

Cryptographic hash functions can be used for secure random number generation—but you have to do it carefully.

Bad $s = H(s)$; `return s`
Very bad—seeing one random number lets the attacker know all future ones

OK `return` H(`ctr++`);
Secure if `ctr` is initialized to a large random number

Using Hash Functions for Random Number Generation

Cryptographic hash functions can be used for secure random number generation—but you have to do it carefully.

Bad $s = H(s)$; `returns`
Very bad—seeing one random number lets the attacker know all future ones

OK `return` $H($`ctr++`$)$;
Secure if `ctr` is initialized to a large random number

Better `return` $HMAC(k,$ `ctr++`$)$;

Using Hash Functions for Random Number Generation

Cryptographic hash functions can be used for secure random number generation—but you have to do it carefully.

Bad	$s = H(s)$; `returns` Very bad—seeing one random number lets the attacker know all future ones
OK	`return` H(`ctr++`); Secure if `ctr` is initialized to a large random number
Better	`return` $HMAC(k,$ `ctr++`); The existence of a key makes this very secure

Coin-Flipping

- We want to flip a coin over the Internet
- We don't want any outside trusted parties—but how do we know the other side isn't cheating?
- Recall that (a) the output of of a hash function is a random(-seeming) number, and (b) because of diffusion, it doesn't leak any information about the input
- Protocol:
  1. Alice and Bob each pick random numbers r_A and r_B, and exchange $H(r_A)$ and $H(r_B)$, thus *committing* to their values
  2. They then exchange r_A and r_B and verify the other's number
  3. Exclusive-OR the low-order bit of r_A and r_B to get 0 or 1: heads or tails

Is Coin-flipping Secure?

- Is this protocol correct? Discuss. . .

Is Coin-flipping Secure?

- Is this protocol correct? Discuss. . .
- What about this variant?
  1. Alice picks the string s as either “red” or “blue”
  2. She commits to it by sending Bob $H(s)$
  3. Bob guesses the color and tells Alice. If he’s right, the coin is “heads”; if he’s wrong, it’s “tails”
  4. Alice discloses s, which yields the result; Bob verifies $H(s)$ matches what he was sent earlier

Is Coin-flipping Secure?

- Is this protocol correct? Discuss. . .
- What about this variant?
  1. Alice picks the string s as either “red” or “blue”
  2. She commits to it by sending Bob $H(s)$
  3. Bob guesses the color and tells Alice. If he’s right, the coin is “heads”; if he’s wrong, it’s “tails”
  4. Alice discloses s, which yields the result; Bob verifies $H(s)$ matches what he was sent earlier
- It’s easy to get these things wrong. . .
- How can we fix that protocol? What assumptions are we making?

Quantum Computers

- Scientists and engineers have been developing quantum computers, which run on quantum mechanical principles
- Quantum computers use “qubits” instead of bits
- Properties such as entanglement and superposition mean that they can, in principle, run much faster than classical computers
- This poses a potential threat to cryptographic algorithms

Grover's Algorithm

- Grover's Algorithm provides, among other things, a way to attack symmetric ciphers such as AES
- It effectively halves the key length against quantum computing brute force attacks: AES-128 could be attack in $O(2^{64})$ time—and 2^{64} is doable
- This isn't a real threat today, though—brute force attacks require massively parallel computers. It's hard enough to build once quantum computer, let alone many thousands
- But—the chance of this is why AES-256 exists, for protection against future massively parallel quantum attacks
- NSA sometimes wants to protect data for 100 years. . .

Shor's Algorithm

- Shor's Algorithm permits efficient factoring of large numbers
- This cracks RSA—and for mathematical reasons, if you can factor efficiently you can also solve discrete log, i.e., you can crack Diffie-Hellman
- Elliptic curve algorithms are also vulnerable

☞ When this technology becomes available, *all* data protected by public key technologies is vulnerable

- This includes the blockchain

Post-Quantum Algorithms

- In late 2016, NIST started an open process—algorithm submissions, conferences, etc.—to pick the best "post-quantum" algorithms
- In July, NIST announced that several algorithms had advanced to Round 3. There will be another conference next year; they hope to announce their final choices by the end of next year
- But they reserve the right to stretch things out more if needed

Why are Quantum Computers Hard to Build?

- Qubits are susceptible to “decoherence”—they lose their quantum properties
- This is caused by environmental interations: heat, magnetic fields, cosmic rays, etc.
- Some quantum computers already require extreme environmental conditions, e.g., a temperature of .02 kelvins
- There is such a thing as quantum error correction—but that requires far more of the hard-to-build qubits
- Some theoreticians think that we will *never* be able to build a big-enough quantum computer to attack real-world encryption
- But we don’t know, and some data must be protected for a very long time

How Long Must Encryption Protect Data?

- There's an important implicit message here: not all data needs to be protected for that long
- Credit card numbers may only need protection for a few years, until they expire
- Cryptography to authenticate traffic only needs to last as long as the session—once the session ends, you can't forge new traffic for it
- Some contracts—mortgages, for example—have to be secure for decades
- Some national security data may need protection for 100 years or more
- Figuring this out is part of engineering security solutions—and often, stronger protection is effectively free

Questions?

(Black-crowned night heron, Morningside Park pond, September 11, 2020)