RSA Encryption

CS70 - Spring 2017

David Dinh

14 February 2017

UC Berkeley

Agenda

Congratulations on your first midterm! Course staff is busy grading them right now and the grades should be ready by tomorrow morning.

Agenda

Congratulations on your first midterm! Course staff is busy grading them right now and the grades should be ready by tomorrow morning.

Today: A practical application of modular arithmetic: RSA encryption.

Agenda

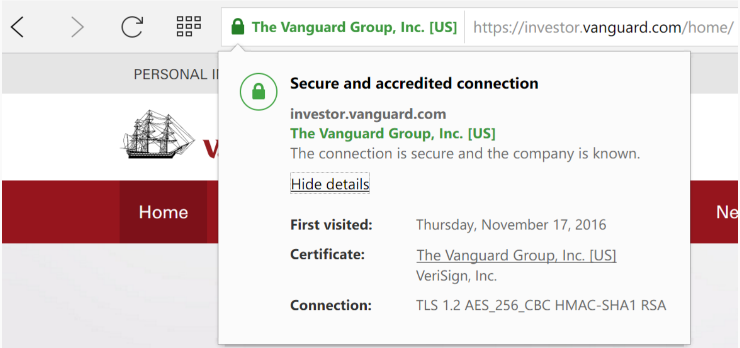
Congratulations on your first midterm! Course staff is busy grading them right now and the grades should be ready by tomorrow morning.

Today: A practical application of modular arithmetic: RSA encryption. You probably use this every day.

Agenda

Congratulations on your first midterm! Course staff is busy grading them right now and the grades should be ready by tomorrow morning.

Today: A practical application of modular arithmetic: RSA encryption. You probably use this every day.



Motivation

Let's say I want to open a bank account online. I need to tell the bank my social security number.

Motivation

Let's say I want to open a bank account online. I need to tell the bank my social security number.

We want a "private channel" where I can send the bank my info, safe from prying eyes.

Motivation

Let's say I want to open a bank account online. I need to tell the bank my social security number.

Unfortunately, real life looks more like this. How do I form a private channel to the bank if there's someone snooping on my connection?

XOR

Consider the bit operation *xor* (denoted $\oplus$): $A \oplus B = 1$ if and only if exactly one of A, B are 1.

XOR

Consider the bit operation *xor* (denoted $\oplus$): $A \oplus B = 1$ if and only if exactly one of A, B are 1. Truth table?

XOR

Consider the bit operation *xor* (denoted $\oplus$): $A \oplus B = 1$ if and only if exactly one of A, B are 1. Truth table?

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

XOR

Consider the bit operation *xor* (denoted $\oplus$): $A \oplus B = 1$ if and only if exactly one of A, B are 1. Truth table?

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

Is $A \oplus B$ equal to $B \oplus A$?

XOR

Consider the bit operation *xor* (denoted $\oplus$): $A \oplus B = 1$ if and only if exactly one of A, B are 1. Truth table?

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

Is $A \oplus B$ equal to $B \oplus A$? Yes!

XOR

Consider the bit operation *xor* (denoted $\oplus$): $A \oplus B = 1$ if and only if exactly one of A, B are 1. Truth table?

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

Is $A \oplus B$ equal to $B \oplus A$? Yes!

What's $A \oplus A$?

XOR

Consider the bit operation *xor* (denoted $\oplus$): $A \oplus B = 1$ if and only if exactly one of A, B are 1. Truth table?

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

Is $A \oplus B$ equal to $B \oplus A$? Yes!

What's $A \oplus A$? 0

One-Time Pad

Simple encryption scheme (”one time pad”): given a *plaintext* we want to encrypt (e.g. SSN, represented as a bitstring) and a *key* of equal length, xor each bit of the plaintext with the corresponding bit of the key to get a *ciphertext*.

One-Time Pad

Simple encryption scheme (”one time pad”): given a *plaintext* we want to encrypt (e.g. SSN, represented as a bitstring) and a *key* of equal length, xor each bit of the plaintext with the corresponding bit of the key to get a *ciphertext*.

How do we decrypt? Notice that $x \oplus y \oplus x = y \oplus x \oplus x = y \oplus 0 = y$. So: just xor the ciphertext with the key, bitwise, to get plaintext back.

Example/Live Demo

OTP: Security

Why is OTP secure?

OTP: Security

Why is OTP secure?

Suppose I have the ciphertext c, but not the key or the plaintext. Can I find out anything about the plaintext?

Why is OTP secure?

Suppose I have the ciphertext c, but not the key or the plaintext. Can I find out anything about the plaintext? No!

OTP: Security

Why is OTP secure?

Suppose I have the ciphertext c, but not the key or the plaintext. Can I find out anything about the plaintext? No!

For every possible plaintext p (of the same length as c), there exists a key k such that $c = p \oplus k$. Why?

OTP: Security

Why is OTP secure?

Suppose I have the ciphertext c, but not the key or the plaintext. Can I find out anything about the plaintext? No!

For every possible plaintext p (of the same length as c), there exists a key k such that $c = p \oplus k$. Why? Just let $k = c \oplus p$.

OTP: Security

Why is OTP secure?

Suppose I have the ciphertext c, but not the key or the plaintext. Can I find out anything about the plaintext? No!

For every possible plaintext p (of the same length as c), there exists a key k such that $c = p \oplus k$. Why? Just let $k = c \oplus p$.

What's Wrong with OTP?

Need a really long key. Same length as input! Fine for SSN, credit card numbers, maybe not so fine if you're trying to transmit the plans for the Death Star...

What’s Wrong with OTP?

Need a really long key. Same length as input! Fine for SSN, credit card numbers, maybe not so fine if you’re trying to transmit the plans for the Death Star...

Can’t reuse key twice without leaking info. Let’s say I send $p_1 \oplus k$ and $p_2 \oplus k$. Then a spy can easily figure out what $p_1 \oplus p_2$ is! Information leaked!

What's Wrong with OTP?

Need a really long key. Same length as input! Fine for SSN, credit card numbers, maybe not so fine if you're trying to transmit the plans for the Death Star...

Can't reuse key twice without leaking info. Let's say I send $p_1 \oplus k$ and $p_2 \oplus k$. Then a spy can easily figure out what $p_1 \oplus p_2$ is! Information leaked!

Needs a key to be shared before the transmission is done. If I need to walk into bank to share a secret key before sending them my SSN, why not just give my SSN to them when I walk in?

Addressing OTP Shortcomings

Long keys can be addressed with ”pseudorandom generators” that take short random strings and turn them into longer strings that ”look random”.

Addressing OTP Shortcomings

Long keys can be addressed with ”pseudorandom generators” that take short random strings and turn them into longer strings that ”look random”. Beyond the scope of this course (CS276 and current workshop at the Simons Institute).

Addressing OTP Shortcomings

Long keys can be addressed with ”pseudorandom generators” that take short random strings and turn them into longer strings that ”look random”. Beyond the scope of this course (CS276 and current workshop at the Simons Institute).

Address the security concerns with **public key crypto**. RSA is an algorithm for that.

Addressing OTP Shortcomings

Long keys can be addressed with ”pseudorandom generators” that take short random strings and turn them into longer strings that ”look random”. Beyond the scope of this course (CS276 and current workshop at the Simons Institute).

Address the security concerns with **public key crypto**. RSA is an algorithm for that.

Big idea: the bank gives everyone a mathematical safe that they can put stuff into, but only the bank can unlock.

The RSA Algorithm

Formally: bank broadcasts a **public key** that anyone can use to encrypt data with. It also has (and keeps secret) a **private key** that they can use to decrypt data that's been encrypted with the public key.

The RSA Algorithm

Formally: bank broadcasts a **public key** that anyone can use to encrypt data with. It also has (and keeps secret) a **private key** that they can use to decrypt data that's been encrypted with the public key.

Key generation: Bank picks two large primes, p and q, and lets $N = pq$. It also chooses some e relatively prime to $(p-1)(q-1)$ (normally small, say, 3), and then computes $d = e^{-1} \mod (p-1)(q-1)$.

The RSA Algorithm

Formally: bank broadcasts a **public key** that anyone can use to encrypt data with. It also has (and keeps secret) a **private key** that they can use to decrypt data that's been encrypted with the public key.

Key generation: Bank picks two large primes, p and q, and lets $N = pq$. It also chooses some e relatively prime to $(p-1)(q-1)$ (normally small, say, 3), and then computes $d = e^{-1} \mod (p-1)(q-1)$.

Puts $N = pq$ and e on their website. Locks up d deep in the bowels of corporate HQ.

The RSA Algorithm

Formally: bank broadcasts a **public key** that anyone can use to encrypt data with. It also has (and keeps secret) a **private key** that they can use to decrypt data that's been encrypted with the public key.

Key generation: Bank picks two large primes, p and q, and lets $N = pq$. It also chooses some e relatively prime to $(p-1)(q-1)$ (normally small, say, 3), and then computes $d = e^{-1} \bmod (p-1)(q-1)$.

Puts $N = pq$ and e on their website. Locks up d deep in the bowels of corporate HQ.

Encrypt: Given plaintext x (say, an SSN), I compute the ciphertext $c = E(x) = mod(x^e, N)$ and sends it to the bank (over an open channel that could be snooped upon).

The RSA Algorithm

Formally: bank broadcasts a **public key** that anyone can use to encrypt data with. It also has (and keeps secret) a **private key** that they can use to decrypt data that's been encrypted with the public key.

Key generation: Bank picks two large primes, p and q, and lets $N = pq$. It also chooses some e relatively prime to $(p-1)(q-1)$ (normally small, say, 3), and then computes $d = e^{-1} \mod (p-1)(q-1)$.

Puts $N = pq$ and e on their website. Locks up d deep in the bowels of corporate HQ.

Encrypt: Given plaintext x (say, an SSN), I compute the ciphertext $c = E(x) = mod(x^e, N)$ and sends it to the bank (over an open channel that could be snooped upon).

Decrypt: Bank computes $D(c) = mod(c^d, N)$. We'll show (next slide) this actually gives the plaintext x back.

Example/Live Demo

Correctness of RSA

Theorem: For the encryption/decryption protocol on the previous slide, $D(E(x)) = x \pmod{N}$ for all $x \in \{0, 1, \ldots n-1\}$.

Correctness of RSA

Theorem: For the encryption/decryption protocol on the previous slide, $D(E(x)) = x \pmod{N}$ for all $x \in \{0, 1, \ldots n-1\}$.

Proof: It suffices to show: $(x^e)^d \equiv x \pmod{n}$ for all $x \in \{0, 1, \ldots n-1\}$.

Correctness of RSA

Theorem: For the encryption/decryption protocol on the previous slide, $D(E(x)) = x \pmod{N}$ for all $x \in \{0, 1, \ldots n-1\}$.

Proof: It suffices to show: $(x^e)^d \equiv x \pmod{n}$ for all $x \in \{0, 1, \ldots n-1\}$.

Consider the exponent ed. We kow that $ed \equiv 1 \mod (p-1)(q-1)$ by definition, so $ed = 1 + k(p-1)(q-1)$ for some integer k.

Correctness of RSA

Theorem: For the encryption/decryption protocol on the previous slide, $D(E(x)) = x \pmod N$ for all $x \in \{0, 1, ... n-1\}$.

Proof: It suffices to show: $(x^e)^d \equiv x \pmod n$ for all $x \in \{0, 1, ... n-1\}$.

Consider the exponent ed. We kow that $ed \equiv 1 \mod (p-1)(q-1)$ by definition, so $ed = 1 + k(p-1)(q-1)$ for some integer k. Therefore,

$$x^{ed} - x = x^{1+k(p-1)(q-1)} - x = x(x^{k(p-1)(q-1)} - 1) \quad .$$

Correctness of RSA

Theorem: For the encryption/decryption protocol on the previous slide, $D(E(x)) = x \pmod{N}$ for all $x \in \{0, 1, \ldots n-1\}$.

Proof: It suffices to show: $(x^e)^d \equiv x \pmod{n}$ for all $x \in \{0, 1, \ldots n-1\}$.

Consider the exponent ed. We kow that $ed \equiv 1 \mod (p-1)(q-1)$ by definition, so $ed = 1 + k(p-1)(q-1)$ for some integer k. Therefore,

$$x^{ed} - x = x^{1+k(p-1)(q-1)} - x = x(x^{k(p-1)(q-1)} - 1) \quad .$$

Suffices to show that this expression is 0 mod N for all x, i.e. that it's a multiple of both p and q. We will show it's a multiple of p.

Correctness of RSA

Theorem: For the encryption/decryption protocol on the previous slide, $D(E(x)) = x \pmod{N}$ for all $x \in \{0,1,\ldots n-1\}$.

Proof: It suffices to show: $(x^e)^d \equiv x \pmod{n}$ for all $x \in \{0,1,\ldots n-1\}$.

Consider the exponent ed. We kow that $ed \equiv 1 \mod (p-1)(q-1)$ by definition, so $ed = 1 + k(p-1)(q-1)$ for some integer k. Therefore,

$$x^{ed} - x = x^{1+k(p-1)(q-1)} - x = x(x^{k(p-1)(q-1)} - 1) \quad .$$

Suffices to show that this expression is 0 mod N for all x, i.e. that it's a multiple of both p and q. We will show it's a multiple of p.

- Case 1: p divides x. Then obviously it also divides $x(x^{k(p-1)(q-1)} - 1)$, as desired.

Correctness of RSA

Theorem: For the encryption/decryption protocol on the previous slide, $D(E(x)) = x \pmod{N}$ for all $x \in \{0, 1, \ldots n-1\}$.

Proof: It suffices to show: $(x^e)^d \equiv x \pmod{n}$ for all $x \in \{0, 1, \ldots n-1\}$.

Consider the exponent ed. We kow that $ed \equiv 1 \mod (p-1)(q-1)$ by definition, so $ed = 1 + k(p-1)(q-1)$ for some integer k. Therefore,

$$x^{ed} - x = x^{1+k(p-1)(q-1)} - x = x(x^{k(p-1)(q-1)} - 1) \quad .$$

Suffices to show that this expression is 0 mod N for all x, i.e. that it's a multiple of both p and q. We will show it's a multiple of p.

- Case 1: p divides x. Then obviously it also divides $x(x^{k(p-1)(q-1)} - 1)$, as desired.
- Case 2: p doesn't divide x. Then $x^{k(p-1)(q-1)} = (x^{p-1})^{k(q-1)}$.

Correctness of RSA

Theorem: For the encryption/decryption protocol on the previous slide, $D(E(x)) = x \pmod{N}$ for all $x \in \{0, 1, \ldots n-1\}$.

Proof: It suffices to show: $(x^e)^d \equiv x \pmod{n}$ for all $x \in \{0, 1, \ldots n-1\}$.

Consider the exponent ed. We kow that $ed \equiv 1 \mod (p-1)(q-1)$ by definition, so $ed = 1 + k(p-1)(q-1)$ for some integer k. Therefore,

$$x^{ed} - x = x^{1+k(p-1)(q-1)} - x = x(x^{k(p-1)(q-1)} - 1) \quad .$$

Suffices to show that this expression is 0 mod N for all x, i.e. that it's a multiple of both p and q. We will show it's a multiple of p.

- Case 1: p divides x. Then obviously it also divides $x(x^{k(p-1)(q-1)} - 1)$, as desired.
- Case 2: p doesn't divide x. Then $x^{k(p-1)(q-1)} = (x^{p-1})^{k(q-1)}$. Applying Fermat's little theorem, $x^{p-1} \equiv 1 \pmod{p}$.

Correctness of RSA

Theorem: For the encryption/decryption protocol on the previous slide, $D(E(x)) = x \pmod{N}$ for all $x \in \{0, 1, \ldots n-1\}$.

Proof: It suffices to show: $(x^e)^d \equiv x \pmod{n}$ for all $x \in \{0, 1, \ldots n-1\}$.

Consider the exponent ed. We kow that $ed \equiv 1 \mod (p-1)(q-1)$ by definition, so $ed = 1 + k(p-1)(q-1)$ for some integer k. Therefore,

$$x^{ed} - x = x^{1+k(p-1)(q-1)} - x = x(x^{k(p-1)(q-1)} - 1) \quad .$$

Suffices to show that this expression is 0 mod N for all x, i.e. that it's a multiple of both p and q. We will show it's a multiple of p.

- Case 1: p divides x. Then obviously it also divides $x(x^{k(p-1)(q-1)} - 1)$, as desired.
- Case 2: p doesn't divide x. Then $x^{k(p-1)(q-1)} = (x^{p-1})^{k(q-1)}$. Applying Fermat's little theorem, $x^{p-1} \equiv 1 \pmod{p}$. So $x^{k(p-1)(q-1)} - 1 \equiv 1^{k(q-1)} - 1 \equiv 0 \pmod{p}$, so $x(x^{k(p-1)(q-1)} - 1)$ must be a multiple of p.

Argument for q is exactly the same. Therefore $p|(x^{ed} - x)$. ■

Implementation Concerns

Key generation: Bank picks two large primes, p and q, and lets $N = pq$. It also chooses some e relatively prime to $(p-1)(q-1)$ (normally small, say, 3), and then computes $d = e^{-1} \mod (p-1)(q-1)$.

Puts $N = pq$ and e on their website. Locks up d deep in the bowels of corporate HQ.

Encrypt: Given plaintext x (say, an SSN), I compute the ciphertext $c = E(x) = mod(x^e, N)$ and sends it to the bank (over an open channel that could be snooped upon).

Decrypt: Bank computes $D(c) = mod(c^d, N)$. We'll show (next slide) this actually gives the plaintext x back.

Implementation Concerns

Key generation: Bank picks two large primes, p and q, and lets $N = pq$. It also chooses some e relatively prime to $(p-1)(q-1)$ (normally small, say, 3), and then computes $d = e^{-1} \bmod (p-1)(q-1)$.

Puts $N = pq$ and e on their website. Locks up d deep in the bowels of corporate HQ.

Encrypt: Given plaintext x (say, an SSN), I compute the ciphertext $c = E(x) = mod(x^e, N)$ and sends it to the bank (over an open channel that could be snooped upon).

Decrypt: Bank computes $D(c) = mod(c^d, N)$. We'll show (next slide) this actually gives the plaintext x back.

Implementation Concerns: Prime-finding

How do we find large primes p and q?

Implementation Concerns: Prime-finding

How do we find large primes p and q?

We don't know whether or not there's an algorithm that's guaranteed to find a prime efficiently at each time!

Implementation Concerns: Prime-finding

How do we find large primes p and q?

We don't know whether or not there's an algorithm that's guaranteed to find a prime efficiently at each time! But... we can pick random numbers, and test that they're prime.

Implementation Concerns: Prime-finding

How do we find large primes p and q?

We don't know whether or not there's an algorithm that's guaranteed to find a prime efficiently at each time! But... we can pick random numbers, and test that they're prime.

Prime number theorem: Let $\pi(x)$ denote the number of prime numbers less than or equal to x. Then as x goes to infinity, $\pi(x)$ converges to $x/\ln x$. (Proof is far beyond the scope of this course.)

Implementation Concerns: Prime-finding

How do we find large primes p and q?

We don't know whether or not there's an algorithm that's guaranteed to find a prime efficiently at each time! But... we can pick random numbers, and test that they're prime.

Prime number theorem: Let $\pi(x)$ denote the number of prime numbers less than or equal to x. Then as x goes to infinity, $\pi(x)$ converges to $x/\ln x$. (Proof is far beyond the scope of this course.)

Main takeaway: primes aren't too uncommon. Pick a bunch of random numbers and one of them will probably be a prime.

Implementation Concerns: Prime-finding

How do we find large primes p and q?

We don't know whether or not there's an algorithm that's guaranteed to find a prime efficiently at each time! But... we can pick random numbers, and test that they're prime.

Prime number theorem: Let $\pi(x)$ denote the number of prime numbers less than or equal to x. Then as x goes to infinity, $\pi(x)$ converges to $x/\ln x$. (Proof is far beyond the scope of this course.)

Main takeaway: primes aren't too uncommon. Pick a bunch of random numbers and one of them will probably be a prime.

How do we test for primality efficiently?

Implementation Concerns: Prime-finding

How do we find large primes p and q?

We don't know whether or not there's an algorithm that's guaranteed to find a prime efficiently at each time! But... we can pick random numbers, and test that they're prime.

Prime number theorem: Let $\pi(x)$ denote the number of prime numbers less than or equal to x. Then as x goes to infinity, $\pi(x)$ converges to $x/\ln x$. (Proof is far beyond the scope of this course.)

Main takeaway: primes aren't too uncommon. Pick a bunch of random numbers and one of them will probably be a prime.

How do we test for primality efficiently? Lots of tests that will tell you "this is definitely not a prime" or "this may or may not be a prime" very quickly - simplest is based on Fermat's little theorem!

Implementation Concerns: Prime-finding

How do we find large primes p and q?

We don't know whether or not there's an algorithm that's guaranteed to find a prime efficiently at each time! But... we can pick random numbers, and test that they're prime.

Prime number theorem: Let $\pi(x)$ denote the number of prime numbers less than or equal to x. Then as x goes to infinity, $\pi(x)$ converges to $x/\ln x$. (Proof is far beyond the scope of this course.)

Main takeaway: primes aren't too uncommon. Pick a bunch of random numbers and one of them will probably be a prime.

How do we test for primality efficiently? Lots of tests that will tell you "this is definitely not a prime" or "this may or may not be a prime" very quickly - simplest is based on Fermat's little theorem! Efficient algorithm for distinguishing between "this is not a prime" and "this definitely is a prime" was found in 2002 by Agrawal, Kayal, Saxena - major breakthrough!

Implementation Concerns: Repeated Squaring

How about encrypting and decrypting? We need to do some pretty big exponents.

Implementation Concerns: Repeated Squaring

How about encrypting and decrypting? We need to do some pretty big exponents.

One way to do this efficiently: repeated squaring. Keep squaring the base and simplifying (since multiplication can easily be simplified under congruence).

Implementation Concerns: Repeated Squaring

How about encrypting and decrypting? We need to do some pretty big exponents.

One way to do this efficiently: repeated squaring. Keep squaring the base and simplifying (since multiplication can easily be simplified under congruence).

Example: compute $51^{43} \pmod{77}$.

Implementation Concerns: Repeated Squaring

How about encrypting and decrypting? We need to do some pretty big exponents.

One way to do this efficiently: repeated squaring. Keep squaring the base and simplifying (since multiplication can easily be simplified under congruence).

Example: compute $51^{43} \pmod{77}$.

$51^1 \equiv 51 \pmod{77}$

Implementation Concerns: Repeated Squaring

How about encrypting and decrypting? We need to do some pretty big exponents.

One way to do this efficiently: repeated squaring. Keep squaring the base and simplifying (since multiplication can easily be simplified under congruence).

Example: compute $51^{43} \pmod{77}$.

$51^1 \equiv 51 \pmod{77}$
$51^2 = (51) * (51) = 2601 \equiv 60 \pmod{77}$

Implementation Concerns: Repeated Squaring

How about encrypting and decrypting? We need to do some pretty big exponents.

One way to do this efficiently: repeated squaring. Keep squaring the base and simplifying (since multiplication can easily be simplified under congruence).

Example: compute $51^{43} \pmod{77}$.

$51^1 \equiv 51 \pmod{77}$
$51^2 = (51) * (51) = 2601 \equiv 60 \pmod{77}$
$51^4 = (51^2) * (51^2) = 60 * 60 = 3600 \equiv 58 \pmod{77}$

Implementation Concerns: Repeated Squaring

How about encrypting and decrypting? We need to do some pretty big exponents.

One way to do this efficiently: repeated squaring. Keep squaring the base and simplifying (since multiplication can easily be simplified under congruence).

Example: compute $51^{43} \pmod{77}$.

$51^1 \equiv 51 \pmod{77}$
$51^2 = (51) * (51) = 2601 \equiv 60 \pmod{77}$
$51^4 = (51^2) * (51^2) = 60 * 60 = 3600 \equiv 58 \pmod{77}$
$51^8 = (51^4) * (51^4) = 58 * 58 = 3364 \equiv 53 \pmod{77}$

Implementation Concerns: Repeated Squaring

How about encrypting and decrypting? We need to do some pretty big exponents.

One way to do this efficiently: repeated squaring. Keep squaring the base and simplifying (since multiplication can easily be simplified under congruence).

Example: compute $51^{43} \pmod{77}$.

$51^1 \equiv 51 \pmod{77}$
$51^2 = (51) * (51) = 2601 \equiv 60 \pmod{77}$
$51^4 = (51^2) * (51^2) = 60 * 60 = 3600 \equiv 58 \pmod{77}$
$51^8 = (51^4) * (51^4) = 58 * 58 = 3364 \equiv 53 \pmod{77}$
$51^{16} = (51^8) * (51^8) = 53 * 53 = 2809 \equiv 37 \pmod{77}$

Implementation Concerns: Repeated Squaring

How about encrypting and decrypting? We need to do some pretty big exponents.

One way to do this efficiently: repeated squaring. Keep squaring the base and simplifying (since multiplication can easily be simplified under congruence).

Example: compute $51^{43} \pmod{77}$.

$51^1 \equiv 51 \pmod{77}$
$51^2 = (51) * (51) = 2601 \equiv 60 \pmod{77}$
$51^4 = (51^2) * (51^2) = 60 * 60 = 3600 \equiv 58 \pmod{77}$
$51^8 = (51^4) * (51^4) = 58 * 58 = 3364 \equiv 53 \pmod{77}$
$51^{16} = (51^8) * (51^8) = 53 * 53 = 2809 \equiv 37 \pmod{77}$
$51^{32} = (51^{16}) * (51^{16}) = 37 * 37 = 1369 \equiv 60 \pmod{77}$

Implementation Concerns: Repeated Squaring

How about encrypting and decrypting? We need to do some pretty big exponents.

One way to do this efficiently: repeated squaring. Keep squaring the base and simplifying (since multiplication can easily be simplified under congruence).

Example: compute $51^{43} \pmod{77}$.

$51^1 \equiv 51 \pmod{77}$
$51^2 = (51) * (51) = 2601 \equiv 60 \pmod{77}$
$51^4 = (51^2) * (51^2) = 60 * 60 = 3600 \equiv 58 \pmod{77}$
$51^8 = (51^4) * (51^4) = 58 * 58 = 3364 \equiv 53 \pmod{77}$
$51^{16} = (51^8) * (51^8) = 53 * 53 = 2809 \equiv 37 \pmod{77}$
$51^{32} = (51^{16}) * (51^{16}) = 37 * 37 = 1369 \equiv 60 \pmod{77}$

$$51^{32} \cdot 51^8 \cdot 51^2 \cdot 51^1 = (60) * (53) * (60) * (51) \equiv 2 \pmod{77} \ .$$

Implementation Concerns: Repeated Squaring II

To compute $x^y \pmod{n}$:

1. x^y: Compute x^1,

Implementation Concerns: Repeated Squaring II

To compute $x^y \pmod{n}$:

1. x^y: Compute $x^1, x^2,$

Implementation Concerns: Repeated Squaring II

To compute $x^y \pmod{n}$:

1. x^y: Compute $x^1, x^2, x^4,$

Implementation Concerns: Repeated Squaring II

To compute $x^y \pmod{n}$:

1. x^y: Compute $x^1, x^2, x^4, \ldots,$

Implementation Concerns: Repeated Squaring II

To compute $x^y \pmod{n}$:

1. x^y: Compute $x^1, x^2, x^4, \ldots, x^{2^{\lfloor \log y \rfloor}}$.

Implementation Concerns: Repeated Squaring II

To compute $x^y \pmod{n}$:

1. x^y: Compute $x^1, x^2, x^4, \ldots, x^{2^{\lfloor \log y \rfloor}}$.
2. Multiply together x^i where the $(\log(i))$th bit of y (in binary) is 1.

Implementation Concerns: Repeated Squaring II

To compute $x^y \pmod{n}$:

1. x^y: Compute $x^1, x^2, x^4, \ldots, x^{2^{\lfloor \log y \rfloor}}$.
2. Multiply together x^i where the $(\log(i))$th bit of y (in binary) is 1. Example:

Implementation Concerns: Repeated Squaring II

To compute $x^y \pmod{n}$:

1. x^y: Compute $x^1, x^2, x^4, \ldots, x^{2^{\lfloor \log y \rfloor}}$.
2. Multiply together x^i where the $(\log(i))$th bit of y (in binary) is 1.
   Example: $43 = 101011$ in binary.

Implementation Concerns: Repeated Squaring II

To compute $x^y \pmod{n}$:

1. x^y: Compute $x^1, x^2, x^4, \ldots, x^{2^{\lfloor \log y \rfloor}}$.
2. Multiply together x^i where the $(\log(i))$th bit of y (in binary) is 1.
   Example: $43 = 101011$ in binary.

   $$x^{43} = x^{32} * x^8 * x^2 * x^1$$

   .

How many multiplications required?

Implementation Concerns: Repeated Squaring II

To compute $x^y \pmod{n}$:

1. x^y: Compute $x^1, x^2, x^4, \ldots, x^{2^{\lfloor \log y \rfloor}}$.
2. Multiply together x^i where the $(\log(i))$th bit of y (in binary) is 1.
   Example: $43 = 101011$ in binary.

   $$x^{43} = x^{32} * x^8 * x^2 * x^1$$

   .

How many multiplications required? $O(\log y)$. Much faster than multiplying y times!

Security of RSA

Why is RSA secure? Even without the private key, we have enough information to decrypt anything we see (we could just take the public key, encrypt every possible string representable as a number under N, and see which one matches the ciphertext).

Security of RSA

Why is RSA secure? Even without the private key, we have enough information to decrypt anything we see (we could just take the public key, encrypt every possible string representable as a number under N, and see which one matches the ciphertext).

The security RSA, like all almost all encryption schemes, relies on *hardness assumptions*. We need to assume something is hard in order to show that decrypting something, or even getting some information about the plaintext, *even with full information*, is hard.

Hardness Assumptions

What hardness assumptions are we making for RSA?

Hardness Assumptions

What hardness assumptions are we making for RSA?

"Given N, e, $c = x^e \pmod{N}$, there is no efficient algorithm for determining x."

Hardness Assumptions

What hardness assumptions are we making for RSA?

"Given N, e, $c = x^e \pmod{N}$, there is no efficient algorithm for determining x."

How would the someone snooping on our connection guess x?

- Brute force: try encrypting every possible string x. There are too many values of x: $2^{|x|}$. Can't do this efficiently*

Hardness Assumptions

What hardness assumptions are we making for RSA?

"Given N, e, $c = x^e \pmod{N}$, there is no efficient algorithm for determining x."

How would the someone snooping on our connection guess x?

- Brute force: try encrypting every possible string x. There are too many values of x: $2^{|x|}$. Can't do this efficiently*
- Factoring: Try determining d by factoring N into p and q, which would allow our spy to compute d the same way the bank did. Factoring large numbers is considered impossible to do efficiently.

Hardness Assumptions

What hardness assumptions are we making for RSA?

"Given N, e, $c = x^e \pmod{N}$, there is no efficient algorithm for determining x."

How would the someone snooping on our connection guess x?

- Brute force: try encrypting every possible string x. There are too many values of x: $2^{|x|}$. Can't do this efficiently*
- Factoring: Try determining d by factoring N into p and q, which would allow our spy to compute d the same way the bank did. Factoring large numbers is considered impossible to do efficiently.
- Direct computation of $(p-1)(q-1)$. Reduces to factoring. Why? If you compute $(p-1)(q-1) = pq - p - q + 1$, you now know what $p+q$ and pq are. Trivial to solve for p and q from here.

Hardness Assumptions

What hardness assumptions are we making for RSA?

“Given N, e, $c = x^e \pmod{N}$, there is no efficient algorithm for determining x.”

How would the someone snooping on our connection guess x?

- Brute force: try encrypting every possible string x. There are too many values of x: $2^{|x|}$. Can’t do this efficiently*
- Factoring: Try determining d by factoring N into p and q, which would allow our spy to compute d the same way the bank did. Factoring large numbers is considered impossible to do efficiently.
- Direct computation of $(p-1)(q-1)$. Reduces to factoring. Why? If you compute $(p-1)(q-1) = pq - p - q + 1$, you now know what $p+q$ and pq are. Trivial to solve for p and q from here.

Security of breaking RSA requires on hardness of factoring large integers.

RSA in Practice: Padding

SSNs are not particularly long. 9 digits. 1 billion possible SSNs.

[1] https://en.wikipedia.org/wiki/Optimal_asymmetric_encryption_padding

RSA in Practice: Padding

SSNs are not particularly long. 9 digits. 1 billion possible SSNs.

An iPhone 7 has a chip that clocks in at 2.34 GHz… wouldn't be too hard to encrypt every single SSN with a single public key and then run a lookup table.

[1] `https://en.wikipedia.org/wiki/Optimal_asymmetric_encryption_padding`

RSA in Practice: Padding

SSNs are not particularly long. 9 digits. 1 billion possible SSNs.

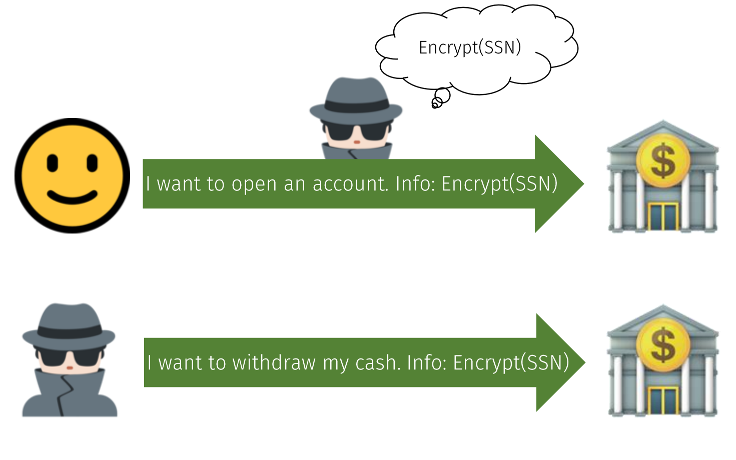
An iPhone 7 has a chip that clocks in at 2.34 GHz… wouldn't be too hard to encrypt every single SSN with a single public key and then run a lookup table.

To address this: "pad" the plaintext by appending extra junk bits to it to make it longer. Determining which junk bits would be secure is not trivial![1]

[1] https://en.wikipedia.org/wiki/Optimal_asymmetric_encryption_padding

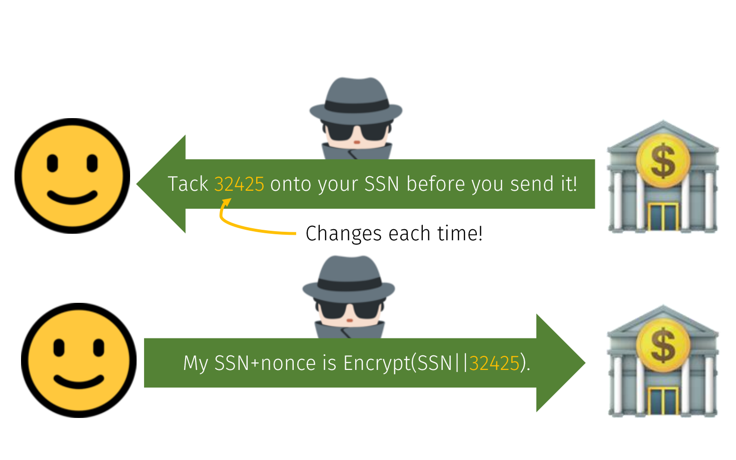
RSA in Practice: Replay Attacks, Nonces

Replay attack: if someone know your ciphertext, he can always send it again...



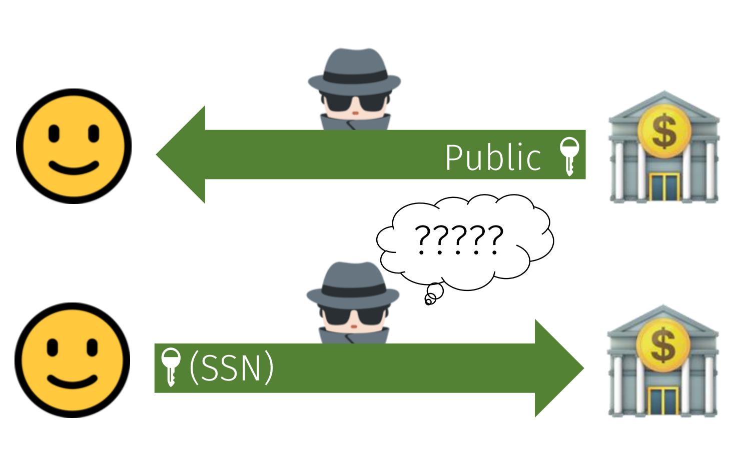
RSA in Practice: Replay Attacks, Nonces

Replay attack: if someone know your ciphertext, he can always send it again...Use a *nonce*: a one-time use string - that is concatenated to the plaintext before encryption.



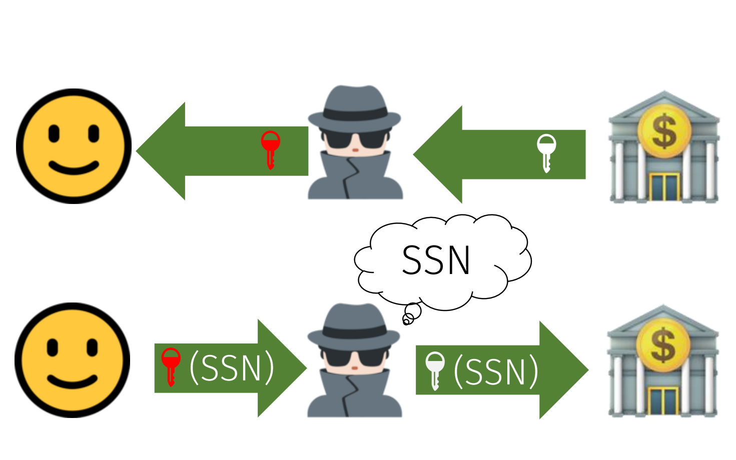
RSA in Practice: MITM

RSA allows you to protect your communication from snooping.



RSA in Practice: MITM

RSA allows you to protect your communication from snooping. It does **not** protect your communication from tampering ("man in the middle", or MITM attacks).



Protecting against MITM: Take One

Naive approach: store all the public keys in your computer (let's say you trust the computer manufacturer. Have it done at the factory).

Protecting against MITM: Take One

Naive approach: store all the public keys in your computer (let's say you trust the computer manufacturer. Have it done at the factory).

Problems?

Protecting against MITM: Take One

Naive approach: store all the public keys in your computer (let's say you trust the computer manufacturer. Have it done at the factory).

Problems?

How many websites are there where you want security? Banks, email, health... anything with a login, basically... you'd need a ton of disk space!

Protecting against MITM: Take One

Naive approach: store all the public keys in your computer (let's say you trust the computer manufacturer. Have it done at the factory).

Problems?

How many websites are there where you want security? Banks, email, health... anything with a login, basically... you'd need a ton of disk space!

What if you want to sign up for an account at a bank that was founded after you got your computer? Where do you get their key?

Protecting against MITM: Take One

Naive approach: store all the public keys in your computer (let's say you trust the computer manufacturer. Have it done at the factory).

Problems?

How many websites are there where you want security? Banks, email, health… anything with a login, basically… you'd need a ton of disk space!

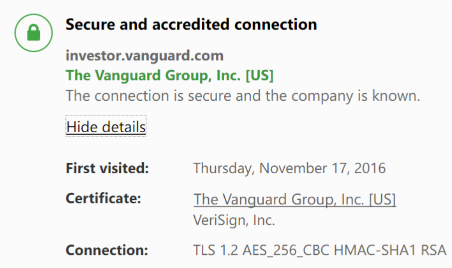
What if you want to sign up for an account at a bank that was founded after you got your computer? Where do you get their key?

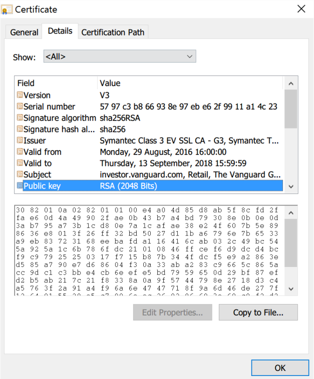
RSA Signatures

Another way to do it: RSA *signatures*. Idea: instead of storing every single public key with you, store the public key of somebody you trust - the “certificate authority”. The CA can then cryptographically endorse other keys to tell you “hey, it’s really them”.

RSA Signatures

Another way to do it: RSA *signatures*. Idea: instead of storing every single public key with you, store the public key of somebody you trust - the “certificate authority”. The CA can then cryptographically endorse other keys to tell you “hey, it’s really them”.





RSA Signatures

Vanguard has a *certificate* C that says “I’m Vanguard, and my public key is K_b.”

RSA Signatures

Vanguard has a *certificate* C that says "I'm Vanguard, and my public key is K_b."

Wants to have it *signed* by Verisign: "I'm Verisign, and I endorse this message." Verisign has an RSA public/private key pair: $K_V = (N, e)$, $k_v = d$, $N = pq$. Verisign's public key, K_v, is known by end users (preloaded into browsers and computers).

RSA Signatures

Vanguard has a *certificate* C that says "I'm Vanguard, and my public key is K_b."

Wants to have it *signed* by Verisign: "I'm Verisign, and I endorse this message." Verisign has an RSA public/private key pair: $K_V = (N, e)$, $k_v = d$, $N = pq$. Verisign's public key, K_v, is known by end users (preloaded into browsers and computers).

Verisign signature of C: $S_v(C) = D(C, k_v) = C^d \pmod{N}$.

RSA Signatures

Vanguard has a *certificate* C that says "I'm Vanguard, and my public key is K_b."

Wants to have it *signed* by Verisign: "I'm Verisign, and I endorse this message." Verisign has an RSA public/private key pair: $K_V = (N, e)$, $k_v = d$, $N = pq$. Verisign's public key, K_v, is known by end users (preloaded into browsers and computers).

Verisign signature of C: $S_v(C) = D(C, k_v) = C^d \pmod{N}$.

Browser receives C, the certificate, and $S_v(C)$, the signature. Check: Does $E(S_v(C), K_v)$ equal C? It should be, since

$$E(S_v(C), K_V = (S_v(C))^e = (C^d)^e = C^{de} = C \pmod{N}$$

RSA Signatures

Vanguard has a *certificate* C that says "I'm Vanguard, and my public key is K_b."

Wants to have it *signed* by Verisign: "I'm Verisign, and I endorse this message." Verisign has an RSA public/private key pair: $K_V = (N, e)$, $k_v = d$, $N = pq$. Verisign's public key, K_v, is known by end users (preloaded into browsers and computers).

Verisign signature of C: $S_v(C) = D(C, k_v) = C^d \pmod{N}$.

Browser receives C, the certificate, and $S_v(C)$, the signature. Check: Does $E(S_v(C), K_v)$ equal C? It should be, since

$$E(S_v(C), K_V = (S_v(C))^e = (C^d)^e = C^{de} = C \pmod{N}$$

What about security?

RSA Signatures

Vanguard has a *certificate* C that says "I'm Vanguard, and my public key is K_b."

Wants to have it *signed* by Verisign: "I'm Verisign, and I endorse this message." Verisign has an RSA public/private key pair: $K_V = (N, e)$, $k_v = d$, $N = pq$. Verisign's public key, K_v, is known by end users (preloaded into browsers and computers).

Verisign signature of C: $S_v(C) = D(C, k_v) = C^d \pmod{N}$.

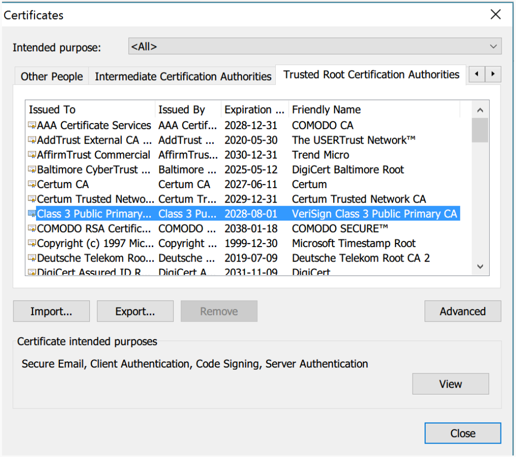
Browser receives C, the certificate, and $S_v(C)$, the signature. Check: Does $E(S_v(C), K_v)$ equal C? It should be, since

$$E(S_v(C), K_V = (S_v(C))^e = (C^d)^e = C^{de} = C \pmod{N})$$

What about security? Making the signature requires computing $D(C, k_v)$ which is hard without k_v. Same security analysis of RSA applies!

Whom do you trust?

You need to trust the browser vendor/computer manufacturer that gave you the list of trusted CAs initially *and* trust the CA to only sign legitimate certificates.



What happens when trust breaks down?

What happens when trust breaks down?

TNW

Superfish admits installing root certificate authority to show ads on secure sites

by OWEN WILLIAMS — Feb 20, 2015 in INSIDER

ars technica

RISK ASSESSMENT / SECURITY & HACKTIVISM

Dell does a Superfish, ships PCs with easily cloneable root certificates

Root certificate debacle that hit Lenovo now visits the House of Dell.

What happens when trust breaks down?

Bugzilla@Mozilla

New Account | Log In | Forgot Password

mozilla

Home New Browse Search

Search [help] Reports

Product Dashboard

Bug 1311713 - CA Comodo used broken OCR and issued certificates to the wrong people

Last Comment

Debian Bug report logs - #744027

Please remove StartCom Certification Authority root certificate

Questions?