Applied Cryptography and Network Security

William Garrison
bill@cs.pitt.edu
6311 Sennott Square

RSA and DSA

University of Pittsburgh

Didn't we learn about RSA last time?

During the last lecture, we saw what RSA does and learned a little bit about how we can use those features

Our goal today will be to explore

- Why RSA actually works
- Why RSA is efficient* to use
- Why it is reasonably safe to use RSA

In short, it's another details day…

Note: Efficiency is a relative term ☺

RSA Overview / Roadmap

How do we choose large, pseudo-random primes?!

Key generation:

- Choose two large prime numbers p and q, compute $n = pq$
- Compute $\phi(n) = (p-1)(q-1)$
- Choose an integer e such that $\gcd(e, \phi(n)) = 1$
- Calculate *d* such that $ed \equiv 1 \pmod{\phi(n)}$
- Public key: n, e
- Private key: p, q, d

Why is $\phi(n) = (p-1)(q-1)$

How can we do this?

This seems tricky, too

Isn't this expensive?

Usage:

- Encryption: $M^e \pmod{n}$
- Decryption: $C^d \pmod{n} = M^{ed} \pmod{n} = M^{k\phi(n)+1} \pmod{n} = M^1 \pmod{n} = M$

Why does this work?

Before we can do anything, we need a few large, pseudo-random primes

If our numbers are small, primality testing is pretty easy

- Try to divide n by all numbers less than $\sqrt{n}$
- The Sieve of Eratosthenes is a general extension of this principle

RSA requires big primes, so brute force testing is not an option (Why?)

To choose the types of numbers that RSA needs, we instead use a probabilistic primality testing method test : Z×Z → {T, F}

- test(n, a) = F means that n is composite based on the witness a
- test(n, a) = T means that n is probably prime based on the witness a

Often with probability ≈ 1/2

To test a number n for primality:

1. Randomly choose a witness a
2. if test(n, a) = F, n is composite
3. if test(n, a) = T, loop until we're reasonably certain that n is prime

k repetitions means Pr(n composite) = 2^{-k}

Fermat's little theorem can help us!

Fermat's little theorem: Given a prime number p and a natural number a such that $1 \leq a < p$, then $a^{p-1} \equiv 1 \bmod p$

How does this help with primality testing?

- If $a^{p-1} \neq 1 \bmod p$, then p is definitely composite
- If $a^{p-1} \equiv 1 \bmod p$, then p is probably prime

Note: Some composite numbers will always pass this test (Yikes!)

- These are called Carmichael numbers
- Carmichael numbers are rare, but may still be found
- Other primality tests (e.g., Miller-Rabin) avoid false positives on these numbers

This helps us test whether some number is prime. But how exactly does this help us generate RSA keys?

Putting it all together…

```
foundPrime = false
while (!foundPrime)
    let r = some large, odd, random number
    foundPrime = true
    for (iters = 0; iters < k; iters++)
        let a = random number less than r
        if (a^(r-1) ≠ 1 mod r)
            foundPrime = false
            break
return r
```

The prime number theorem tells us that, on average, the number of primes less than n is approximately n/ln(n)

- That is, P(n is prime) ≈ 1/ln(n)
- Searching for a prime is hard, but not ridiculously so

RSA Overview / Roadmap

How do we choose large, pseudo-random primes?!

Key generation:

- ✔ Choose two large prime numbers p and q, compute $n = pq$
- Compute $\phi(n) = (p-1)(q-1)$
- Choose an integer e such that $\gcd(e, \phi(n)) = 1$
- Calculate *d* such that $ed \equiv 1 \pmod{\phi(n)}$
- Public key: n, e
- Private key: p, q, d

Why is $\phi(n) = (p-1)(q-1)$

How can we do this?

This seems tricky, too

Isn't this expensive?

Usage:

- Encryption: $M^e \pmod{n}$
- Decryption: $C^d \pmod{n} = M^{ed} \pmod{n} = M^{k\phi(n)+1} \pmod{n} = M^1 \pmod{n} = M$

Why does this work?

$\varphi(n)$ is called Euler's totient function

Definition: The totient function, $\varphi(n)$, counts the number of elements less than n that are relatively prime to n

For an RSA modulus $n = pq$, calculating $\varphi(n)$ is actually pretty simple

Consider each of the pq numbers $\leq n$

- All multiples of p share a common factor with n
  - There are q such numbers $\{p, 2p, 3p, \ldots, qp\}$
- Similarly, all multiples of q share a common factor with n
  - There are p such numbers $\{q, 2q, 3q, \ldots, pq\}$
- So, we have that $\varphi(n) = pq - p - q + 1$
- As a result, $\varphi(n) = pq - p - q + 1 = (p-1)(q-1)$

The +1 controls for subtracting pq twice

Note: Calculating $\varphi(n)$ is easy because we know how to factor n!

RSA Overview / Roadmap

Key generation:

- ✔ Choose two large prime numbers p and q, compute $n = pq$
- ✔ Compute $\phi(n) = (p-1)(q-1)$
- Choose an integer e such that $\gcd(e, \phi(n)) = 1$
- Calculate *d* such that $ed \equiv 1 \pmod{\phi(n)}$
- Public key: n, e
- Private key: p, q, d

Why is $\phi(n) = (p-1)(q-1)$

How can we do this?

This seems tricky, too

Isn't this expensive?

Usage:

- Encryption: $M^e \pmod{n}$
- Decryption: $C^d \pmod{n} = M^{ed} \pmod{n} = M^{k\phi(n)+1} \pmod{n} = M^1 \pmod{n} = M$

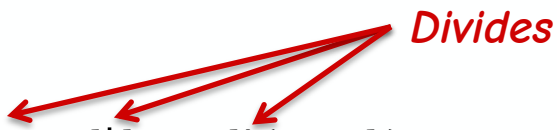
Why does this work?

Luckily, computing GCDs is not all that hard…

Intuition: Rather than computing the GCD of two big numbers, we can instead compute the GCD of smaller numbers that have the same GCD!

Interesting observation: $\gcd(a, b)$ is the same as $\gcd(a - b, b)$

Wait, what?



First, we must show that $d|a \wedge d|b \rightarrow d|(a - b)$

- If $d|a$ and $d|b$, then $a = kd$ and $b = jd$
- Then $a - b = kd - jd = (k - j)d$
- So, $d|a \wedge d|b \rightarrow d|(a - b)$

Ok, so d is a divisor of $(a - b)$, but is it the greatest divisor?

- The divisors of $(a - b)$ are a subset of of the divisors of a and the divisors of b
- Since $d = \gcd(a, b)$, it is the greatest of the remaining divisors

Euclid's algorithm optimizes this process!

Euclid's algorithm finds $\gcd(a, b)$ as follows:

- Set $r_{-1} = a,\ r_{-2} = b,\ n = 0$
- While $r_{n-1} \neq 0$
  - divide r_{n-2} by r_{n-1} to find q_n and r_n such that $r_{n-2} = q_n r_{n-1} + r_n$
  - $n = n + 1$
- $\gcd(a, b) = r_{n-2}$

Example: Computing gcd(414, 662)

n	q_n	r_n
-2	-	662
-1	-	414
0	1	248
1	1	166
2	1	82
3	2	2
4	41	0

How do we use this when choosing e?

Method 1: Use Euclid's algorithm

- Choose a random e
- Use Euclid's algorithm to determine whether $\gcd\big(e, \phi(n)\big) = 1$
- Repeat as needed

Method 2: We can just choose a large prime number, $r > \max(p, q)$

Why does method 2 work?

- r is a prime number, so it has no divisors other than itself and 1
- r is larger than p and q, so $r \neq p$ and $r \neq q$

RSA Overview / Roadmap

Key generation:

- ✔ Choose two large prime numbers p and q, compute $n = pq$
- ✔ Compute $\phi(n) = (p-1)(q-1)$
- ✔ Choose an integer e such that $\gcd(e, \phi(n)) = 1$
- Calculate *d* such that $ed \equiv 1 \pmod{\phi(n)}$
- Public key: n, e
- Private key: p, q, d

How can we do this?

This seems tricky, too

Isn't this expensive?

Usage:

- Encryption: $M^e \pmod{n}$
- Decryption: $C^d \pmod{n} = M^{ed} \pmod{n} = M^{k\phi(n)+1} \pmod{n} = M^1 \pmod{n} = M$

Why does this work?

It turns out that Euclid's algorithm can help us compute $d \equiv e^{-1}$, too

If we maintain a little extra state, we can figure out numbers u_n and v_n such that $r_n = u_n a + v_n b$ when calculating $\gcd(a, b)$

If a and b are relatively prime, this will allow us to calculate a^{-1}

- $1 = u_n a + v_n b$ // If a and b are relatively prime, $r_n = 1$
- $u_n a = 1 - v_n b$ // Subtract $v_n b$ from both sides
- $u_n a \equiv 1 \bmod b$ // Definition of congruence
- So $u_n = a^{-1}$ // Definition of inverse

This makes $r_n = u_n a + v_n b$ for $n = -1$ and $n = -2$

The extended Euclid's algorithm works as follows:

- Set $r_{-1} = b,\ r_{-2} = a,\ n = 0,\ u_{-2} = 1,\ v_{-2} = 0,\ u_{-1} = 0,\ v_{-1} = 1$
- While $r_{n-1} \neq 0$
  - divide r_{n-2} by r_{n-1} to find q_n and r_n such that $r_{n-2} = q_n r_{n-1} + r_n$
  - $u_n = u_{n-2} - q_n u_{n-1}$
  - $v_n = v_{n-2} - q_n v_{n-1}$
  - $n = n + 1$
- $\gcd(a, b) = r_{n-2} = u_{n-2} a + v_{n-2} b$

How about an example?

Example: Find the inverse of 797 mod 1047

n	q_n	r_n	u_n	v_n
-2		797	1	0
-1		1047	0	1
0	0	797	1	0
1	1	250	-1	1
2	3	47	4	-3
3	5	15	-21	16
4	3	2	67	-51
5	7	1	-490	373

So, 1 = -490*797 + 373*1047

- -490*797 = 1 + (-373)*1047
- -490*797 ≡ 1 mod 1047
- In other words, -490 is the inverse of 797 mod 1047

RSA Overview / Roadmap

Key generation:

- ✔ Choose two large prime numbers p and q, compute $n = pq$
- ✔ Compute $\phi(n) = (p-1)(q-1)$
- ✔ Choose an integer e such that $\gcd(e, \phi(n)) = 1$
- ✔ Calculate *d* such that $ed \equiv 1 \pmod{\phi(n)}$
- Public key: n, e
- Private key: p, q, d

This seems tricky, too

Isn't this expensive?

Usage:

- Encryption: $M^e \pmod{n}$
- Decryption: $C^d \pmod{n} = M^{ed} \pmod{n} = M^{k\phi(n)+1} \pmod{n} = M^1 \pmod{n} = M$

Why does this work?

Isn't exponentiation really expensive?

Exponentiation can be sped up using a trick called successive squaring

```
int pow(int m, int e)
    if(e is even)
        return pow(m*m, e/2)
    else
        return m * pow(m, e – 1)
```

For example, consider computing 2^{15}

- Naive method: 2 * 2 * 2 * … * 2 = 32,768
- Fast method: $2^{15} = 2 * 4^7$
  - $= 2 * 4 * 4^6$
  - $= 2 * 4 * 16^3$
  - $= 2 * 4 * 16 * 16^2$
  - $= 2 * 4 * 16 * 256$
  - $= 32{,}768$

O(e) multiplications

O(log(e)) multiplications

This only gets us partway there. Various other algorithmic tricks enable modulo exponentiation to be efficient! (e.g., take the mod every time, rather than at the end)

RSA Overview / Roadmap

Key generation:

- ✔ Choose two large prime numbers p and q, compute $n = pq$
- ✔ Compute $\phi(n) = (p-1)(q-1)$
- ✔ Choose an integer e such that $\gcd(e, \phi(n)) = 1$
- ✔ Calculate *d* such that $ed \equiv 1 \pmod{\phi(n)}$
- Public key: n, e
- Private key: p, q, d

Isn't this expensive?

Usage:

- ✔ Encryption: $M^e \pmod{n}$
- Decryption: $C^d \pmod{n} = M^{ed} \pmod{n} = M^{k\phi(n)+1} \pmod{n} = M^1 \pmod{n} = M$

Why does this work?

Why *does* decryption work?

Note: Decryption will work if and only if $C^d \bmod n = M$

$$
\begin{aligned}
C^d \bmod n &= M^{ed} \bmod n && // \; C = M^e \bmod n \\
&= M^{k\phi(n)+1} \bmod n && // \; ed \equiv 1 \bmod \phi(n), \text{ so } ed = k\phi(n) + 1 \\
&= M^1 \bmod n && // \; ?!? \\
&= M \bmod n && // \; M^1 = M
\end{aligned}
$$

The only hitch in showing the correctness of the decryption process is proving that $M^{k\phi(n)+1} \bmod n = M^1 \bmod n$

Fortunately, two smart guys can help us out with this…

Pierre de Fermat
160? - 1665

Leonhard Euler
1707 - 1783

RSA Overview / Roadmap

Key generation:

- ✔ Choose two large prime numbers p and q, compute $n = pq$
- ✔ Compute $\phi(n) = (p-1)(q-1)$
- ✔ Choose an integer e such that $\gcd(e, \phi(n)) = 1$
- ✔ Calculate *d* such that $ed \equiv 1 \pmod{\phi(n)}$
- Public key: n, e
- Private key: p, q, d

Usage:

- ✔ Encryption: $M^e \pmod{n}$
- ✔ Decryption: $C^d \pmod{n} = M^{ed} \pmod{n} = M^{k\phi(n)+1} \pmod{n} = M^1 \pmod{n} = M$

But why is RSA *safe* to use?

Now, why exactly is RSA safe to use?

In the original RSA paper*, the authors identify four avenues for attacking the mathematics behind RSA

1. Factoring n to find p and q
2. Determining $\phi(n)$ without factoring n
3. Determining d without factoring n or learning $\phi(n)$
4. Learning to take e^{th} roots modulo n

As it turns out, all of these attacks are thought to be hard to do

- But you shouldn't take my word for it…
- Let's see why!

*R.L. Rivest, A. Shamir, and L. Adleman, A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, Communications of the ACM 21(2): 120-126, Feb. 1978.

It turns out that factoring is a hard* problem

First of all, why is factoring an issue?

- n is the public modulus of the RSA algorithm
- If we can factor n to find p and q, we can compute $\phi(n)$
- Given $\phi(n)$ and e, we can easily compute the decryption exponent d

---

Fortunately, mathematicians believe that factoring numbers is a very difficult problem. History backs up this belief.

The fastest general-purpose algorithm for integer factorization is called the general number field sieve. This algorithm has running time:

$$O\left(e^{(c+o(1))(\log n)^{\frac{1}{3}}(\log\log n)^{\frac{2}{3}}}\right)$$

Note: This running time is sub-exponential

- i.e., Factoring can be done faster than brute force
- This explains why RSA keys are larger than AES keys
  - RSA: Typically ≥3072 bits
  - AES: Typically ≥128 bits
  - https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r5.pdf

Also true for discrete log / Diffie-Hellman

What about computing $\phi(n)$ without factoring?

Question: Why would the ability to compute $\phi(n)$ be a bad thing?

- It would allow us to easily compute d, since $ed \equiv 1 \bmod \phi(n)$

Good news: If we can compute $\phi(n)$, it will allow us to factor n

- Note 1: $\phi(n) = n - p - q + 1$
  $= n - (p + q) + 1$
- Rewriting gives us $(p + q) = n - \phi(n) + 1$
- Note 2: $(p - q) = \sqrt{(p + q)^2 - 4n}$
- Note 3: $(p + q) - (p - q) = 2q$
- Finally, given q and n, we can easily compute p

$$(p+q)^2 - 4n = p^2 + 2pq + q^2 - 4n$$
$$= p^2 + 2pq + q^2 - 4pq$$
$$= p^2 - 2pq + q^2$$
$$= (p-q)^2$$

What does this mean?

- If factoring is actually hard, then so is computing $\phi(n)$ without factoring
- (Recall the concept of reduction)

What about computing d without factoring n or knowing $\phi(n)$?

As it turns out, if we can figure out d without knowing $\phi(n)$ and without factoring n, d can be used to help us factor n

---

Given d, we can compute $ed - 1$, since we know e

Note: $ed - 1$ is a multiple of $\phi(n)$

- $ed \equiv 1 \bmod \phi(n)$
- $ed = 1 + k\phi(n)$
- $ed - 1 = k\phi(n)$ ✔

It has been shown that n can be efficiently factored using any multiple of $\phi(n)$. As such, if we know e and d, we can efficiently factor n.

Are there any other attacks that we need to worry about?

Recall: $C = M^e \bmod n$

- e is part of the public key, so the adversary knows this
- If we could compute e^{th} roots mod n, we could decrypt without d

It is not known whether breaking RSA yields an efficient factoring algorithm, but the inventors conjecture that this is the case

- This conjecture was made in 1978
- To date, it has either been proved or disproved

Conclusion: Odds are that breaking RSA efficiently implies that factoring can be done efficiently. Since factoring is hard, RSA is probably safe to use.

Implementation concerns, and padding

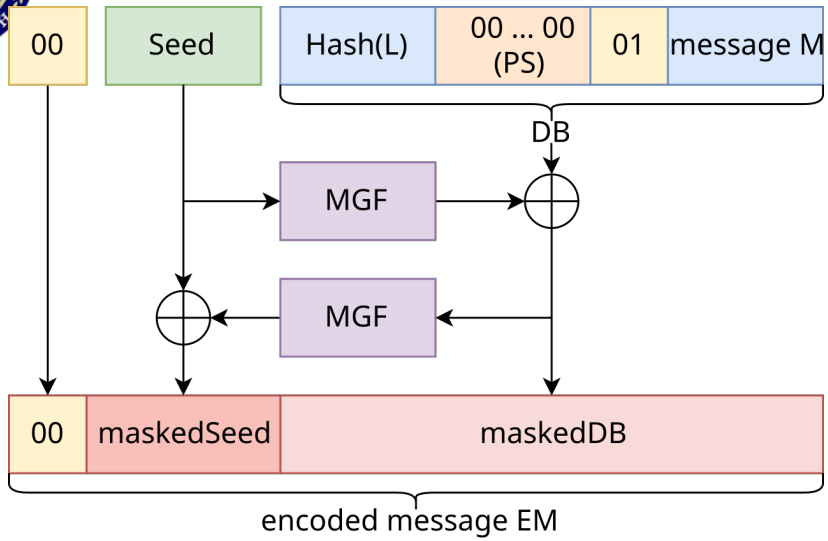
For many reasons, we very rarely encrypt or sign raw messages

- (We discussed hybrid cryptography last time; even there, we don't encrypt the raw symmetric key or sign the raw hash)
- Small m and small e can combine to create easy decryptions
- Multiplicative homomorphism
  - $m_1^e * m_2^e \equiv (m_1 m_2)^e \pmod{n}$
  - We'll see issues this causes later in the term
- The same value encrypts the same every time
  - No semantic security!

Instead, PKCS#1 defines padding that is applied before encryption/signing

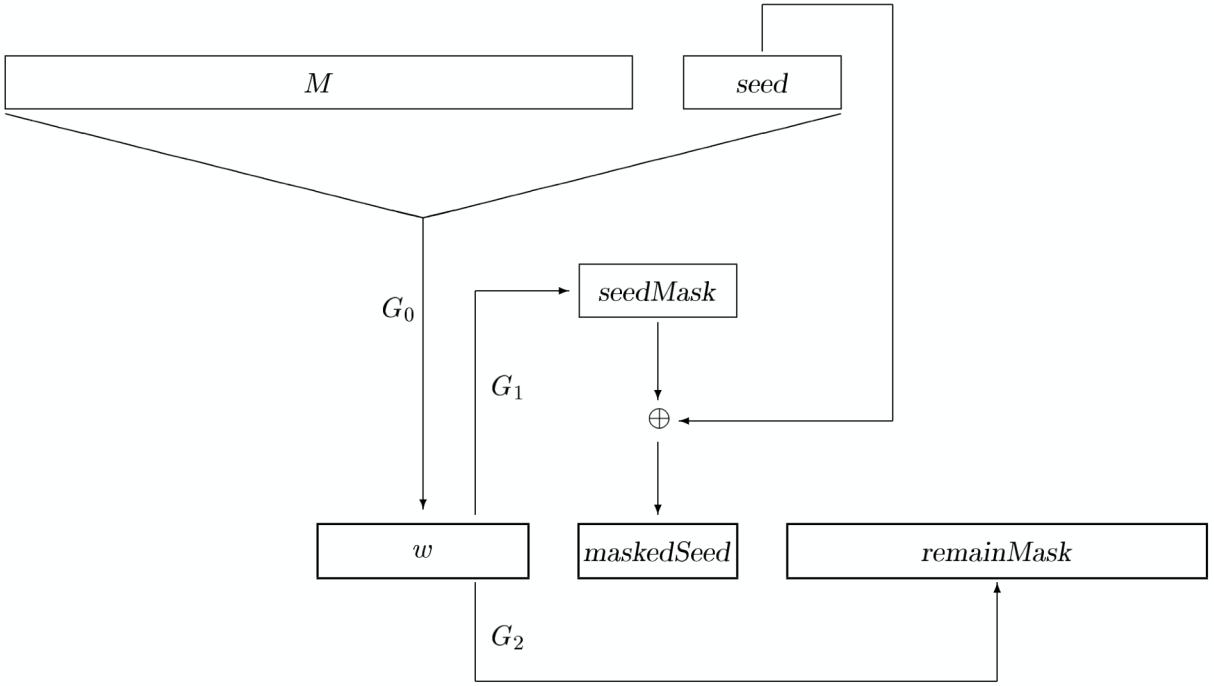
- OAEP for encryption: Adds randomness, pads to fixed length, breaks mult. homomorphism, easy to undo after decryption
- PSS for signing: One-way, adds randomness and padding, breaks mult. homomorphism

OAEP and PSS block diagrams



OAEP

PSS



ElGamal: A predecessor to DSA using Diffie-Hellman-style keys

Assume this setup:

- Agreed-upon g and p
- Signer has generated keypair a, $A = g^a \bmod p$ and shared A

Producing a signature:

- $M = \text{hash}(m)$, and t is a per-message secret
  - t must be coprime with $p - 1$
- $T = g^t \bmod p$, $S = (M - aT) * t^{-1} \bmod (p - 1)$
- The signature is $\langle T, S \rangle$

Verifying a signature:

- (Check that $0 < T < p - 1$ and $0 < S < p - 1$)
- Recompute M
- Verify that $g^M = A^T * T^S \bmod p$

Why does ElGamal work?

Let's do some algebra!

- $A^T * T^S \bmod p = (g^a)^T * (g^t)^S \bmod p = g^{aT+tS} \bmod p$
- $A^T * T^S \bmod p = g^{aT+tS \bmod (p-1)} \bmod p$, by FLT
- Notes that: $tS = (t * (M - aT) * t^{-1}) \bmod (p-1) = M - aT \bmod (p-1)$ since $t * t^{-1} \equiv 1 \bmod (p-1)$ (modl'r inverses)
- So $A^T * T^S \bmod p = g^{aT+M-aT \bmod (p-1)} \bmod p = g^{M \bmod (p-1)} \bmod p$
- Since M is a hash and p is large*, we can assume $M < p - 1$
- $A^T * T^S \bmod p = g^M \bmod p$ 🥳🎉

But why is it secure? This is less obvious and left as an exercise; we want to be convinced that:

- If m (and thus M) is changed, verification will fail
- Without a, one cannot compute T and S
- Given A and a collection of signatures, one can't determine a

In 1991–94, NIST standardized DSA, a more efficient derivative of ElGamal

DSA: Digital Signature Algorithm, also based on discrete log (DH-like)

- For efficiency, some operations use a second smaller prime, q, which divides $p - 1$
  - $|p|$ as usual for discrete log, $|q|$ is 2× the security level
- Keypairs are the same as ElGamal ($A = g^a \bmod p$)

Signing:

- $T = \left((g^t \bmod p) \bmod q\right)$ (Note the addition of $\bmod q$)
- $S = (M + aT) * t^{-1} \bmod q$ (Rather than $\bmod p - 1$)
- The signature is, again, $\langle T, S \rangle$

Verification:

- Compute $x = M * S^{-1} \bmod q$, $y = T * S^{-1} \bmod q$
- Verify that $T = (g^x * A^y \bmod p) \bmod q$
  - Try the algebra to prove that this works!

A brief note about efficiency…

Cryptosystems based on factoring and discrete log require keys much larger than security level

- (as discussed previously)
- For 128-bit security, $|p| = 3072$ (and $|q| = 256$ for DSA)

Can we achieve the same security with smaller keys?

- Yes!
- … but it requires different math
- ~~Individual numbers~~ → points on an elliptic curve
- ~~Multiplication~~ → specialized formulas for "multiplying" two points

Elliptic-curve cryptography (ECC) is faster than RSA (etc.), especially for the private operations

- (for the same security level, thanks to smaller keys)
- We'll learn more about ECC later in the term!

Conclusions

We've looked at the details of the "first-generation" public-key cryptosystems

- Diffie-Hellman (key exchange)
- RSA (encryption/decryption, signing/verifying)
- DSA (signing/verifying, based on ElGamal)

We'll see these, and other public-key systems, again later

- DSA is better, but all are slow (see Elliptic-curve cryptography)
- Although security is based on conditional proofs, primitives can be misused, and implementations can be broken (see Subverting cryptography)
  - And today's hard problems might not be hard forever! (see Post-quantum cryptography)

We're done with the main cryptographic primitives!

Next: Authentication and password storage