Introduction to Public-Key Cryptography

Nadia Heninger

University of Pennsylvania

June 11, 2018

"We stand today on the brink of a revolution in cryptography."

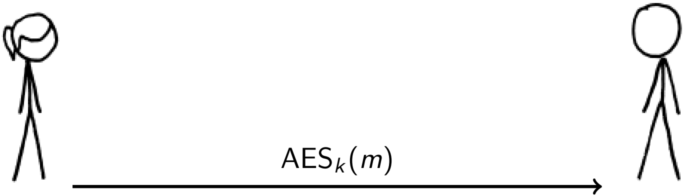
— Diffie and Hellman, 1976

New Directions in Cryptography

Invited Paper

WHITFIELD DIFFIE AND MARTIN E. HELLMAN, MEMBER, IEEE

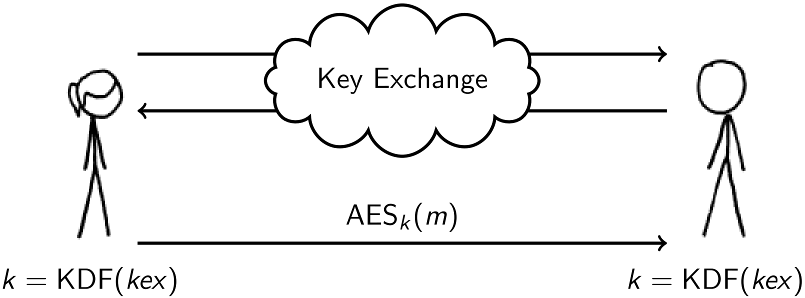
Symmetric cryptography



* Toy protocol for illustration purposes only; not secure.

Public key crypto idea # 1: Key exchange

Solving key distribution without trusted third parties



* Toy protocol for illustration purposes only; not secure.

Textbook Diffie-Hellman

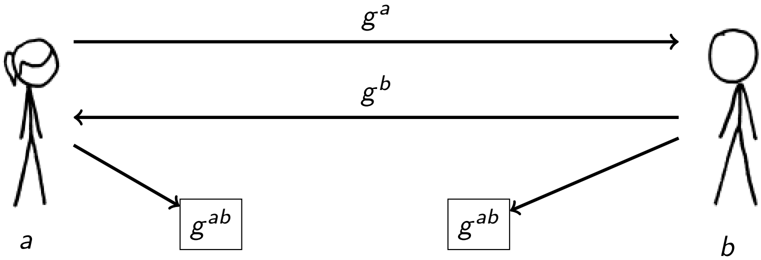
[Diffie Hellman 1976]

Public Parameters

G a cyclic group (e.g. $\mathbb{F}_p^*$, or an elliptic curve)

g group generator

Key Exchange

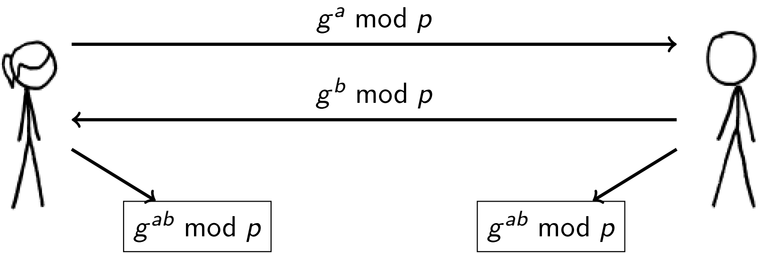


Finite-Field Diffie-Hellman

Public Parameters

- p a prime
- q a subgroup order; $q \mid (p-1)$
- g a generator of multiplicative group of order $q \in \mathbb{F}_p^*$

Key Exchange



The Discrete Log Problem

Problem: Given $g^a \bmod p$, compute a.

- Solving this problem permits attacker to compute shared key by computing a and raising $(g^b)^a$.
- Discrete log is in NP and coNP $\rightarrow$ not NP-complete (unless P=NP or similar).
- Shor's algorithm solves discrete log with a quantum computer in polynomial time.

The Computational Diffie-Hellman problem

Problem: Given $g^a \bmod p$, $g^b \bmod p$, compute $g^{ab} \bmod p$.

- Exactly problem of computing shared key from public information.
- Reduces to discrete log in some cases:
  - "Diffie-Hellman is as strong as discrete log for certain primes" [den Boer 1988] "both problems are (probabilistically) polynomial-time equivalent if the totient of $p-1$ has only small prime factors"
  - "Towards the equivalence of breaking the Diffie-Hellman protocol and computing discrete logarithms" [Maurer 1994] "if . . . an elliptic curve with smooth order can be construted efficiently, then . . . [the discrete log] can be reduced efficiently to breakingthe Diffie-Hellman protocol"
- Computational Diffie-Hellman Assumption: No efficient algorithm to solve this problem.

Decisional Diffie-Hellman problem

Problem: Given $g^a \bmod p$, $g^b \bmod p$, distinguish $g^{ab} \bmod p$ from random.

- Decisional Diffie-Hellman Assumption: No efficient algorithm has better than negligible advantage.
- Required for most security proofs.

Selecting parameters for finite-field Diffie-Hellman

For 128-bit security:

- Choose $\geq$ 256-bit q.
  - Pollard rho/Baby step-giant step algorithm: $O(\sqrt{q})$

Selecting parameters for finite-field Diffie-Hellman

For 128-bit security:

- Choose $\geq$ 256-bit q.
  - Pollard rho/Baby step-giant step algorithm: $O(\sqrt{q})$
- Choose prime group order q.
  - (Pohlig-Hellman algorithm: as secure as largest factor of q.)

Selecting parameters for finite-field Diffie-Hellman

For 128-bit security:

- Choose $\geq$ 256-bit q.
  - Pollard rho/Baby step-giant step algorithm: $O(\sqrt{q})$
- Choose prime group order q.
  - (Pohlig-Hellman algorithm: as secure as largest factor of q.)
- Choose $\geq$ 256-bit exponents a, b.
  - Pollard lambda algorithm: $O(\sqrt{a})$

Selecting parameters for finite-field Diffie-Hellman

For 128-bit security:

- Choose $\geq$ 256-bit q.
  - Pollard rho/Baby step-giant step algorithm: $O(\sqrt{q})$
- Choose prime group order q.
  - (Pohlig-Hellman algorithm: as secure as largest factor of q.)
- Choose $\geq$ 256-bit exponents a, b.
  - Pollard lambda algorithm: $O(\sqrt{a})$
- Choose $\geq$ 2048-bit prime modulus p.
  - Number field sieve algorithm: $O(\exp(1.92 \ln p^{1/3} (\ln \ln p)^{2/3}))$

Selecting parameters for finite-field Diffie-Hellman

For 128-bit security:

- Choose $\geq$ 256-bit q.
  - Pollard rho/Baby step-giant step algorithm: $O(\sqrt{q})$
- Choose prime group order q.
  - (Pohlig-Hellman algorithm: as secure as largest factor of q.)
- Choose $\geq$ 256-bit exponents a, b.
  - Pollard lambda algorithm: $O(\sqrt{a})$
- Choose $\geq$ 2048-bit prime modulus p.
  - Number field sieve algorithm: $O(\exp(1.92 \ln p^{1/3} (\ln \ln p)^{2/3}))$
- Choose nothing-up-my-sleeve p (e.g. digits of π, e)
  - Special number field sieve: $O(\exp(1.53 \ln p^{1/3} (\ln \ln p)^{2/3}))$

Real-world finite field DH implementation choices

- 1024-bit primes remain common in practice.
- Many standardized, hard-coded primes.
- 1024-bit primes baked into SSH, IPsec, but have been deprecated by some implementations.
- NIST recommends working within smaller order subgroup (e.g. 160 bits for 1024-bit prime)
- Many implementations use short exponents (e.g. 256 bits)
- DDH often false in practice: many implementations generate full group mod p.
- Support for FF-DH has dropped rapidly in TLS in favor of ECDH.

My personal recommendation

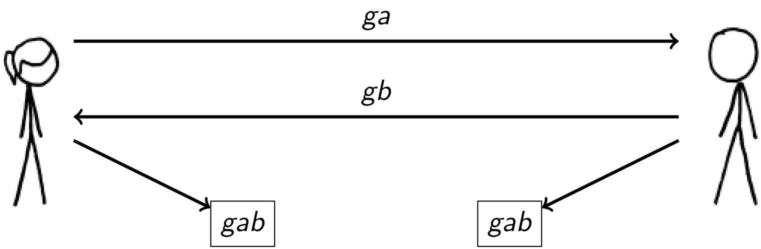
- **Don't use prime-field Diffie-Hellman at all.**
- Too many implementation vulnerabilities.
- ECDH is more secure (classically) as far as we know.

Elliptic-Curve Diffie-Hellman

Public Parameters

$\mathbb{E}$ an elliptic curve

g a group generator



Selecting parameters for elliptic-curve Diffie-Hellman

For 128-bit security:

- Choose a 256-bit curve.
  - (ECDH keys are shorter because fewer strong attacks.)
- See Craig's talk later today!

Real-world implementation choices for ECDH

- ECDH rapidly becoming more common than FF-DH.
- NIST p256 most common curve.

Post-quantum Diffie-Hellman

- Promising Candidate: Supersingular Isogeny Diffie-Hellman

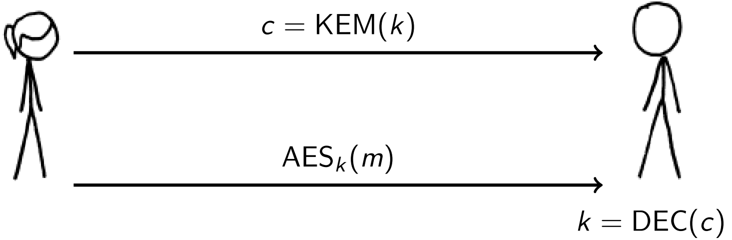
  See Craig's talk on Friday for more!

- Diffie-Hellman from lattices: situation is complex.

  See Douglas's talk later today for more!

Idea # 2: Key encapsulation/public-key encryption

Solving key distribution without trusted third parties



* Toy protocol for illustration purposes only; not secure.

A Method for Obtaining Digital Signatures and Public-Key Cryptosystems

R.L. Rivest, A. Shamir, and L. Adleman*

Textbook RSA Encryption

[Rivest Shamir Adleman 1977]

Public Key

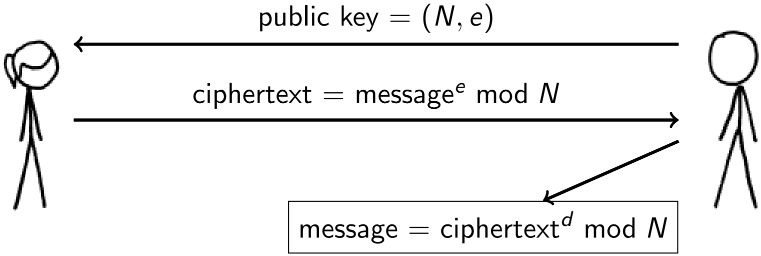
$N = pq$ modulus

e encryption exponent

Private Key

p, q primes

d decryption exponent
$(d = e^{-1} \bmod (p-1)(q-1))$



Factoring

Problem: Given N, compute its prime factors.

- Computationally equivalent to computing private key d.
- Factoring is in NP and coNP $\rightarrow$ not NP-complete (unless P=NP or similar).
- Shor's algorithm factors integers on a quantum computer in polynomial time.

eth roots mod N

Problem: Given N, e, and c, compute x such that $x^e \equiv c \bmod N$.

- Equivalent to decrypting an RSA-encrypted ciphertext.
- Not known whether it reduces to factoring:
  - "Breaking RSA may not be equivalent to factoring" [Boneh Venkatesan 1998]
    "an algebraic reduction from factoring to breaking low-exponent RSA can be converted into an efficient factoring algorithm"
  - "Breaking RSA generically is equivalent to factoring" [Aggarwal Maurer 2009]
    "a generic ring algorithm for breaking RSA in $\mathbb{Z}_N$ can be converted into an algorithm for factoring"
- "RSA assumption": This problem is hard.

A garden of attacks on textbook RSA

Unpadded RSA encryption is homomorphic under multiplication.
Let's have some fun!

Attack: Malleability

Given a ciphertext $c = \mathsf{Enc}(m) = m^e \bmod N$, attacker can forge ciphertext $\mathsf{Enc}(ma) = ca^e \bmod N$ for any a.

Attack: Chosen ciphertext attack

Given a ciphertext $c = \mathsf{Enc}(m)$ for unknown m, attacker asks for $\mathsf{Dec}(ca^e \bmod N) = d$ and computes $m = da^{-1} \bmod N$.

So in practice **always use padding on messages**.

RSA PKCS #1 v1.5 padding

```
m = 00 02 [random padding string] 00 [data]
```

- Encrypter pads message, then encrypts padded message using RSA public key.
- Decrypter decrypts using RSA private key, strips off padding to recover original data.

Q: What happens if a decrypter decrypts a message and the padding isn't in correct format?

A: Throw an error?

RSA PKCS #1 v1.5 padding

```
m = 00 02 [random padding string] 00 [data]
```

- Encrypter pads message, then encrypts padded message using RSA public key.
- Decrypter decrypts using RSA private key, strips off padding to recover original data.

Q: What happens if a decrypter decrypts a message and the padding isn't in correct format?

A: ~~Throw an error?~~ **Bleichenbacher padding oracle attack.**

OAEP and variants are CCA-secure padding, but nobody uses them.

Selecting parameters for RSA encryption

- Choose $\geq$ 2048-bit modulus N.
  - Number field sieve factoring: $O(\exp(1.92 \ln p^{1/3} (\ln \ln p)^{2/3}))$
- Choose $e \geq 65537$.
  - Avoids Coppersmith-type small exponent attacks.
- If you can, use Shoup RSA-KEM or similar.
  - Send $r^e \bmod N$, derive $k = \text{KDF}(r)$.

My personal recommendation:

- **Just don't use RSA.**
- (ECDH is probably better for key agreement.)

Real-world implementation choices for RSA

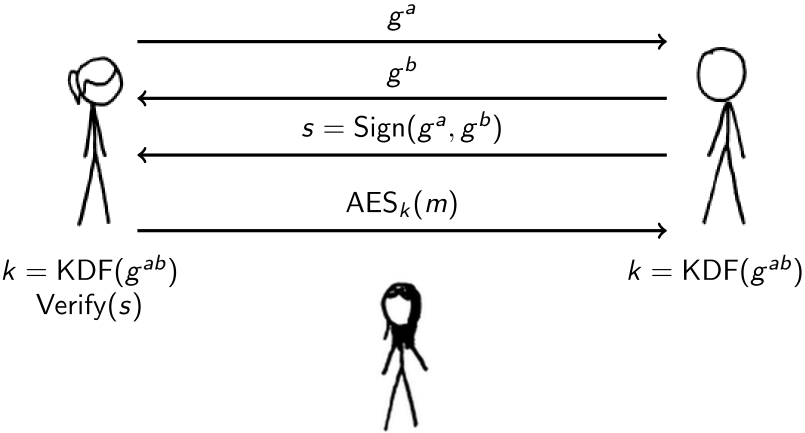
- Most of the internet has moved to at least 2048-bit keys.
- Nearly everyone uses $e = 65537$. Almost nobody uses $e > 32$ bits.
- RSA key exchange supported by default for TLS.
- PKCS#1v1.5 is universally used.
- Padding oracle protection: if padding error, generate random secret and continue handshake with random secret.
- Many implementations use "safe" primes ($p - 1 = 2q$) or have special form ($p - 1 = hq$) for prime q.

Other PKE/KEM systems

- ElGamal: Public-key encryption from discrete log.
  - Weirdly only used by PGP.
- Post-Quantum KEMs:
  - Ring-LWE, etc.
  - See Douglas's talk later today.

Idea #3: Digital Signatures

Solving the authentication problem.



* Toy protocol for illustration purposes only; not secure.

Textbook RSA Signatures

[Rivest Shamir Adleman 1977]

Public Key

$N = pq$ modulus

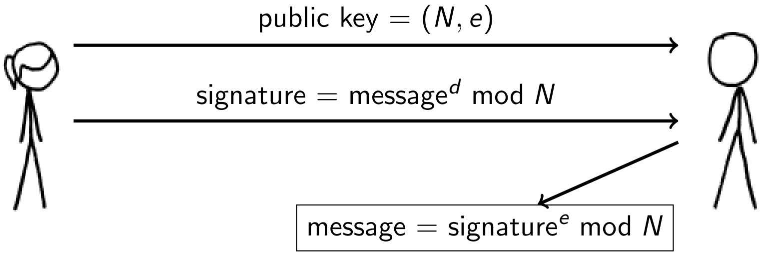
e encryption exponent

Private Key

p, q primes

d decryption exponent

$(d = e^{-1} \bmod (p-1)(q-1))$



eth roots mod N

Problem: Given N, e, and c, compute x such that $x^e \equiv c \bmod N$.

- Equivalent to selective forgery of RSA signatures.

Attacking textbook RSA signatures

Attack: Signature forgery

1. Attacker wants $\text{Sign}(x)$.
2. Attacker computes $z = xy^e \bmod N$ for some y.
3. Attacker asks signer for $s = \text{Sign}(z) = z^d \bmod N$.
4. Attacker computes $\text{Sign}(x) = sy^{-1} \bmod N$.

Countermeasures:

- **Always use padding with RSA.**
- **Hash-and-sign paradigm.**

Positive viewpoint:

- Signature blinding.

RSA PKCS #1 v1.5 signature padding

```
m = 00 01 [FF FF FF ...  FF FF] 00 [data H(m)]
```

- Signer hashes and pads message, then signs padded message using RSA private key.
- Verifier verifies using RSA public key, strips off padding to recover hash of message.

Q: What happens if a decrypter doesn't correctly check padding length?

RSA PKCS #1 v1.5 signature padding

```
m = 00 01 [FF FF FF ...  FF FF] 00 [data H(m)]
```

- Signer hashes and pads message, then signs padded message using RSA private key.
- Verifier verifies using RSA public key, strips off padding to recover hash of message.

Q: What happens if a decrypter doesn't correctly check padding length?

A: Bleichenbacher low exponent signature forgery attack.

Setting parameters for RSA signatures

- Same guidance as RSA encryption.
- Use ECDSA instead.

Real-world implementation choices for RSA signatures

- RSA remains default signature scheme for most protocols.
- Some highly important keys remain 1024-bit. (DNSSEC root was 1024 bits until 2016, long-lived TLS certs, etc.)
- Nearly everyone uses exponent $e = 65537$.
- PKCS#1v.1.5 padding used everywhere.
- Same RSA keys used for encryption and signatures in TLS.

FIPS PUB 186-3

FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION

Digital Signature Standard (DSS)

CATEGORY: COMPUTER SECURITY **SUBCATEGORY: CRYPTOGRAPHY**

DSA (Digital Signature Algorithm)

Public Key

- p prime
- q prime, divides $(p-1)$
- g generator of subgroup of order q mod p
- $y = g^x \bmod p$

Private Key

- x private key

Verify

$$u_1 = H(m)s^{-1} \bmod q$$
$$u_2 = rs^{-1} \bmod q$$
$$r \stackrel{?}{=} g^{u_1}y^{u_2} \bmod p \bmod q$$

Sign

Generate random k.

$$r = g^k \bmod p \bmod q$$
$$s = k^{-1}(H(m) + xr) \bmod q$$

DSA Security Assumptions

Discrete Log

- Breaking DSA is equivalent to computing discrete logs in the random oracle model. [Pointcheval, Vaudenay 96]

A garden of attacks on DSA nonces

Public Key

p, q, g domain parameters

$y = g^x \bmod p$

Private Key

x private key

Signature: (r, s_1)

$r = g^k \bmod p \bmod q$
$s_1 = k^{-1}(H(m_1) + xr) \bmod q$

- DSA nonce known $\rightarrow$ easily compute private key.

A garden of attacks on DSA nonces

Public Key

p, q, g domain parameters

$y = g^x \bmod p$

Private Key

x private key

Signature: (r, s_1)

$r = g^k \bmod p \bmod q$

$s_1 = k^{-1}(H(m_1) + xr) \bmod q$

Signature: (r, s_2)

$r = g^k \bmod p \bmod q$

$s_2 = k^{-1}(H(m_2) + xr) \bmod q$

- DSA nonce known → easily compute private key.

$$s_1 - s_2 = k^{-1}(H(m_1) - H(m_2)) \bmod q$$

- DSA nonce reused → easily compute nonce.

A garden of attacks on DSA nonces

Public Key

p, q, g domain parameters

$y = g^x \bmod p$

Private Key

x private key

Signature: (r, s_1)

$r = g^k \bmod p \bmod q$

$s_1 = k^{-1}(H(m_1) + xr) \bmod q$

Signature: (r, s_2)

$r = g^k \bmod p \bmod q$

$s_2 = k^{-1}(H(m_2) + xr) \bmod q$

- DSA nonce known $\rightarrow$ easily compute private key.
- DSA nonce reused $\rightarrow$ easily compute nonce.
- Biased DSA nonces $\rightarrow$ compute nonces. (Hidden number problem and variants.)

Setting parameters for (EC)DSA

- Same security considerations as Diffie-Hellman.
- Prefer ECDSA over DSA for classical adversaries.
- Generate k deterministically.
  - RFC 6979: $k = \text{HMAC}_x(m)$

Real-world implementation choices for (EC)DSA.

- FF-DSA widely supported in SSH, but not other protocols (TLS or IPsec).
- ECDSA is rapidly becoming more common.
- NIST p256 most common curve.
- Nonce generation remains a common source of implementation vulnerabilities.

Post-quantum signatures

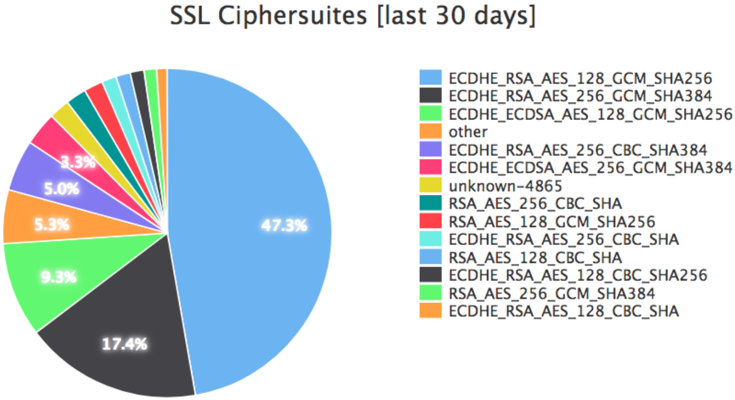
Many candidates:

- Hash-based signatures.
- Lattice-based signatures.
- . . .

Future cryptographic best practices TBD.

See Douglas's talk later today.

TLS cipher suite statistics from the ICSI notary



Summary of Public Key Algorithms in Practice

	Old and busted	Current practice	Future hotness
Key exchange	FF-DH	ECDH	SIDH
Key encapsulation	RSA		Ring-LWE
Signatures	RSA	ECDSA	Hashes? Lattices?