Legal Notice

The Zoom session for this class will be recorded and made available asynchronously on Canvas to registered students.
Announcements

1. HW 5 is due Wednesday!

2. HW 6 will be online this afternoon!
Last time:
  • RSA

This time:
  • Attacks on RSA
  • CCA security
Reminder: Textbook RSA Encryption

• Key Generation:
  1. $N = pq$
  2. Choose $e$ s.t. $\gcd(e, \phi(N)) = 1$
  3. $d = e^{-1} \mod \phi(N) = \frac{1}{e} \mod \phi(N)$
  4. $pk = (N, e), sk = (N, d)$.

• Encryption: $c = m^e \mod N$

• Decryption: $m = c^d \mod N = c^{\frac{1}{e}} \mod N$

\[
\begin{align*}
pk &= (N, e) \\
\text{c} &= \text{m}^e \mod N \\
\text{m} &= \text{c}^d \mod N
\end{align*}
\]
Textbook RSA is insecure

**Small $e$ attack:**
If $e = 3$ and $m < N^{1/3}$, $m = c^{1/3}$ over $\mathbb{Z}$.

- e.g. $2^{958} \cdot 5 + N$ m is 256-bit AES key = not mod N
- cube roots over $\mathbb{Z}$: poly-time
- cube roots over $\mathbb{Z}/N\mathbb{Z}$: not known to be efficient
  (RSA assumption: taking elt in subgroup is hard)

Conclusion: mod N required for RSA security.
Textbook RSA is insecure

**Small $e$ with Chinese Remainder Theorem**
Assume three parties have RSA keys with $e = 3$:

$$(3, N_1) (3, N_2) (3, N_3)$$

And the same (full-length) message is encrypted to each:

$$c_1 = m^3 \mod N_1 \quad c_2 = m^3 \mod N_2 \quad c_3 = m^3 \mod N_3$$

Use the Chinese remainder theorem to reconstruct

$$c \equiv m^3 \mod N_1 N_2 N_3$$

Now since $m^3 < N_1 N_2 N_3$ we are in the same situation as before.

$$c^\frac{1}{3} = m \mod \mathbb{Z}. $$
Textbook RSA is insecure

Meet-in-the-middle attack for random $m$

Input ciphertext $c$.

- Compute $x_i = c / r^e \mod N$ for $r$ from $1, \ldots, \sqrt{m}$.
- Compute $y_i = s^e \mod N$ for $s$ from $1, \ldots, \sqrt{m}$.
- If $y_i = x_j$ return $r \cdot s \cdot c^i \mod N$

If $m = r \cdot s$, $m^e = r^e \cdot s^e$.

For a randomly chosen $m$, good probability it has no large factors.
Common moduli, different exponents

If $pk_1 = (e_1, N)$ and $pk_2 = (e_2, N)$

Factorization of $N$ reveals $d = e^{-1} \mod (p - 1)(q - 1)$ for any $e$. 
RSA Key Generation Vulnerabilities

Common moduli, different exponent and encryption

Let $pk_1 = (e_1, N)$ and $pk_2 = (e_2, N)$.

Encrypt the same $m$ to both keys above:

$$c_1 = m^{e_1} \mod N \quad c_2 = m^{e_2} \mod N$$

If $\gcd(e_1, e_2) = 1$ compute $ae_1 + be_2 = 1$

$$c_1^a c_2^b = m^{e_1a} m^{e_2b} = m \mod N$$
RSA is homomorphic under multiplication

If we have a ciphertext \( c = m^e \mod N \), can forge encryption of \( mr \) by computing

\[
    c r^e \mod N = m^e r^e \mod N = (mr)^e \mod N
\]

Implications:

- Positive use: blinding. Can blind ciphertexts before decryption to try to prevent side-channel attacks, or blind signatures before signing. (More later.)
- Negative use: Chosen ciphertext attacks.
Definition

(Enc, Dec) is CCA-secure if
\[ |\Pr[A = 1|b = 0] - \Pr[A = 1|b = 1]| \text{ is negligible.} \]
Chosen ciphertext attack on textbook RSA

1. Input challenge ciphertext $c = m^e \mod N$.

2. Submit ciphertext $c' = r^e c \mod N$ for decryption.

3. Receive message $m' = rm$.

4. Original message is $m' r^{-1} \mod N = m$. 
CCA-Secure RSA encryption

Our hybrid RSA encryption from last lecture is also CCA secure.

- Key Generation:
  1. Generate primes $p$, $q$; $N = pq$
  2. Choose odd $e$ s.t. $\gcd(e, \phi(N)) = 1$
  3. $d = e^{-1} \mod \phi(N)$
  4. $pk = (N, e)$, $sk = (N, d)$.

- Encryption: Choose random $x$, $y = x^e \mod N$; $k = H(x)$;
  $c = \text{SymEnc}_k(m) \quad \text{Send} \ (y, c)$

- Decryption: $x = y^d \mod N$; $k = H(x)$; $m = \text{SymDec}_k(c)$
CCA-Secure RSA encryption

Our hybrid RSA encryption from last lecture is also CCA secure.

- **Key Generation:**
  1. Generate primes $p, q$; $N = pq$
  2. Choose odd $e$ s.t. $\gcd(e, \phi(N)) = 1$
  3. $d = e^{-1} \mod \phi(N)$
  4. $pk = (N, e), sk = (N, d)$.

- **Encryption:** Choose random $x$, $y = x^e \mod N$; $k = H(x)$;
  $c = \text{SymEnc}_k(m)$

- **Decryption:** $x = y^d \mod N$; $k = H(x)$; $m = \text{SymDec}_k(c)$

Unfortunately, nobody actually uses this in practice.
RSA Padding Schemes

To protect against RSA malleability, RSA is universally used with a padding scheme in practice.

Instead of $\text{Enc}_{\text{pk}}(m) = m^e \mod N$, we define:

- $\text{Enc}_{\text{pk}}(m) = (\text{pad}(m))^e \mod N$
- $\text{Dec}_{\text{sk}}(m)$:
  1. Compute $p = c^d \mod N$.
  2. If $p$ has correct padding format, return $\text{unpad}(p)$.
  3. Else return “failure”.

You have seen this result in problems before.
RSA Padding Schemes

To protect against RSA malleability, RSA is universally used with a padding scheme in practice.

Instead of $\text{Enc}_{pk}(m) = m^e \mod N$, we define:

- $\text{Enc}_{pk}(m) = (\text{pad}(m))^e \mod N$
- $\text{Dec}_{sk}(m)$:
  1. Compute $p = c^d \mod N$.
  2. If $p$ has correct padding format, return $\text{unpad}(p)$.
  3. Else return “failure”.

You have seen this result in problems before.
PKCS #1 v. 1.5 padding

PKCS #1 v. 1.5 padding is the most common padding scheme for RSA in practice.

\[ m = 00 \ 02 \text{ [random padding string] } 00 \text{ [data]} \]

To decrypt, implementation checks padding format:

- First two bytes correct.
- Padding string contains no null bytes.
- Presence of null byte.
- data is typically symmetric key data.
Bleichenbacher PKCS #1 v. 1.5 chosen ciphertext attack
[Bleichenbacher 1998]

\[
m = 00 \ 02 \ [\text{random padding string}] \ 00 \ [\text{data}]
\]

Attack setup:

- Attacker has a valid ciphertext \( c \) which is an encryption of a 48-byte SSL “premaster secret”.
- Victim is a SSL 3.0 server with the private key.
Bleichenbacher PKCS #1 v. 1.5 chosen ciphertext attack
[Bleichenbacher 1998]

\[ m = 00 \ 02 \ \text{[random padding string]} \ 00 \ \text{[data]} \]

Attack setup:

- Attacker has a valid ciphertext \( c \) which is an encryption of a 48-byte SSL "premaster secret".
- Victim is a SSL 3.0 server with the private key.

1. Attacker queries server with candidates \( c r^e \bmod N \).
2. Server \( \begin{cases} \text{aborts if padding incorrect} \\ \text{continues if padding correct} \end{cases} \)
3. Server is padding oracle that leaks information about plaintext.
   \[ m \cdot r = 00 \ 02 \ \ldots \ \bmod N \] Every \( r \) is small. This \( m \cdot r \) never \( = 00 \ 02 \ \bmod N \).

With a few million queries can decrypt a 2048-bit RSA ciphertext.
TLS countermeasures against Bleichenbacher attack

TLS 1.0–1.2 countermeasure:

- If padding incorrect, server generates fake plaintext and continues connection with that fake plaintext.
- Since client doesn’t know secret, connection will fail later.
TLS countermeasures against Bleichenbacher attack

TLS 1.0–1.2 countermeasure:

- If padding incorrect, server generates fake plaintext and continues connection with that fake plaintext.
- Since client doesn’t know secret, connection will fail later.

Q: Why didn’t they use a CCA-secure padding scheme?
A: Fears about backwards compatibility.
TLS countermeasures against Bleichenbacher attack

TLS 1.0–1.2 countermeasure:

- If padding incorrect, server generates fake plaintext and continues connection with that fake plaintext.
- Since client doesn’t know secret, connection will fail later.

Q: Why didn’t they use a CCA-secure padding scheme?
A: Fears about backwards compatibility.

2016: DROWN Attack

- Since servers use the same RSA keys with old versions of SSL/TLS, attacker can mount Bleichenbacher attack against servers supporting SSL 2.0 to decrypt a TLS ciphertext.

TLS 1.3 countermeasure: Eliminate RSA key exchange entirely.

DH or ECDH only for key exchange.
OAEP: CCA-secure RSA padding

[Bellare Rogaway 1994], [Fujisaki et al.]

Uses hash functions $H$, $W$, optional associated data $d$.

**Theorem**

*OAEP padding is CCA-secure in the random oracle model assuming that RSA is “partially one-way”.*

TLS, SSH, IPsec, etc. all default to PKCS#1 v. 1.5 padding.
Elementary factoring algorithms: Trial division

Input: $N \in \mathbb{Z}$
Output: $p, q \in \mathbb{Z}$ s.t. $pq = N$

**Trial division:**
For $i \leq \sqrt{N}$ check if $i \mid N$. 
Elementary factoring algorithms: Pollard rho

Input: $N \in \mathbb{Z}$
Output: $p, q \in \mathbb{Z}$ s.t. $pq = N$

Pollard rho:
Take a random walk mod $N$, hope to find a cycle modulo $p \mid N$.

Problem: Want a collision modulo $p$, but we don’t know $p$!
Solution: $a_i \equiv a_j \mod p \implies p \mid \gcd(a_i - a_j, N)$
Elementary factoring algorithms: Pollard rho

Input: \( N \in \mathbb{Z} \)

Output: \( p, q \in \mathbb{Z} \) s.t. \( pq = N \)

Pollard rho:
Take a random walk mod \( N \), hope to find a cycle modulo \( p \mid N \).

Problem: Want a collision modulo \( p \), but we don’t know \( p \)!

Solution: \( a_i \equiv a_j \mod p \implies p \mid \gcd(a_i - a_j, N) \)

Try \#1: Generate \( \sqrt{p} = O(N^{1/4}) \) elements \( a_i \).
Check \( \gcd(a_i - a_j, N) \). Problem: \( O(\sqrt{N}) \) time.
Elementary factoring algorithms: Pollard rho

Input: $N \in \mathbb{Z}$
Output: $p, q \in \mathbb{Z}$ s.t. $pq = N$

**Pollard rho:**
Take a random walk mod $N$, hope to find a cycle modulo $p \mid N$.

**Try #2:** Pseudorandom walk.
Define $f(x) = x^2 + c \mod N$, our pseudorandom function.

1. Choose random starting point $s$, constant $c$. $x_1 = x_2 = s$
2. Iterate walk: $x_1 = f(x_1), x_2 = f(f(x_2))$, compute $g = \gcd(a_1 - a_2, N)$.
   If $g = N$ start over. If $g \neq 1$ return $g$.

If $f$ is sufficiently random, expect collision after $O(\sqrt{p})$ steps. $N$ must have a factor $p$ of size at most $O(\sqrt{N})$. 
Elementary factoring algorithms: Pollard $p - 1$

Input: $N \in \mathbb{Z}$
Output: $p, q \in \mathbb{Z}$ s.t. $pq = N$

Recall Fermat’s little theorem: $a^{p-1} \equiv 1 \mod p$.

1. Choose random $a$.
2. Compute $M(k) = \text{lcm}(1 \ldots k) = \prod_i p_i^{e_i}$, $p_i^{e_i} < k$
3. Compute $b = a^{M(k)} - 1 \mod N$.
4. Compute $\gcd(b, N) = g$.
5. If $g \neq 1$ or $N$ return $g$.

Factors $N$ if $p - 1 \mid M(k) \implies p - 1$ has all small factors.

Countermeasure: Choose $p$ so that $p - 1$ has some big prime factors.
Advanced factoring algorithms: Number field sieve

Running time: $O(\exp(c(\lg N)^{1/3}(\lg \lg N)^{2/3}))$
Current record: RSA-250, 829 bits (February 2020)
RSA and factoring

Public Key
\( (N = pq, e) \)

Private Key
\( (p, q, d \equiv e^{-1} \mod (p - 1)(q - 1)) \)

If two RSA moduli share a common factor, \( N_1 = pq_1 \) and \( N_2 = pq_2 \), then
\[
\gcd(N_1, N_2) = p
\]
You can factor both keys with GCD algorithm.

Time to factor an 829-bit RSA modulus: 2700 core-years [Boudot et al. 2020]

Time to calculate GCD for 1024-bit RSA moduli: 15 \( \mu s \)
RSA and factoring

Public Key
$(N = pq, e)$

Private Key
$(p, q, d \equiv e^{-1} \mod (p - 1)(q - 1))$

If two RSA moduli share a common factor,

$N_1 = pq_1$
$N_2 = pq_2$

Time to factor an 829-bit RSA modulus: 2700 core-years

Time to calculate GCD for 1024-bit RSA moduli: 15 µs

[Boudot et al. 2020]
RSA and factoring

Public Key
\((N = pq, e)\)

Private Key
\((p, q, d \equiv e^{-1} \mod (p - 1)(q - 1))\)

If two RSA moduli share a common factor,

\[ N_1 = pq_1 \quad N_2 = pq_2 \]

\[ \gcd(N_1, N_2) = p \]

You can factor both keys with GCD algorithm.

Time to factor
829-bit RSA modulus:
2700 core-years
[Boudot et al. 2020]

Time to calculate GCD
for 1024-bit RSA moduli:
15\(\mu\)s