Announcements

1. HW 5 is due in one week!

2. HW 6 is online!
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. $d = e^{-1} \mod \phi(N)$
  4. $pk = (N, e)$, $sk = (N, d)$.

- **Encryption:** $c = m^e \mod N$

- **Decryption:** $m = c^d \mod N$
RSA Key Generation Vulnerabilities

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

$$cr^e \mod N = m^e r^e \mod N = (mr)^e \mod N$$

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.
Definitions

\((\text{Enc}, \text{Dec})\) 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 \( x, y = x^e \mod N \); \( k = H(x) \);
  \( c = \text{SymEnc}_k(m) \). Send \( (y, c) \).

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

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

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 $\text{Enc}_{pk}(m) = m^e \mod N$, 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.
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.

Encryption:

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

Signatures:

\[ m = 00 \ 01 \ FF \ldots FF \ 00 \ [\text{data}] \]

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 \( cr^e \mod N \).
2. 
   
   \[
   \text{server} \begin{cases} 
   \text{aborts if padding incorrect} \\
   \text{continues if padding correct}
   \end{cases}
   \]

3. Server is padding oracle that leaks information about plaintext.

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.

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.
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.
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 \( c \). \( a_1 = a_2 = s \)
2. Iterate walk: \( a_1 = f(a_1), a_2 = f(f(a_2)) \), 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 $M(k) = \text{lcm}(1 \ldots k) = \prod_i p_i^{e_i}, \quad p_i^{e_i} < k$
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: \(O\left(\exp(c \lg N^{1/3} \lg \lg N^{2/3})\right)\)

Current record: RSA-250, 829 bits (February 2020)
RSA and GCDs

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,
$N_1 = pq_1$
$N_2 = pq_2$
$\gcd(N_1, N_2) = p$

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
RSA and GCDs

**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,

\[
N_1 = pq_1 \quad \quad N_2 = pq_2
\]
RSA and GCDs

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\(\mu s\)
Naively computing pairwise GCDs

Euclid’s algorithm $\text{gcd}(a, b)$

```python
if b = 0:
    return a
else:
    return $\text{gcd}(b, a \mod b)$
```

$a, b$ have $n$ bits $\rightarrow O(n^2)$ time.
Naively computing pairwise GCDs

Euclid’s algorithm $\text{gcd}(a, b)$

```python
if b == 0:
    return a
else:
    return \text{gcd}(b, a \mod b)
```

$a, b$ have $n$ bits $\rightarrow O(n^2)$ time.
Naively computing pairwise GCDs

Euclid’s algorithm \( \text{gcd}(a, b) \)

\[
\text{if } b = 0: \\
\quad \text{return } a \\
\text{else:} \\
\quad \text{return } \text{gcd}(b, a \mod b)
\]

\( a, b \) have \( n \) bits \( \rightarrow O(n^2) \) time.

Use fast integer arithmetic for \( O(n(\log n)^2 \log \log n) \) time.

“Fast multiplication and its applications” Bernstein 2008
Naively computing pairwise GCDs

Euclid’s algorithm $\gcd(a, b)$

\[
\text{if } b = 0: \\
\quad \text{return } a \\
\text{else:} \\
\quad \text{return } \gcd(b, a \mod b)
\]

\[a, b \text{ have } n \text{ bits } \rightarrow O(n^2) \text{ time.}\]

Naive pairwise GCDs:

for all pairs $(N_i, N_j)$:

\[
\text{if } \gcd(N_i, N_j) \neq 1: \\
\quad \text{add } (N_i, N_j) \text{ to list}
\]

Use fast integer arithmetic for $O(n(\lg n)^2 \lg \lg n)$ time.

“Fast multiplication and its applications” Bernstein 2008
Naively computing pairwise GCDs

Euclid’s algorithm $\text{gcd}(a, b)$

$$\text{if } b = 0:$$
$$\text{return } a$$
$$\text{else:}$$
$$\text{return } \text{gcd}(b, a \mod b)$$

$a, b$ have $n$ bits $\rightarrow O(n^2)$ time.

Naive pairwise GCDs:

for all pairs $(N_i, N_j)$:
$$\text{if } \text{gcd}(N_i, N_j) \neq 1:$$
$$\text{add } (N_i, N_j) \text{ to list}$$

Use fast integer arithmetic for $O(n(lg n)^2 lg lg n)$ time.

“Fast multiplication and its applications” Bernstein 2008

$$15 \mu s \times \left(\frac{14 \times 10^6}{2}\right) \text{ pairs}$$
$$\approx 1100 \text{ years}$$
Naively computing pairwise GCDs

Euclid’s algorithm $\text{gcd}(a, b)$

if $b = 0$:
    return $a$
else:
    return $\text{gcd}(b, a \mod b)$

$a, b$ have $n$ bits $\rightarrow O(n^2)$ time.

Naive pairwise GCDs:

for all pairs $(N_i, N_j)$:
    if $\text{gcd}(N_i, N_j) \neq 1$:
        add $(N_i, N_j)$ to list

Use fast integer arithmetic for $O(n(\lg n)^2 \lg \lg n)$ time.

“Fast multiplication and its applications” Bernstein 2008

$15\mu s \times \left( \frac{14 \times 10^6}{2} \right) \text{ pairs}$

$\approx 1100 \text{ years}$
Efficiently computing pairwise GCDs

An efficient algorithm due to [Bernstein 2004].

\[ \frac{N_1 N_2 N_3 N_4}{N_4 N_3} \times \frac{N_1 N_2 N_3 N_4}{N_2 N_1} \mod N_2 \]

\[ \frac{N_1 N_2 N_3 N_4}{N_3 N_4} \times \frac{N_1 N_2 N_3 N_4}{N_1 N_2} \mod N_2 \]

\[ \gcd(, N_1) \gcd(, N_2) \gcd(, N_3) \gcd(, N_4) \]

\( O(mn \text{ polylog}(mn)) \) time for \( m n \)-bit integers, a few hours for internet-wide scan data.
Should we expect to find prime collisions in the wild?

**Experiment:** Compute GCD of each pair of $M$ RSA moduli randomly chosen from $P$ primes.

What *should* happen? **Nothing.**
Should we expect to find prime collisions in the wild?

**Experiment:** Compute GCD of each pair of $M$ RSA moduli randomly chosen from $P$ primes.

What *should* happen? **Nothing.**

**Prime Number Theorem:**
\[ \sim 10^{150} \] 512-bit primes

**Birthday bound:**
\[ \Pr[\text{nontrivial gcd}] \approx 1 - e^{-2M^2/P} \]
What happened when we GCDed RSA keys in 2012?

Computed private keys for

- **64,081** HTTPS servers (0.50%).
- **2,459** SSH servers (0.03%).
- **2** PGP users (and a few hundred invalid keys).

What has happened since?

- **103** Taiwanese citizen smart card keys
  - [Bernstein, Chang, Cheng, Chou, Heninger, Lange, van Someren 2013]
- **90** export-grade HTTPS keys
  - [Albrecht, Papini, Paterson, Villanueva-Polanco 2015]
- **313,330** HTTPS, SSH, IMAPS, POP3S, SMTPS keys
  - [Hastings Fried Heninger 2016]
- **3,337** Tor relay RSA keys
  - [Kadianakis, Roberts, Roberts, Winter 2017]
What happened when we GCDed RSA keys in 2012?

Computed private keys for

- **64,081** HTTPS servers (0.50%).
- **2,459** SSH servers (0.03%).
- **2** PGP users (and a few hundred invalid keys).

What has happened since?

- **103** Taiwanese citizen smart card keys [Bernstein, Chang, Cheng, Chou, Heninger, Lange, van Someren 2013]
- **90** export-grade HTTPS keys.
  [Albrecht, Papini, Paterson, Villanueva-Polanco 2015]
- **313,330** HTTPS, SSH, IMAPS, POP3S, SMTPS keys
  [Hastings Fried Heninger 2016]
- **3,337** Tor relay RSA keys.
  [Kadianakis, Roberts, Roberts, Winter 2017]
Widespread RNG failures on low resource devices

We accidentally found *multiple independent cascading PRNG failures*.

**Factor #1:** Weak keys generated by low resource devices (> 50 manufacturers).

1. Linux PRNG inputs: keyboard, mouse, disk
2. OpenSSL inputs: time, pid, OS PRNG
3. Headless or embedded devices lack these entropy sources.

**Factor #2:** Boot-time entropy hole on Linux PRNG

- Devices automatically generated keys on first boot.
- Linux PRNG had not yet been seeded when queried by OpenSSL.
- Fixed since July 2012.
“Random number generator enhancements for Linux 5.17 and 5.18”
https://www.zx2c4.com/projects/linux-rng-5.17-5.18/

• “the RNG can seed itself using cycle counter jitter in a second or so if it hasn’t already been seeded by other entropy sources”
• “apparently we cannot yet unify /dev/random and /dev/urandom, because the day after this change made it to mainline breakage was detected on arm, m68k, microblaze, sparc32, and xtensa”
• “swapping out SHA-1 for BLAKE2s”
• “is ‘premature next’ a real world rng concern, or just an academic exercise?”
https://lore.kernel.org/lkml/YmlMGx6+uigkGiZ0@zx2c4.com/
• Widespread RSA key generation and random number generation vulnerabilities were hiding in plain sight for years.

• Patching rates are low to nonexistent for networked devices.

• Gaps between theory and practice.