CISS451: Cryptography

A. Sharma (January 29, 2026)

Contents

Chapter 1

RSA

1.1 RSA

RSA is one of the most important encryption algorithms. We work with integers. A message in its real form is just bits, which we view as an integer x.

1. Alice wants to send a message, which we represent by the integer x.
2. Bob selects two prime numbers p and q, and computes
$$N = p\,q.$$
3. He computes $\phi(N) = (p-1)(q-1)$ and chooses integers e and d such that
$$e\,d \equiv 1 \pmod{\phi(N)}.$$
4. Bob publishes the public key (N, e) and keeps the private key (N, d) secret.
5. Encryption: Alice computes
$$c = E_{N,e}(x) = x^e \bmod N.$$
6. Decryption: Bob computes
$$x = D_{N,d}(c) = c^d \bmod N.$$

Why RSA Works

Theorem. Let p and q be primes, let $N = p\,q$, and let e, d satisfy $e\,d \equiv 1 \pmod{\phi(N)}$. Then
$$(x^e)^d \equiv x \pmod{N}$$
for all integers x.

Proof. Since $e\,d = 1 + k\,\phi(N)$ for some integer k, we have

$$(x^e)^d = x^{ed} = x^{1+k\,\phi(N)} = x \cdot \left(x^{\phi(N)}\right)^k.$$

If $\gcd(x, N) = 1$, Euler's theorem gives $x^{\phi(N)} \equiv 1 \pmod{N}$, so the right side is congruent to x. If $\gcd(x, N) \neq 1$, then $p \mid x$ or $q \mid x$ (or both). In either case $(x^e)^d \equiv 0 \equiv x$ modulo that prime. The conclusion follows by the Chinese remainder theorem.

Hybrid Encryption

In practice, RSA is too slow for large messages. Instead, one uses a hybrid approach:

1. Generate a random symmetric key k.
2. Encrypt k with RSA: $c = k^e \bmod N$.
3. Encrypt the message M with a fast symmetric cipher using key k.
4. Send $(c, \text{Encrypted}(M))$.

Divisibility Proposition

Proposition

1. If p and q are distinct primes with $p \mid a$ and $q \mid a$, then $p\,q \mid a$.
2. If $x \mid a$ and $y \mid a$ with $\gcd(x, y) = 1$, then $x\,y \mid a$.
3. If $x \mid a$ and $y \mid a$, then $\frac{x\,y}{\gcd(x,y)} \mid a$.

Proof.

1. Since $p \mid a$, write $a = p\,k$. Then $q \mid a$ implies $q \mid p\,k$. Since q is prime and $q \neq p$, $q \mid k$, so $k = q\,m$ and $a = p(q\,m) = (p\,q)\,m$.
2. Similar to part 1, using Euclid's lemma.
3. Let $g = \gcd(x, y)$ and write $x = gx'$, $y = gy'$ with $\gcd(x', y') = 1$. Using parts 1 and 2, one shows $\frac{xy}{g} \mid a$.

Exercise 1.1.1. Eve saw Alice sent the ciphertext 230539333248 to Bob. Eve checks Bob's website and found out that his public RSA key is $(N, e) = (10000016300000148701, 7)$. Help Eve compute the plaintext. In fact, compute the private key (N, d). You only need to compute d since N is known. How much time did you use? (Hint: Why is factoring N crucial?) (Go to solution, page **??**) $\square$

1.2 Implementation issues debug: rsa-implementation-issues.tex

RSA is a good cyptosystem but does cause alot of problems when it comes to implementing, fom hardware limitations(not important) to other issues, The implementation has too make the encryption and decryption fast and it should also sutain all types of attack.

1.3 Baby Asymptotic Analysis debug: baby-asymptotic-analysis.tex

Asymptotic analysis studies algorithms by their time or space usage as a function of input size n, independent of hardware.

For example, sorting 10 elements with bubble sort might be as fast as merge sort, but on very large inputs (e.g., 500×10^{12} elements), merge sort is much faster.

- **Big-O:** $T(n) = O(f(n))$ means there exist constants $c > 0$ and n_0 such that for all $n \geq n_0$,
$$T(n) \leq c\, f(n).$$
- **Big-Ω:** $T(n) = \Omega(f(n))$ means there exist constants $c > 0$ and n_0 such that for all $n \geq n_0$,
$$T(n) \geq c\, f(n).$$
- **Big-Θ:** $T(n) = \Theta(f(n))$ means $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$, i.e., there exist constants $c_1, c_2 > 0$ and n_0 such that for all $n \geq n_0$,
$$c_1\, f(n) \leq T(n) \leq c_2\, f(n).$$

1.4 Multiplication: Karatsuba algorithm debug: karatsuba.tex

Multiplying two n-digit integers via the classical grade-school method takes $O(n^2)$ single-digit multiplications. Karatsuba's insight reduces this to approximately $O(n^{1.585})$.

- Let x and y be two nonnegative integers each of at most n digits in base T (e.g., $T = 10$ or $T = 2^{16}$).
- Write
$$x = a\,T^m + b, \quad y = c\,T^m + d,$$
where $m = \lceil n/2 \rceil$, and a, b, c, d are roughly half-size.
- Observe the product:
$$x\,y = a\,c\,T^{2m} + (a\,d + b\,c)\,T^m + b\,d.$$
- Instead of computing the four products ac, ad, bc, and bd, Karatsuba shows we need only three:
  1. $A = a\,c$
  2. $B = b\,d$
  3. $C = (a + b)\,(c + d)$
- Then $a\,d + b\,c = C - A - B$. Recombine:
$$x\,y = A\,T^{2m} + (C - A - B)\,T^m + B.$$
- Each recursion halves the digit-length, leading to the recurrence
$$T(n) = 3\,T\big(\lceil n/2 \rceil\big) + O(n).$$

2. Complexity Analysis

- Solve $T(n) = 3T(n/2) + cn$ by the Master Theorem:
$$T(n) = \Theta\big(n^{\log_2 3}\big) \approx \Theta(n^{1.585}).$$
- For large n, this outperforms $\Theta(n^2)$. In practice, a cutoff to classical multiplication is used for small sizes to minimize overhead.
- Typical base threshold: when $n \le 32$ (or machine-word size), revert to $O(n^2)$ multiply.

3. Pseudocode

```
function Karatsuba(x, y):
    if x < T or y < T:
        return x * y            # base-case single-digit
    m = ceil(max(len(x), len(y)) / 2)
    (a, b) = split_at(x, m)
    (c, d) = split_at(y, m)
    A = Karatsuba(a, c)
    B = Karatsuba(b, d)
    C = Karatsuba(a + b, c + d)
    return A * T^(2*m) + (C - A - B) * T^m + B
```

4. Threshold

- *Threshold Choice:* Recursive overhead outweighs savings below a certain digit count. Benchmark on target hardware.
- *Base Case Multiply:* For $n < n_0$, perform schoolbook $O(n^2)$ multiply directly.
- *Balanced Splits:* Use $m = \lceil n/2 \rceil$ to handle odd lengths.
- *Efficient Addition:* Recombining requires only two additions and one subtraction of $O(n)$ cost.
- *Radix Selection:* Use a large base $T = 2^{16}$ or 2^{32} if implementing on binary machines to reduce recursion depth.

5. Code Examples

Python

```python
def karatsuba(x, y):
    # Base case: single-digit
    if x < 10 or y < 10:
        return x * y
    # Determine split size m
    n = max(len(str(x)), len(str(y)))
    m = (n + 1) // 2
    # Split x and y
    high1, low1 = divmod(x, 10**m)
    high2, low2 = divmod(y, 10**m)
    # Three recursive calls
    A = karatsuba(high1, high2)
    B = karatsuba(low1, low2)
    C = karatsuba(high1 + low1, high2 + low2)
    # Combine results
    return A * 10**(2*m) + (C - A - B) * 10**m + B
```

Exercise 1.4.1. Compute $1122334455667788 \times 8765432187654321$ using Karasuba multiplication to continually breakdown the integers up to integers of length 2; use your calculator to perform multiplication of length 2 integers. (Go to solution, page **??**) □

Exercise 1.4.2. Implement a long integer class where multiplication uses Karasuba. (Go to solution, page **??**) □

Exercise 1.4.3. The algorithmic analysis above is basically correct but some details are actually missing. Write a research paper on Karatsuba, describing the algorithm in detail, and analyze the runtime performance exactly. Research also on efficient implementation of Karatsuba. (Go to solution, page **??**) □

Exercise 1.4.4. * Since the surprising (shocking?) discovery of Karatsuba, several improvements to his algorithm has appeared since the 60s. Write a research paper on the various new-fangled integer multiplication algorithms, analyze and comparison their runtime performance. (Go to solution, page **??**) □

1.5 Exponentiation: the squaring method debug: exponentials-squaring-method.tex

We have a common challenge in RSA cryptography - computing extremely large powers efficiently. When we need to calculate a^n for $n > 0$.

$$a^n = a^{n-1} \cdot a \tag{1.1}$$

This method requires us to perform $n-1$ multiplications, giving us a runtime of $O(n)$ (actually $\Theta(n)$). When working in modular arithmetic with modulus N, we must take the modulus after each multiplication to keep our intermediate values manageable:

$$a^k = a^{k-1} \cdot a \pmod N \tag{1.2}$$

We can use recursion

$$a^n = \begin{cases} 1 & n = 0 \\ (a^{n/2})^2 & n > 0 \text{ and } n \text{ is even} \\ a \cdot (a^{(n-1)/2})^2 & n > 0 \text{ and } n \text{ is odd} \end{cases} \tag{1.3}$$

Let us walk through two examples to illustrate how this recursive approach works:

Example 1: Computing 2^{27}

1. $2^{27} = 2 \cdot (2^{13})^2$
2. $2^{13} = 2 \cdot (2^6)^2$
3. $2^6 = (2^3)^2$
4. $2^3 = 2 \cdot (2^1)^2$
5. $2^1 = 2 \cdot (2^0)^2 = 2 \cdot (1)^2 = 2$
6. Working backwards: $2^1 = 2$, $2^3 = 2 \cdot 2^2 = 2 \cdot 4 = 8$
7. $2^6 = 8^2 = 64$
8. $2^{13} = 2 \cdot 64^2 = 2 \cdot 4096 = 8192$
9. $2^{27} = 2 \cdot 8192^2 = 2 \cdot 67108864 = 134217728$

Example 2: Computing 3^{20} (using the fact that 20 is even)

1. $3^{20} = (3^{10})^2$
2. $3^{10} = (3^5)^2$

3. $3^5 = 3 \cdot (3^2)^2$
4. $3^2 = (3^1)^2$
5. $3^1 = 3 \cdot (3^0)^2 = 3 \cdot 1 = 3$
6. Working backwards: $3^1 = 3$, $3^2 = 3^2 = 9$
7. $3^5 = 3 \cdot 9^2 = 3 \cdot 81 = 243$
8. $3^{10} = 243^2 = 59049$
9. $3^{20} = 59049^2 = 3486784401$

Let us now present an algorithm that implements this recursive approach:

```
ALGORITHM: power
INPUTS: a, n where n >= 0
if n == 0:
    return 1
else:
    if n is even:
        x = power(a, n / 2)
        return x * x
    else:
        x = power(a, (n - 1) / 2)
        return a * x * x
```

We can see number of recursions is $O(\log n)$, recursion step costing one or two multiplications. We can rewrite this recursive algorithm using a loop, which often provides better performance in practice.

To compute a^x, we first write x as a binary number:

$$x = (x_k \cdots x_0)_2 = x_k 2^k + \ldots + x_1 2^1 + x_0 2^0 \tag{1.4}$$

where each x_i is either 0 or 1.

For example, if we take $x = 27 = (11011)_2 = 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$, then:

$$a^{27} \equiv a^{1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0} \equiv a^{2^4} \cdot a^{2^3} \cdot a^{2^1} \cdot a^{2^0} \tag{1.5}$$

Notice that the computation of a^{27} depends on powers of the form a^{2^i} where bit i in the binary representation of 27 is 1. In general, $a^{2^{i+1}} = (a^{2^i})^2$, which allows us to efficiently compute these powers with repeated squaring.

As another example, let us calculate 5^{42} using this binary method:

$$42 = (101010)_2 = 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \tag{1.6}$$
$$5^{42} = 5^{2^5} \cdot 5^{2^3} \cdot 5^{2^1} \tag{1.7}$$

We compute these powers as follows:

$$5^{2^0} = 5 \tag{1.8}$$
$$5^{2^1} = (5^{2^0})^2 = 5^2 = 25 \tag{1.9}$$
$$5^{2^2} = (5^{2^1})^2 = 25^2 = 625 \tag{1.10}$$
$$5^{2^3} = (5^{2^2})^2 = 625^2 = 390625 \tag{1.11}$$
$$5^{2^4} = (5^{2^3})^2 = 390625^2 = 152587890625 \tag{1.12}$$
$$5^{2^5} = (5^{2^4})^2 = 152587890625^2 = 23283064365386962890625 \tag{1.13}$$

Therefore:

$$5^{42} = 5^{2^5} \cdot 5^{2^3} \cdot 5^{2^1} \tag{1.14}$$
$$= 23283064365386962890625 \cdot 390625 \cdot 25 \tag{1.15}$$
$$= 227373675443232059478759765625 \tag{1.16}$$

now present the iterative algorithm that implements this binary method:

```
ALGORITHM: power
INPUTS: a, n where n >= 0
OUTPUT: a^n
p = 1
b = a
while n is not 0:
    bit = n % 2
    n = n / 2 (integer division)
    if bit == 1:
        p = p * b
    b = b * b
return p
```

This algorithm is quite elegant and efficient. Let us trace through it step by step for the simple example of computing 3^7:

Iteration	n	bit	p	b	Operation
Initial	7	-	1	3	-
1	7	1	3	9	p = p * b, b = b * b
2	3	1	27	81	p = p * b, b = b * b
3	1	1	2187	6561	p = p * b, b = b * b
4	0	-	2187	-	return p

For exponentiation in modular arithmetic, take the modulus as frequently as possible to keep intermediate values manageable:

```
ALGORITHM: power-mod
INPUTS: a, n, N where n >= 0 and N is the modulus
OUTPUT: (a^n) % N
p = 1
b = a % N
while n is not 0:
    if n % 2 == 1:
        p = (p * b) % N
    n = n // 2
    b = (b * b) % N
return p
```

For the case of negative exponents, when a is a real number:

$$a^{-n} = (a^{-1})^n \tag{1.17}$$

And for modular arithmetic, if a is invertible in $\mathbb{Z}/N$ (meaning $\gcd(a, N) = 1$):

$$a^{-n} \equiv (a^{-1})^n \pmod{N} \tag{1.18}$$

where a^{-1} represents the modular multiplicative inverse of a modulo N, which can be computed efficiently using the Extended Euclidean Algorithm.

1.6 Applications in Cryptography

This squaring method for exponentiation is fundamental to many cryptographic systems, including:

- **RSA Encryption and Decryption:** Both encryption ($c = m^e \bmod n$) and decryption ($m = c^d \bmod n$) operations in RSA require modular

exponentiation with very large exponents.
- **Diffie-Hellman Key Exchange:** Computing $g^a \bmod p$ and $g^b \bmod p$ efficiently.
- **ElGamal Cryptosystem:** Similar to RSA, requires efficient modular exponentiation.
- **Digital Signature Algorithms:** Many signature schemes rely on modular exponentiation operations.

Exercise 1.6.1. Leetcode 50.
https://leetcode.com/problems/powx-n/
Implement `pow(x, n)`, which calculates `x` raised to the power `n`. (Go to solution, page ??) □

Exercise 1.6.2. * In the above, the computation of a^x depends on writing x is base 2. What if you write x in base 3? (Go to solution, page ??) □

Exercise 1.6.3. Implement an exponentiation function using the squaring method. Test it. After you are done, implement an exponentiation function in $\mathbb{Z}/N$ using the squaring method. (Go to solution, page ??) □

1.7 Inverse in modulo arithmetic debug: inverse-in-modular-arithmetic.tex

In the key generation for RSA, Bob has to compute the multiplicative inverse of $e \bmod \phi(n)$. This is just the Extended Euclidean Algorithm. (See previous notes).

1.8 The Prime Number Theorem and finding primes debug: primality-test.tex

Finding Primes for RSA:

- RSA requires generating huge prime numbers (typically 1024-2048 bits)
- Process: generate random odd integer of desired bit length, test for primality, if not prime try $n + 2$, etc.

Prime Number Theorem (PNT):

- Discovered by Gauss in 1792/3, proven by Hadamard and de la Vallée Poussin in 1896
- States: $\pi(x) \sim \frac{x}{\ln x}$ as $x \to \infty$
- Where $\pi(x)$ = number of primes $\leq x$

Density of Primes:

- Density of primes up to x is approximately $\frac{1}{\ln x}$
- When searching only odd integers, probability of finding a prime is approximately $\frac{2}{\ln x}$

Practical Implications:

- For 1024-bit integers, approximately 0.14% of odd integers are prime
- Finding a prime typically requires < 1000 primality tests

Primality Testing:

- Modern systems use probabilistic primality tests (faster than deterministic tests)
- Two important tests: Fermat primality test and Miller-Rabin primality test
- Miller-Rabin is used in real-world applications (builds on Fermat test principles)

Implementation Considerations:

- Start with random odd integer (least significant bit set to 1)
- Apply primality test
- If not prime, try next odd integer $(n + 2)$
- Continue until prime is found

Exercise 1.8.1. Write a function `rand_odd_int` that accept L and return an odd positive integer with L random bits. Hint: For python, try this:

```
n = int("0b111", 2) # the "0b" is optional
print(n)
n = int("0b100", 2)
print(n)
```

Try a few more examples to understand what is happening. (Go to solution, page ??) □

Exercise 1.8.2. Write a function `eratosthenes` that accepts an integer n and returns a bool array `isprime` of size n such that `isprime[i]` is True iff `i` is prime. (Go to solution, page ??) □

Exercise 1.8.3. Write a function `primes` that accepts x and returns an array of primes from 2 up to x (inclusive) in ascending order. For instance `primes(10)` return `[2, 3, 5, 7]`. (Go to solution, page ??) □

Exercise 1.8.4. Write a function `write_primes` that accepts `x` and a path `p` and store primes up to `x` at path `p` in comma-separated format. For instance `write_primes(10, 'primes-10.txt')` writes "2,3,5,7" to the file `primes-10.txt`. Write another function `read_primes` that accepts a path `p` and returns a list of primes stored at path `p`. Create a file of primes up to 10,000,000. After you are done with the above make a slight optimization by storing integer in hex. While a decimal (base-10 digit) can store 10 patterns, a hexadecimal can store 16. Try this:

```
i = int("0x1a", 16) # the "0x" is optional
print(i)
s = hex(i)
print(s)
```

(It's even better to store the integer directly in binary format, but that makes the file non-human readable.) (Go to solution, page ??) □

Exercise 1.8.5. Write a function `pi` that accept `x` and returns the number of primes up to `x` (inclusive). (Go to solution, page ??) □

1.9 Fermat Prime Test debug: fermat-primality-test.tex

Exercise 1.9.1. Let $n = 18801105946394093459$. Prove that n is composite in three ways:

(a) Use division compositeness test. First try to randomly generate a potential divisor and test it. If it fails, try to do a brute force iteration from 2 to $\sqrt{n}$ to locate a divisor.
(b) Use GCD compositeness test. First try to randomly generate a potential a and test if $\gcd(a, n) > 1$. If it fails, try to do a brute force iterate a from 2 to $\sqrt{n}$ and test if $\gcd(a, n) > 1$.
(c) Use Fermat primality test with $t = 1$.

(Go to solution, page **??**) □

Exercise 1.9.2. Let $n = 586455633175643098473344787149390694952432006747 93$. Prove that n is prime or probably prime in three ways:

(a) Use division compositeness test. Do a brute force iteration from 2 to $\sqrt{n}$ to locate a potential divisor.
(b) Use GCD compositeness test. Iterate a from 2 to $\sqrt{n}$ and test if $\gcd(a, n) > 1$.
(c) Use Fermat primality test with $t = 10$.

(Go to solution, page **??**) □

Exercise 1.9.3. How often is 2 a liar? Write a program to print all composite n such that 2 lies for n. Do you think 2 lie for infinitely many Fermat pseudoprimes? Can you prove it? Compare your experiment with 3, say up to $n = 100000$, how often does 3 lie for n when compared to 2?

(Go to solution, page **??**) □

Exercise 1.9.4. Show that 1729 is a Carmichael number. (Go to solution, page **??**) □

Exercise 1.9.5. Use Corollary **??** to show that 1729 is a Carmichael number. (Go to solution, page **??**) □

Exercise 1.9.6. Use Corollary **??** to find your own Carmichael number. Check what you have discovered with known Carmichael numbers on the web. (Go to solution, page **??**) □

Exercise 1.9.7. Implement Fermat's primality test. (Go to solution, page **??**) □

1.10 Miller-Rabin primality test debug: miller-rabin-primality-test.tex

Exercise 1.10.1. Are the Miller-Rabin computations for the case of $a = 50$ and $n = 561$? What does Miller-Rabin conclude in this case? (Go to solution, page **??**) ☐

Exercise 1.10.2. Compare the results from Fermat primality test and Miller–Rabin primality test for $n = 1729$. (Go to solution, page **??**) ☐

Exercise 1.10.3. Implement the Miller–Rabin primality test algorithm. (Go to solution, page **??**) ☐

1.11 Monte-Carlo algorithms debug: monte-carlo.tex

As noted earlier, Fermat and Miller–Rabin prime-testing algorithms are probabilistic in the sense that:

- If the return value is "n is composite," then you know for sure n is composite.
- If the return value is "n is probably prime," then n may be prime or composite.

Such tests are called *Monte–Carlo algorithms* because their "prime" verdict is only probabilistic. Since the "composite" verdict is always correct, they are sometimes called *false-biased* Monte–Carlo algorithms.

There are many primality tests; Miller–Rabin is just one. A completely deterministic, polynomial-time algorithm (the AKS test) was published in 2002 by Agrawal, Kayal, and Saxena, proving

$$\text{Primes} \in P.$$

However, AKS is not used in practice because its constants are large. In real-world use, testing a random 2048-bit odd number with $t = 10$ rounds of Miller–Rabin suffices.

Exercise 1.11.1. * Good research project: Study the AKS algorithm. (Go to solution, page **??**) □

1.12 Carmichael function debug: carmichael-function.tex

Definition 1.12.1. The *multiplicative order* of a mod n (if it exists) is the smallest positive integer k such that

$$a^k \equiv 1 \pmod{n}.$$

Not every a has an order (e.g. $0^k \not\equiv 1 \pmod{n}$). Recall Euler's theorem: if $\gcd(a, n) = 1$, then

$$a^{\varphi(n)} \equiv 1 \pmod{n}.$$

But $\varphi(n)$ is not always minimal.

Definition 1.12.2. The *Carmichael function* $\lambda(n)$ is the least common multiple of the orders of all $a \in \{1, 2, \ldots, n\}$ with $\gcd(a, n) = 1$.

Theorem 1.12.1.

1. If $\gcd(m, n) = 1$, then $\lambda(mn) = \operatorname{lcm}\big(\lambda(m), \lambda(n)\big)$.
2. If p is prime and $k \geq 1$, then

$$\lambda(p^k) = \begin{cases} p^{k-1}(p-1), & p > 2, \\ 1, & p = 2,\ k = 1, \\ 2, & p = 2,\ k = 2, \\ 2^{k-2}, & p = 2,\ k \geq 3. \end{cases}$$

Theorem 1.12.2.

1. If $\gcd(a, n) = 1$ and $a^k \equiv 1 \pmod{n}$, then $k \mid \lambda(n)$.
2. If $a^k \equiv 1 \pmod{n}$ for all $\gcd(a, n) = 1$, then $\lambda(n) \mid k$.
3. $\lambda(n) \mid \varphi(n)$.

1.13 OpenSSL debug: openssl.tex

1.14 RSA security debug: rsa-security.tex

Fundamental Security Basis:

- RSA security relies on the difficulty of factoring large composites ($n = pq$).
- As of 2025, the largest factored RSA modulus was 829 bits (RSA-250, factored in 2020).
- Current minimum recommended key size is 2048 bits.

Key Factoring Algorithms:

- General Number Field Sieve (GNFS): fastest known, runtime $e^{O\left((\ln n)^{1/3}(\ln\ln n)^{2/3}\right)}$.
- Fermat Factorization: only efficient if the two primes are very close.

Critical Vulnerabilities:

- Small private exponent ($d < n^{0.292}$): Wiener's attack.
- Textbook RSA (no padding): malleable, chosen-ciphertext attacks.
- Side-channels (timing, power, faults) can leak keys.
- Poor RNG leads to weak or repeated keys.

Quantum Threat:

- Shor's algorithm breaks RSA in polynomial time.
- 2048-bit RSA needs millions of logical qubits.
- 2025 quantum hardware has only thousands of noisy qubits.

Security Levels & Key Sizes:

- 2048 bits: $\approx$ 112-bit security (minimum).
- 3072 bits: $\approx$ 128-bit security.
- 7680 bits: $\approx$ 192-bit security.
- 15360 bits: $\approx$ 256-bit security.

Implementation Best Practices:

- Use standard padding (OAEP for encryption, PSS for signatures).
- Choose p, q of similar—but not too similar—size.
- Write constant-time code; apply blinding.
- Store keys securely with proper access controls.

Future-Proofing:

- Plan migration to post-quantum algorithms.
- Consider hybrid RSA + PQ schemes.
- For 10–15+ year security, evaluate alternatives now.

1.15 Fermat factorization debug: fermat-factorization.tex

This method uses that if we can express a number as the difference of two squares, we can easily factor it.

Suppose we want to factorize a number n and we know that n can be written as a difference of two squares:

$$n = x^2 - y^2 \tag{1.19}$$

Then we immediately know that:

$$n = (x + y)(x - y) \tag{1.20}$$

This gives us a factorization of n. Of course, if $x - y = 1$, then this factorization becomes $n = n \times 1$, which is not helpful.

To achieve our goal of writing n as a difference of squares, we can use a systematic approach:

- Check if $1^2 - n$ is a square.
- Check if $2^2 - n$ is a square.
- Check if $3^2 - n$ is a square.
- And so on...

Why does this work? Because if $x^2 - n$ is a square, say y^2, then we have $x^2 - n = y^2$, which gives us:

$$n = x^2 - y^2 \tag{1.21}$$

An interesting observation is that if $n = ab$ is odd (which is the case for an RSA modulus), then we can always write n as a difference of two squares since:

$$n = \left(\frac{a+b}{2}\right)^2 - \left(\frac{a-b}{2}\right)^2 \tag{1.22}$$

Exercise 1.15.1. Write a parallel program for the Fermat factorization algorithm. (Go to solution, page **??**) □

Exercise 1.15.2. What if you test for the condition $n = x^3 - y^3$? Is there are version of factorization using difference of cubes? (Go to solution, page **??**)

□

Bibliography