Number Theory I

Cryptographic Hardware for Embedded Systems
ECE 3170 A

Fall 2025

Assoc. Prof. Vincent John Mooney III

Georgia Institute of Technology

Reading Assignment

- Please read Chapter 11 of the course textbook by Schneier

Entropy or Randomness

- Consider an example encoding of the days of the week
  - 000 = Sunday
  - 001 = Monday
  - 010 = Tuesday
  - 011 = Wednesday
  - 100 = Thursday
  - 101 = Friday
  - 110 = Saturday
  - 111 is unused
- In this example, one bit pattern (i.e., 111) will never appear

Shannon's Definition of Entropy

- $H(M) = \log_2(n)$ where M is the message and n is the number of distinct possible meanings of the message
- In our example of days of the week, $H(\text{day of the week}) = \log_2(7) = 2.8$
- Shannon assumes a binary representation of a message
- If we use ASCII to encode only the days of the week, then the entropy is still 2.8073554922 even though each day consists of multiple ASCII characters each of which is 8 bits in length
  - In a storage system, there are practical issues such as a unique way of indicating the end of a file

Natural Languages

- With this definition of entropy, we can define the **rate of the language** as follows:
  - $r = H(M)/N$ where N is the length of the message
  - English messages tend to have values ranging between 1.0 bits/letter and 1.5 bits/letter
- The **absolute rate of the language** is the maximum number of bits that can be coded in each character, assuming each character sequence is equally likely
  - $R = \log_2(L)$ where R =is the absolute rate and L is the number of letters
  - For English with 26 letters, the absolute rate is $\log_2(26) = 4.7$ bits per letter

Security of a Cryptosystem

- Adversary goal: discover key *K*, plaintext *P*, or both
- In practice, the adversary has some knowledge about *P*, e.g., there may appear to be commands exchanged between Underwater Unmanned Autonomous Vehicles (UUAVs)
- To have bits reveal nothing to an adversary, Shannon theorized that the keysize has to be a large as the message size
  - Only a one-time pad appears to satisfy this requirement
- Cryptography goal: keep knowledge about *P* small – so small that no useful or actionable information is provided to the adversary
- The entropy of a cryptosystem depends on it keyspace
  - $H(K) = \log_2(K)$ where *K* is the number if distinct possible key values

Complexity Theory

- Two variables
  - *T* for *time complexity*
  - *S* for *space complexity*
- Both *T* and *S* are commonly expressed as functions of *n* where n is the size of the input
- So-called "big-O" notation: order of magnitude of computational complexity
  - E.g., $4n^2 + 7n + 12$ is $O(n^2)$
- If *T* = O(n), then doubling the input size doubles the time to compute
- If *T* = $O(n^2)$, then doubling the input size quadruples the time to compute

Table 11.2 from page 239 of Schneier

Table 11.2
Running Times of Different Classes of Algorithms

Class	Complexity	# of Operations for $n = 10^6$	Time at 10^6 O/S
Constant	$O(1)$	1	1 μsec.
Linear	$O(n)$	10^6	1 sec.
Quadratic	$O(n^2)$	10^{12}	11.6 days
Cubic	$O(n^3)$	10^{18}	32,000 yrs.
Exponential	$O(2^n)$	$10^{301,030}$	$10^{301,006}$ times the age of the universe

Complexity Classes

- Constant
  - For example, *c*
- Linear
  - For example, *n* where *n* = number of inputs
- Polynomial (includes quadratic, cubic, etc.)
  - For example, n^c where if *c* = 3 then the complexity is cubic
- Superpolynomial
  - For example $n^{f(n)}$ where *f(n)* is more than a constant but less than linear
- Exponential
  - For example, 2^n

Complexity of Problems

- Problems that can be solved in polynomial time or less are considered **tractable**
  - Class **P**
- Problems that have no known solution techniques in polynomial time or less are considered **intractable**
  - Class **NP**
  - Further subdivisions: NP-Complete, NP-Hard, etc.
- Conjecture: **P** $\neq$ **NP**

Modular Arithmetic

- No computer has infinite numbers
  - Typically the number representation is a power of two
  - Often the smallest number of bits that can be read or written by an instruction set processor is eight, i.e., a byte
- What happens to max value (e.g., 11111111) plus one?
  - 255 + 1 (mod 256) = 256 (mod 256) = 0
  - For cryptographic reasons, often want a particular value, e.g., $n = pq$, then make calculations mod n
- What about the inverse of a number?
  - With rational numbers, $n^{-1} = \frac{1}{n}$, e.g., $5^{-1} = 0.2$
  - What about inverses of integers?

Multiplicative Inverses in Modular Arithmetic

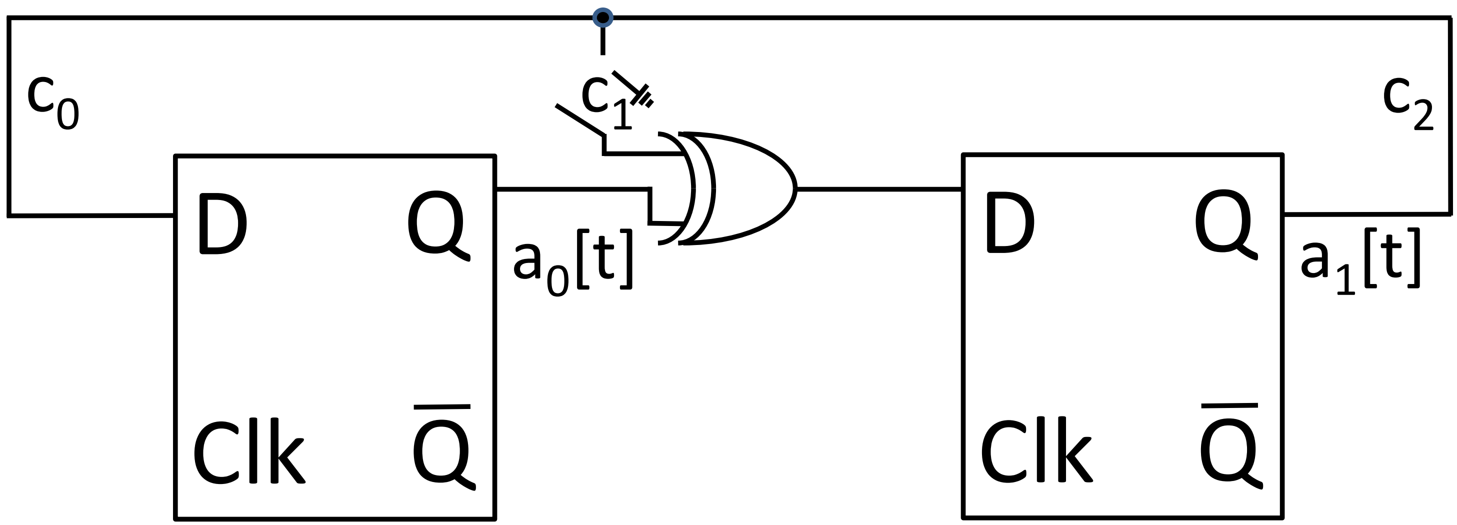
- The mathematical definition of the multiplicative inverse of a is a^{-1} such that $aa^{-1} = 1$
- However, with integers and infinite range, multiplicative inverses may exist
- What about a finite set of numbers, e.g., on a computer?
  - It turns out that in modular arithmetic, integers have inverses
  - The modular inverse of a mod n is a^{-1} such that $aa^{-1} = 1$ mod n
  - For example, for 4 in a space mod 7, 4^{-1} = 2 since 4*2 mod 7 = 8 mod 7 = 1
  - Note that sometimes there is no solution, e.g., for 4 in a space mod 8, 4^{-1} does not exist because there is no integer $x \in \{0,1,2,3,4,5,6,7\}$ such that $4x$ = 1 mod 8; in particular, for any x, the result of $4x$ is always an even number

Computing in a Galois Field

- Given *n* which is prime or is the power of a large prime, we have a finite field
  - Instead of *n*, we will now use *p*
- This type of finite field is known as a Galois Field (GF)
  - Évariste Galois was a mathematician in France in the 1800s who died at age 20 in a duel
  - He was able to prove that there is no general formula to solve a quintic polynomial
- In a GF, addition, multiplication and inverses for nonzero elements are well defined
  - Every nonzero element has a unique multiplicative inverse (this would not be true if *p* were not prime)
- Advantages of GF arithmetic include all mathematical operations work, all numbers are limited to a finite size, and multiplication by an inverse (which can be considered as a form of division) has no rounding errors

Computation in GF(2^n)

- Can be quickly implemented in hardware with feedback shift registers

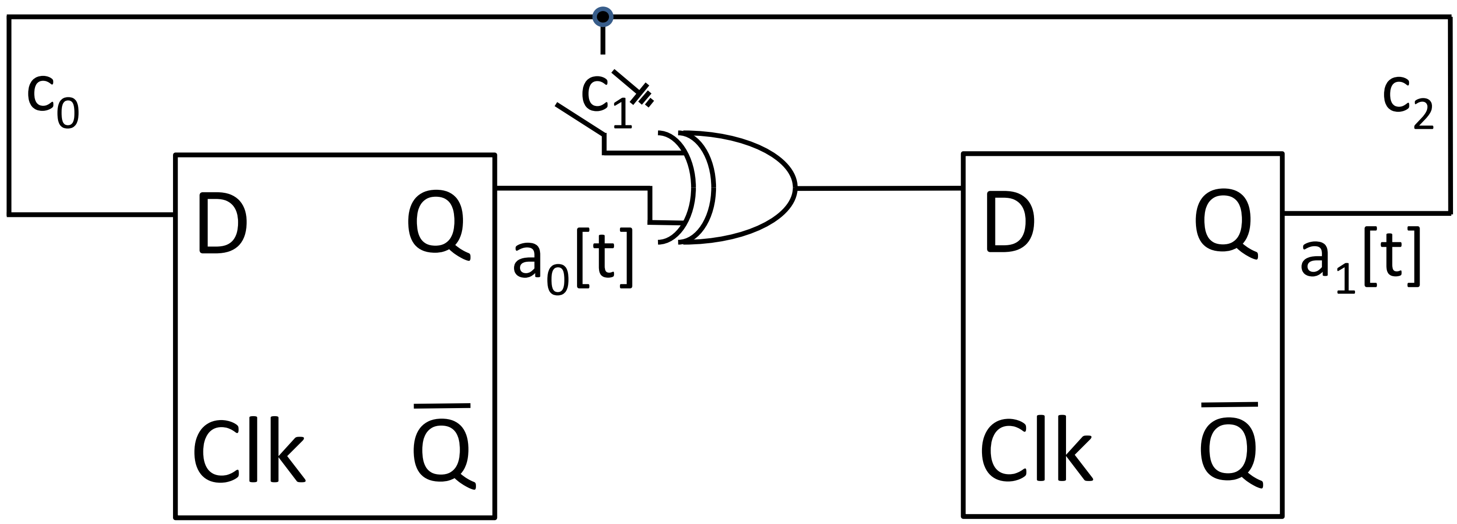


- $f(x)[t] = a_1[t]x + a_0[t]$, $P(x) = c_2x^2 + c_1x + c_0$, $f(x)[t+1] = xf(x)[t] \bmod P(x)$

Polynomial Representation

- Galois tried to find the roots of the quintic equation $a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0 = 0$ using the coefficients for a general formula, similar to $ax^2 + bx + c$ where the quadratic formula is expressed in terms of a, b and c
- Can view the bits in a feedback shift register as coefficients in a polynomial equation where x^5, x^4, x^3, x^2, x^1, x^0, etc., are placeholders (i.e., not evaluated or substituted for with numbers)
- Multiplication by x, modulus the characteristic polynomial, calculates the next state
  - $f(x)[t] = \sum_{i=0}^{n-1} a_i[t]x^i$
  - $P(x) = \sum_{i=0}^{n} c_i x^i$
  - $f(x)[t+1] = xf(x)[t] \bmod P(x)$

- $f(x)[t] = a_1[t]x + a_0[t]$, $P(x) = c_2x^2 + c_1x + c_0$, $f(x)[t+1] = xf(x)[t] \bmod P(x)$



Math describes the state sequence

- Can be quickly implemented in hardware with feedback shift registers

Factoring

- Finding prime factors
  - 10 = 2 * 5
  - 60 = 2 * 2 * 3 * 5
  - 252601 = 41 * 61 * 101
  - 2^{113}-1 = 3391 * 23279 * 65993 * 1868569 * 1066818132868207
- All known algorithms have superpolynomial/exponential run-time, but the constants in the exponent can be quite small

Prime Numbers

- In 512 bits, there exist approximately 10^{151} primes
- For your chosen prime, if selected randomly, the chance of an adversary correctly guessing your prime number is exceedingly small
- It turns out that generating a prime number is dramatically easier than factoring
- Approach to prime number generation
    - Generate a candidate prime number randomly
    - Test it
        - There exist fast tests which err less than one in 2^{50} tries

Practical Prime Number Generation

1) Generate a random *n*-bit number *p*
2) Set the high-order bit to 1; set the low-order bit to 1
3) Check *p*'s divisibility by the small primes: 3, 5, 7, 11, etc. (e.g., check all primes less than 2000)
4) Run your favorite test sequence such as Rabin-Miller

Discrete Logarithm in a Finite Field

- Modular exponentiation: $a^x \bmod n$
- Inverse of modular exponentiation:
  - Find x where $a^x \equiv b \pmod{n}$
  - Example: if $3^x \equiv 15 \pmod{17}$, then $x = 6$
  - Note that some discrete logarithms have no valid solution, i.e., no integer solution, e.g., consider $3^x \equiv 7 \pmod{13}$
- As with factoring, all known approaches to calculating the inverse of modular exponentiation have superpolynomial/exponential run-time, but the constants in the exponent can be quite small