Supplemental Notes for Trappe and Washington

Prof. Shahed Sharif

0 Introduction

The purpose of this document is to give a more rigorous foundation for cryptography. The textbook we are using is overall quite good, but uses a more informal, conversational format. Since this is an upper-division math class, we need something a bit more formal. Hence these notes!

None of the below is meant to be standalone, so make sure to refer back to the text if there are any terms or theorems that I don't explain here. Accordingly, I will use the same section numbers here as in the text.

1 Overview

1.1 Secure communications

We classify attacks by how much capability Eve has:

1. default: Eve knows only the ciphertext.
2. known plaintext (KPA): Eve knows one plaintext/ciphertext pair.
3. chosen plaintext (CPA): Eve can simulate encryption of plaintexts of her choosing.
4. chosen ciphertext (CCA): Eve can simulate decryptions of ciphertexts of her choosing (with possibly one exception).

A word on CCA: suppose Eve wants to decrypt one specific ciphertext C. Then the phrase "with possibly one exception" should be clear: the exception is C. Otherwise, nothing is CCA secure!

We use "bits" (binary digits) in computer science, not decimal digits. But we are used to decimal digits, so it's good to have a way of relating the two. We first characterize the number of bits/digits in a number in a technical way.

Theorem 1.1. *For n a positive integer, the number of decimal digits in n is $1 + \lfloor \log_{10} n \rfloor$. The number of bits to write down n is $1 + \lfloor \log_2 n \rfloor$.*

Recall that $\lfloor x \rfloor$ means to round x down to an integer, so $\lfloor 2 \rfloor = \lfloor 2.1 \rfloor = \lfloor 2.999 \rfloor = 2$. Our proof uses the fact that 10^{d-1} is the smallest number with exactly d digits. For instance, $10^{3-1} = 10^2 = 100$ has 3 digits, and smaller numbers have fewer digits.

Proof. Suppose n has d digits. Then

$$10^{d-1} \leq n < 10^d.$$

The log function is increasing, so we can take logs of the inequality to get

$$d - 1 \leq \log_{10} n < d.$$

Therefore $\lfloor \log_{10} n \rfloor = d - 1$. Adding 1 to both sides completes the first part of the proof.

The second part is similar: just swap out 10 with 2. □

All logs are proportional; that is, if a, b, x are positive real numbers, then $\log_a x = c \log_b x$ for some constant c depending only on a and b. To figure out the constant, plug in $x = b$ (or $x = a$, but that requires a little more work). In particular,

$$\log_2 x = \log_2 10 \cdot \log_{10} x.$$

Since $\log_2 10 \approx 3$,

$$\log_2 x \approx 3 \log_{10} x. \tag{1}$$

So the number of bits in a number is roughly 3 times the number of digits in that number. The number 1,000,000 has 7 digits, so in binary we expect it to have about 21 bits. In fact, it has 20 bits, so close.

Theorem 1.2. *In a probabilistic trial, suppose outcome X has a probability of p of occurring. If we repeat the trial until X occurs, assuming the trials are independent of each other, the expected number of trials is $1/p$.*

To make this more concrete, say we flip a coin and want to get heads. The probability is $1/2$, so we expect to have to flip the coin on average

$$\frac{1}{1/2} = 2$$

times before we get heads. That's an *average*, of course; we could get it on the first try, or it might take 10 flips.

Another example is rolling dice. Say we roll two dice and want to get snake eyes (both 1s). The probability is $1/36$, so on average we'll have to roll the dice 36 times before we get snake eyes.

The proof of this theorem is beyond the scope of the class, so we omit it.

2 Basic number theory

2.1 Basic notions

Theorem 2.1 (Quotient-Remainder Theorem)**.** *Suppose $n, d \in \mathbb{Z}$ with $n \geq 0$ and $d > 0$. Then there exist unique $q, r \in \mathbb{Z}$ with $0 \leq r < d$ such that*

$$n = qd + r.$$

This is proved in previous courses, so I will not give the full proof here. But here is the outline:

- Take $R = \{m \in \mathbb{Z} : m \geq 0, \exists Q \in \mathbb{Z} \text{ such that } m = n - Qd\}$.
- Note that taking $Q = 0$, we have $n \in R$, so $R \neq \varnothing$.
- By Well-Ordering Principle, R has a least element, call it r, and let q be the corresponding Q.
- By definition of R, $r \geq 0$. If $r \geq d$, we can obtain a contradiction by showing that $r - d \in R$.
- For uniqueness, if there are two different rs, subtract the expressions from each other to show that the difference of the rs is a small number divisible by d, and so must be 0.
- Then show the qs are the same.

It's a good exercise to fill in the details.

The following Python code computes the gcd of two integers, a, b, not both zero.

```
def gcd(a,b):
    """Compute the gcd of a and b."""
    while b != 0:
        a, b = b, a%b
    return a
```

Note that the algorithm works even if a, b are negative!

This is the first new mathematical algorithm we are covering. The good news is the algorithm is short. The bad news is that proving that it works is long! But hopefully the proof will be instructive in terms of how to prove algorithms in the future. The steps are as follows:

- Show the algorithm terminates.
- Show that as the values of the computer variables `a` and `b` change, the value of their gcd does *not* change.
- Show that the output is what we want.

Theorem 2.2. *The code for* `gcd` *computes the gcd of* a *and* b *when* $a, b \geq 0$ *and are not both zero.*

Proof. Let a_j, b_j be the values of `a` and `b` after j iterations of the loop; so for instance, $a_0 = a$ and $b_0 = b$. If there are n executions of the loop, then the output is a_n.

Since the program uses a **`while`** loop, we should check that the algorithm terminates. Observe that for each $j \geq 1$, b_j is the remainder when we divide a_{j-1} by b_{j-1}. By the Quotient-Remainder Theorem, $0 \leq b_j < b_{j-1}$. Therefore the sequence of b_js is a strictly decreasing sequence of nonnegative integers. This means that the sequence must eventually reach 0, which is precisely when the loop terminates.

Next, I will prove that the gcd doesn't change; that is, for all j, $\gcd(a_{j-1}, b_{j-1}) = \gcd(a_j, b_j)$. We do this by proving that any common divisor of a_{j-1}, b_{j-1} is a common divisor of a_j, b_j, and vice versa. Once we know this, the greatest of the common divisors must be the same.

So let d be a common divisor of a_{j-1}, b_{j-1}. Since $d \mid b_{j-1}$ and $a_j = b_j$, we get $d \mid a_j$. By definition of remainder, $\exists q, r \in \mathbb{Z}$ such that

$$a_{j-1} = qb_{j-1} + b_j.$$

Thus $b_j = a_{j-1} - qb_{j-1}$. Since $d \mid a_{j-1}$ and $d \mid b_{j-1}$, we get $d \mid (a_{j-1} - qb_{j-1}) = b_j$. This shows that d is a common divisor of a_j, b_j.

The converse argument is similar, and is left as an exercise.

Suppose there are n iterations of the loop before the program ends. We prove by induction on j that $\gcd(a_j, b_j) = \gcd(a, b)$ for $0 \leq j \leq n$. The base case is obvious. For the inductive step, we assume that $\gcd(a_{j-1}, b_{j-1}) = \gcd(a, b)$. We already showed that $\gcd(a_j, b_j) = \gcd(a_{j-1}, b_{j-1})$, so by transitivity $\gcd(a_j, b_j) = \gcd(a, b)$.

Finally, we show that the output is correct. We know by the termination condition that $b_n = 0$. That means

$$\gcd(a_n, b_n) = \gcd(a_n, 0) = a_n.$$

By the previous paragraph, $a_n = \gcd(a, b)$. As a_n is the output of the program, the output is correct. □

2.2 Extended Euclidean algorithm

You may recall the following:

Theorem 2.3. *Let a, b be nonnegative integers, not both zero. If $d = \gcd(a, b)$, then $\exists s, t \in \mathbb{Z}$ such that*

$$d = sa + tb.$$

In other words, the gcd is a linear combination of a and b. The coefficients s and t are very much *not* unique! You should generate your own example to see this.

This theorem is often called the Euclidean algorithm. But it isn't an algorithm! We'll fix this. I will give an *informal* description of the algorithm. I'll let you translate my description into Python. I'll use matrices in my algorithm, but you should not: just reference the variables which make up the entries of all matrices in sight.

This is probably the most complicated algorithm we'll see in the course. It's unfortunate that it comes so early, but what can you do.

INPUT: $a, b \in \mathbb{Z}$, not both zero.

1. Initialize a matrix M_{-1} to be $\left[\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right]$.

2. Apply the regular Euclidean algorithm. At each stage, when we compute the remainder of a_j by b_j, let q_j be the quotient:

$$a_j = q_j b_j + b_{j+1}.$$

3. When we do the previous step, also define

$$M_j = \begin{bmatrix} 0 & 1 \\ 1 & -q_j \end{bmatrix} M_{j-1}.$$

4. Suppose $M_n = \left[\begin{smallmatrix} s & t \\ u & v \end{smallmatrix}\right]$. Then output s and t.

Rather mysterious! The key fact is that for *every* j, the top row of M_j give the coefficients to get the jth remainder in terms of a and b; that is, if r_j is the remainder at the jth loop of the Euclidean algorithm, and

$$M_j = \begin{bmatrix} s_j & t_j \\ u_j & v_j \end{bmatrix},$$

then $r_j = s_j a + t_j b$. The very last nonzero remainder r_n is the gcd, so this will prove the claim.

A few observations before we do the proof. First, $r_j = b_{j+1}$. For this reason, it makes sense to initialize $r_{-1} = a$ and $r_0 = b$. Actually, to cut down on the number of variables, we'll do $a = b_{-1}$ and $b = b_0$, and get rid of the r variable completely. Note that our Euclidean algorithm recurrence is then

$$b_{j-1} = q_j b_j + b_{j+1}.$$

So for instance when $j = 0$, this yields $a = q_0 b + b_1$. It will be useful to rewrite this as

$$b_{j+1} = b_{j-1} - q_j b_j. \tag{2}$$

Lemma 2.4. *For* $1 \le j \le n$, $u_{j-1} = s_j$ *and* $v_{j-1} = t_j$. *Additionally, if* $1 \le j \le n-1$, *then*

$$s_{j+1} = s_{j-1} - q_j s_j \qquad t_{j+1} = t_{j-1} - q_j t_j.$$

Proof. We have

$$M_j = \begin{bmatrix} 0 & 1 \\ 1 & -q_j \end{bmatrix} M_{j-1}.$$

and so

$$\begin{bmatrix} s_j & t_j \\ u_j & v_j \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & -q_j \end{bmatrix} \begin{bmatrix} s_{j-1} & t_{j-1} \\ u_{j-1} & v_{j-1} \end{bmatrix} = \begin{bmatrix} u_{j-1} & v_{j-1} \\ s_{j-1} - q_j u_{j-1} & t_{j-1} - q_j v_{j-1} \end{bmatrix}.$$

Comparing entries, the claim follows. □

Theorem 2.5. *The extended Euclidean algorithm works.*

Proof. The termination condition is the same as for the Euclidean algorithm, so this algorithm ends after a finite amount of time.

For correctness, I will show by strong induction that for all $-1 \leq j \leq n$, we get

$$s_j a + t_j b = b_j.$$

The case where $j = -1$ looks like $1 \cdot a + 0 \cdot b = a$, clearly true. The case where $j = 0$ looks like $0 \cdot a + 1 \cdot b = b$, also clearly true. That takes care of the base case.

So let us suppose that for $j \leq n-1$, for all $i \leq j$, we have

$$s_i a + t_i b = b_i.$$

In particular,

$$\begin{aligned} s_{j-1} a + t_{j-1} b &= b_{j-1} \text{ and} \\ s_j a + t_j b &= b_j. \end{aligned}$$

Multiply the second equation by q_j and subtract from the first equation to get

$$(s_{j-1} - q_j s_j)a + (t_{j-1} - q_j t_j)b = b_{j-1} - q_j b_j.$$

By the previous lemma, the left hand side is

$$s_{j+1} a + t_{j+1} b.$$

By equation (2), the right side is b_{j+1}. Thus

$$s_{j+1} a + t_{j+1} b = b_{j+1}.$$

This proves the inductive conclusion. By induction, $s_j a + t_j b = b_j$ is true for all j. The Euclidean algorithm outputs $a_n = b_{n-1}$, and the extended Euclidean algorithm outputs s_{n-1}, t_{n-1}. Therefore the algorithm is correct. □