Cipher

Welcome to CSCI 1515!

Throughout the semester, you'll implement numerous systems using production-grade cryptographic primitives. This assignment is meant to get you up to speed with C++ and the schemes you'll spend the semester with. If you ever have any questions, don't hesitate to ask a TA for support!

Due Date: *Tuesday, February 3rd.*

Contents

1 Background Knowledge

In this assignment you'll be implementing a few foundational cryptographic schemes. In order to fully understand why these schemes are correct and secure, we review some of the number theory underlying these constructions. Don't worry — the rest of the course won't rely on a deep understanding of the math behind these schemes. Critically, we don't go over any advanced or involved proofs in this handout or course. Rather, we introduce the results that are useful and ask that you take them at face value.

If you are interested in the advanced mathematics or theoretical proofs, we recommend taking CSCI 1510 or MATH 1580. It is, however, crucial that you understand the purpose and use of each scheme, and knowing some about how they work under the hood can help you gain some of that understanding.

1.1 Elementary Number Theory

The following is an overview of the number theory necessary to understand the cryptographic schemes in this homework. **This section may seem intimidating: to reiterate, you do not need to understand this math deeply to implement this assignment, and you certainly don't need it for the rest of the course.**

1.1.1 Divisibility and GCDs

Consider two integers $a, b \in \mathbb{Z}$. We say that a **divides** b if there exists an integer $c \in \mathbb{Z}$ such that $a \cdot c = b$. We denote this by $a \mid b$.

Given integers $a, b, m \in \mathbb{Z}$. We say that a and b are **congruent mod** m if there exists an integer $k \in \mathbb{Z}$ such that $a + km = b$. In other words, it means that a and b differ by a multiple of m, or that when divided by m, they yield the same remainder. We denote this by $a \equiv b \mod m$.

Recall **greatest common divisors (GCDs)**. Given two integers $a, b \in \mathbb{Z}$, the GCD of a and b is the largest integer $d \in \mathbb{Z}$ such that $d \mid a$ and $d \mid b$. We say that two integers are **coprime** if their GCD is 1. Calculating the GCD of two integers can be done efficiently using the Euclidean Algorithm, and calculating integers s, t such that $s \cdot a + t \cdot b = \gcd(a, b)$ can be done efficiently using the Extended Euclidean Algorithm. We eschew a detailed explanation of either algorithm in favor of the Wikipedia articles.

1.1.2 Groups

To work with some theorems more nicely in general, and to allow us to generalize some of the following schemes, we introduce the notion of a group. A **group** is defined as a set $\mathbb{G}$ along with a binary operation $\circ : \mathbb{G} \times \mathbb{G} \to \mathbb{G}$, such that the following properties hold:

1. **Closure:** for any two elements $a, b \in \mathbb{G}$, we have that $a \circ b \in \mathbb{G}$.

2. **Identity:** there exists an identity element $e \in \mathbb{G}$ such that for any element $a \in \mathbb{G}$ we have that $e \circ a = a \circ e = a$.

3. **Associativity:** for any $a, b, c \in \mathbb{G}$, we have that $(a \circ b) \circ c = a \circ (b \circ c)$.

4. **Inverses:** for any $a \in \mathbb{G}$, there exists an inverse element $b \in \mathbb{G}$ such that $a \circ b = b \circ a = e$. We denote the inverse of a by a^{-1}.

For example, $\mathbb{Z}$ under addition (where our operation $\circ$ is $+$) is a group. The set of integers modulo a prime p (excluding 0) under multiplication is also a group, denoted as $\mathbb{Z}_p^* = \{1, 2, \ldots, p-1\}$. More generally, if you consider the integers from $[1, N-1]$ that are coprime to N, then we can construct a special group $\mathbb{Z}_N^* = \{a \mid a \in [1, N-1], \gcd(a, N) = 1\}$, under multiplication. To find the inverse of an element in this group, we can simply run the Extended Euclidean Algorithm. Given that $\gcd(a, N) = 1$, taking the relation $s \cdot a + t \cdot N = 1 \mod N$, we get $s \cdot a \equiv 1 \mod N$ where s is the inverse of a.

If group $\mathbb{G}$ has a finite number of elements, we say $\mathbb{G}$ is a **finite group** and let $|\mathbb{G}|$ denote the **order** of the group (i.e., the number of elements in $\mathbb{G}$). A set $\mathbb{H} \subseteq \mathbb{G}$ is a **subgroup** of $\mathbb{G}$ if $\mathbb{H}$ forms a group under the same operation $\circ$.

Let $\mathbb{G}$ be a finite group of order m. For any $g \in \mathbb{G}$, it holds that $g^m = 1$ (we defer the proof to a mathematical cryptography course). Consider the set $\langle g \rangle := \{g^0, g^1, \ldots, g^{m-1}\}$. We note that $\langle g \rangle$ contains at most m elements, and it is not hard to verify that $\langle g \rangle$ is a subgroup of $\mathbb{G}$. We say $\mathbb{G}$ is a **cyclic group** if there exists $g \in \mathbb{G}$ that **generates** the whole group, namely $\langle g \rangle = \mathbb{G}$. In other words, every element in $\mathbb{G}$ is some power of g (for any $a \in \mathbb{G}$, $a = g^r$ for some r). In this case, g is called a **generator** of $\mathbb{G}$.

The multiplicative group $\mathbb{Z}_p^*$ (for prime p) is a cyclic group of order $p - 1$. If $\mathbb{G}$ is a finite group of prime order, then $\mathbb{G}$ is a cyclic group, and every element other than the identity is a generator. In later sections, we will be working with prime-order subgroups of $\mathbb{Z}_p^*$.

1.1.3 Euler's Theorem

We end this section with one more useful result, which we state without proof. Euler's Theorem states that given any integers m and a where $\gcd(a, m) = 1$, it is the case that $a^{\phi(m)} \equiv 1 \mod m$. Note that $\phi(m)$ is Euler's totient function, defined as the number of positive integers from $[1, m-1]$ that are coprime to m. In particular, for prime p, $\phi(p) = p - 1$. If $m = p \cdot q$ for distinct primes p, q, then $\phi(m) = (p-1)(q-1)$.

1.2 Diffie-Hellman Key Exchange

We now step away from number theory and consider some real cryptographic protocols. Let's say two parties, Alice and Bob (by convention we name our honest parties Alice and Bob, and any adversaries Eve), want to decide on a shared key to encrypt some messages. For example, they may want to apply the one-time pad and so they need a shared, secret k-bit integer to do so. The **Diffie-Hellman key exchange** protocol, developed in 1976, is one method of coming to a shared secret. Diffie-Hellman will be used *extensively* throughout the rest of the course to compute shared secrets.

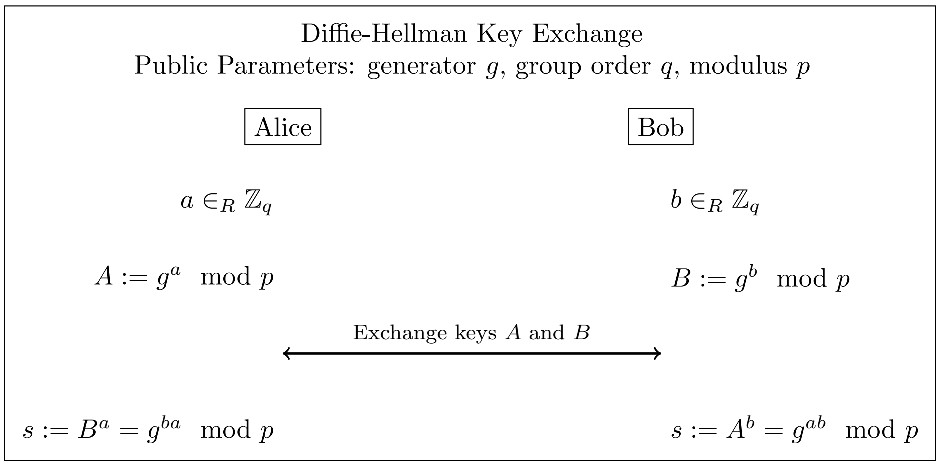
Diffie-Hellman is quite simple. Alice and Bob first come to agreement on a cyclic group $\mathbb{G}$ of order q with a generator g. In general, we wish to keep our groups large enough where an adversary can't brute-force their way into finding out the secrets. Alice and Bob then each pick a secret random integer from $\mathbb{Z}_q$, denoted as a, b respectively. Alice will compute and send g^a to Bob, and Bob will compute and send g^b to Alice. Finally, both parties will compute g^{ab} by exponentiating what they receive from the other party with their secret integer. This value, g^{ab}, is the shared secret. (Note that fast-powering is what makes this efficient; otherwise, computing large exponents will take a long time!)

Correctness is clear since the operations clearly end up with the same values on both parties. What might not be clear is why this is secure. Can an adversary Eve, who is eavesdropping on the messages that are *sent* (namely g^a and g^b), figure out g^{ab}?

In fact, we don't know whether Eve can efficiently solve this problem. The hardness of this problem is called the Diffie-Hellman assumption (decisional, computational), which inherently assumes that discrete logarithm is computationally hard. For specific groups that we use in practice, all *known* (classical) algorithms take too long (longer than the age of the universe!) to break the Diffie-Hellman assumption.

Recall that the multiplicative group $\mathbb{Z}_p^*$ (for prime p) is a cyclic group of order $p - 1$. Unfortunately, the decisional Diffie-Hellman (DDH) assumption does **not** hold in $\mathbb{Z}_p^*$, hence it is unacceptable for the Diffie-Hellman key exchange. To address this issue, we work with a prime-order *subgroup* of $\mathbb{Z}_p^*$.

Specifically, let $p = 2q+1$ where both p, q are primes (p is called a *safe prime*). Consider the set of quadratic residues modulo p, namely $\mathbb{G} := \{x^2 \mod p \mid x \in \mathbb{Z}_p^*\}$. It can be proved that $\mathbb{G}$ is a subgroup of $\mathbb{Z}_p^*$ with prime order q, hence every element in $\mathbb{G}$ (other than 1) is a generator. The DDH assumption is believed to hold in $\mathbb{G}$.


Diffie-Hellman Key Exchange
Public Parameters: generator g, group order q, modulus p
Alice
Bob
$a \in_R \mathbb{Z}_q$
$b \in_R \mathbb{Z}_q$
$A := g^a \mod p$
$B := g^b \mod p$
Exchange keys A and B
$s := B^a = g^{ba} \mod p$
$s := A^b = g^{ab} \mod p$

1.3 ElGamal Encryption

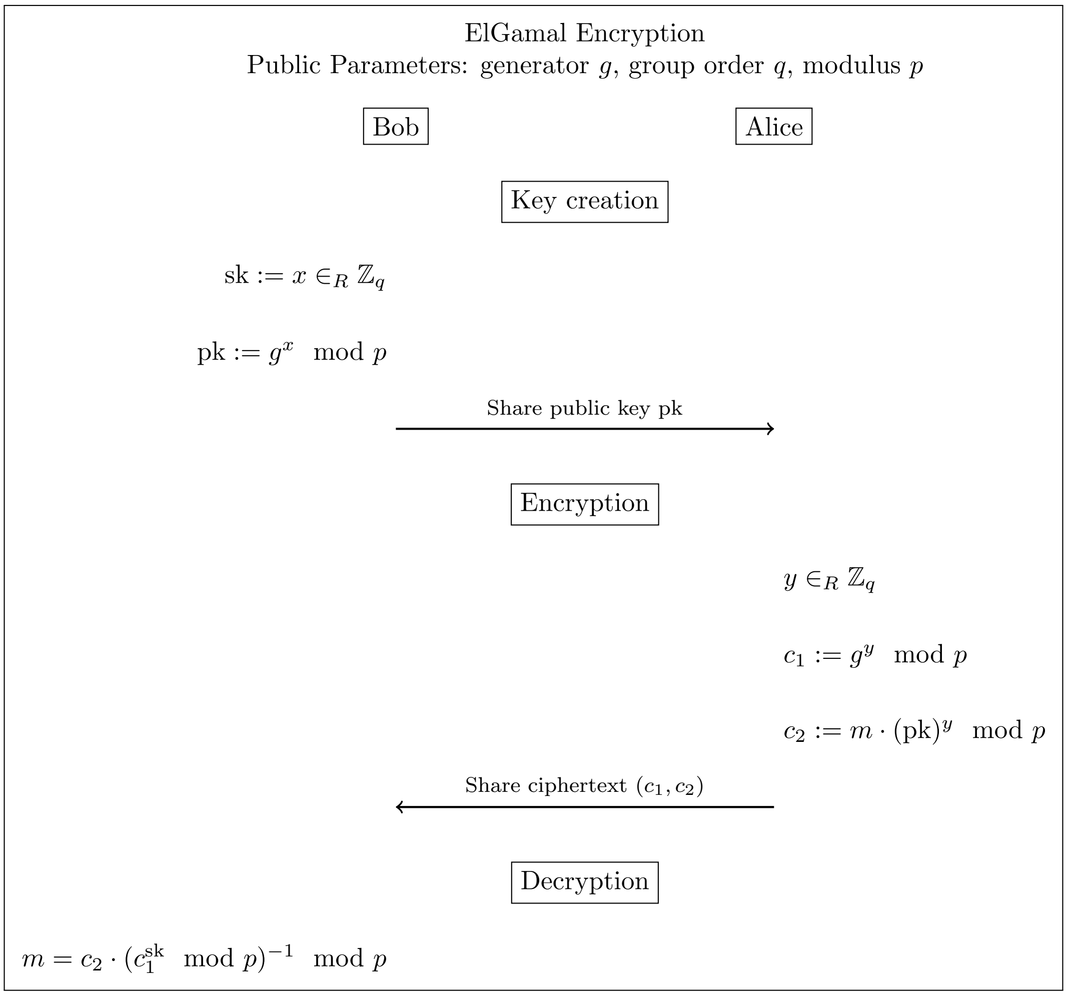
With Diffie-Hellman key exchange, we can come to a shared key in order to send a message. This is useful for **symmetric-key encryption**, which is where both Alice and Bob encrypt and decrypt using a shared key. However, what if we can't communicate to find a shared key in advance? We can instead rely on **asymmetric-key encryption**, also known as **public-key encryption**, which allows Alice to send messages to Bob but not vice versa for a given public and secret key pair.

In general, Bob will have some secret key, sk, and some public key, pk, and publish only pk. Alice will then use pk to encrypt messages to Bob that can only be decrypted with sk. We explore an example of such a system called the **ElGamal encryption** scheme, developed in 1985. It is based on the Diffie-Hellman assumption and operates in a very similar way.

We begin by describing how Bob generates his public key, pk, and secret key, sk. First, Bob will choose a cyclic group $\mathbb{G}$ of order q (e.g., subgroup of $\mathbb{Z}_p^*$ of prime order q) and a generator g. He then chooses a random integer x from $\mathbb{Z}_q$ and computes g^x. We then have that $\mathrm{pk} = g^x$ and $\mathrm{sk} = x$, so Bob publishes pk.

When Alice wants to encrypt a message $m \in \mathbb{G}$, which can be any element in the group $\mathbb{G}$, she first chooses a random integer y from $\mathbb{Z}_q$. Then, she computes $c_1 = g^y$ and

$c_2 = m \cdot \text{pk}^y$ and sends both to Bob. To decrypt, Bob computes $c_2 \cdot (c_1^{\text{sk}})^{-1}$ (where inverses are computed using the Extended Euclidean Algorithm in $\mathbb{Z}_p^*$).


ElGamal Encryption
Public Parameters: generator g, group order q, modulus p
Bob
Alice
Key creation
$\text{sk} := x \in_R \mathbb{Z}_q$
$\text{pk} := g^x \mod p$
Share public key pk
Encryption
$y \in_R \mathbb{Z}_q$
$c_1 := g^y \mod p$
$c_2 := m \cdot (\text{pk})^y \mod p$
Share ciphertext (c_1, c_2)
Decryption
$m = c_2 \cdot (c_1^{\text{sk}} \mod p)^{-1} \mod p$

Correctness is seen when we expand. Notice that $c_2 \cdot (c_1^{\text{sk}})^{-1} = m \cdot g^{xy} \cdot g^{-xy} = m$, so Bob recovers the original message.

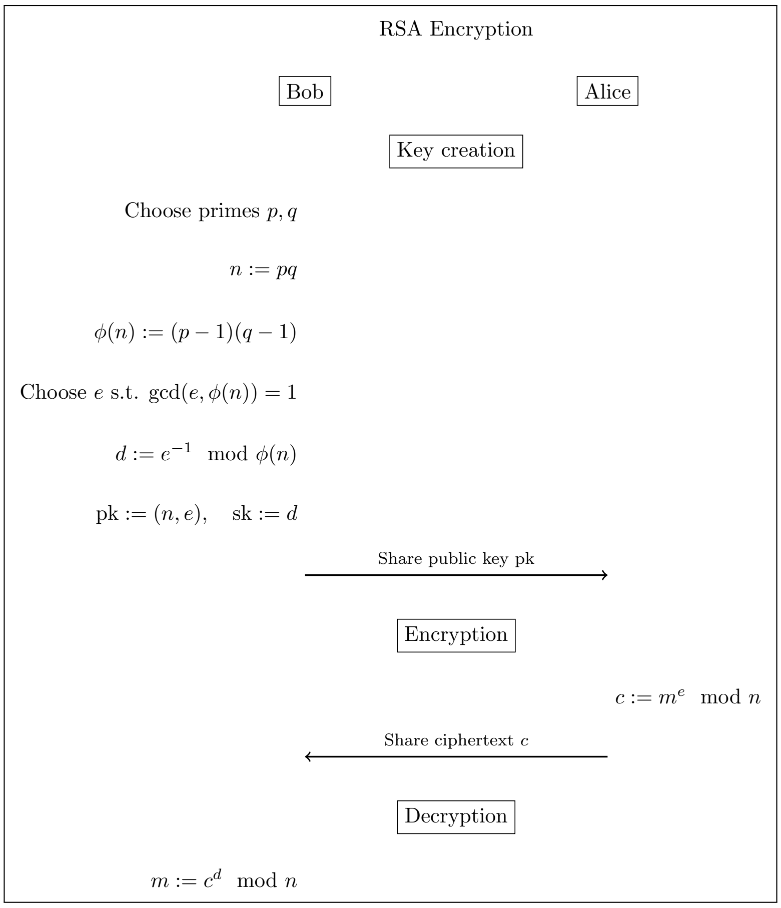
Security of this scheme relies on the same assumptions as the Diffie-Hellman key exchange: we eschew a rigorous security proof in favor of a theoretical cryptography course (CSCI 1510, MATH 1580).

1.4 RSA Encryption

We explore one more public-key cryptosystem known as RSA (Rivest-Shamir-Adleman). Unlike Diffie-Hellman key exchange and ElGamal encryption, RSA doesn't rely on the hardness of the discrete logarithm problem. Rather, it relies on the hardness of factoring large integers. That is, given the product of two distinct large primes $n = p \cdot q$, it is computationally hard to find p and q.

We start with how Bob calculates his public and secret keys. First, Bob will generate system parameters by choosing two distinct large primes p, q. He then computes $n = p \cdot q$ and $\phi(n) := \phi(p \cdot q) = (p-1)(q-1)$. Bob then chooses some integer e such that $\gcd(e, \phi(n)) = 1$. Lastly, since e is coprime to $\phi(n)$, we can find some d such that $e \cdot d \equiv 1 \mod \phi(n)$. We then set $\mathrm{pk} = (n, e)$ and $\mathrm{sk} = d$, so Bob publishes pk.

When Alice wants to encrypt a message m, which can be any integer in $\mathbb{Z}_n^*$, Alice simply computes $c = m^e \mod n$ and sends it to Bob. To decrypt, Bob computes $c^d \mod n$.



Correctness relies on Euler's theorem. We have that $e \cdot d \equiv 1 \mod \phi(n)$, or that $e \cdot d = k \cdot \phi(n) + 1$ for some integer k. Then, $c^d \equiv m^{e \cdot d} \equiv m^{k \cdot \phi(n)+1} \equiv m \mod n$ by Euler's theorem.

In terms of security, intuitively speaking, it relies on the assumption that factoring n is computationally hard. Without factoring n, an adversary Eve cannot discover a suitable decryption exponent d, and is stuck. However, the above *plain* RSA encryption turns out to be *insecure* because an adversary Eve can compute $\tilde{c} = \tilde{m}^e \mod n$ for any possible message $\tilde{m}$ and compare $\tilde{c}$ with c. We eschew a secure RSA encryption scheme with a rigorous security proof in favor of a theoretical cryptography course (CSCI 1510, MATH 1580).

1.5 RSA Signature

Orthogonal to the problem of message secrecy is message integrity. Let's say all of our channels are being controlled by an adversary Eve; how can we protect our messages from being tampered with? One solution to this problem is to **sign** our messages; by having the signature be difficult to compute without knowledge of a secret key, Eve will not be able to sign altered messages. We explore the RSA Signature scheme.

The RSA signature scheme is quite similar to RSA encryption, with the caveat that to sign a message, we use the secret key, and to verify a signature, we use the public key. However, it is important to note that we must first **hash** our message before signing it; otherwise, an adversary Eve with access to chosen messages can forge signatures. We explore this in more detail below.

1.5.1 The importance of hashing

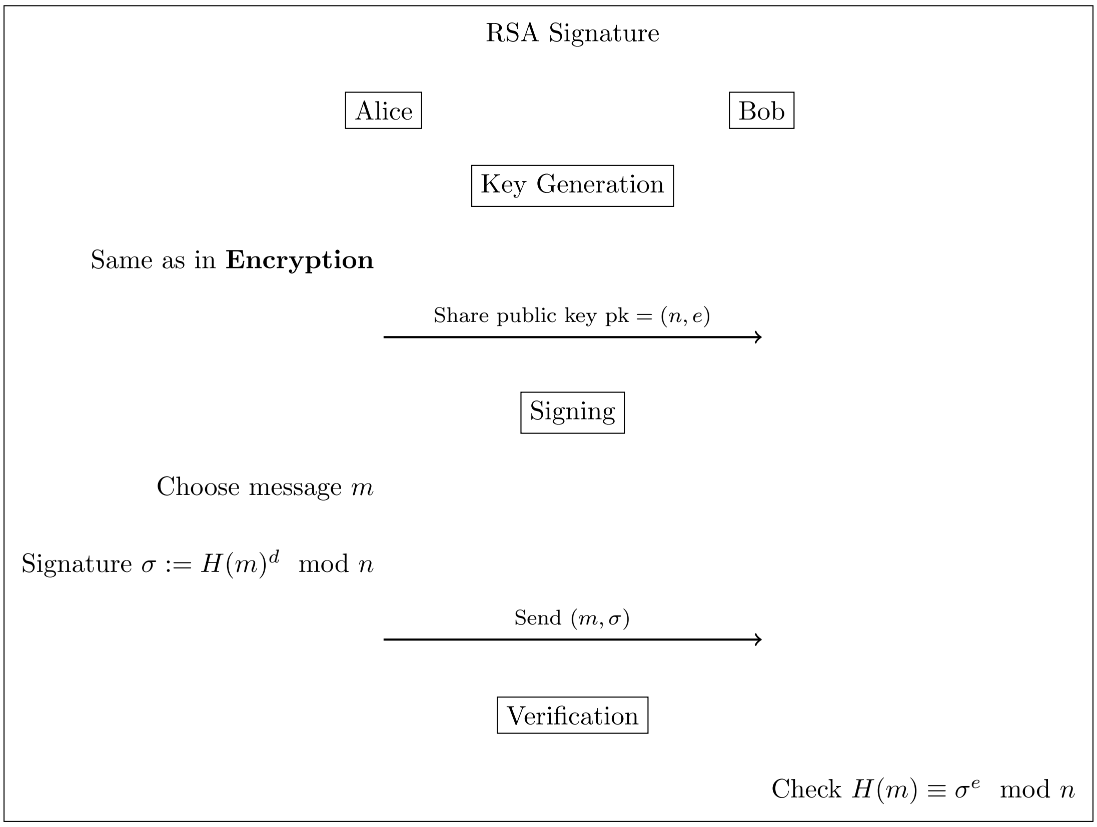
Consider if we didn't hash our messages before signing them and assume an adversary Eve has access to message signature pairs (m_0, σ_0) and (m_1, σ_1) but does not know the signing key d.

Then Eve can compute a new message $m := m_0 \cdot m_1 \mod n$ and a new signature $\sigma := \sigma_0 \cdot \sigma_1 \mod n$ and send (m, σ) to Bob.

Bob will then verify, using the verification key e, that $\sigma^e \equiv m \mod n$, since $\sigma^e \equiv \sigma_0^e \cdot \sigma_1^e \equiv m_0 \cdot m_1 \equiv m \mod n$. Thus, Eve can forge signatures without knowing the signing key.

This can be remedied by hashing our messages before signing them. Thus, instead of signing m, we sign $H(m)$ and verify that $\sigma^e \equiv H(m) \mod n$.

Now even if Eve knew $H(m_0)$ and $H(m_1)$, computing $H(m_0 \cdot m_1)$ will be intractible assuming our hash function is a *random oracle* that produces a random output on any input, so Eve cannot forge signatures without knowing the signing key.


RSA Signature
Alice
Bob
Key Generation
Same as in **Encryption**
Share public key pk = (n, e)
Signing
Choose message m
Signature $\sigma := H(m)^d \mod n$
Send (m, σ)
Verification
Check $H(m) \equiv \sigma^e \mod n$

Correctness relies on the same arguments from RSA encryption. Security relies on the RSA assumption and the security of the hash function used. We eschew a rigorous security proof in favor of a theoretical cryptography course.

1.6 A Word of Caution

While we are having you implement some schemes on your own, know that this is an exercise to help you understand these algorithms, not a warrant to use home-rolled schemes in the wild. Building these schemes so that they are efficient and work securely all the time is a fool's errand, and we are all better off using standardized implementations from well-vetted libraries (as we will do for the rest of the course with the CryptoPP library).

A pledge for another scheme, AES, rings true:

I promise that once I see how simple AES really is, I will not implement it in production code even though it will be really fun. This agreement will remain in effect until I learn all about side-channel attacks and countermeasures to the point where I lose all interest in implementing AES myself.

2 Assignment Specification

Please note that you may *NOT* change any of the function headers defined in the stencil. Doing so will break the autograder; if you don't understand a function header, please ask us what it means and we'll be happy to clarify.

2.1 Cryptographic Schemes

In this assignment you will implement four cryptographic schemes: Diffie-Hellman key exchange, ElGamal encryption, RSA encryption, and RSA signature. Using what you know about these schemes from class and from the descriptions above, implement the function headers in 🗀 `src/cipher.cxx`. We recommend doing them in the order they are introduced, but there is no best way to complete this assignment. In particular, you should edit the following functions:

- `diffie_hellman(...)`
- `elgamal_encrypt(...)`
- `elgamal_decrypt(...)`
- `rsa_encrypt(...)`
- `rsa_decrypt(...)`
- `rsa_sign(...)`
- `rsa_verify(...)`

Remember to use the provided functions to compute random values and powers. Using outside functions for any of the schemes isn't permitted.

2.2 C++

Throughout this course, we will use C++. We use C++ because it is the language in which most cryptographic libraries are written, especially those used later in the course. Moreover, it is a highly performant language that affords us great control over the systems we build.

Our development environment makes it very easy for you to write and build C++. In terms of syntax, we recommend cppreference and learncpp as good resources to help you learn. In general, we won't be using very advanced C++ features, but it is good to understand basic syntax. If you ever have questions, do not hesitate to reach out.

2.3 Libraries: CryptoPP

In this and future assignments, we will be using CryptoPP as our library of choice for our basic cryptographic primitives. CryptoPP is a widely used and trusted suite of cryptographic primitives; others like it include OpenSSL. It operates at a nice level for learning, where it abstracts away computation specifics while still allowing users to write technical cryptographic programs. We will introduce other libraries as the course goes along and make sure that you have all of the documentation you need on hand to build what we ask you to build.

As of late August 2024, CryptoPP stopped hosting their documentation. If you are searching for documentation online, it will likely be helpful to use the internet archive or similar. We have provided links to the archived version below.

You may find the following functions useful:

- `CryptoPP::EuclideanMultiplicativeInverse`
- `CryptoPP::ModularExponentiation`

You may find the following wiki pages useful during this assignment:

- CryptoPP Integer
- CryptoPP nbtheory

3 Getting Started

First, make sure you have a local development environment set up! See the development environment guide for help, and let us know via EdStem or TA Hours if you need help getting your dev environment set up.

To get started, get your stencil repository here and clone it into the 🗀 `devenv/home` folder. From here you can access the code from both your computer and from the Docker container.

3.1 Running

We use CMake in this course to manage builds. You won't need to know any CMake beyond what we detail in this section, but be aware that it exists and will be used to build your projects.

To build the project, `cd` into the 🗀 `build` folder and run `cmake ..`. This will generate a set of Makefiles to build the whole project. From here, you can run `make` to generate a binary you can run, and you can run `make check` to run any tests you write in the 🗀 `test` folder.

3.2 Testing

You may write tests in any of the 🗀 `test/**.cxx` files in the Doctest format. We provide 🗀 `test/test_provided.cxx`, feel free to write more tests in this file directly. If you want to add any new test *files*, make sure to add the file to the `cmake` variable, `TESTFILES`, on line 7 of 🗀 `test/CMakeLists.txt` so that CMake can pick up on the new files. Examples have been included in the assignment stencil. To build and run the tests, run `make check` in the `build` directory. The specifics of tests run on Gradescope are hidden, as they reveal how implementations should look. As such, we encourage you to write your own tests as you go. You will not be graded on any tests you write.

3.3 Note on Submitting

Before you submit to gradescope, you *must* return a value for every function. Otherwise, the autograder will segfault and you won't be able to see your score.

If you wish to submit to the autograder before you are fully finished, we suggest you put temporary return values for functions that haven't been fully implemented.