Chaining Modes

- Block chaining modes
- Extending chaining modes to bytes

Public-key Cryptography

RSA

Big Numbers

- Factoring Assumption
- Computing with Big Numbers

Fast Exponentiation Algorithms

Number Theory Needed for RSA

Appendix
Chaining Modes
Block chaining modes

Encrypting sequences of blocks

Recall from Lecture 5: A chaining mode tells how to encrypt a sequence of plaintext blocks $m_1, m_2, \ldots, m_t$ to produce a corresponding sequence of ciphertext blocks $c_1, c_2, \ldots, c_t$, and conversely, how to recover the $m_i$’s given the $c_i$’s.

Electronic Code Book (ECB) mode encrypts/decrypts each block separately.

Output Feedback (OFB) mode repeatedly applies the block cipher to a fixed initialization vector (IV) to produce a sequence of subkeys. Each block is encrypted/decrypted by XORing with the corresponding subkey.
Removing cipher block dependence from ECB

ECB has the undesirable property that identical plaintext blocks yield identical ciphertext blocks.

_Cipher Block Chaining Mode (CBC)_ breaks this relationship by mixing in the _previous_ ciphertext block when encrypting the current block.

- To encrypt, Alice applies $E_k$ to the XOR of the current plaintext block with the previous ciphertext block. That is, $c_i = E_k(m_i \oplus c_{i-1})$.
- To decrypt, Bob computes $m_i = D_k(c_i) \oplus c_{i-1}$.

To get started, we take $c_0 = IV$, where IV is a fixed _initialization vector_ which we assume is publicly known.
Block chaining modes

Removing cipher block dependence from OFB

OFB has the undesirable property that two messages with identical plaintext blocks in corresponding block positions will yield identical ciphertext blocks in those same positions.

**Cipher Feedback (CFB)** mode breaks this relationship by choosing the current subkey \( k_i \) to be the encryption of the previous ciphertext block \( c_{i-1} \) rather than as the encryption of the previous subkey as is done with OFB.
Block chaining modes

Cipher Feedback (CFB)

- To encrypt, Alice computes $k_i = E_k(c_{i-1})$ and $c_i = m_i \oplus k_i$.
  $c_0$ is a fixed initialization vector.
- To decrypt, Bob computes $k_i = E_k(c_{i-1})$ and $m_i = c_i \oplus k_k$.

Note that Bob is able to decrypt without using the block decryption function $D_k$. In fact, it is not even necessary for $E_k$ to be a one-to-one function (but using a non one-to-one function might weaken security).
OFB, CFB, and stream ciphers

Both OFB and CFB are closely related to stream ciphers. In both cases, \( c_i = m_i \oplus k_i \), where subkey \( k_i \) is computed from the master key and the data that came before stage \( i \).

Like a one-time pad, OFB is insecure if the same key is ever reused, for the sequence of \( k_i \)'s generated will be the same. If \( m \) and \( m' \) are encrypted using the same key \( k \), then \( m \oplus m' = c \oplus c' \).

CFB partially avoids this problem, for even if the same key \( k \) is used for two different message sequences \( m_i \) and \( m'_i \), it is only true that \( m_i \oplus m'_i = c_i \oplus c'_i \oplus E_k(c_{i-1}) \oplus E_k(c'_{i-1}) \), and the dependency on \( k \) does not drop out. However, the problem still exists when \( m \) and \( m' \) share a prefix.
Propagating Cipher-Block Chaining Mode (PCBC)

Here is a more complicated chaining rule that nonetheless can be deciphered.

- To encrypt, Alice XORs the current plaintext block, previous plaintext block, and previous ciphertext block. That is, $c_i = E_k(m_i \oplus m_{i-1} \oplus c_{i-1})$. Here, both $m_0$ and $c_0$ are fixed initialization vectors.
- To decrypt, Bob computes $m_i = D_k(c_i) \oplus m_{i-1} \oplus c_{i-1}$. 
Recovery from data corruption

In real applications, a ciphertext block might be damaged or lost. An interesting property is how much plaintext is lost as a result.

- With ECB and OFB, if Bob receives a bad block $c_i$, then he cannot recover the corresponding $m_i$, but all good ciphertext blocks can be decrypted.

- With CBC and CFB, Bob needs good $c_i$ and $c_{i-1}$ blocks in order to decrypt $m_i$. Therefore, a bad block $c_i$ renders both $m_i$ and $m_{i+1}$ unreadable.

- With PCBC, bad block $c_i$ renders $m_j$ unreadable for all $j \geq i$.

Error-correcting codes applied to the ciphertext are often used in practice since they minimize lost data and give better indications of when irrecoverable data loss has occurred.
Other modes

Other modes can easily be invented.

In all cases, $c_i$ is computed by some expression (which may depend on $i$) built from $E_k()$ and $\oplus$ applied to available information:

- ciphertext blocks $c_1, \ldots, c_{i-1}$,
- message blocks $m_1, \ldots, m_i$,
- any initialization vectors.

Any such equation that can be “solved” for $m_i$ (by possibly using $D_k()$ to invert $E_k()$) is a suitable chaining mode in the sense that Alice can produce the ciphertext and Bob can decrypt it.

Of course, the resulting security properties depend heavily on the particular expression chosen.
Stream ciphers from OFB and CFB block ciphers

OFB and CFB block modes can be turned into stream ciphers.

Both compute \( c_i = m_i \oplus k_i \), where

- \( k_i = E_k(k_{i-1}) \) (for OFB);
- \( k_i = E_k(c_{i-1}) \) (for CFB).

Assume a block size of \( b \) bytes. Number the bytes in block \( m_i \) as \( m_{i,0}, \ldots, m_{i,b-1} \) and similarly for \( c_i \) and \( k_i \).

Then \( c_{i,j} = m_{i,j} \oplus k_{i,j} \), so each output byte \( c_{i,j} \) can be computed before knowing \( m_{i,j'} \) for \( j' > j \); no need to wait for all of \( m_i \).

One must keep track of \( j \). When \( j = b \), the current block is finished, \( i \) must be incremented, \( j \) must be reset to 0, and \( k_{i+1} \) must be computed.
Byte chaining modes

Extended OFB and CFB modes

Simpler (for hardware implementation) and more uniform stream ciphers result by also computing $k_i$ a byte at a time.

**The idea:** Use a shift register $X$ to accumulate the feedback bits from previous stages of encryption so that the full-sized blocks needed by the block chaining method are available.

$X$ is initialized to some public initialization vector.

Details are in the [appendix](#).
Public-key Cryptography
Public-key cryptography

Classical cryptography uses a single key for both encryption and decryption. This is also called a *symmetric* or *1-key* cryptography.

There is no logical reason why the encryption and decryption keys should be the same.

Allowing them to differ gives rise to *asymmetric* cryptography, also known as *public-key* or *2-key* cryptography.
Asymmetric cryptosystems

An **asymmetric cryptosystem** has a pair \( k = (k_e, k_d) \) of related keys, the **encryption key** \( k_e \) and the **decryption key** \( k_d \).

Alice encrypts a message \( m \) by computing \( c = E_{k_e}(m) \).
Bob decrypts \( c \) by computing \( m = D_{k_d}(c) \).

We sometimes write \( e \) and \( d \) as shorthand for \( k_e \) and \( k_d \), respectively.

As always, the decryption function inverts the encryption function, so \( m = D_d(E_e(m)) \).
Security requirement

Should be hard for Eve to find $m$ given $c = E_e(m)$ and $e$.

- The system remains secure even if the encryption key $e$ is made public!
- $e$ is said to be the public key and $d$ the private key.

Reason to make $e$ public.

- Anybody can send an encrypted message to Bob. Sandra obtains Bob’s public key $e$ and sends $c = E_e(m)$ to Bob.
- Bob recovers $m$ by computing $D_d(c)$, using his private key $d$.

This greatly simplifies key management. No longer need a secure channel between Alice and Bob for the initial key distribution (which I have carefully avoided talking about so far).
Man-in-the-middle attack against 2-key cryptosystem

An active adversary Mallory can carry out a nasty *man-in-the-middle* attack.

- Mallory sends his own encryption key to Sandra when she attempts to obtain Bob’s key.
- Not knowing she has been duped, Sandra encrypts her private data using Mallory’s public key, so Mallory can read it (but Bob cannot)!
- To keep from being discovered, Mallory intercepts each message from Sandra to Bob, decrypts using his own decryption key, re-encrypts using Bob’s public encryption key, and sends it on to Bob. Bob, receiving a validly encrypted message, is none the wiser.
Passive attacks against a 2-key cryptosystem

Making the encryption key public also helps a passive attacker.

1. **Chosen-plaintext attacks** are always available since Eve can generate as many plaintext-ciphertext pairs as she wishes using the public encryption function $E_e()$.

2. The public encryption function also gives Eve a foolproof way to check validity of a potential decryption. Namely, Eve can verify $D_d(c) = m_0$ for some candidate message $m_0$ by checking that $c = E_e(m_0)$.

Redundancy in the set of meaningful messages is no longer necessary for brute force attacks.
Facts about asymmetric cryptosystems

Good asymmetric cryptosystems are much harder to design than good symmetric cryptosystems.

All known asymmetric systems are orders of magnitude slower than corresponding symmetric systems.
Hybrid cryptosystems

Asymmetric and symmetric cryptosystems are often used together. Let \((E^2, D^2)\) be a 2-key cryptosystem and \((E^1, D^1)\) be a 1-key cryptosystem.

Here’s how Alice sends a secret message \(m\) to Bob.

- Alice generates a random session key \(k\).
- Alice computes \(c_1 = E_k^1(m)\) and \(c_2 = E_e^2(k)\), where \(e\) is Bob’s public key, and sends \((c_1, c_2)\) to Bob.
- Bob computes \(k = D_d^2(c_2)\) using his private decryption key \(d\) and then computes \(m = D_k^1(c_1)\).

This is much more efficient than simply sending \(E_e^2(m)\) in the usual case that \(m\) is much longer than \(k\).

Note that the 2-key system is used to encrypt random strings!
RSA
Overview of RSA

Probably the most commonly used asymmetric cryptosystem today is **RSA**, named from the initials of its three inventors, Rivest, Shamir, and Adelman.

Unlike the symmetric systems we have been talking about so far, RSA is based not on substitution and transposition but on arithmetic involving very large integers—numbers that are hundreds or even thousands of bits long.

To understand why RSA works requires knowing a bit of number theory. However, the basic ideas can be presented quite simply, which I will do now.
RSA spaces

The message space, ciphertext space, and key space for RSA is the set of integers $\mathbb{Z}_n = \{0, \ldots, n - 1\}$ for some very large integer $n$.

For now, think of $n$ as a number so large that its binary representation is 1024 bits long.

Such a number is unimaginably big. It is bigger than $2^{1023} \approx 10^{308}$.

For comparison, the number of atoms in the observable universe\(^1\) is estimated to be “only” $10^{80}$.

Encoding bit strings by integers

To use RSA as a block cipher on bit strings, Alice must convert each block to an integer $m \in \mathbb{Z}_n$, and Bob must convert $m$ back to a block.

Many such encodings are possible, but perhaps the simplest is to prepend a “1” to the block $x$ and regard the result as a binary integer $m$.

To decode $m$ to a block, write out $m$ in binary and then delete the initial “1” bit.

To ensure that $m < n$ as required, we limit the length of our blocks to 1022 bits.
RSA key generation

Here’s how Bob generates an RSA key pair.

- Bob chooses two sufficiently large distinct prime numbers $p$ and $q$ and computes $n = pq$. For security, $p$ and $q$ should be about the same length (when written in binary).

- He computes two numbers $e$ and $d$ with a certain number-theoretic relationship.

- The public key is the pair $k_e = (e, n)$. The private key is the pair $k_d = (d, n)$. The primes $p$ and $q$ are no longer needed and should be discarded.
RSA encryption and decryption

To encrypt, Alice computes $c = m^e \mod n$. ²

To decrypt, Bob computes $m = c^d \mod n$.

Here, $a \mod n$ denotes the remainder when $a$ is divided by $n$.

This works because $e$ and $d$ are chosen so that, for all $m$,

$$m = (m^e \mod n)^d \mod n. \quad (1)$$

That's all there is to it, once the keys have been found.

Most of the complexity in implementing RSA has to do with key generation, which fortunately is done only infrequently.

²For now, assume all messages and ciphertexts are integers in $\mathbb{Z}_n$. 
RSA questions

You should already be asking yourself the following questions:

▶ How does one find \( n, e, d \) such that (1) is satisfied?
▶ Why is RSA believed to be secure?
▶ How can one implement RSA on a computer when most computers only support arithmetic on 32-bit or 64-bit integers, and how long does it take?
▶ How can one possibly compute \( m^e \mod n \) for 1024 bit numbers. \( m^e \), before taking the remainder, has size roughly

\[
(2^{1024})^{2^{1024}} = 2^{1024 \times 2^{1024}} = 2^{2^{10} \times 2^{1024}} = 2^{2^{1034}}.
\]

This is a number that is roughly \( 2^{1034} \) bits long! No computer has enough memory to store that number, and no computer is fast enough to compute it.
Tools needed to answer RSA questions

Two kinds of tools are needed to understand and implement RSA.

**Algorithms:** Need clever algorithms for primality testing, fast exponentiation, and modular inverse computation.

**Number theory:** Need some theory of $\mathbb{Z}_n$, the integers modulo $n$, and some special properties of numbers $n$ that are the product of two primes.
Big Numbers
Factoring assumption

The \textit{factoring problem} is to find a prime divisor of a composite number \( n \).

The \textit{factoring assumption} is that there is no probabilistic polynomial-time algorithm for solving the factoring problem, even for the special case of an integer \( n \) that is the product of just two distinct primes.

The security of RSA is based on the factoring assumption. No feasible factoring algorithm is known, but there is no proof that such an algorithm does not exist.
How big is big enough?

The security of RSA depends on \( n, p, q \) being sufficiently large.

What is sufficiently large? That’s hard to say, but \( n \) is typically chosen to be at least 1024 bits long, or for better security, 2048 bits long.

The primes \( p \) and \( q \) whose product is \( n \) are generally chosen be roughly the same length, so each will be about half as long as \( n \).
Algorithms for arithmetic on big numbers

The arithmetic built into typical computers can handle only 32-bit or 64-bit integers. Hence, all arithmetic on large integers must be performed by software routines.

The straightforward algorithms for addition and multiplication have time complexities $O(N)$ and $O(N^2)$, respectively, where $N$ is the length (in bits) of the integers involved.

Asymptotically faster multiplication algorithms are known, but they involve large constant factor overheads. It’s not clear whether they are practical for numbers of the sizes we are talking about.
Big number libraries

A lot of cleverness is possible in the careful implementation of even the $O(N^2)$ multiplication algorithms, and a good implementation can be many times faster in practice than a poor one. They are also hard to get right because of many special cases that must be handled correctly!

Most people choose to use big number libraries written by others rather than write their own code.

Two such libraries that you can use in this course:

1. GMP (GNU Multiple Precision Arithmetic Library);
2. The big number routines in the openssl crypto library.
GMP provides a large number of highly-optimized function calls for use with C and C++. It is preinstalled on all of the Zoo nodes and supported by the open source community. Type `info gmp` at a shell for documentation.
Openssl crypto package

OpenSSL is a cryptography toolkit implementing the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) network protocols and related cryptography standards required by them.

It is widely used and pretty well debugged. The protocols require cryptography, and OpenSSL implements its own big number routines which are contained in its crypto library.

Type `man crypto` for general information about the library, and `man bn` for specifics of the big number routines.
Fast Exponentiation Algorithms
Modular exponentiation

The basic operation of RSA is modular exponentiation of big numbers, i.e., computing $m^e \mod n$ for big numbers $m$, $e$, and $n$.

The obvious way to compute this would be to compute first $t = m^e$ and then compute $t \mod n$.

This has two serious drawbacks.
Computing $m^e$ the conventional way is too slow

The simple iterative loop to compute $m^e$ requires $e$ multiplications, or about $2^{1024}$ operations in all. This computation would run longer than the current age of the universe (which is estimated to be 15 billion years).

Assuming one loop iteration could be done in one microsecond (very optimistic seeing as each iteration requires computing a product and remainder of big numbers), only about $30 \times 10^{12}$ iterations could be performed per year, and only about $450 \times 10^{21}$ iterations in the lifetime of the universe. But $450 \times 10^{21} \approx 2^{79}$, far less than $e - 1$. 
The result of computing $m^e$ is too big to write down.

The number $m^e$ is too big to store! This number, when written in binary, is about $1024 \times 2^{1024}$ bits long, a number far larger than the number of atoms in the universe (which is estimated to be only around $10^{80} \approx 2^{266}$).
Controlling the size of intermediate results

The trick to get around the second problem is to do all arithmetic modulo $n$, that is, reduce the result modulo $n$ after each arithmetic operation.

The product of two length $\ell$ numbers is only length $2\ell$ before reduction mod $n$, so in this way, one never has to deal with numbers longer than about 2048 bits.

Question to think about: Why is it correct to do this?
Efficient exponentiation

The trick to avoiding the first problem is to use a more efficient exponentiation algorithm based on repeated squaring.

For the special case of $e = 2^k$, one computes $m^e \mod n$ as follows:

\[
\begin{align*}
m_0 &= m \\
m_1 &= (m_0 \times m_0) \mod n \\
m_2 &= (m_1 \times m_1) \mod n \\
&\vdots \\
m_k &= (m_{k-1} \times m_{k-1}) \mod n.
\end{align*}
\]

Clearly, $m_i = m^{2^i} \mod n$ for all $i$. 
Combining the $m_i$ for general $e$

For values of $e$ that are not powers of 2, $m^e \mod n$ can be obtained as the product modulo $n$ of certain $m_i$'s.

Express $e$ in binary as $e = (b_s b_{s-1} \ldots b_2 b_1 b_0)_2$. Then $e = \sum_i b_i 2^i$, so

$$m^e = m^{\sum_i b_i 2^i} = \prod_i m^{b_i 2^i} = \prod_i (m^{2^i})^{b_i} = \prod_{i: b_i=1} m_i.$$

Since each $b_i \in \{0, 1\}$, we include exactly those $m_i$ in the final product for which $b_i = 1$. Hence,

$$m^e \mod n = \prod_{i: b_i=1} m_i \mod n.$$
Towards greater efficiency

It is not necessary to perform this computation in two phases. Rather, the two phases can be combined together, resulting in slicker and simpler algorithms that do not require the explicit storage of the $m_i$'s.

We give both a recursive and an iterative version.
A recursive exponentiation algorithm

Here is a recursive version written in C notation, but it should be understood that the C programs only work for numbers smaller than $2^{16}$. To handle larger numbers requires the use of big number functions.

```c
/* computes m^e mod n recursively */
int modexp( int m, int e, int n) {
    int r;
    if ( e == 0 ) return 1; /* m^0 = 1 */
    r = modexp(m*m % n, e/2, n); /* r = (m^2)^(e/2) mod n */
    if ( (e&1) == 1 ) r = r*m % n; /* handle case of odd e */
    return r;
}
```
An iterative exponentiation algorithm

This same idea can be expressed iteratively to achieve even greater efficiency.

/* computes m^e mod n iteratively */
int modexp( int m, int e, int n) {
    int r = 1;
    while ( e > 0 ) {
        if ( (e&1) == 1 ) r = r*m % n;
        e /= 2;
        m = m*m % n;
    }
    return r;
}
Correctness

The loop invariant is

\[ e > 0 \land (m_0^{e_0} \mod n = rm^e \mod n) \]  \hspace{1cm} (2)

where \( m_0 \) and \( e_0 \) are the initial values of \( m \) and \( e \), respectively.

Proof of correctness:

- It is easily checked that (2) holds at the start of each iteration.
- If the loop exits, then \( e = 0 \), so \( r \mod n \) is the desired result.
- Termination is ensured since \( e \) gets reduced during each iteration.
A minor optimization

Note that the last iteration of the loop computes a new value of \( m \) that is never used. A slight efficiency improvement results from restructuring the code to eliminate this unnecessary computation. Following is one way of doing so.

```c
/* computes \( m^e \mod n \) iteratively */
int modexp( int m, int e, int n ) {
    int r = ( (e&1) == 1 ) ? m % n : 1;
    e /= 2;
    while ( e > 0 ) {
        m = m*m % n;
        if ( (e&1) == 1 ) r = r*m % n;
        e /= 2;
    }
    return r;
}
```
Number Theory Needed for RSA
Number theory overview

In this and following sections, we review some number theory that is needed for understanding RSA.

I will provide only a high-level overview. Further details are contained in course handouts and the textbooks.
Bare-bones definitions

The following definitions apply to the RSA parameters.

- $p$, $q$ are distinct large primes of roughly the same length.
- $n = pq$.
- $\mathbb{Z}_n = \{0, 1, \ldots, n - 1\}$.
- $\phi(n) = (p - 1)(q - 1)$.
- $\mathbb{Z}_n^*$ are the numbers in $\mathbb{Z}_n$ that are relatively prime to $n$ (that is, not divisible by either $p$ or $q$, which is most of them).

Fact: $|\mathbb{Z}_n^*| = \phi(n)$.
Summary of what is needed

Here’s a summary of the number theory needed to understand RSA and its associate algorithms.

- Greatest common divisor, $\mathbb{Z}_n$, mod $n$, $\phi(n)$, $\mathbb{Z}_n^*$, and how to add, subtract, multiply, and find inverses mod $n$.
- Euler’s theorem: $a^{\phi(n)} \equiv 1 \pmod{n}$ for $a \in \mathbb{Z}_n^*$.
- How to generate large prime numbers: density of primes and testing primality.
Appendix
Extended OFB and CFB notation

Details for extended modes.

Assume block size $b = 16$ bytes.

Define two operations: $L$ and $R$ on blocks:

- $L(x)$ is the leftmost byte of $x$;
- $R(x)$ is the rightmost $b - 1$ bytes of $x$. 
Extended OFB and CFB similarities

The extended versions of OFB and CFB are very similar.

Both maintain a one-block shift register $X$.

The shift register value $X_s$ at stage $s$ depends only on $c_1, \ldots, c_{s-1}$ (which are now single bytes) and the master key $k$.

At stage $i$, Alice

- computes $X_s$ according to Extended OFB or Extended CFB rules;
- computes byte key $k_s = L(E_k(X_s))$;
- encrypts message byte $m_s$ as $c_s = m_s \oplus k_s$.

Bob decrypts similarly.
Shift register rules

The two modes differ in how they update the shift register.

**Extended OFB mode**

\[ X_s = R(X_{s-1}) \cdot k_{s-1} \]

**Extended CFB mode**

\[ X_s = R(X_{s-1}) \cdot c_{s-1} \]

(‘·’ denotes concatenation.)

**Summary:**

- Extended OFB keeps the most recent \( b \) key bytes in \( X \).
- Extended CFB keeps the most recent \( b \) ciphertext bytes in \( X \),
Comparison of extended OFB and CFB modes

The differences seem minor, but they have profound implications on the resulting cryptosystem.

▶ In eOFB mode, $X_s$ depends only on $s$ and the master key $k$ (and the initialization vector IV), so loss of a ciphertext byte causes loss of only the corresponding plaintext byte.

▶ In eCFB mode, loss of ciphertext byte $c_s$ causes $m_s$ and all succeeding message bytes to become undecipherable until $c_s$ is shifted off the end of $X$. Thus, $b$ message bytes are lost.
Downside of extended OFB

The downside of eOFB is that security is lost if the same master key is used twice for different messages. CFB does not suffer from this problem since different messages lead to different ciphertexts and hence different keystreams.

Nevertheless, eCFB has the undesirable property that the keystreams are the same up to and including the first byte in which the two message streams differ.

This enables Eve to determine the length of the common prefix of the two message streams and also to determine the XOR of the first bytes at which they differ.
Possible solution

Possible solution to both problems: Use a different initialization vector for each message. Prefix the ciphertext with the (unencrypted) IV so Bob can still decrypt.