Wireless Systems Security

EE/NiS/TM-584-A/WS

Bruce McNair
bmcnair@stevens.edu

Week 4:
More Security Topics

This week we will continue the discussion of general security topics.

Evolution of Cryptography

- Monoalphabetic substitution, e.g.,
  - Caesar cipher {a,b,c,d,e,f,…,x,y,z} -> {b,c,d,e,f,g,…,y,z,a}
  - Atbash cipher {a,b,c,d,e,f,…x,y,z} -> (z,y,x,w,v,….c,b,a}
  - Any permutation of the alphabet
  - Easily solved by observing single and double letter frequencies
  - English (like most other non-ideograph languages) have distinct letter frequencies over a small alphabet.
  - Encoding English letters requires $\log_2(26) \sim 4.7$ bits/letter,
  - but information content in English text is

    $\Sigma\ p \log_2(p)$

    With unequal letter probabilities, actual information content is much lower.

    Equivocation of source is the effective information content

In order to understand some of the issues that exist in current cryptographic systems, it is useful to understand how the technology evolved. To do this, we have to go back to the earliest encryption systems. Two of the first known ciphers are the so-called Caesar cipher and the Atbash cipher. The Caesar cipher is named for one of the first people known to have used this technique – Julius Caesar, about 2000 years ago. In the same time frame, the Atbash cipher was used with the Hebrew alphabet.

What both of these ciphers have in common is that they were simple permutations of the alphabet. The Caesar cipher shifted the position of each letter by one, while the Atbash cipher reversed pairs of letters.

Both are special cases of a permutation of the alphabet. Although it might seem that the 26! (26 factorial = 26*25*24*…*3*2*1) possible permutations would make this a very difficult method of encryption to break, in fact it is rather simple.

DPOTJEFS UIJT TJNQMF NFTTBHF BOE TFF IPX FBTZ JU JT UP CSFBL

If we recognize that E is the most common letter in the English language, TH is the most common two letter combination, the fact that A and I are the only one letter words, and a two letter word must contain an A, E, I, O, or U, with only certain of those occurring as the first or last letter, we have tremendous start in identifying many of the letters in a cryptogram. It is the inherent redundancy in the language that makes this cipher easy to break.

It took until about 1400 before people had a general understanding of how to break monoalphabetic ciphers. The Dark Ages delayed progress a bit.

Evolution of Cryptography

- Monoalphabetic substitution, e.g.,
  - Caesar cipher {a,b,c,d,e,f,…,x,y,z} -> {b,c,d,e,f,g,…,y,z,a}
  - Atbash cipher {a,b,c,d,e,f,…x,y,z} -> (z,y,x,w,v,….c,b,a}
  - Any permutation of the alphabet
  - Easily solved by observing single and double letter frequencies
  - English (like most other non-ideograph languages) have distinct letter frequencies over a small alphabet.
  - Encoding English letters requires $\log_2(26) \sim 4.7$ bits/letter,
  - but information content in English text is

    $\Sigma\ p \log_2(p)$

    With unequal letter probabilities, actual information content is much lower.
    Equivocation of source is the effective information content

- Polyalphabetic substitution:

  `thisisamessagetobeencrypted` - plaintext
  `badbadbadbadbadbadbadbadbad` - key stream
  `vimujwcniuteifxqcigogtztvfh` - ciphertext

  - Correlation-like techniques find the length of the key stream, k
  - Problem then reduces to solving k monoalphabetic ciphers
  - Using running text (e.g., from an agreed to book) makes solution harder, but with enough ciphertext, both the plaintext as well as the key stream are easily found

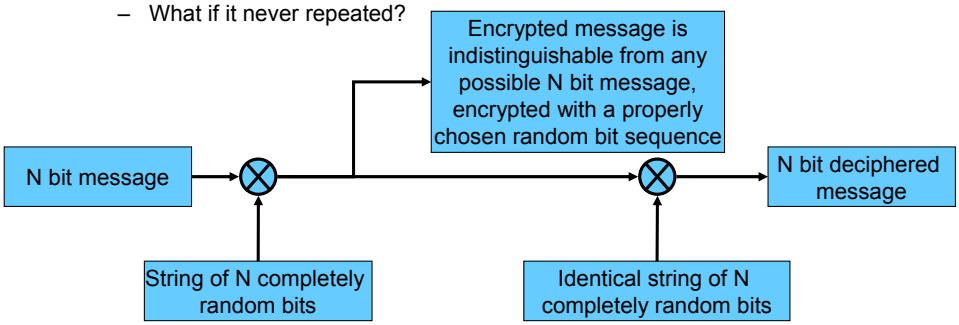
Recognizing the weakness of a monoalphabetic cipher, the polyalphabetic cipher was invented. Here, the same transformation is not applied to every letter in a message. There are several (and thus, poly-, rather than mono-) transformations applied to the message, changing for every successive letter. We need a standardized transformation method, so we can easily encrypt and decrypt, so a short repeating string was first used. This makes the monoalphabetic letter frequency attack fail. However, it turns out that there are correlation-like techniques that allow the cryptanalyst to examine only the encrypted message to find the length of the repeating key. In the case shown here, the key stream is 3 characters long. Once the cryptanalyst knows this, they can separately attack each 1/3rd of the message using the same monoalphabetic techniques as before.

The next improvement was to use a key stream that didn't repeat so quickly. For this, it became even more important to agree between sender and receiver on how the message would be encrypted. Generally, a commonly available book would be used as a source of key stream, e.g., a given passage in the Bible. Unfortunately, now the message and the key stream have some known letter statistics, so the same type of attack is still possible.

The polyalphabetic cipher didn't stand up as long as the monoalphabetic cipher. It took until the 1700s before it was easily attacked.

Evolution of Cryptography - 2

- Weakness of polyalphabetic cipher is repetition of the key stream
  - What if it never repeated?



- One-time-pad is the only provably secure cryptographic system

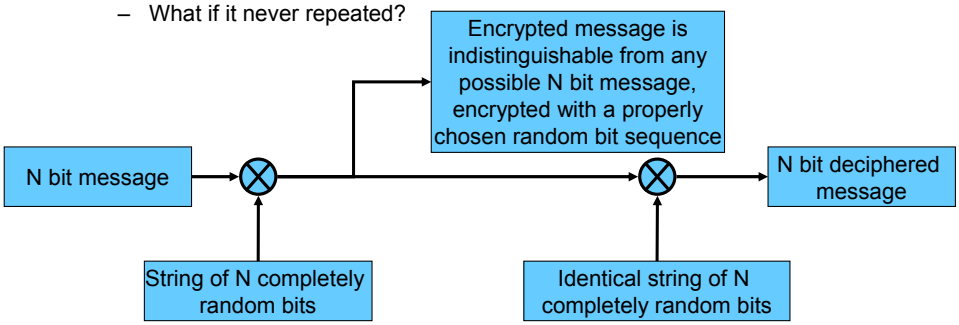
The attack against the polyalphabetic cipher eventually led to the invention of the only known provably secure cipher – the one-time pad. This concept was invented in 1917 by someone working for an organization that would later become Bell Labs.

The concept here is to use a COMPLETELY random stream of characters to encipher a message, with an identical stream of characters to decipher the message. The cipher is called the ONE-time pad because the random sequence of characters is used exactly once and then discarded. The strength of this cipher is that for any possible N bit message and any possible N bit encrypted message, there exists a random string of characters that could have transformed the message into the cipher. Since all random strings are possible, it is not possible to tell which one was used. If I receive a message

“D7HACUM0RLGRP”, it might mean “ATTACKTONIGHT” or it could just as well mean “SEND12BANANAS” or “BOMBISPLANTED” – each is a correct decryption of the message with an appropriately chosen key stream. Thus, a one-time pad, when properly used can never be decrypted.

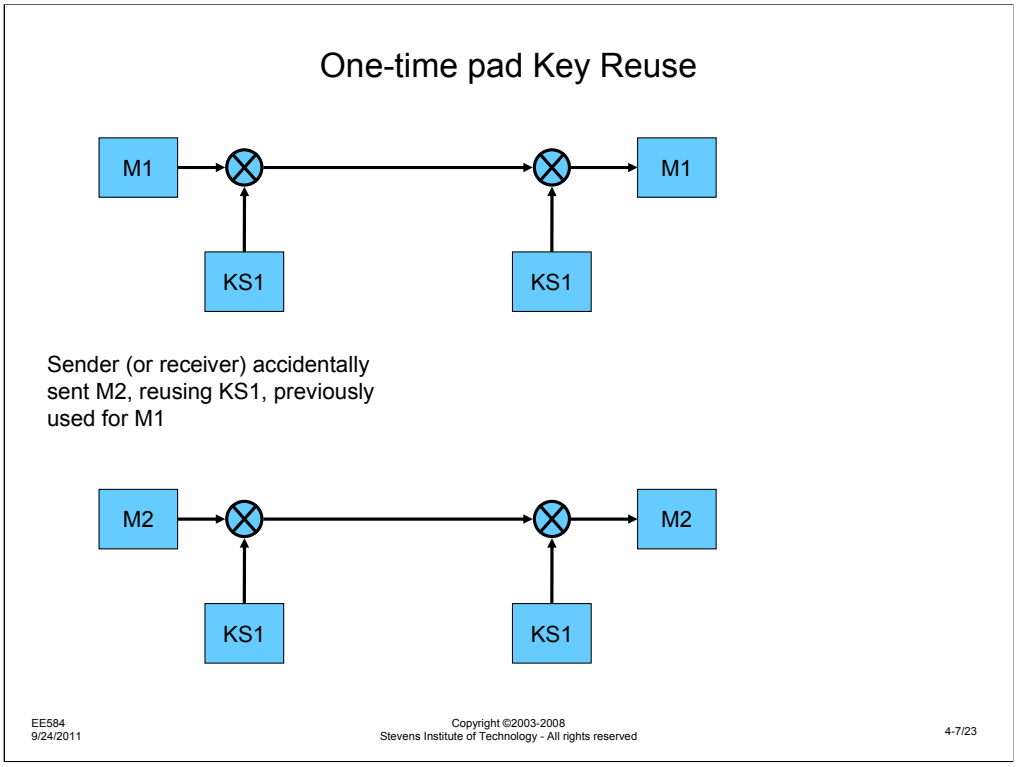
Evolution of Cryptography - 2

- Weakness of polyalphabetic cipher is repetition of the key stream
  - What if it never repeated?

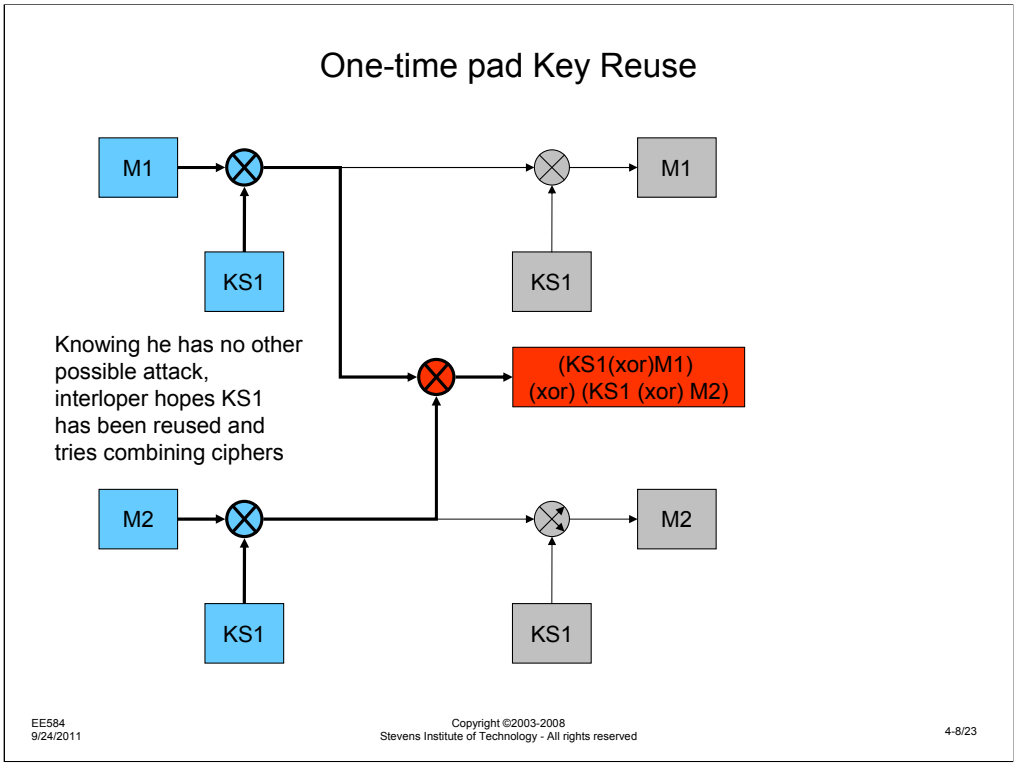


- One-time-pad is the only provably secure cryptographic system
  What happens if key sequence is (accidentally) reused?

The “key” (ouch) point here is, the one-time pad must be correctly used to remain completely secure. What if a simple mistake, like reuse of the random key sequence, is made? How might this influence the security of a one-time pad? As we will see in the next few slides, the result can render a one-time pad useless. The attack was discovered in the early 1920s by someone who was working for what would later become the National Security Agency.

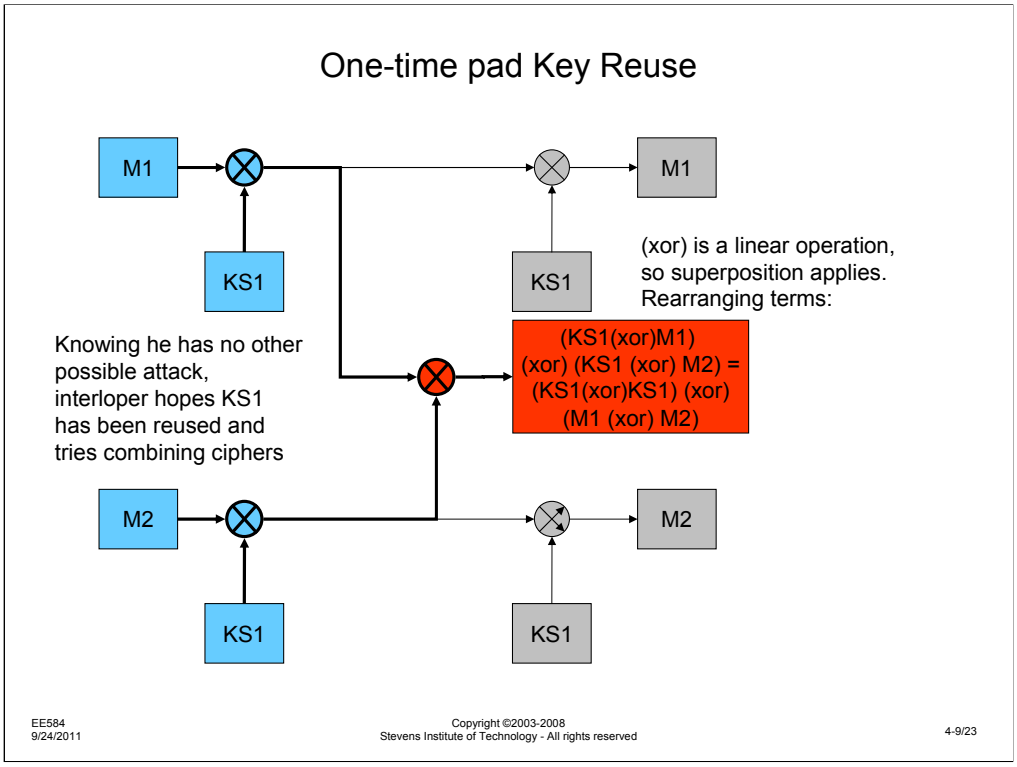


Consider that we have two messages, M1 and M2, that are to be sent using the one-time pad system. Accidentally, one of the two people who have access to the one-time pad key sequence, KS1, that was only to be use for M1, use it to encrypt M2. This is now no longer a one-time pad, since it has been used twice.

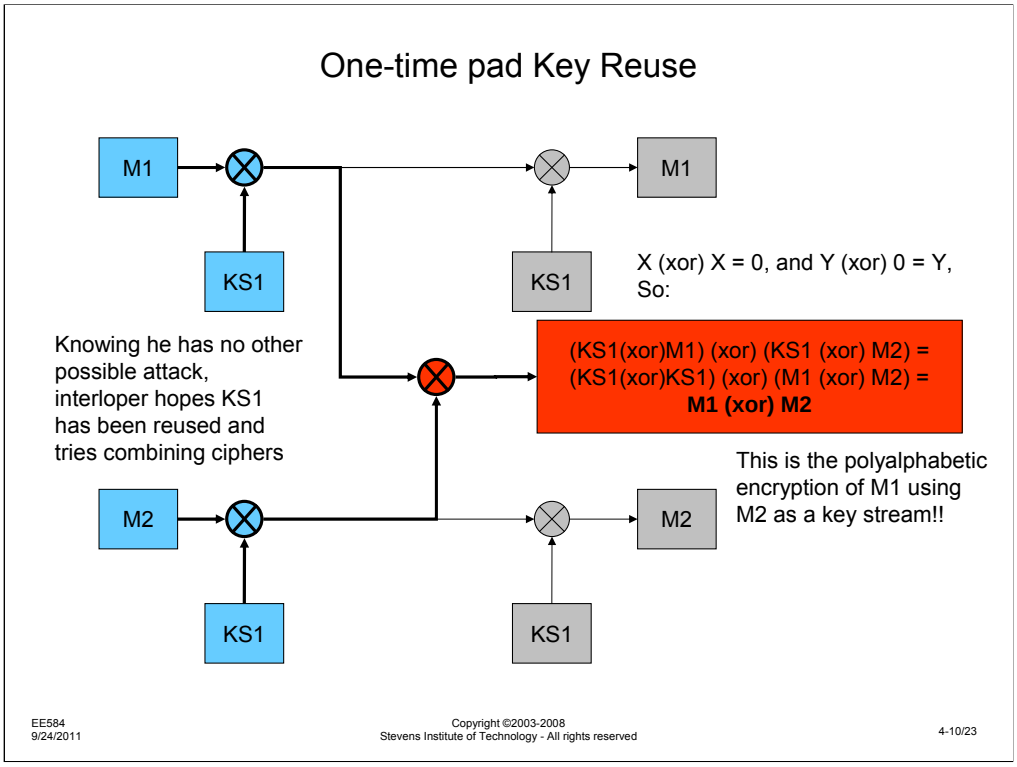


The idle cryptanalyst has been monitoring all the traffic that the parties are generating. He has tried all the tricks he knows, but sees that nothing is working. Perhaps he knows the material being sent is very sensitive and a one-time pad is in use, so he knows that no possible attack exists. Yet, his bosses are telling him that he MUST try to decrypt the messages. So, he tries the only way he can think of to try to attack the one-time pad. He assumes that an operational mistake MAY have been made, reusing a one-time pad.

The cryptanalyst tries combining the two encrypted messages with each other. The resulting string is shown in the red box, above. So far it doesn't look too interesting.



Since the XOR operation is a linear operation, we can reorder the variables, as shown. We find that, by XORing one encrypted message with the other, we have produced the string (KS1 xor KS1) xor (M1 xor M2).



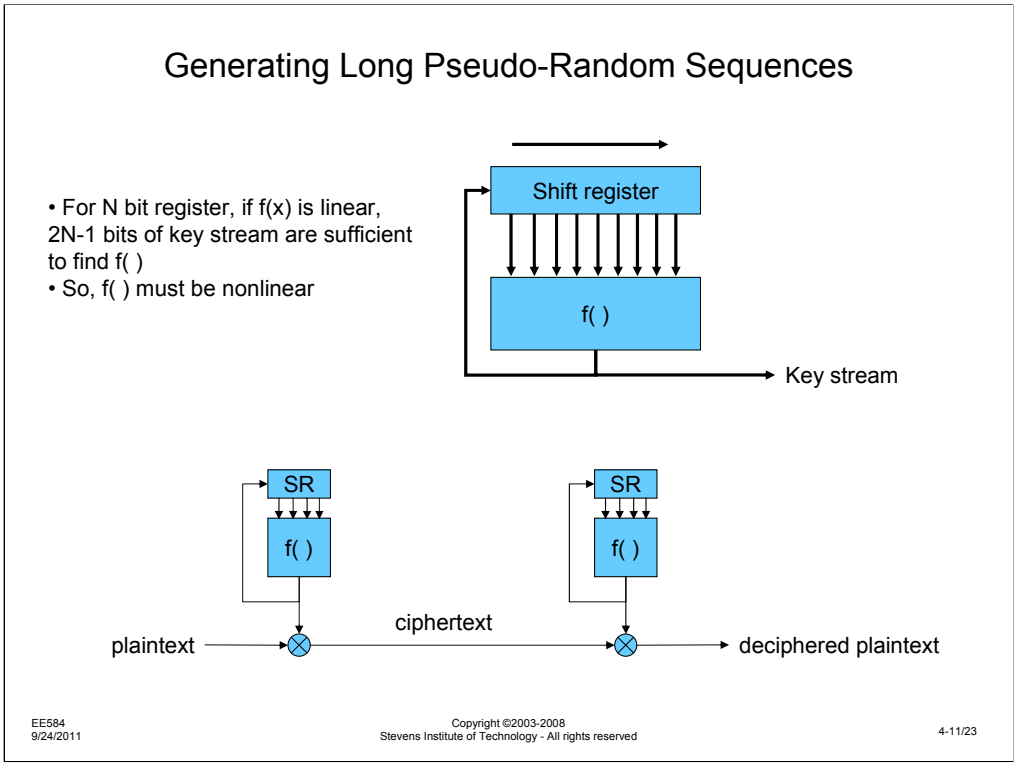
The thing that makes encryption and decryption work in the first place is the fact that X xor X = 0 (((1 xor 1 = 0) xor) 0 = 0, while ((1 xor 0 = 1) xor 1) = 0).

This means that by XORing a message twice with the same key sequence, we get the message back.

Looking at the result of XORing the two messages with each other, we see that KS1 xor KS1 = 0. What we are left with from this accidental one-time pad reuse is M1 xor M2. This is equivalent to a polyalphabetic cipher with running text, an attack that was understood 300 years ago.

Thus, reuse of a one-time pad compromises the security of the provably secure system.

As we will see later, this 80 year old attack has more to do with wireless system security than just about anything else.



Let's put aside the one-time pad for a while. While it is provably secure, it is completely impractical for day-to-day operation. As we saw, we can never reuse a random string or the entire security of the system is compromised. This means that we need as much secret key sequence as we intend to transmit data from one station to another. If we ignore the bookkeeping just needed to keep things secure, we still need to generate and securely store enormous amounts of random data.

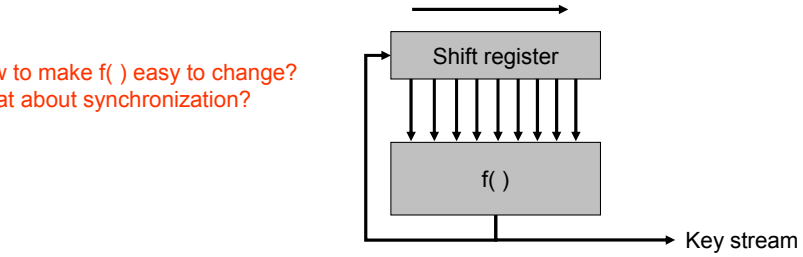
The alternative is to generate pseudorandom sequences as they are needed. The easiest way to imagine doing this is with a feedback shift register. The first that comes to mind is a linear feedback shift register. These devices have been well studied and much is known about their sequence length. An N bit register is capable of generating a sequence of length (2^N)-1 which, if N is large, can become enormous. While this sequence meets the requirement of not repeating for a long time, since the generation of values is linear, it doesn't take too much to break the sequence. If the attacker knows or can guess 2*N-1 bits of the plaintext data sequence, they have enough information to figure out the current state of the shift register (N bits) and can observe the next N-1 bits to verify the feedback taps on the register. Even though a 32 bit register has a sequence that is a billion bits long, only 4 ASCII characters are enough to break this system, so we can't use a linear feedback register.

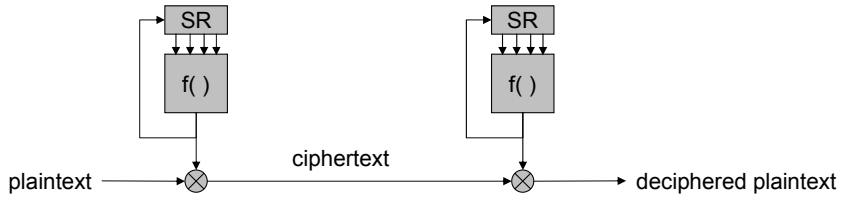
For this reason, a nonlinear feedback shift register is used. Here, we can imagine any complex, nonlinear function f( ) to generate the feedback. Picking a function that is maximal length is now difficult, but even if the sequence is not maximal length, we can make it quite long.

As shown at the bottom of the slide, if we pick the same f( ) at both ends of the link and keep the registers in the same state, we have an encryption system.

Generating Long Pseudo-Random Sequences

- How to make f( ) easy to change?
- What about synchronization?

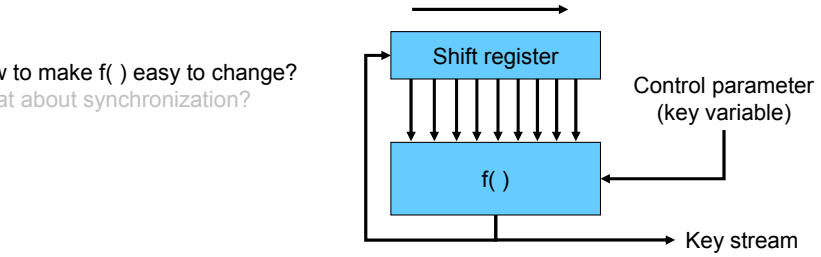


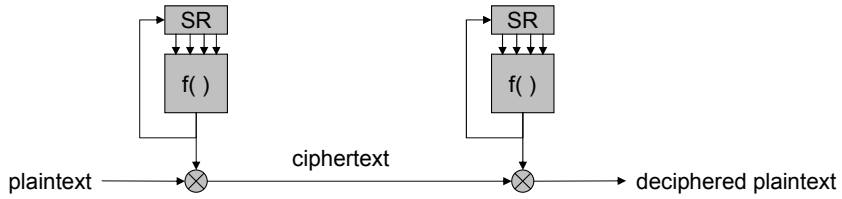


There are still two considerations to make this method usable. (1) we want to design a system that allows many terminals to use the same hardware with minimal redesign, so we want a function f( ) that is easy to change for each application. (2) for this system to work, the transmit and receive shift register have to be in the same state at the same time. If the state of the shift register at the transmitter and the receiver differ by only one bit, a well designed f( ) will generate an output sequence that is completely different between the transmitter and receiver. **Discussion topic**: why?

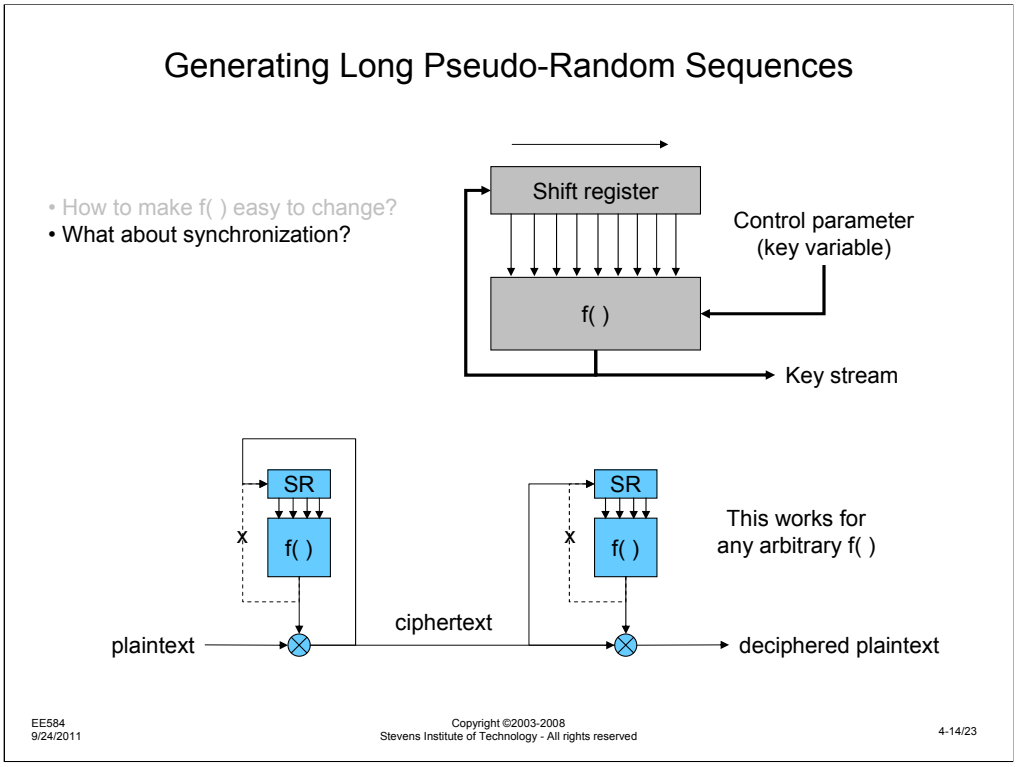
Generating Long Pseudo-Random Sequences

- How to make f( ) easy to change?
- What about synchronization?





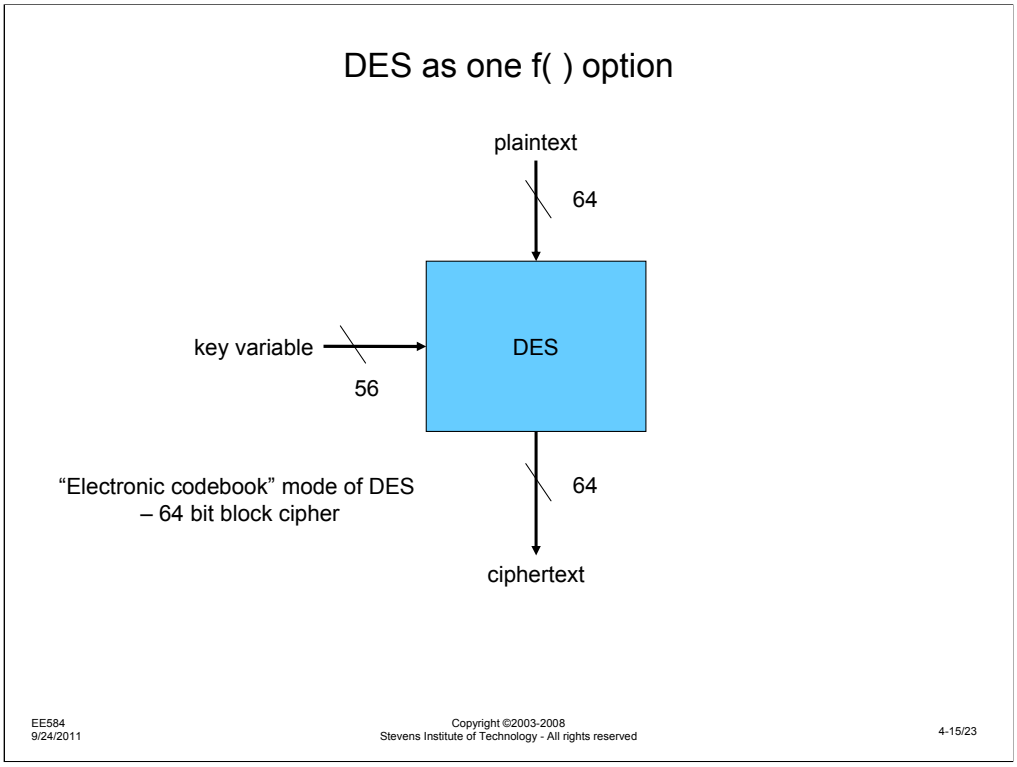
Let's address the first requirement: We want f( ) to be easy to change. In essence, we want a simple parameter to be able to control how the function operates. Basically, we can express f( ) as a function with two arguments: output_state = f(key_variable, input_state). We want the nonlinearity of f( ) to apply to both of the input parameters – changing either the key_variable parameter or the input_state parameter will have drastic effects on the output. In a few slides, we will see one representative structure that makes this requirement easy to satisfy.



The second requirement is the ability to synchronize the state of the encryptor at the receiver and transmitter.

The structure shown previously uses the output of the feedback generator to set the next state. This mode of operation is known as key-autokey in military systems and is called output-feedback in non-military systems. The state of the key generator can be conveyed from the transmitter to the receiver at the beginning of a session, but there is no guarantee that something won't happen to disrupt the synchronization. For instance, if the channel is subject to timing error, a "bit slip" can occur – 1,000,000,000,000 bits are sent, but 1,000,000,000,001 are received (or equivalently, 999,999,999,999 may be received). This is common on wireless channels, where the received signal is noisy and the receiver must make a best guess at the transmitter timing, derived from an independent clock. In wired systems, the signals are often more stable, less noisy, and other channels can be used to convey timing information.

An alternate structure is shown at the bottom of the slide. Here, instead of basing the next state on the previous state, the next state is based on the encrypted data. If there are no errors on the channel, both the transmitter and receiver have access to the same data stream, since they both need it to communicate, anyway. If there is an error on the channel, the receiver will be subject to the effect of the error momentarily, but eventually, it will be shifted out of the receive register. This method is known in military systems as ciphertext-autokey or in non-military systems as cipher-feedback mode. In this case, the encryptors synchronize automatically without any other coordination. However, the effect of channel errors is multiplied. Depending on the channel conditions and application, this may be an acceptable tradeoff. **Discussion topic**: what characteristics of an application may make this error multiplication acceptable?

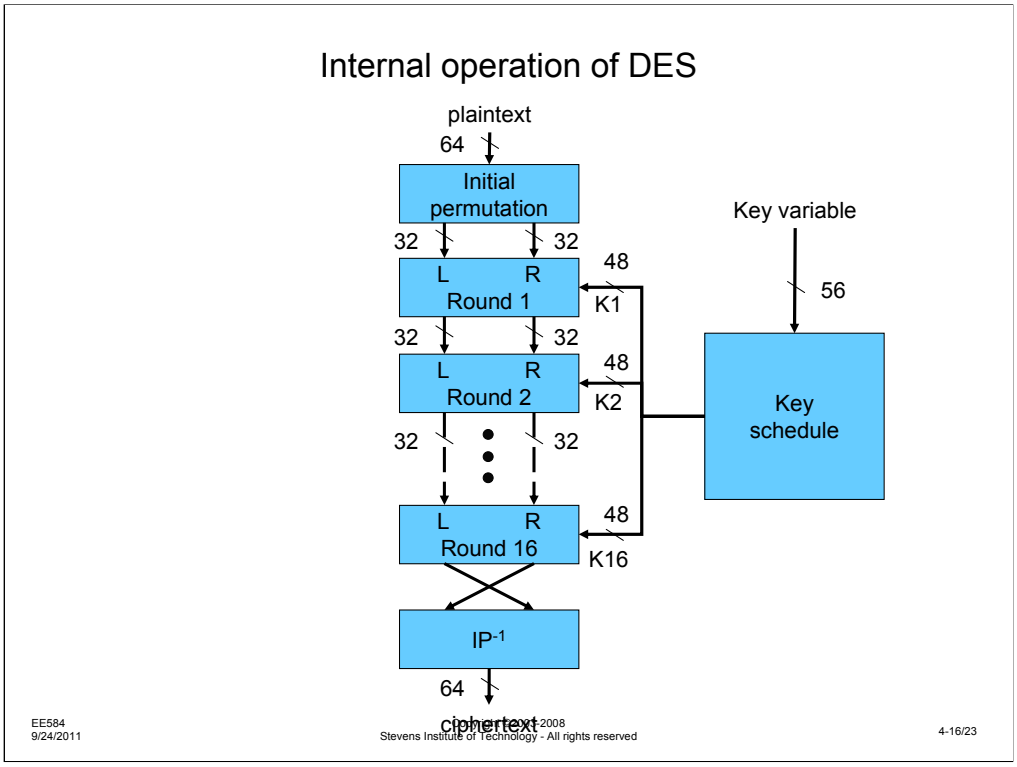


Let's examine what type of function might be suitable for f( ).

I'll use the Data Encryption Standard here, since there has been a lot of information published on this algorithm. DES was standardized in the late 1970s and has been broken by brute-force attack, trying all possible keys variables with specialized hardware, but the underlying algorithm is still secure. It is only the key length that has made it weak.

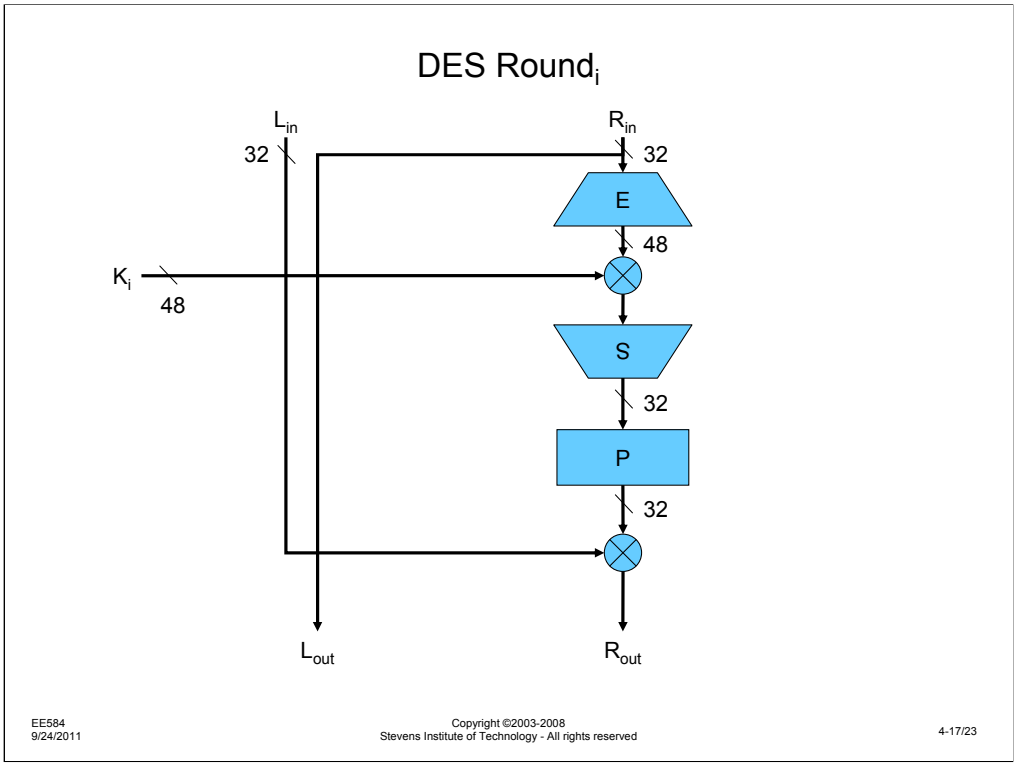
DES uses a 56 bit key variable to transform a 64 bit plaintext input to a 64 bit ciphertext output. Thus, DES can be thought of as a family of functions with 2^56 members that maps a 2^64 input space to a 2^64 output space.

On the next slide, we will look inside the DES block to examine the details of the algorithm.



There are a small number of basic components: the key schedule algorithm takes the 56 bit input key variable and creates a set of 16 48-bit "subkeys" that are used in the 16 "rounds" of the DES algorithm. The rounds are 16 repetitions of the basic functions that make the DES secure. Finally, there are permutations at the input and output of the DES algorithm.

Within the DES algorithm, the 64-bit input and output are actually treated as a 32-bit left half and a 32-bit right half. We will next examine what is inside each "round."



Each round of the DES is structurally the same. The 32-bit right-half input is passed unchanged as the left-half output. It is also used as input to the Expansion box. This E-box copies the 32-bit input to its output, but chooses 16 of the input for replication to create a total of 48 bits output. Between the permutation of the 32 bit lines, as well as the additional copies of the inputs, it is easy to see that the E-box mapping function is nonlinear.

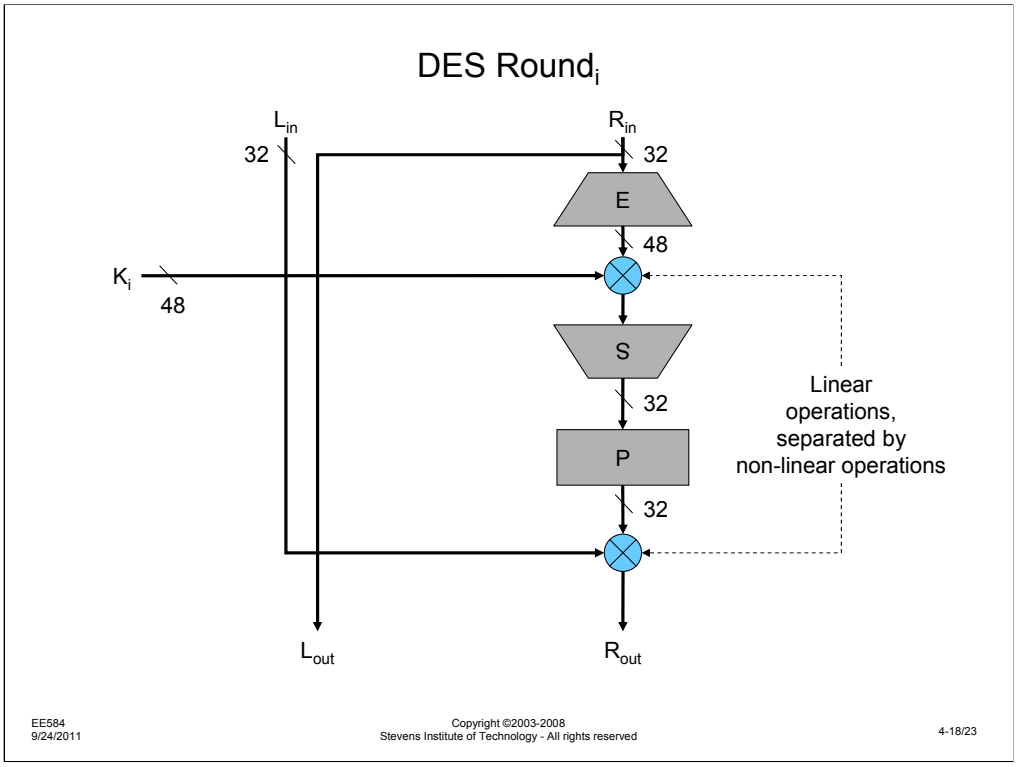
The 48-bit E-box output is XORed, bit-by-bit with the proper 48-bit subkey for this round of the DES. Each round uses a different subkey. Although the XOR function is a linear combination, its linearity cannot be easily exploited, since this linear operation is bracketed by the E-box and the following S-box.

The S-box is another nonlinear operation, this time, mapping a 48-bit input space to a 32-bit output space. The S-box function is public information, but the reason for the specific values in the S-box are secret. Most of the security of the DES comes from the complex mapping of the S-box.

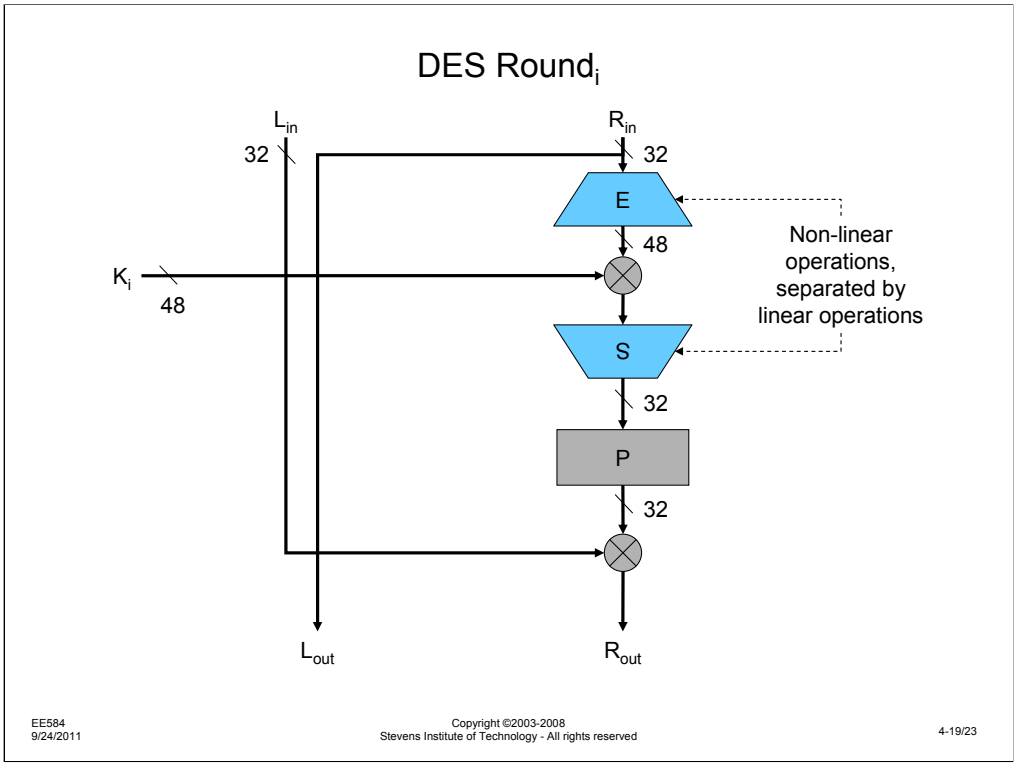
The 32-bit output of the S-box is permuted by the P-box, another nonlinear operation. In essence, the P-box routes 32 inputs individually to 32 outputs with a fixed but seemingly random interconnection.

Finally, the 32-bit output of the P-box is XORed with the 32-bit left-half input to the round, creating the right-half output.

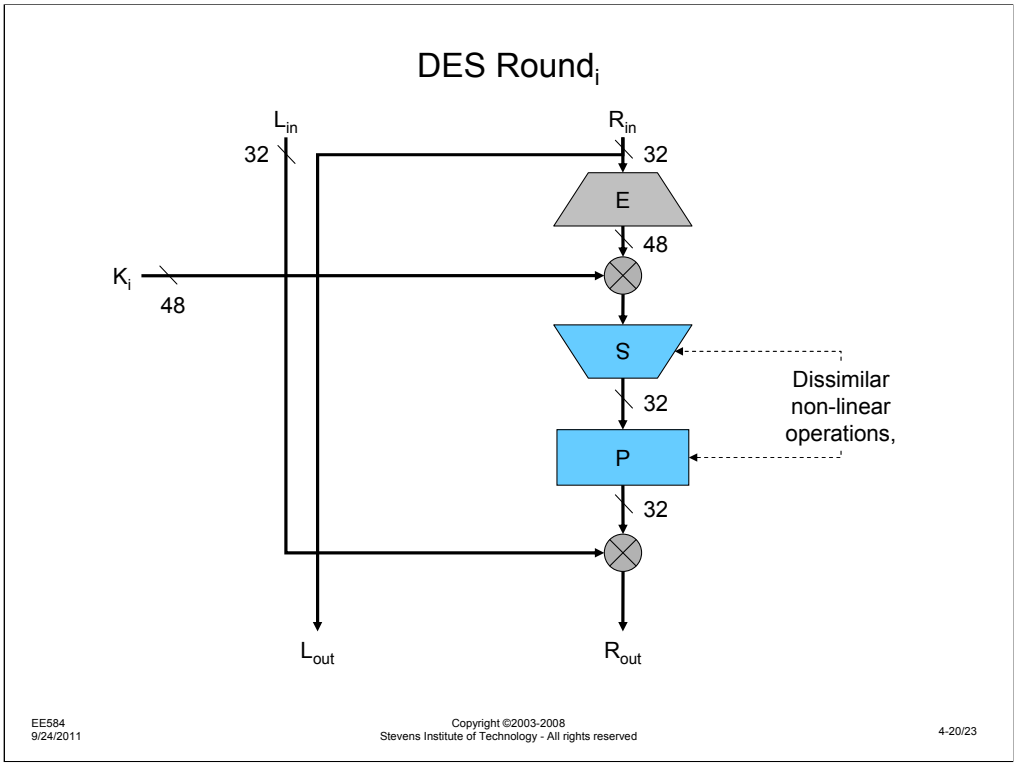
To sum up the operation of the DES round, there are multiple linear and nonlinear operations, with various pieces of data and key thoroughly intermixed with each other. While the particular operations of the algorithm are well defined and easy to replicate, the underlying structure and reason for various mappings are not public, making it very difficult to attack.



As shown above, the two linear operations in the DES, the XORs, are separated by nonlinear operations. If the linear operations were adjacent, it would be possible to replace them with one combined linear operation. Since the nonlinear operations separate them this is not possible.



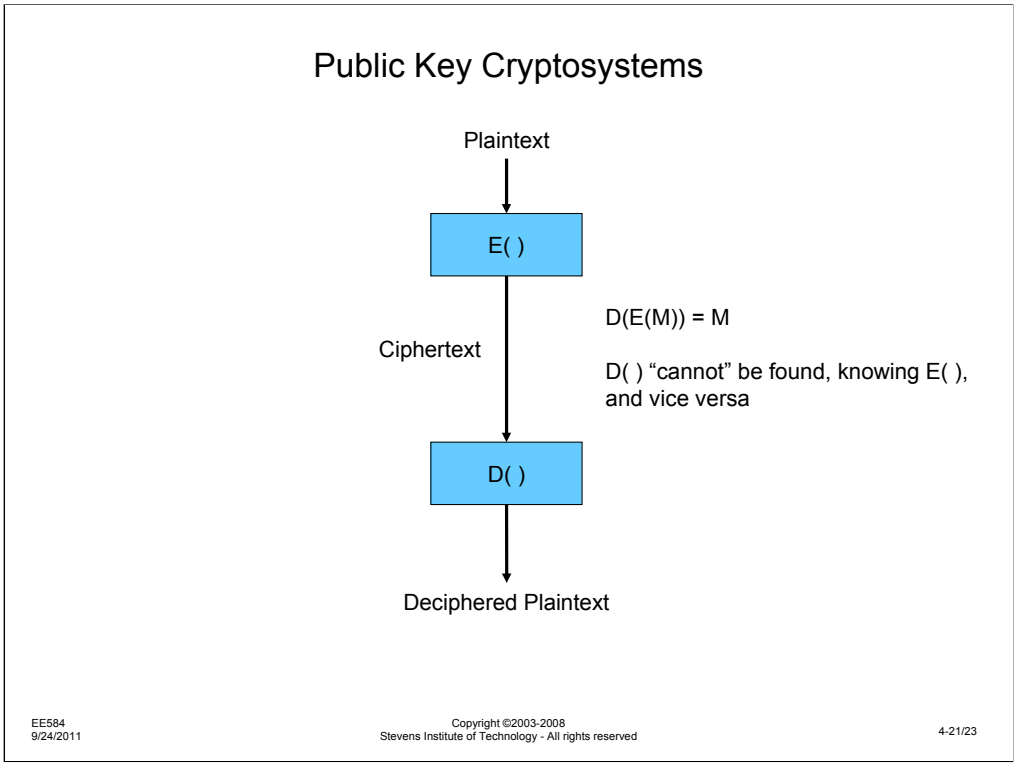
Likewise, the E box and S box transform a 32-bit number to a 48-bit number and back to a 32-bit number. If they were adjacent, they could be replaced by a single 32-bit to 32-bit mapping, simplifying the analysis. Since there is a linear operation in between, which is based on a secret 48-bit subkey for each round, the combining of the two boxes is not possible.



Finally, the S-box and P-box are both nonlinear operations. One is a compressive substitution, the other a permutation. The difference of these two operations makes analysis of the combined operations difficult.

Although the DES is no longer recommended for new applications, it retains its original design security. In fact, the underlying design of the DES is based on work by Claude Shannon, the person credited with creating the field of information theory in a classic Bell System Technical Journal paper in 1949. His less well known 1948 BSTJ paper, “The Theory of Secrecy Systems” first suggested the structure of the DES as a “product cipher.” By combining (forming the product of) two different ciphers, and by repeatedly applying one after the other, as done in the DES, a much stronger cipher is created than either underlying cipher would be by itself. The current standard for non-classified crypto systems is the Advanced Encryption Standard (AES). Essentially, the AES has the same architecture as DES, with a two-dimensional set of building blocks, as opposed to the one-dimensional set used in DES.

See http://www.esat.kuleuven.ac.be/~rijmen/rijndael/ and http://home.ecn.ab.ca/~jsavard/crypto/co040401.htm for a very good descriptions of the internals of “Rijndael,” the algorithm that has been standardized as AES.



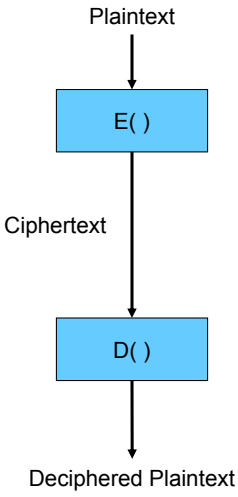
The encryption systems discussed to this point are known as "classical," "one-key," or "symmetrical" cryptosystems. What they have in common is the key variable used to encrypt a message is also used to decrypt the message. What is different about "public key," "two-key" or "asymmetrical" cryptosystems is that the process used to encrypt a message is separate from the process used to decrypt a message. It is possible to reveal one without the other.

Here's a physical analogy: I want to be able to allow each student to leave their homework outside my office in a secure manner. I want to be sure one student cannot access another student's work and I want a way to be sure that the work left could only have been left by a specific student or their agent. I will have N boxes with standard lock hasps on them, which I leave without locks on them. I obtain N different locks and keys, and give each student a lock, keeping the key for myself and noting which lock I gave them. Students can leave their homework in one of the boxes and lock it with their lock. No one else can unlock the lock, since only I have the key. When I open the lock, I know which of the N keys I used, so I know which student was responsible for leaving the work.

We can turn this analogy around to consider the other use of Public Key Cryptosystems (PKCs). Instead of giving the students the locks, I can give them the keys and keep the locks. Now, I can leave information on their grades in the boxes. Only the authorized student will be able to unlock the box to retrieve their grade.

Thus, depending on which PKC key is made public and which is kept secret, PKCs can be used to send secrets or sign messages.

Public Key Cryptosystems



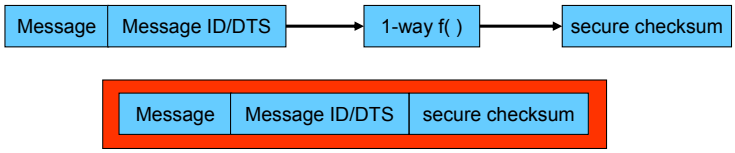
- D( ) and E( ) must be built on commutative functions:
  f(g(x)) = g(f(x))
- Multiplication and exponentiation work – are there others?
  These form bases for Rivest-Shamir-Adleman (RSA) and Diffie-Hellman PKCs
- The apparent security of PKCs come from difficulty of computing logarithms and factoring composite numbers in a finite field. ***Thought*** to be NP-Complete problems, Which ***might*** make them mathematically intractable
- E.g.,
  $E(M) = M^e$
  $D(C) = C^d$
  $D(E(M)) = (M^e)^d = M^{ed} = M^1$, if $d=e^{-1}$ in the field

Just like the physical lock analogy, the locking and unlocking functions are intimately related to each other. The analogy breaks down if you examine a key and see the height of the cuts, but unless you can open a lock and measure the height of the pins, you don't know how to match a key to it.

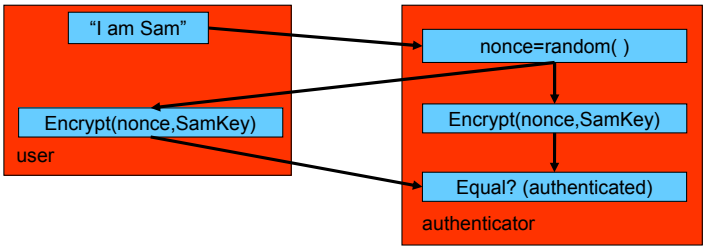
For PKCs, the encryption and decryption functions are generally based on the commutativity of multiplication. A*B = B*A, so they can be applied in either order. The related function of exponentiation is an alternative, one used in the Rivest-Shamir-Adleman algorithm, the other in the Diffie-Hellman algorithm. In either case, the apparent security of PKCs come from the property of multiplicative inverses in a finite field. While it is easy to perform corresponding operations in an infinite number field, finding the multiplicative inverse in a finite field is hard unless one knows how the field was generated.

Applications of cryptography to security

- Confidentiality – the most obvious application
- Integrity



- Non-repudiation
  - Same as integrity, but seal the message: with user ID and user-specific key
- Authentication Challenge-response



With this brief background, we can start to discuss how we might apply cryptography to provide various security services.

Confidentiality is the simplest – by properly encrypting a message, only the intended recipient can decrypt it.

Integrity is also straightforward. We can use cryptography to add a checksum to a message. If the message is modified in any way, the checksum will not be correct. But only someone who knows how the checksum is (cryptographically) generated can create a valid checksum. Thus, the originator and the intended recipient can generate and verify checksums, but the attacker cannot.

Likewise, non-repudiation services (digital signature) can be provided by sealing a message with a user ID and a user key.

Finally, cryptography is frequently used for user authentication with challenge-response systems. As shown, the user makes an identity claim, "I am Sam." The authentication server creates a random challenge, which it sends to the user and simultaneously encrypts with the claimed user's key. If the user is who they claim to be, they will know their key and can encrypt the challenge properly. They return the encrypted challenge to the authenticator, who compares it with the value they calculated. If they match, the user is who they claim to be with high probability.