Hash algorithms

EECE 457

Hash rules

0. “Easy” to compute hash(X)

1. Given Y, “hard” to find X with Y=hash(X)

2. Generally “hard” to find $X_1 \neq X_2$ with $hash(X_1)=hash(X_2)$

But what does “hard” mean?

Hash rules

- You can always solve Y=hash(X) using brute force, which can be really easy if you are brute-forcing over a tiny space

  (e.g., your password is an English word)

- “hard” means that we don’t know of any reasonable algorithm—for hashes, it means that nobody knows a better method than brute force

- Need a large output length, so that brute force can be computationally unfeasible

Applications (ctd)

- Password hashes

```
user:alice  password:hash(asdf,alice,salt)
```

- Tamper evidence (hash as checksum for file)
- Commitments

Commitment

1. Alice generates secret X

2. Alice sends Bob: Y=hash(X)

3. Time passes

4. Alice sends Bob: X

Alice convinces Bob that she knew X as far back as step 2, and has not changed it since step 2

Commitment

1. Alice generates random coin flip X

2. Alice→Bob: Y=hash(X)

3. Bob→Alice: guess for value of X

4. Alice→Bob: X

X must be large enough to prevent brute force, e.g. a coin flip combined with a long random string:
`FLIP=HEADS::2b00042f7481c7b056c4b410d28f33cf`

Message surrogates

1. Alice sends Bob: 30 page EULA X

2. Bob signs Y=hash(X)

“I hereby agree to the contract whose hash is 84da1f9a4a891ab7a5c873730d9e9a4e”

We will use cryptography to “sign” messages, and it’s far easier to sign short ones.

“Time” stamping, forcing order of messages

- Alice publishes message A
- Bob publishes message B,hash(A)
- Carol publishes message C,hash(B,hash(A))
- Dave publishes D,hash(C,hash(B,hash(A)))

Each document must have been created after the previous one.

Proof of work (hashcash)

- Alice sends Bob message A
- Bob finds a random nonce N where hash(A,N) ends with 12 zero bits
- Bob sends Alice N
- Alice computes hash(A,N) to verify Bob's work

Bob must find N through brute force search; Alice's verification requires a single quick step

The random oracle model

- A machine M contains a random bit generator and inexhaustible memory
- Given input X:
  - If M never saw X before, set hash(X) equal to a string of 128 random bits, and output this
  - If M saw X before, output the value previously generated

This satisfies the requirements of a secure hash function

Brute force

- Suppose there are 2^n hash outputs, and each one is equally likely to appear
- Given Y, every guess X has a 2^{-n} probability of outputting Y
- Expected number of trials until collision is $1/(2^{-n})$

You are "weakly collision-free" if your space of outputs is too big to brute-force!

Proof

- Event has probability p of occurring
- In repeated trials, probability that event occurs in trial K is $p(1-p)^{K-1}$
- Expected number of trials = $\sum_K K \cdot p(1-p)^{K-1}$
  $= p \sum_K K \cdot (1-p)^{K-1}$
- Trick: $Kx^{K-1} = d/dx\ x^K$
  $\sum Kx^{K-1} = d/dx \sum x^K$

Brute force

- Suppose there are 2^n hash outputs, and each one is equally likely to appear
- Choose X_1, X_2, … X_m, until two inputs match
- Expected number of attempts about ($2^{-n/2}$)

This is the square root of the size of your space of outputs!

Proof

- Sample uniformly from a set of M outcomes

- If the first K outcomes are unique, the next outcome is a collision with probability K/M

- The probability of no collision with K+1 samples:

  $Pr = (1-0/M)(1-1/M)(1-2/M)\ldots(1-K/M)$

- With M much larger than K, $(1+K/M) \sim e^{K/M}$, and

  $Pr = (e^{-0/M})(e^{-1/M})\ldots(e^{-K/M}) = e^{-(1+2+\ldots+K)/M} = e^{-K(K+1)/2M}$

Or

- $(1-a/M)(1-(K-a)/M) <= (1-K/2M)(1-K/2M)$

  $Pr = (1-1/M)(1-2/M)\ldots(1-K/M) <= (1-K/2M)^K$

- $e^x = \lim(1-x/N)^N$

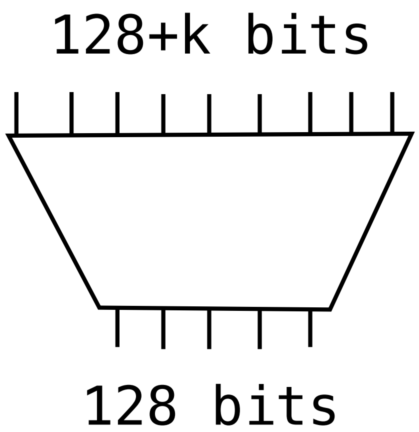
- $Pr <= (1-K/2M)^K = (1-(K^2/2M)/K)^K \sim e^{-(KK/2M)}$

But how do hash algorithms work?

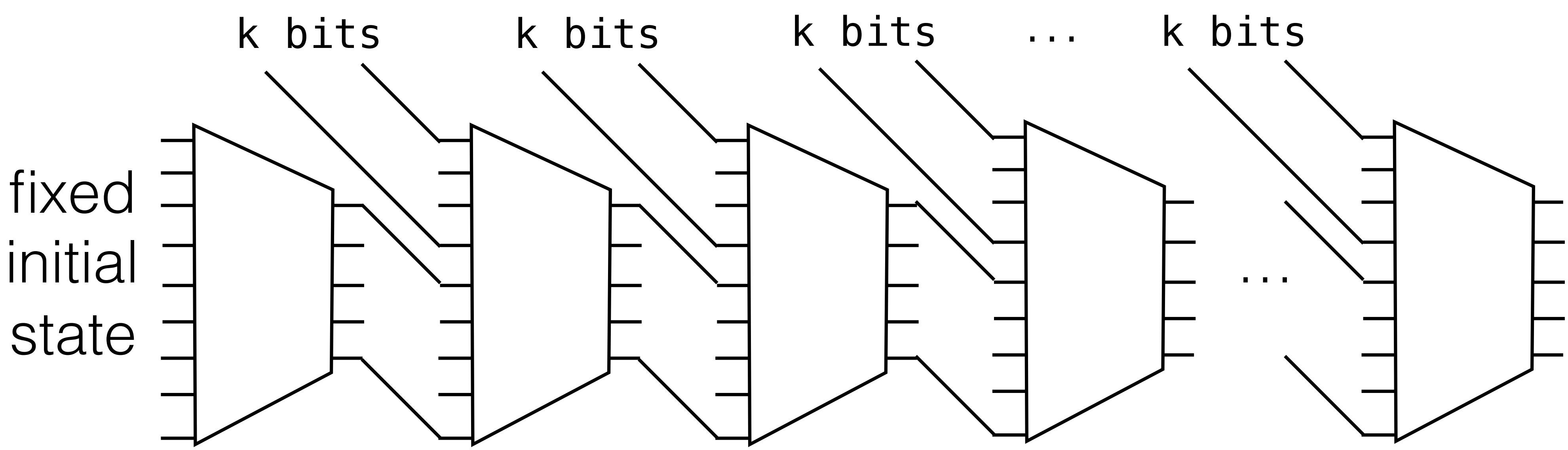
The Merkle Damgård construction

- Let M be a finite state machine with a very large state (hundreds of bits) that accepts m input bits per clock
- Given input X:
  - Partition X into m-bit blocks
  - Feed all input blocks to M
  - Use the state of M as the “hash”

Example

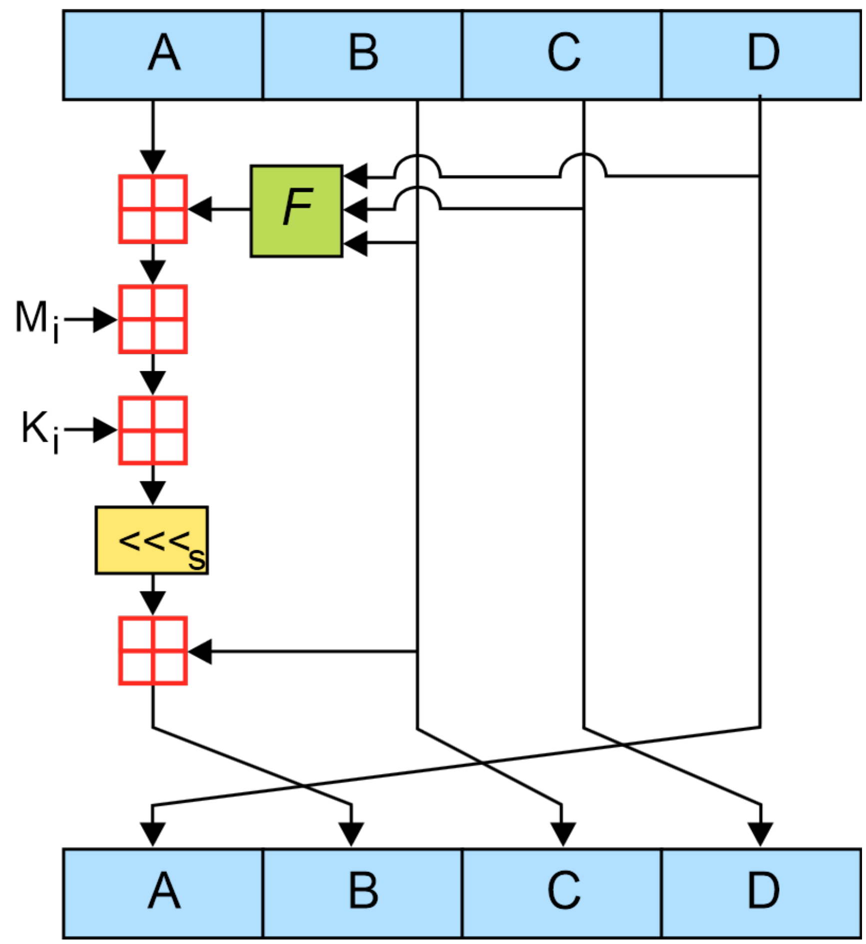


Garbled combinational circuit — represents the computation of a new FSM state from the previous state plus k bits of input (from string to be hashed)



Example: MD5

- 128+32 bit input, 128-bit state output
- A 512-bit block is blended into the state by breaking it into 16 32-bit inputs M_i
- The block is folded in 4 times, for a total of 64 stages
- Values Ki are constants, a different one per round; shift value s is different each round



Example: MD5

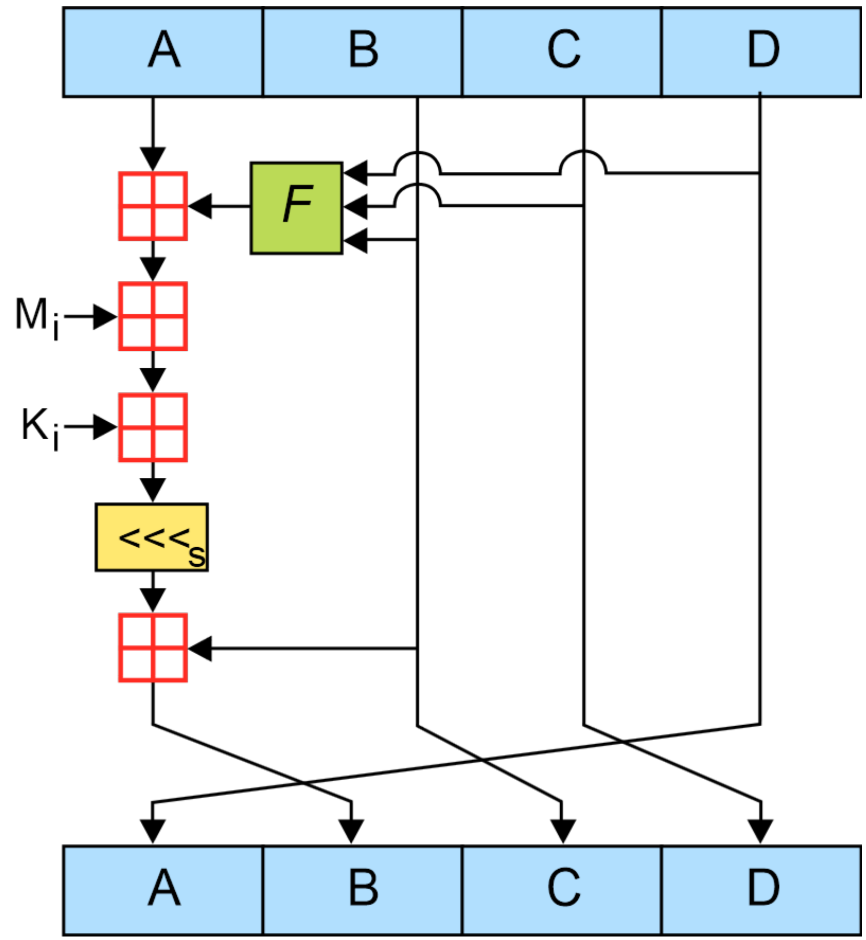
- Function F varies in each of the 4 passes

  pass 1, rounds 0-15:
  `(B and C) or ((not B) and D)`

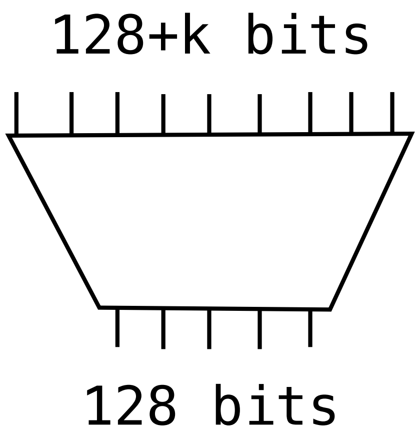
  pass 2, rounds 16-31:
  `(D and B) or ((not D) and C)`

  pass 3, rounds 32-47:
  `B xor C xor D`

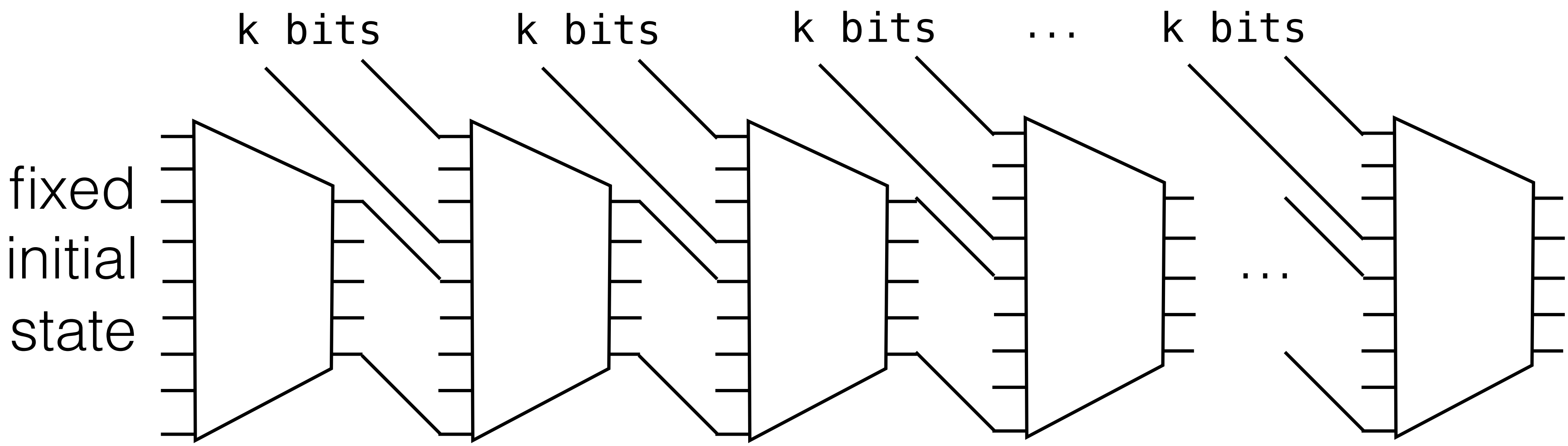
  pass 4, rounds 48-63:
  `C xor (B or (not D))`



Example



Garbled combinational circuit — represents the computation of a new FSM state from the previous state plus k bits of input (from string to be hashed)



The Merkle Damgård construction

- Why?
  - If we come up with a good way to hash a fixed-length input, we can use this trick to have it accept arbitrary-length inputs
  - Result can be sensitive to every bit of input
  - Downside: if we find a collision hash(X)=hash(Z), then hash(XY)=hash(ZY)!

Pseudo random generators

Pseudo-random generators

- Used to generate long string of “random-ish” data deterministically, from a small seed
- Function PRNG(seed,k) = bit_k
- Requirements:

  0: PRNG is easy to compute
  1: “hard” to determine seed from output
  2: “hard” to predict subsequent bits from past bits

Pseudo-random generators

- Simple methods using hashes (but unnecessarily compute-intensive):
  - bit_k = hash(seed,k)&0x01
  - or: S_0=seed, S_k=hash(S_{k-1}), bit_k=S_k&0x01
- Hashes can be used for PRNGs, if a hash is all you have

Simple generators

- These are NOT CRYPTOGRAPHICALLY SECURE!
  - Linear congruential generators
  - Linear feedback shift registers (LFSRs)
  - Combinations of simple, “unsafe” generators may produce secure ones

Linear congruential generators

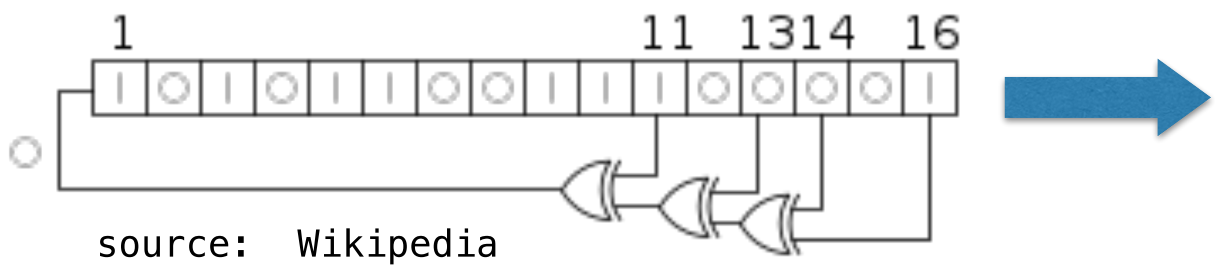
- X_0 = initial state; $X_{k+1} = (aX_k+c)$ modulo m

- Example: byte $B_{k+1} = (5B_k+1)$

- Advantage: with proper choice of a, c and m, the generator has a maximal period (m).

- Disadvantage: easy for an adversary to estimate state from output values

Linear feedback shift registers



source: Wikipedia

outgoing bit is generator output

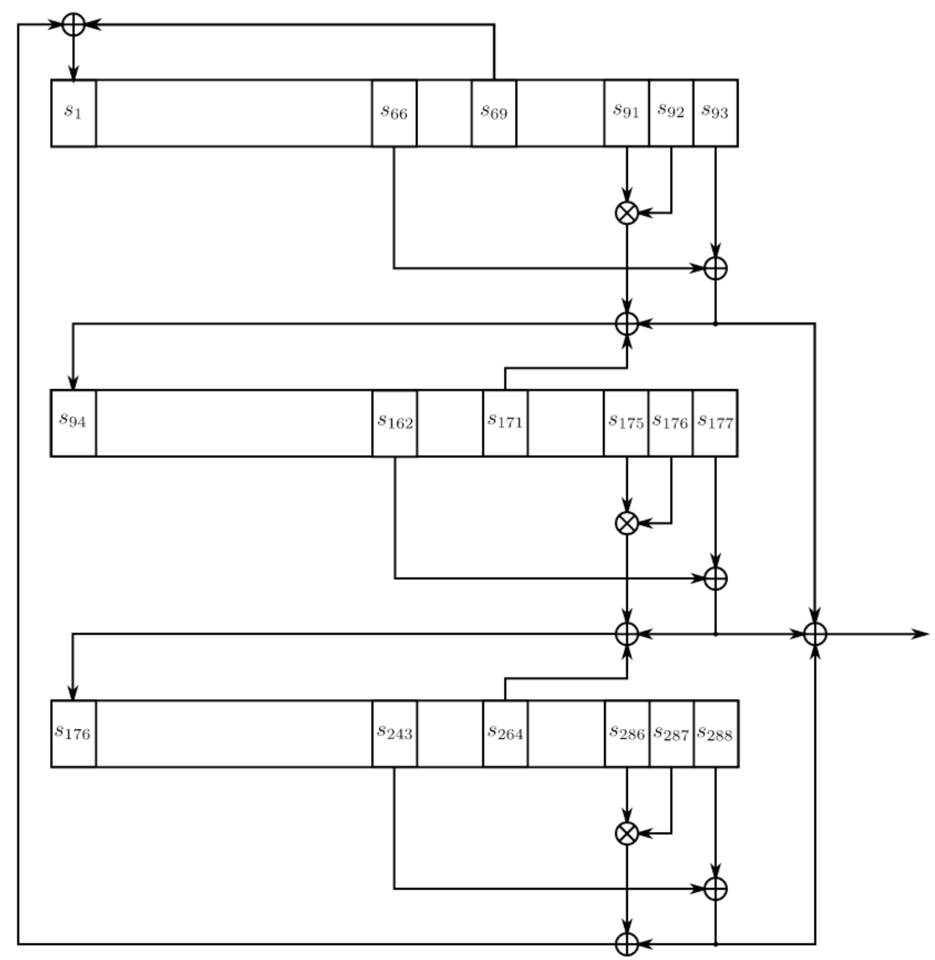
- Select bits are tapped and XORed to produce input for shift register, output bits produce PRNG stream
- Advantage: maximal period 2^n-1 if taps are chosen correctly
- Disadvantage: burps out secret seed value in n clocks

Combinations of LFSRs

- Some hopefully secure PRNGs such as Trivium (right) attempt to combine multiple LFSRs nonlinearly

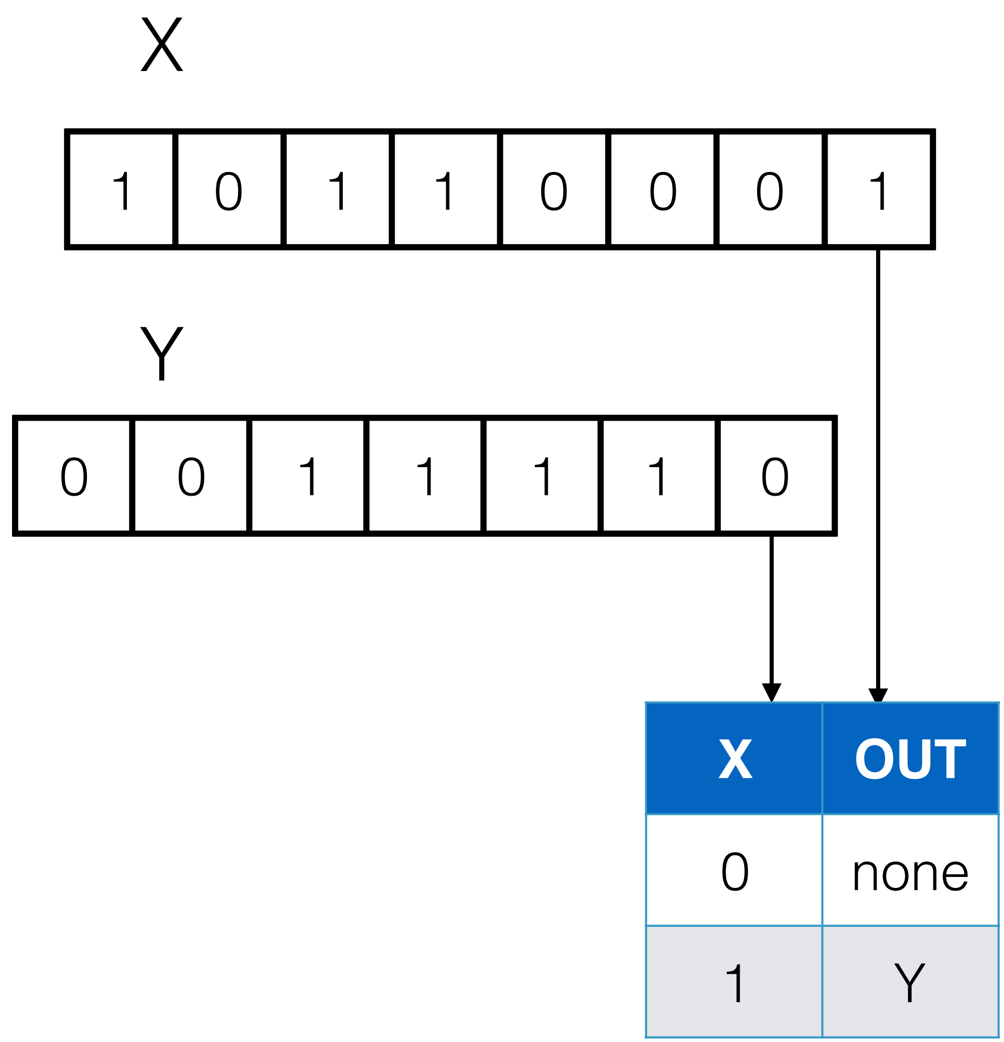
- Simple example: shrinking generator

- Such designs are desirable because they would be computationally very simple, easy to realize in hardware.



Combinations of LFSRs

- Some hopefully secure PRNGs such as Trivium attempt to combine multiple LFSRs nonlinearly
- Simple example: shrinking generator (right)
- Such designs are desirable because they would be computationally very simple, easy to realize in hardware.



More complicated methods

- Salsa20: uses 16 32-bit registers, with add-xor-shift operations

```
x[ 4] ⊕= (x[ 0] ⊞ x[12])<<<7;    x[ 9] ⊕= (x[ 5] ⊞ x[ 1])<<<7;
x[14] ⊕= (x[10] ⊞ x[ 6])<<<7;    x[ 3] ⊕= (x[15] ⊞ x[11])<<<7;
x[ 8] ⊕= (x[ 4] ⊞ x[ 0])<<<9;    x[13] ⊕= (x[ 9] ⊞ x[ 5])<<<9;
x[ 2] ⊕= (x[14] ⊞ x[10])<<<9;    x[ 7] ⊕= (x[ 3] ⊞ x[15])<<<9;
x[12] ⊕= (x[ 8] ⊞ x[ 4])<<<13;   x[ 1] ⊕= (x[13] ⊞ x[ 9])<<<13;
x[ 6] ⊕= (x[ 2] ⊞ x[14])<<<13;   x[11] ⊕= (x[ 7] ⊞ x[ 3])<<<13;
x[ 0] ⊕= (x[12] ⊞ x[ 8])<<<18;   x[ 5] ⊕= (x[ 1] ⊞ x[13])<<<18;
x[10] ⊕= (x[ 6] ⊞ x[ 2])<<<18;   x[15] ⊕= (x[11] ⊞ x[ 7])<<<18;

x[ 1] ⊕= (x[ 0] ⊞ x[ 3])<<<7;    x[ 6] ⊕= (x[ 5] ⊞ x[ 4])<<<7;
x[11] ⊕= (x[10] ⊞ x[ 9])<<<7;    x[12] ⊕= (x[15] ⊞ x[14])<<<7;
x[ 2] ⊕= (x[ 1] ⊞ x[ 0])<<<9;    x[ 7] ⊕= (x[ 6] ⊞ x[ 5])<<<9;
x[ 8] ⊕= (x[11] ⊞ x[10])<<<9;    x[13] ⊕= (x[12] ⊞ x[15])<<<9;
x[ 3] ⊕= (x[ 2] ⊞ x[ 1])<<<13;   x[ 4] ⊕= (x[ 7] ⊞ x[ 6])<<<13;
x[ 9] ⊕= (x[ 8] ⊞ x[11])<<<13;   x[14] ⊕= (x[13] ⊞ x[12])<<<13;
x[ 0] ⊕= (x[ 3] ⊞ x[ 2])<<<18;   x[ 5] ⊕= (x[ 4] ⊞ x[ 7])<<<18;
x[10] ⊕= (x[ 9] ⊞ x[ 8])<<<18;   x[15] ⊕= (x[14] ⊞ x[13])<<<18;
```

More complicated methods

- Salsa20 setup: block of 16 words z[0]..z[15]:
  Z[1,2,3,4,11,12,13,14] = 256 bit key (seed)
  Z[0,5,10,15] = Salsa constants
  Z[6,7] = extra nonce (can be part of seed)
  Z[8,9] = block number
- Copy words into registers x[0]..x[15]
- Perform 20 rounds (right)
- Add z[i]+=x[i]
- Output z[i]

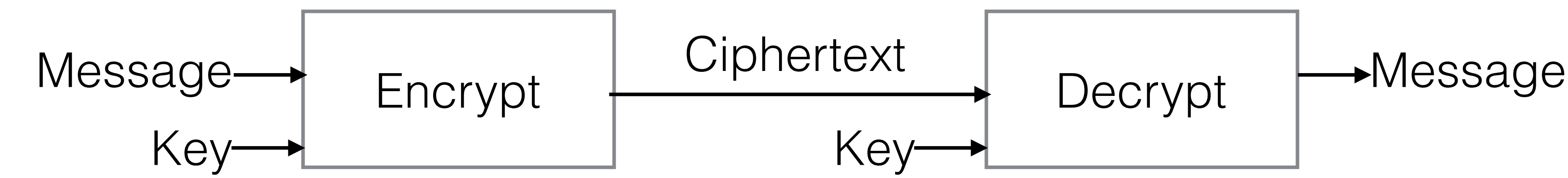
```
x[ 4] ⊕= (x[ 0] ⊞ x[12])<<<7;    x[ 9] ⊕= (x[ 5] ⊞ x[ 1])<<<7;
x[14] ⊕= (x[10] ⊞ x[ 6])<<<7;    x[ 3] ⊕= (x[15] ⊞ x[11])<<<7;
x[ 8] ⊕= (x[ 4] ⊞ x[ 0])<<<9;    x[13] ⊕= (x[ 9] ⊞ x[ 5])<<<9;
x[ 2] ⊕= (x[14] ⊞ x[10])<<<9;    x[ 7] ⊕= (x[ 3] ⊞ x[15])<<<9;
x[12] ⊕= (x[ 8] ⊞ x[ 4])<<<13;   x[ 1] ⊕= (x[13] ⊞ x[ 9])<<<13;
x[ 6] ⊕= (x[ 2] ⊞ x[14])<<<13;   x[11] ⊕= (x[ 7] ⊞ x[ 3])<<<13;
x[ 0] ⊕= (x[12] ⊞ x[ 8])<<<18;   x[ 5] ⊕= (x[ 1] ⊞ x[13])<<<18;
x[10] ⊕= (x[ 6] ⊞ x[ 2])<<<18;   x[15] ⊕= (x[11] ⊞ x[ 7])<<<18;

x[ 1] ⊕= (x[ 0] ⊞ x[ 3])<<<7;    x[ 6] ⊕= (x[ 5] ⊞ x[ 4])<<<7;
x[11] ⊕= (x[10] ⊞ x[ 9])<<<7;    x[12] ⊕= (x[15] ⊞ x[14])<<<7;
x[ 2] ⊕= (x[ 1] ⊞ x[ 0])<<<9;    x[ 7] ⊕= (x[ 6] ⊞ x[ 5])<<<9;
x[ 8] ⊕= (x[11] ⊞ x[10])<<<9;    x[13] ⊕= (x[12] ⊞ x[15])<<<9;
x[ 3] ⊕= (x[ 2] ⊞ x[ 1])<<<13;   x[ 4] ⊕= (x[ 7] ⊞ x[ 6])<<<13;
x[ 9] ⊕= (x[ 8] ⊞ x[11])<<<13;   x[14] ⊕= (x[13] ⊞ x[12])<<<13;
x[ 0] ⊕= (x[ 3] ⊞ x[ 2])<<<18;   x[ 5] ⊕= (x[ 4] ⊞ x[ 7])<<<18;
x[10] ⊕= (x[ 9] ⊞ x[ 8])<<<18;   x[15] ⊕= (x[14] ⊞ x[13])<<<18;
```

Why?

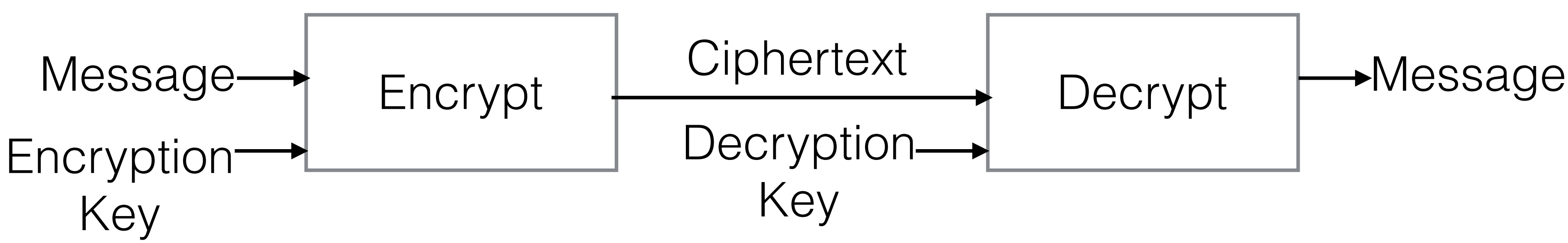
- Requires a constant number of register operations per block, only uses registers for rounds.
- Also allows user to “jump” to any 512-byte block in the PRNG stream

Encryption

- Symmetric

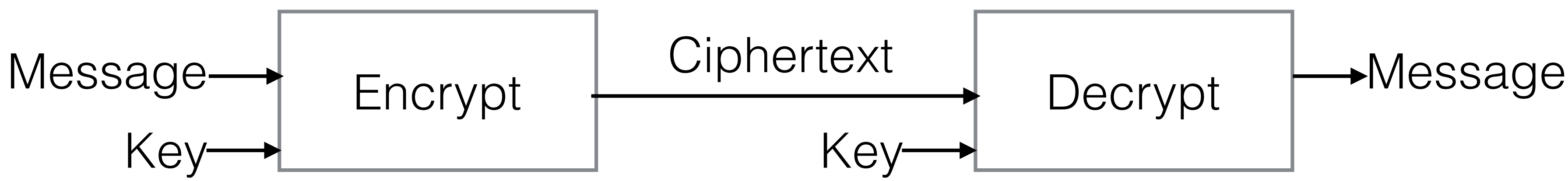


- Asymmetric (“public key”)



Encryption

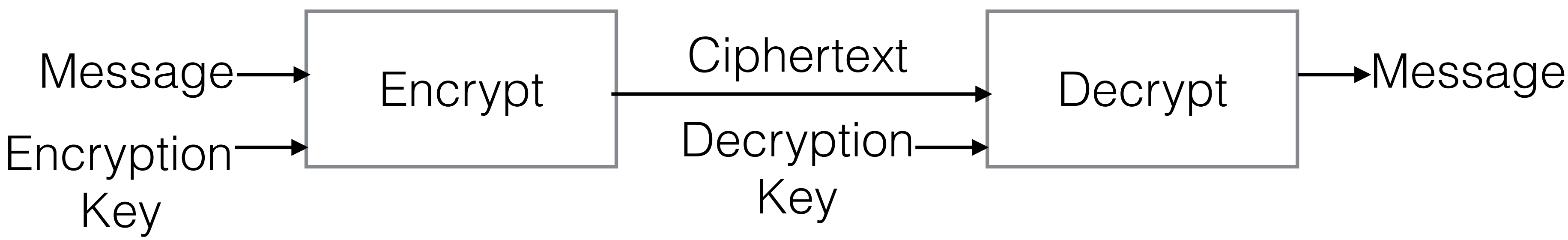
- Symmetric



- Easy to compute Encrypt(M,K), Decrypt(C,K)
- Hard to determine Encrypt(M,K) or Decrypt(C,K) given M or C, but without knowledge of K
- Hard to determine K given both C and M (why?)

Encryption

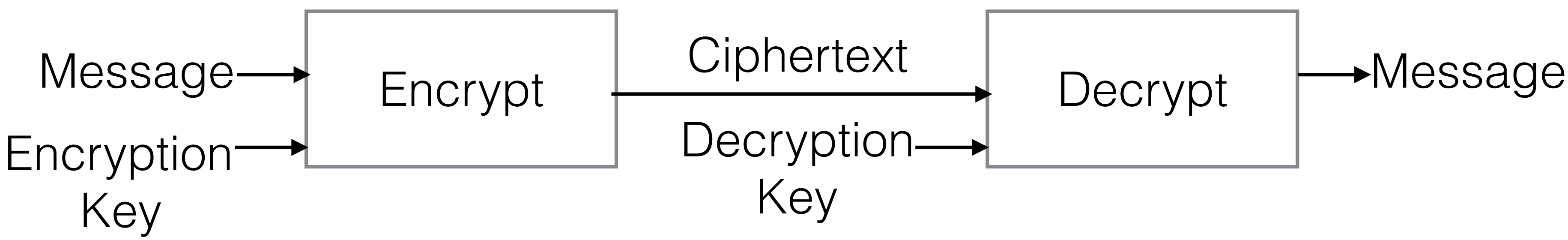
- Asymmetric ("public key")



- Easy to compute Encrypt(M,K_E), Decrypt(C,K_D)
- Hard to compute Decrypt(C,K_D) without K_D;
  Hard to compute Encrypt(M,K_E) without K_E;
- Thus, hard to derive K_D from K_E, and vice versa

Encryption

- Asymmetric ("public key")



- Allows you to publish an encryption key, keeping a decryption key secret.

- Anyone can send you an encrypted message, without arranging a shared secret key in advance

Key management

- For everyone to communicate with symmetric cryptography, you need to manage a key for every pair of users, or develop a key management system.
- With asymmetric cryptography, you still need a way for Alice to look up Bob's public key, but:
  - Alice doesn't need a key of her own
  - The secret key is stored by a single party, and is easier to keep secure

Symmetric vs asymmetric

- Symmetric algorithms tend to be much faster
- Asymmetric algorithms are possible because of large instances of difficult computational problems—and thus require large keys, more serious computation.
- We typically use asymmetric encryption just to send a key for symmetric encryption!

  Alice→Bob: EncryptAsymmetric($K_{SESSION}$,K_E),
  EncryptSymmetric(M,$K_{SESSION}$)

Fun encryption tricks

1. Alice: Message M, public key K_E private key K_D
2. Compute H = hash(M) (message surrogate)
3. Compute S = Decrypt(H,K_D)
4. Wait, what?
5. Alice→Bob: <M,S>

Fun encryption tricks

1. Alice: Stores public key K_E with remote host
2. Alice sends login request to remote host Bob
3. Bob→Alice: Random string R
4. Alice→Bob: S=Decrypt(R,K_D)
5. Bob verifies that Encrypt(S,K_E) = challenge R

Equivalence of primitives

- Can implement PRNG using a hash
- Can implement symmetric encryption with a PRNG
- Can implement a hash using symmetric encryption

Kerckhoffs’s Criterion

- A system must be designed under the assumption that an adversary will know everything about it, save for secret keys.
- Secrecy of keys (seeds, passwords, etc) provide security of overall system
- This does not mean everything else must or should be made public, but that it should remain secure if that should happen

Keys

- What is a key?

  A brief, portable parameter that can be generated randomly from a well-defined set/distribution, upon whose secrecy the security of a system depends in a consistent manner.

- Portability and ease of generation are important because *keys can go bad, and must be replaced*.

Keys

- What is a key?

  A brief, portable parameter that can be generated randomly from a well-defined set/distribution, upon whose secrecy the security of a system depends in a consistent manner.

- Random generation is important, so that the "secrecy" of a key can be quantified. It's not a key if you can't analyze the difficulty of guessing it!

Kerckhoffs's Criterion

- A system must be designed under the assumption that an adversary will know everything about it, save for secret keys.

- A violation of Kerckhoff's criterion is called an *obscurity* tactic, or "security through obscurity"

- Examples: hiding a spare key by a door, placing critical data in a "hidden" directory, hard-coding a password in an executable file

Reasons for Kerckhoffs's Criterion

- It works! Systems that violate this principle are often broken.
- Following this rule gives you quantifiable secrecy
- Non-key components (e.g. algorithms) are hard to swiftly regenerate, or randomly sample with consistent levels of security.
- Allowing an architecture to be made public allows peer review
- It is hard to control who knows your architecture.

Cryptographic protocols

- Protocols are typically short multi-step processes performed by multiple parties (“principals”), using a communications channel.
- These steps use cryptographic primitives, wrapping them up into a larger application
- Goal is to achieve some guaranteed security property (e.g. that Alice has authorization to log in)
- Difficulty: nobody trusts anyone, and people may spy, cheat, and misbehave to achieve sinister goals

Principals

- Alice, Bob, Carol, Dave

  Primary participants in protocol

- Eve

  Unseen eavesdropper, assumed to exist.

- All principals, including those we don’t expressly mention, are potential adversaries.

The Dolev-Yao model

- All communication takes place over a public, modifiable channel, with no reliable indicator of a message's date, direction, or sender
- Analogy: public bulletin board
- Eve can read, modify, and forge any message.
- Sometimes described as "the adversary carries the message."

Example protocol

1. Alice$\rightarrow$Bob: KA_E

2. Bob$\rightarrow$Alice: $C = Encrypt(K_{SESSION}, KA_E)$

3. Alice $\Leftrightarrow$ Bob: $Encrypt(M, K_{SESSION})$

Example protocol

1. Alice→Bob: KA_E
    1. Eve: replace KA_E with KE_E
2. Bob→Alice: $C=\mathrm{Encrypt}(K_{SESSION},KE_E)$
    1. Eve: replace with $\mathrm{Encrypt}(K_{SESSION},KA_E)$
3. Alice ⇔ Bob ⇔ Eve: $\mathrm{Encrypt}(M,K_{SESSION})$

Standard Protocol Notation

1. Alice: [stuff] — Alice has or computes [stuff]
2. Bob$\rightarrow$Alice: [stuff] — Bob sends Alice [stuff]
3. Alice $\Leftrightarrow$ Bob: [stuff] — Alice and bob share [stuff]
4. $\{M,N\}_K$ — "M,N" encrypted with K
   Also use $\text{Encrypt}_K(M,N)$ or Encrypt(M,N,K)

Example protocol

1. Alice→Bob: $\{ K \}_{EB}$

2. Bob→Alice: $\{ K \}_{EA}$

K is a random session key, EB and EA the public encryption keys of Bob and Alice

Confirms to Alice that Bob has decrypted her message and fashioned a response — does not tell Bob that he is speaking with Alice

A simple replay attack

1. Alice→Bob: $\{ K \}_{EB}$ ←Eve records this
2. Bob→Alice: $\{ K \}_{EA}$
3. Alice→Bob: $\{ M \}_{K}$ ←Eve records this

1. Eve(Alice)→Bob: $\{ K \}_{EB}$
2. Bob→Alice: $\{ K \}_{EA}$
3. Eve(Alice)→Bob: $\{ M \}_{K}$

A simple replay attack

1. Alice→Bob: $\{ K \}_{EB}$ ←Eve records this
2. Bob→Alice: $\{ K \}_{EA}$
3. Alice→Bob: $\{ M \}_{K}$

1. Eve→Bob: $\{ K \}_{EB}$
2. Bob→Eve: $\{ K \}_{EE}$

Possible fixes:

1. Alice→Bob: $\{$ Alice, K, Date $\}_{EB}$

2. Bob→Alice: $\{$ Bob, K, Date $\}_{EA}$

We might as well expressly designate sender/receiver information in the encrypted packet

A datestamp could be used to prevent a message from being replayed far in the past. But then we have to trust in clocks, deal with clock differences, choose a cutoff time, etc.

Possible fixes:

1. Alice→Bob: $\{$ Alice, K $\}_{EB}$

2. Bob→Alice: $\{$ Bob, K, N $\}_{EA}$

3. Alice→Bob: $\{$ Alice, N $\}_{K}$

Let N be a random nonce. A third step demonstrates to Bob (without a date stamp) that the packet from Alice is *fresh.*

Further complications

1. Alice→Bob: $\{ \text{Alice}, \boldsymbol{\{ K \}_{EB}} \}_{EB}$
2. Bob→Alice: $\{ \text{Bob}, K, N \}_{EA}$
3. Alice→Bob: $\{ \text{Alice}, N \}_{K}$

By the way, does it help to add an extra encryption layer? Even if it makes no difference, does it hurt at all, security-wise?

An oracle attack

1. Alice→Bob: { Alice, { K }$_{EB}$ }$_{EB}$ ←Eve records this

1. Eve→Bob: { Eve, **{ Alice, { K }$_{EB}$ }$_{EB}$** }$_{EB}$
2. Bob→Eve: { Bob, "Alice, { K }$_{EB}$", N }$_{EE}$

1. Eve→Bob: { Eve, **{ K }$_{EB}$** }$_{EB}$
2. Bob→Eve: { Bob, K, N }$_{EE}$

Some general types of attack

- Replay attack: record and resend a message from another party, to produce an unusual effect

- Reflection attack: a message from Alice is sent back to Alice, and mistaken for a message from someone else

- MITM attack: someone forwards messages and key datagrams, and is able to modify them to read traffic or later data

- Oracle attack: tricking a principal into decrypting a message or performing some task for your benefit

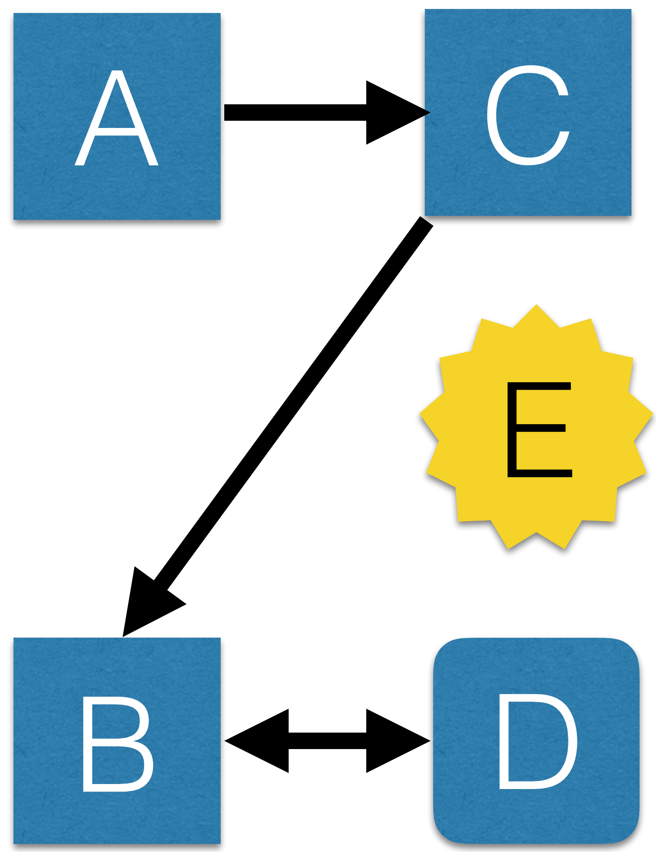
Weirder protocols

Secret splitting

- Want to encrypt a file and distribute to N people
- Goal: any subset of K people have the information needed to decrypt the file
    - Any subset of <K people do not have the information needed to decrypt the file
- Applications: fail-safe security polices

Authentication

- Alice: user wants to perform financial transaction
- Bob: Bank wants to authenticate Alice
- Carol: Clerk at POS terminal, initiates connection to Bob
- Database: information needed to authenticate Alice
- Eve: the eavesdropper



Simple authentication

1. Alice→Bob: password

2. Bob: checks database for hash of password.

- Bob and Carol both have information they need to impersonate Alice

- Eve can eavesdrop connection

Simple authentication

1. Alice→Bob: Alice

2. Bob→Alice: $\{ N \}_{EA}$

3. Alice→Bob: hash[N]

- Neither Carol nor Eve can use information to impersonate Alice later

- Bob's database only needs to contain E_A

Zero-knowledge protocols

- Can you prove you know a secret without revealing it, even partially?
- In previous protocol, Alice must decrypt something that Bob can choose. Technically she provides information Bob didn't previously have
  - Bob could send Alice an encrypted message {M}that he can't read, and get back hash[M]
- A zero-knowledge protocol provably conveys no information while demonstrating that I know the secret