DIMACS Security & Cryptography Crash Course – Day 3

Resiliency to Corruptions and Key Exposures

Prof. Amir Herzberg

Computer Science Department, Bar Ilan University

http://amir.herzberg.name

Outline

- Motivation
- Using Insecure Servers
- Private key exposure
- Certificates and revocation
- Shared key refresh
- Forward security
- Resiliency via Redundancy
- Secret Sharing
- Distributed PKC
- Distributed El-Gamal
- Proactive security
- Proactive secret sharing
- Proactive PKC, signatures
- Applying proactive security
- Conclusions

Why Resiliency?

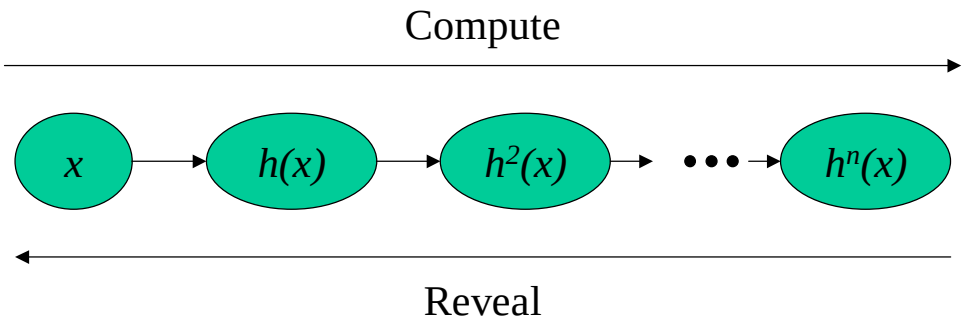
- Existing operating systems have weak security
  - By design
  - By typical implementations
  - More on this in follow-up courses
- Many (or most) computer crimes are by insiders
  - With access/control over computer/OS/Application
- Corruptions to data, key exposures possible
- Questions:
  - How to avoid / minimize damages?
  - How to recover?

Secure Service from Insecure Servers

- Can we provide secure services using insecure servers?
- Idea: insecure ≠corrupted!
- Insecure servers: servers that *may* be corrupted
- Provide secure services…
  - If server is not corrupted *at time of service*
  - By multiple servers – assuming not *all* are corrupted
  - Assume *enough* are *not* corrupted (One? Majority? 2/3?)
  - By multiple servers – assuming *enough are Ok at time of service*
- Example: Unix and S/Key Password Schemes…

Recall: Login to Insecure Server

- Unix Password Scheme
  - Unix passwords file is public (or `less secret`)
  - Keep only hash of passwords
  - Password file contains $<salt, h_{salt}(password)>$
  - Random `salt` added to prevent dictionary attacks
  - Hide passwords (or salt) for further protection (how?)
- S/Key Login: Hash Chain (One-time Passwords)
  - Protects against exposing server password file
  - And: eavesdropping on login communication

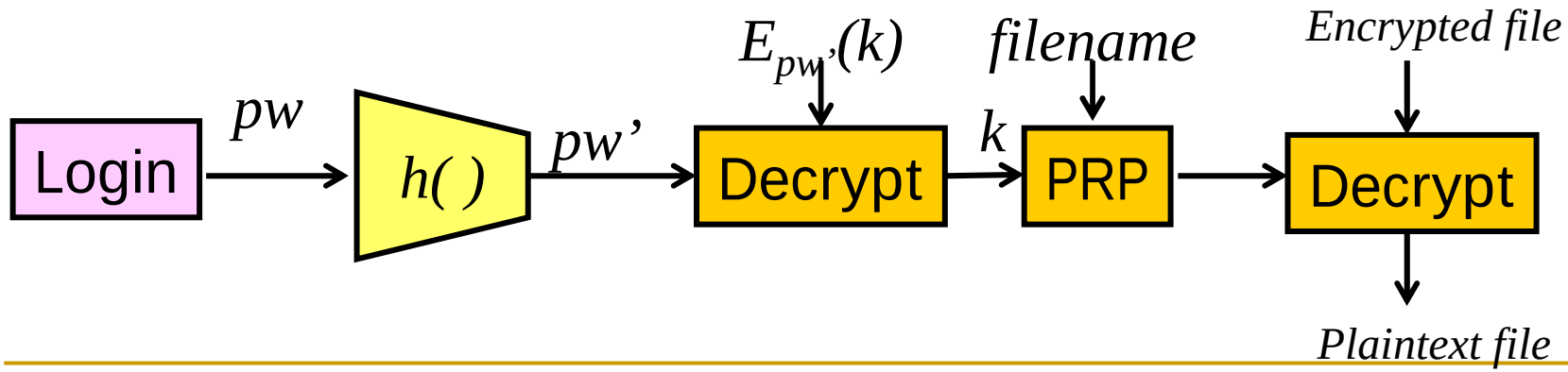


Protecting User Data on Insecure Server

- Encrypt using user's password; Decrypt during session
- Password (key) not kept in server (as before)
  - User provides *pw;* use as key in server $pw'=h(pw)$
  - Why not use directly?
- Decrypt master key k using pw' as key, i.e. $k=D_{pw'}(E_{pw'}(k))$
  - To allow changing of pw (only re-encrypt $E_{pw}(k)$)
  - To allow changing of k
- Encrypt each file by its own key, encrypted by master key
  - Why not use same key for all files?
  - Method 1: attach encrypted key to file
  - Method 2: Derive key from file-name: $k_{file}=PRP_k(filename)$
  - Advantages ?
- *Let's summarize this process…*

Protecting User Data on Insecure Server

- Encrypt using user's password; Decrypt during session
- User's password (*pw*) not kept in server (as before)
- Derive pw-key from password: $pw'=h(pw)$
- Decrypt master key k using pw' as key, i.e. $k=D_{pw'}(E_{pw'}(k))$
- Encrypt/Decrypt *filename* using: $k_{file}=PRP_k(filename)$
- *Problem? (Hint: user-chosen passwords…)*



Dictionary Attacks on User Data

- Problem: user passwords pw are often from dictionary
- Attacker can test against all $pw \in Dictionary$
  - For every guess u, compute $pw'=h(pw)$
  - Decrypt master key k, i.e. $k=D_{pw'}(E_{pw'}(k))$
  - Decrypt *filename* using: $k_{file}=PRP_k(filename)$
  - Validate guess of user password upw by viewing plaintext
- Countermeasures:
  - System-only access to encrypted passwords and files
    - Identify guessing attacks
  - Standard salt technique: $pw'=h(pw,salt)$
  - Use very slow h and/or decryption of master key $k=D_{pw}(E_{pw}(k))$

Exposure of Secret Keys

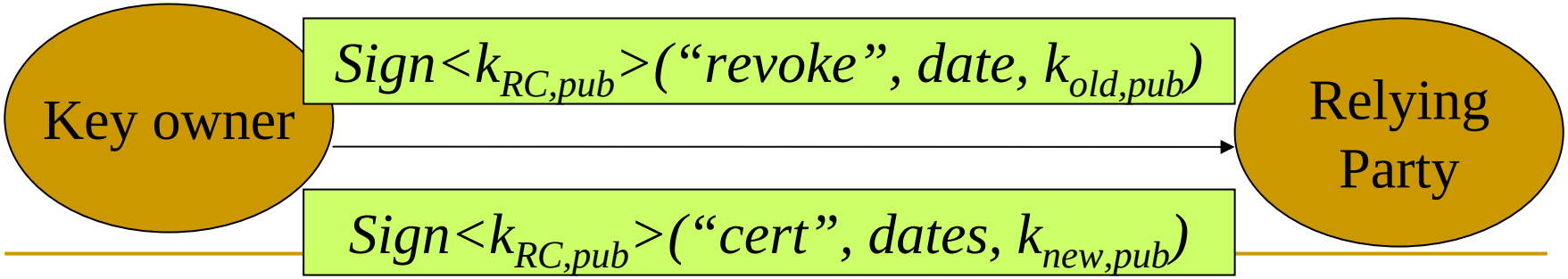
- Exposure of shared secret key: replace key
  - How to agree on new key without exposing it?
  - Can we protect past traffic?
  - Should we wait until detecting exposure to replace key?
  - Bad idea! Better refresh keys periodically (proactively)
- Exposure of private key using PK cryptosystem
  - How to revoke exposed public key, distribute new public key?
  - Can we protect past traffic?
- Exposure of private key using PK signatures
  - How to revoke exposed public key, distribute new public key?
  - Can old signatures remain effective?

Dealing with Private Key Exposure

- Private key of PK cryptosystem or signature scheme
- Can we wait until detecting exposure to replace / revoke (public) key?
  - Bad idea for exposure of shared secret key
  - But… changing public key, informing everybody, is a mess
  - Better: revoke and replace key only when exposed
- How to revoke exposed public key?
- How to distribute new public key?
- More advanced issues – later…
  - Change only *private* key, not modifying public key (periodically and proactively – without detecting exposure)
  - Can we protect past traffic?
  - Can old signatures remain effective?

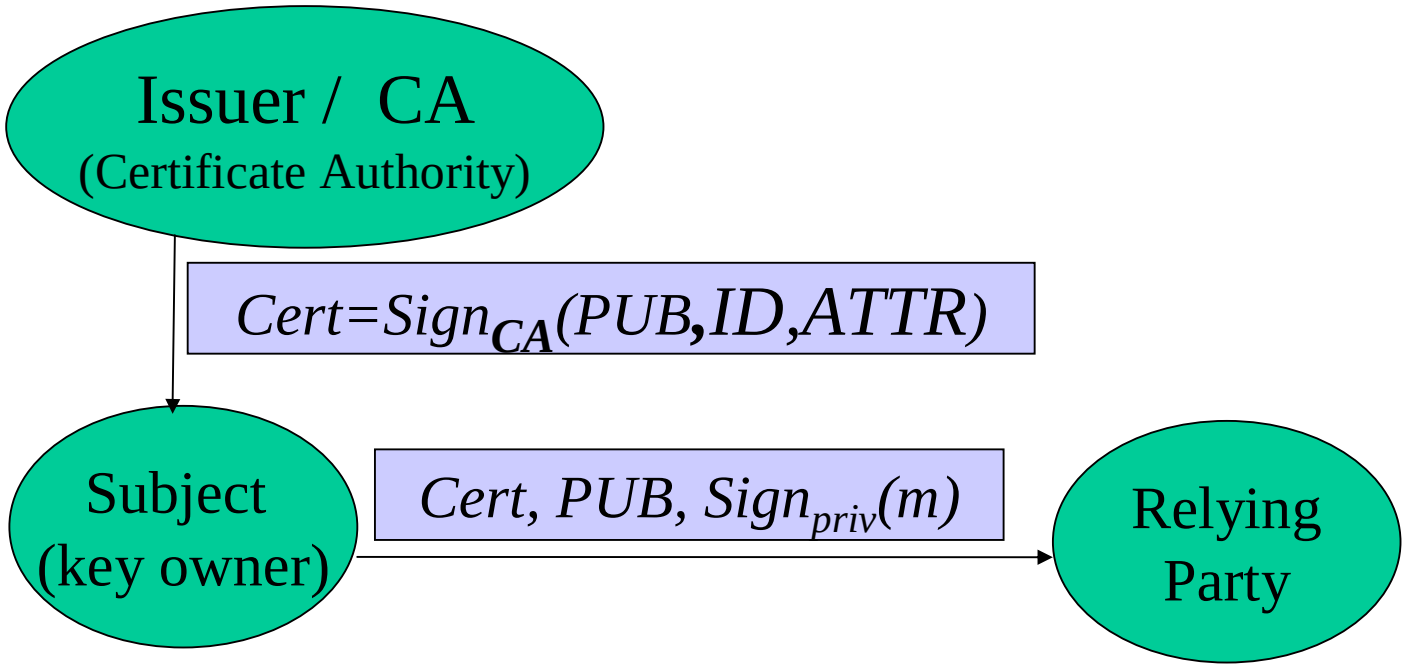
Self Certification and Revocation

- How to revoke exposed public key?
- How to distribute new public key?
- Distribute signed messages…
  - Self revocation message for exposed key *["revoke", date, $k_{old,pub}$]*
  - Self certificate of new key: *["cert", dates, $k_{new,pub}$]*
  - Both signed using $k_{RC,pub}$ – signing key for Revoke/Certify
- Problem: what if $k_{RC,pub}$ itself is exposed?



Certification & Revocation Authority

- *Certification Authority (CA, Issuer*) issues *certificate* to subject's public key
- *Relying party* validates *certificate* using CA public key
- Revocations also signed – by CA, subject, or other
- Also: limit certificate to validity period (an attribute)
- How to protect the key of the CA itself?

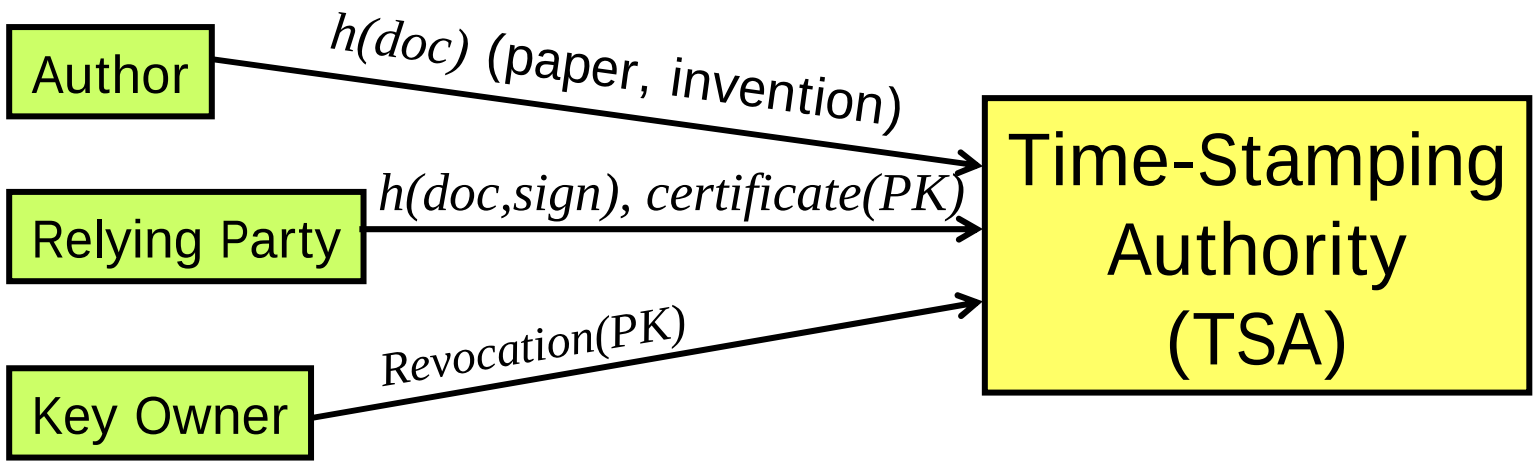


Preventing Invalidation of Signatures

- Problem: public key revoked… are signatures invalid??
  - If not… revoking does not prevent signing with stolen key
  - If yes… signer can revoke key (claim exposed) to deny signing
  - Fair solutions: signatures validated before revocation remain valid
- Solution 1: time-stamping of signature and revocation
  - Third-party evidence: date when document was signed
  - Third-party evidence: date when key was revoked
- Solution 2: limited validity/revocation periods for keys
  - Divide time into *periods,* e.g. day / month
  - Different keys for each period t: $Priv[t], Pub[t]$
  - Exposure of $Priv[t]$ does not enable signing with $Priv[t'], t'<t$
  - ➔ Even if key revoked at period t, previous signatures are Ok

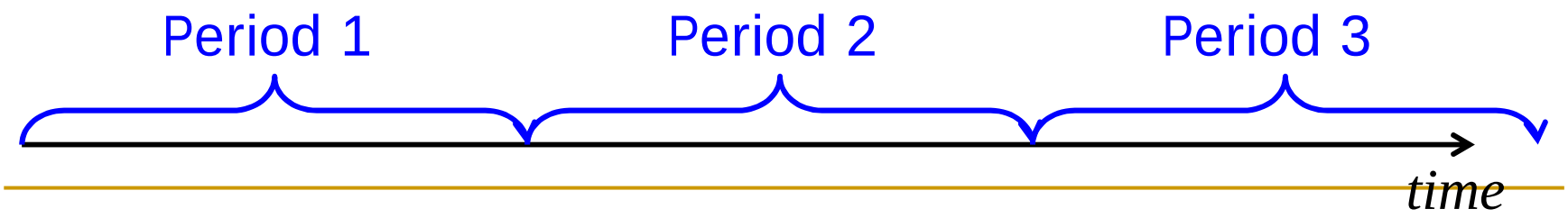
Time-Stamping of Signatures, Revocations and other documents

- Goal: non-repudiated proof of document creation date
  - Proof document/signature/revocation existed at/before date
  - If signature on contract existed (was validated) before public key was revoked, then contract remains valid!
- Timestamp signed by Time-stamping Authority
- Hash document to protect confidentiality



Limited validity periods for keys

- Goal: signatures validated before revocation are valid
- Prevent invalidation by *changing keys periodically*
  - Divide time into *periods,* e.g. day / month
  - Different keys for each period t: $Priv[t]$, $Pub[t]$
  - Exposure of $Priv[t]$ does not enable signing with $Priv[t']$, $t'<t$
  - ➔ Even if key revoked at period t, previous signatures are Ok
- How? $Cert[t]=Sign_{CA}(Pub[t],ID,ATTR[period=t])$



Forward Secure Signatures

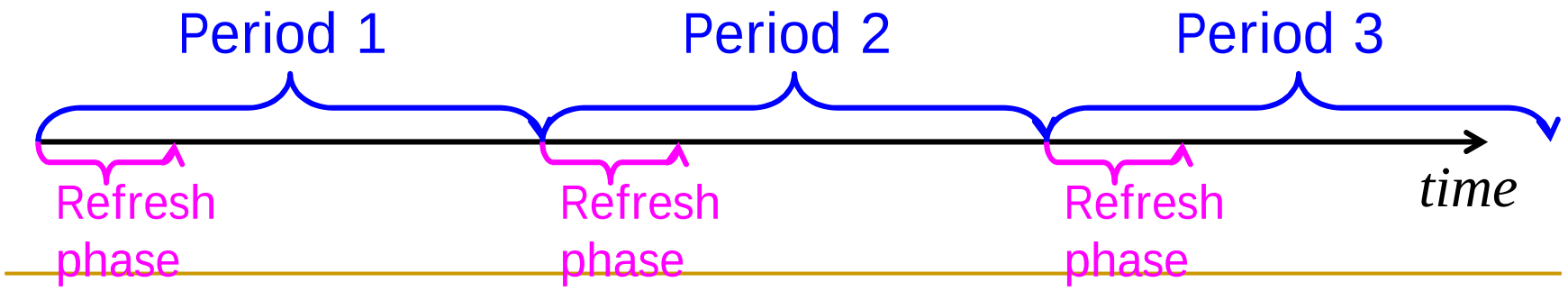
- Problem: many public keys… inconvenient
- Forward Secure Signatures:
  - Different private key for each period t: $Priv[t]$
  - Exposure of $Priv[t]$ does not enable signing with $Priv[t']$, $t'<t$
  - ➔ Even if key revoked at period t, previous signatures are Ok
  - But: use fixed public key Pub (and certificate?)
  - Validation function depends on period
  - Naïve version: deteriorating private key $Priv[t]=\{Priv'(i)\}_{i=t,..T}$
  - Public key $Pub=\{Pub'(i)\}_{i=1,...T}$
- Evolving area of research: using short Pub, $Priv[t]$
- Existing proposals incompatible with RSA / DSA etc.
  - Not in this course (or at least not in this lecture)

Remaining problems

- Recovery from private key exposure
  - Detected exposure: revoke and certify new key
  - Undetected : limited validity period, proactively change keys
- High dependency on Certificate Authority:
  - CA's private key is critical… We later show how to protect it.
  - Secure communication with the CA is critical…
  - How to secure it?
- In general: how to maintain secret shared key?

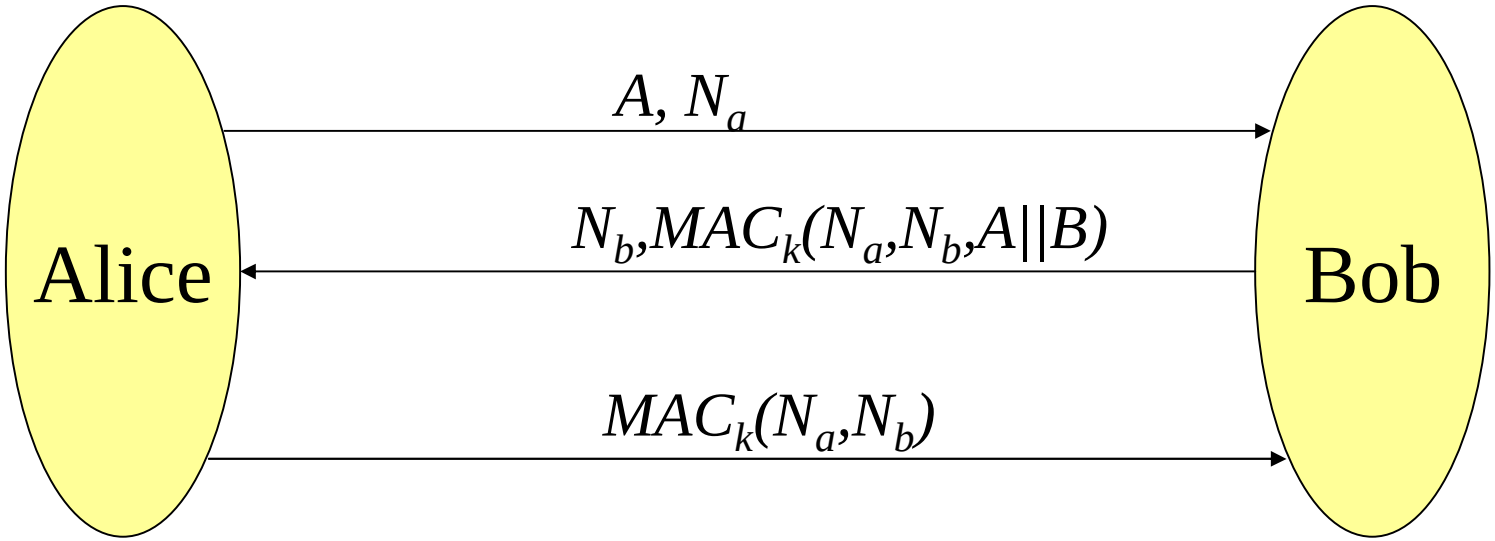
Exposure of Shared Secret Keys

- Exposure of shared secret key: replace key
  - How to agree on new key without exposing it?
  - Can we protect past traffic?
  - Refresh periodically (proactively, *before* detecting exposure)
- Divide time into *periods*
- Each period begins with *refresh/recovery phase*
  - Forward security: protect past/present from future exposure
  - Proactive security: periodically *recover* security



2PP handshake protocol

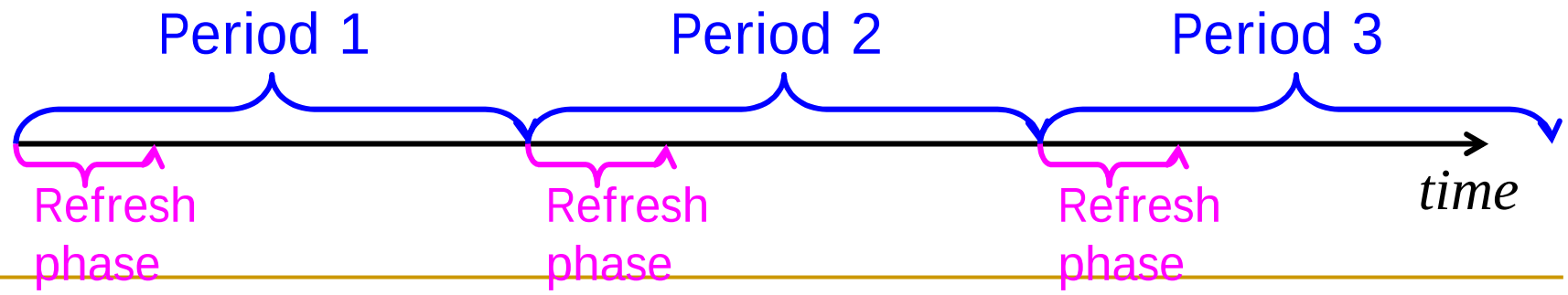
- Use master key k to securely distribute session key $k[i]$ for period i
- $k[i] = PRP_k(\text{“session”}, i, N_a, N_b)$
- Independent session keys – Adversary cannot learn anything about m, given $E_{k[j]}(m)$ and $k[i]$ for $j \neq i$
- But what if master key k is exposed?



Weak Forward Security

- Protect past traffic from future exposure of all keys
- Divide time into *periods;* $k[i]$ is session key in period i
- Each period begins with *refresh/recovery phase*
- Weak Forward security: Any adversary, given…
  - All keys in server during period i, *after refresh phase*
  - Encryption $E_{k[j]}(m)$ for $j<i$

  Cannot learn anything about *m.*

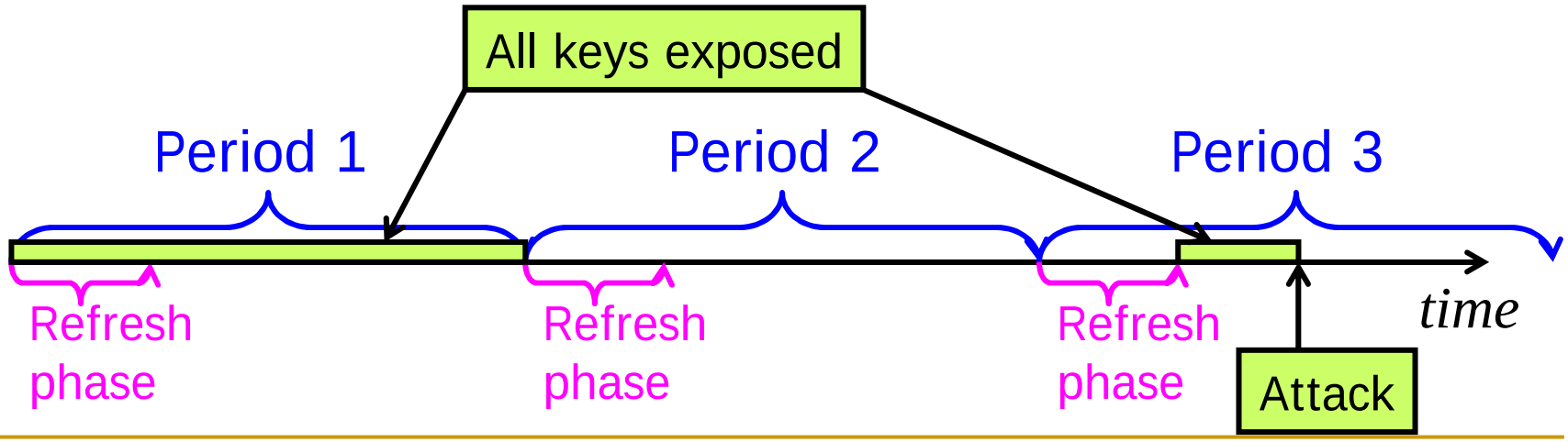


Ensuring Weak Forward Security

- Evolving master key: $MK[i]=PRP_{MK[i-1]}(\text{“}next\text{”})$
- Derive session key: $k[i]= PRP_{MK[i]}(\text{“}s\text{”}, N_a, N_b)$
- Adversary cannot learn anything on m, given $E_{k[j]}(m)$ and…
  - All keys in server after refresh phase of period $j+1$,
  - And session keys $k[i]$ for all periods $i \neq j$
- However… if all keys of period i are exposed (incl. Master key $K[i]$) adversary can decrypt all traffic after period i
- Can we limit the value of $K[i]$ to period i??
- Suppose keys of period i are exposed only *after* period i…

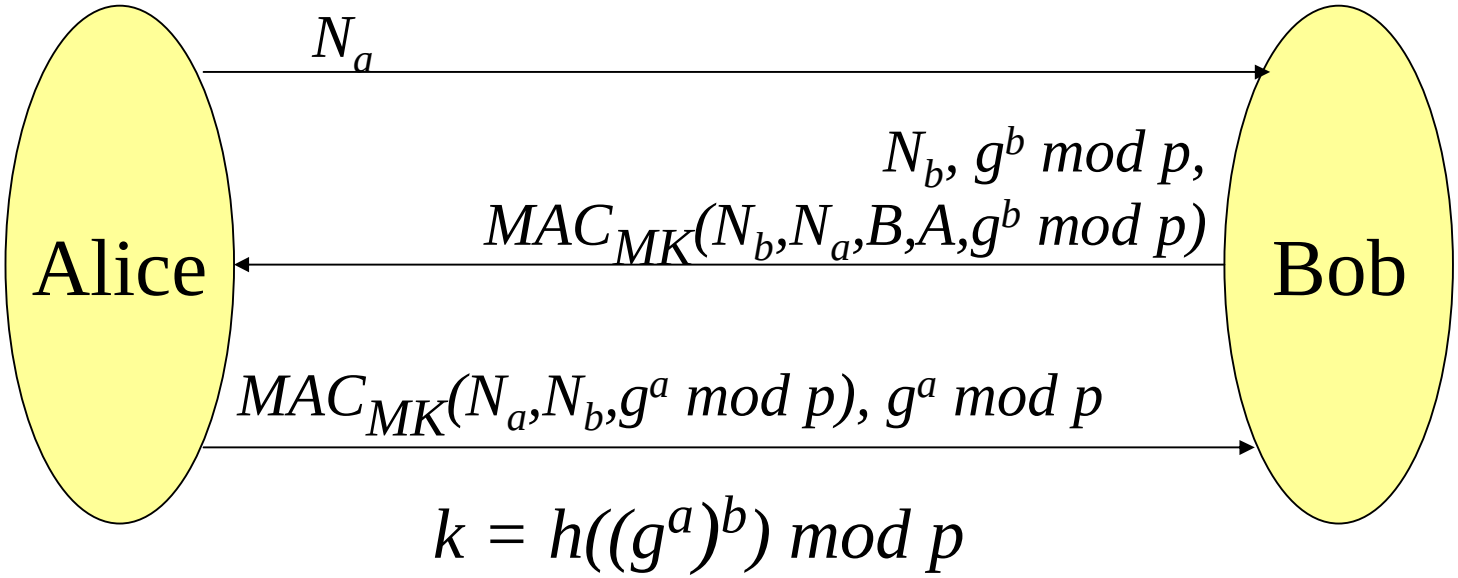
Strong (Perfect) Forward Security

- Protect traffic of period *i* from exposure of all keys of all periods $j \neq i$, as long as exposure happens after refresh phase of period $i+1$
- Motivation: some keys may persist from past periods; plus, attacker may expose old key by time-consuming methods, e.g. cryptanalysis, reading from `erased` data



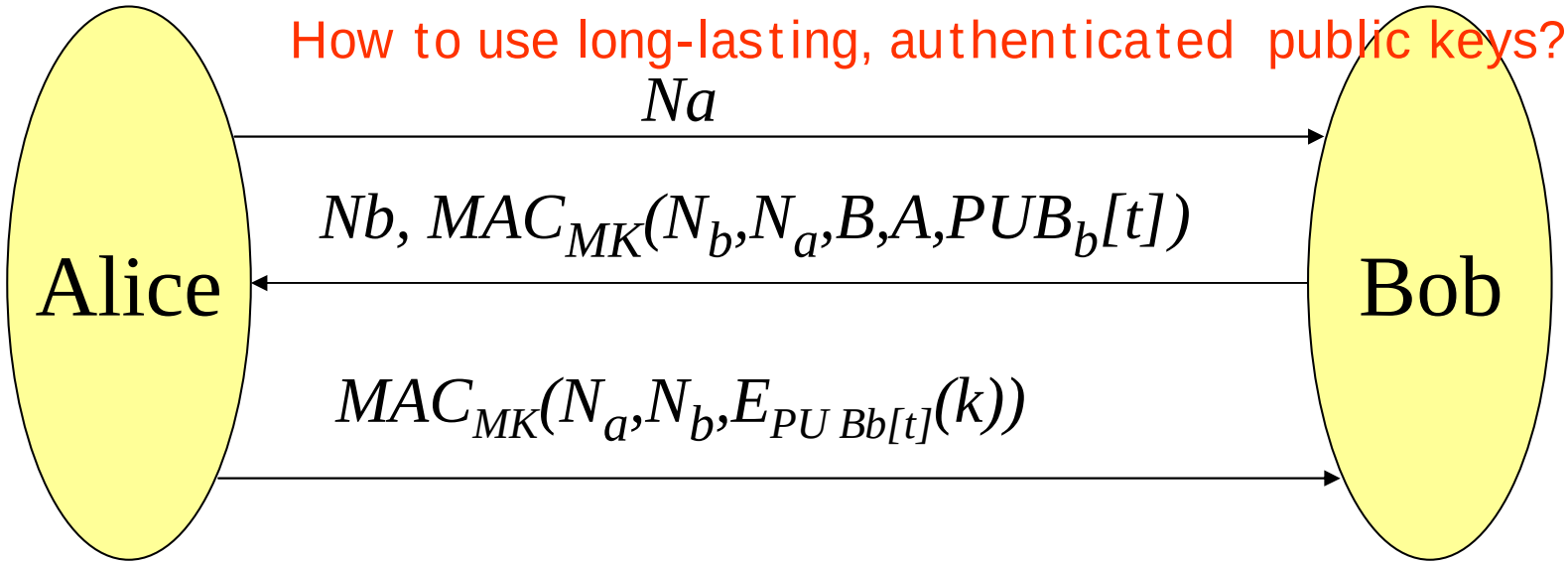
Strong Forward Secrecy with DH

- Strong (Perfect) Forward secrecy: exposure after period i of all keys in all periods $j \neq i$ exposes nothing about messages sent during i.
- How? Periodical Authenticated Key Agreement [DH]
  - a,b selected randomly each time by Alice, Bob respectively



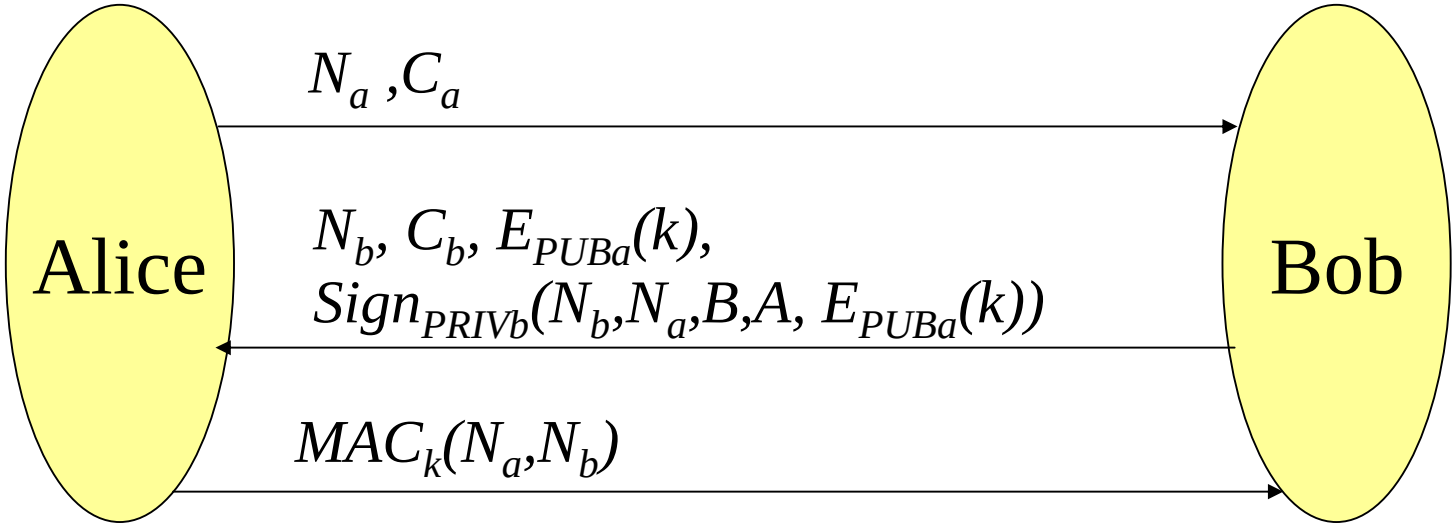
Forward Secrecy with Pub Key

- Each period *t,* Bob generates new, ephemeral public key $PUB_b[t]$ and sends to Alice
- Alice may also generate public key and send to Bob
- Public key generation →substantial overhead (esp. RSA)
- *k* exposed → adversary can impersonate (fake $PUB_b[t]$)



Session Keys from Public Keys

- Public keys PUB_a, PUB from certificates



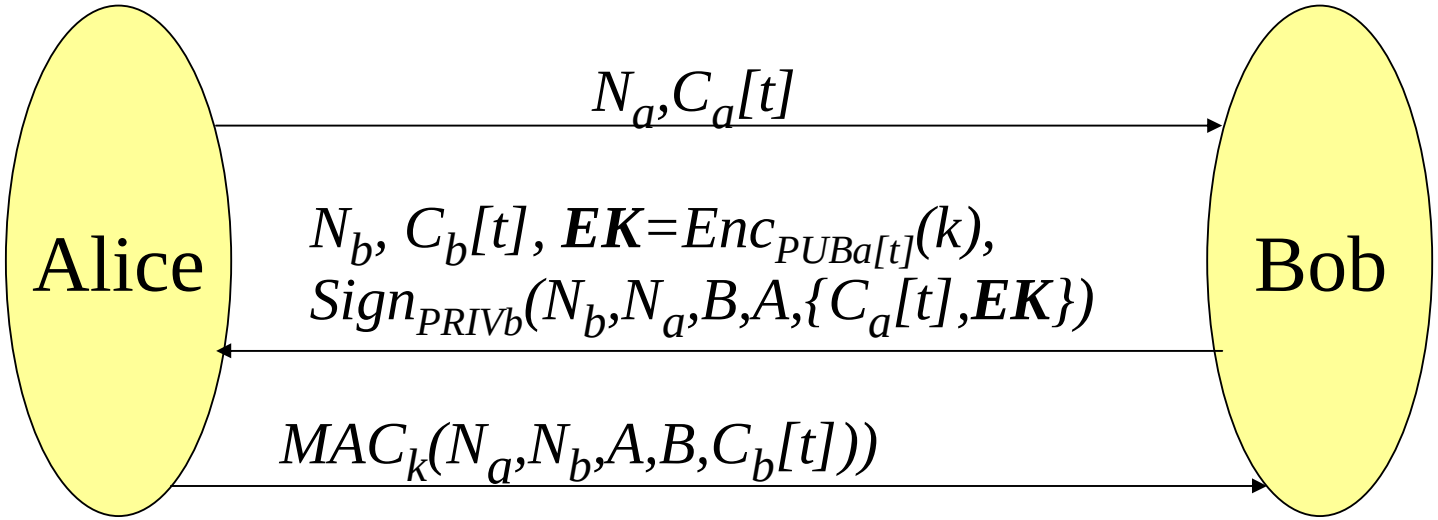
- Simple extension to protect k even if attacker learns $PRIV_a$.
- What if attacker learns both $PRIV_a$ and $PRIV_b$?

→ Can extend with forward secrecy to protect past traffic...

→ Can we do better - change private keys?

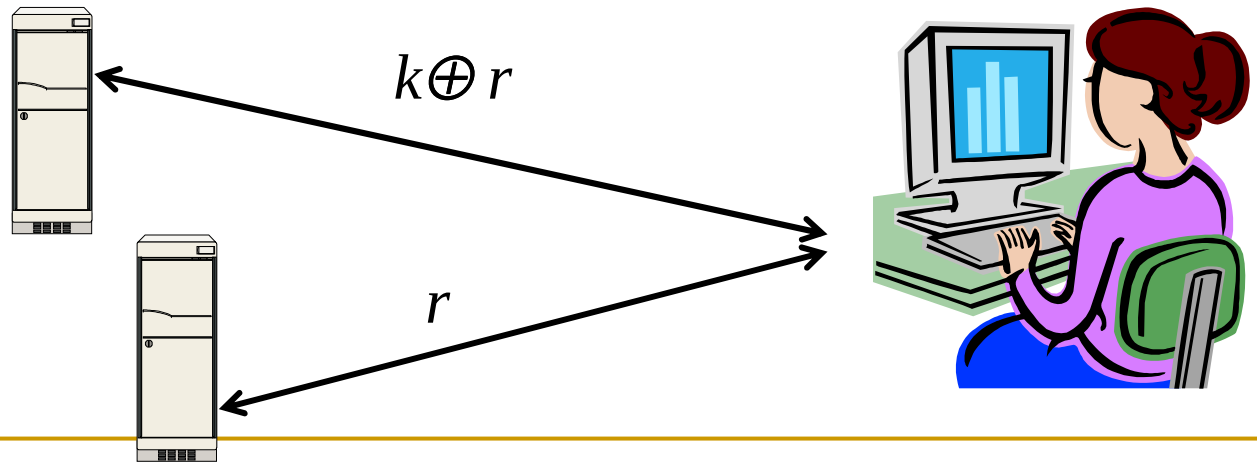
Proactive Secure Communication

- Each period t, Bob and Alice generate new public keys $PUB_b[t]$, $PUB_a[t]$ and certificates $C_a[t]=Sign_{CA}\{PUB_a[t],t\}$, $C_b[t]=Sign_{CA}\{PUB_b[t],t\}$
- Exchange keys, certificates, encrypted session key…
- Critical: Proactively secure Certification Authority
- Idea: use redundant, multiple servers for secure CA!



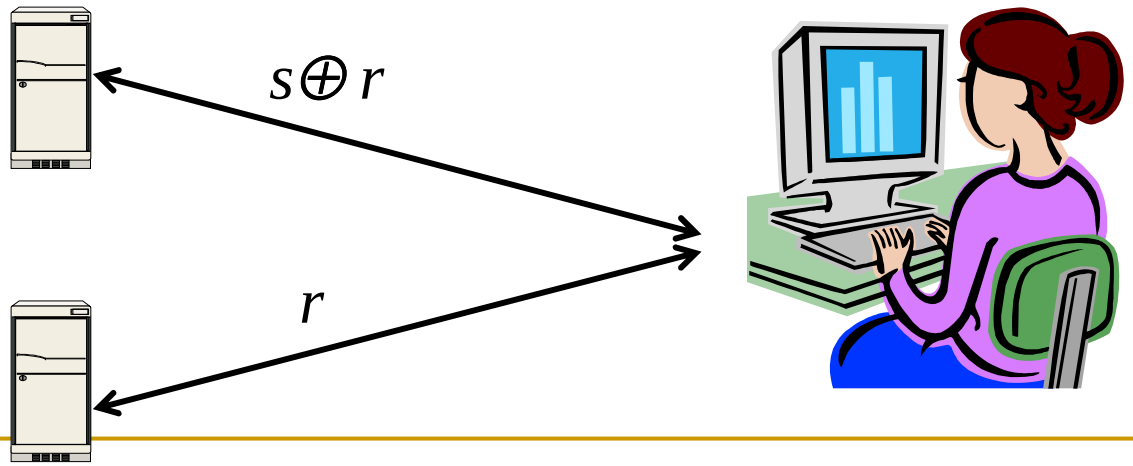
Security By Redundancy

- Problem: server corrupted *at time of service*
  - Certificate Authority: capture CA's private key, issue certificates
  - Secure login: access to account, change password
  - Secure user data: capture password and/or key
  - Database with private index: capture index at query
- Solution: use multiple, redundant servers
  - Assume not *all* servers are corrupted
- Example: share secret key k among two servers:



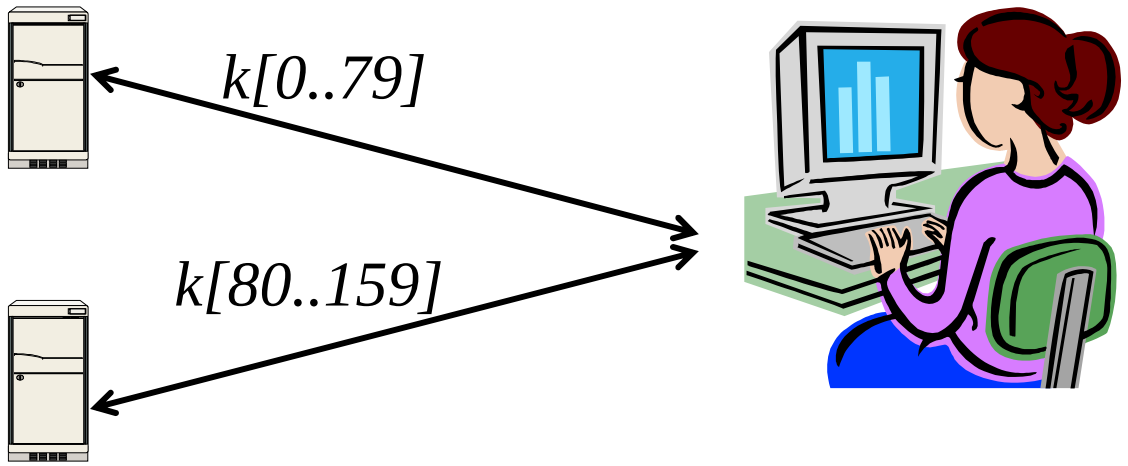
Secret Sharing

- Idea: share secret among multiple, redundant servers
  - Do not put all your eggs in one basket!
- Below: share secret s among two servers
- Questions:
  - Share among $n>2$ servers [easy…]
  - How to use / retrieve / store key securely? [Later]
  - Require only threshold $t<n$ servers for recovery [Next]



Question

- In many cases the secret shared is a key
- When sharing a key, why not simply give each server half of the bits of the key?

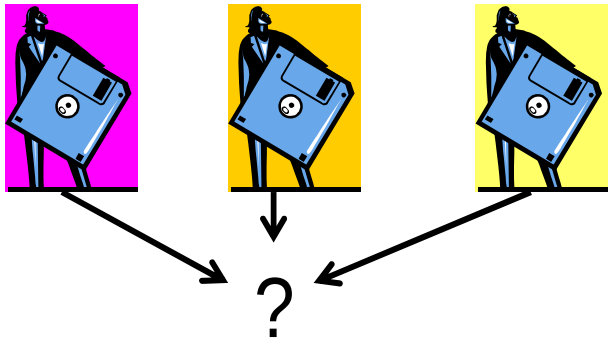


Secret Sharing with Threshold

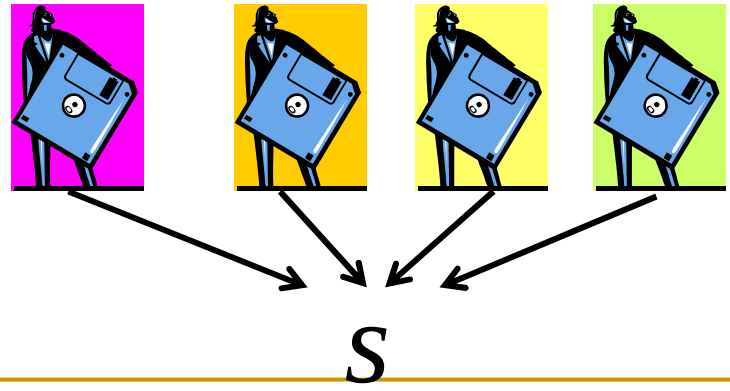
- N users and a threshold t
- Any group of t users can jointly obtain the secret
- Any group of less than t users cannot jointly obtain any information about the secret
- Assume we have a dealer who has the secret
- $(N,1)$-scheme: give secret to all users…
- (N,N)-scheme: as before (XOR), or…
  - Let s be the secret, let p be a large number ($s<p$)
  - Let $a_1,\ldots, a_N$ be random numbers s.t. $a_1+ a_2+\ldots+ a_N = s \bmod p$
  - Assign a_i to the i^{th} user

Threshold Secret Sharing protects:

- *Secrecy*: no t-1 shareholders can learn the secret

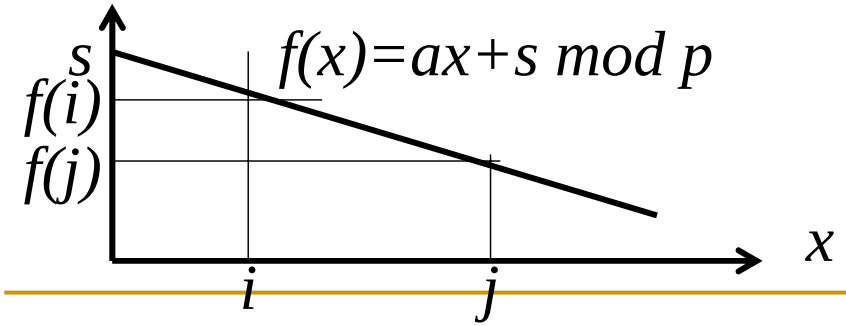


- *Integrity*: Every t shareholders can reconstruct the secret (no N-t shareholders can destroy the secret)



Secret Sharing with Threshold *2*

- *(N,2)*-scheme: N users and a threshold $t=2$
- Need (exactly) two users to recover secret s
- Idea [Shamir]: share points on a line
  - Let s be the secret, let p be a large prime (s, $N<p$)
  - Let $a \in_R Z_p$ (i.e. $\{0,\ldots(p-1)\}$)
  - Hide the secret as points on the line $f(x)=ax+s \bmod p$
  - User $i \in \{1,..N\}$ receives $s_i=f(i)=ai+s \bmod p$
  - Every two users have two points ➔ can find $s=f(0)$

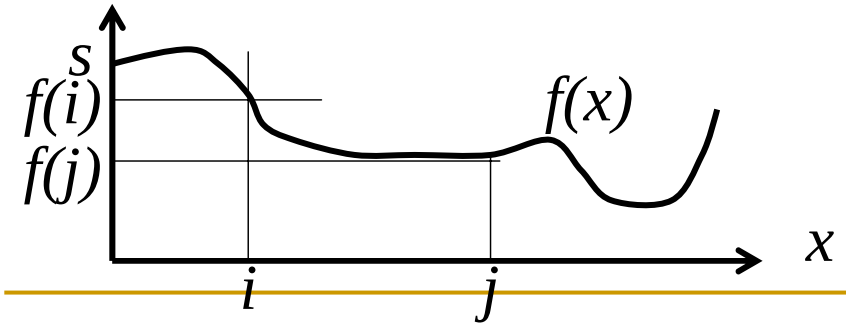


Polynomial Secret Sharing [S79]

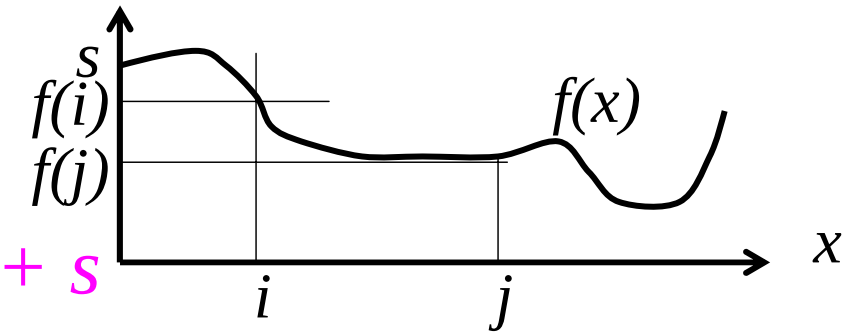
- Generalize from *(N,2)* to *(N,t)*
- Instead of a line, use a *polynomial of degree (t-1)*
- Pick random polynomial

$$f(x) = a_{t-1}x^{t-1} + a_{t-2}x^{t-2} + \ldots + s \bmod p$$

- As before:
  - $s = f(0)$
  - User i receive $s_i = f(i) \bmod p$
  - t points allow interpolation ➔ computing $f(x)$ ➔ $s=f(0)$



Lagrange Interpolation



- If
  $f(x) = a_{t-1}x^{t-1} + a_{t-2}x^{t-2} + \ldots + s$
- Then given $\{f(x_i), x_i\}$ for $i=1,..t$,

$$f(x) = \sum_{i=1}^{t} f(x_i) \prod_{\substack{j=1 \\ j \neq i}}^{t} \frac{(x_j - x)}{(x_j - x_i)}$$

- In particular,

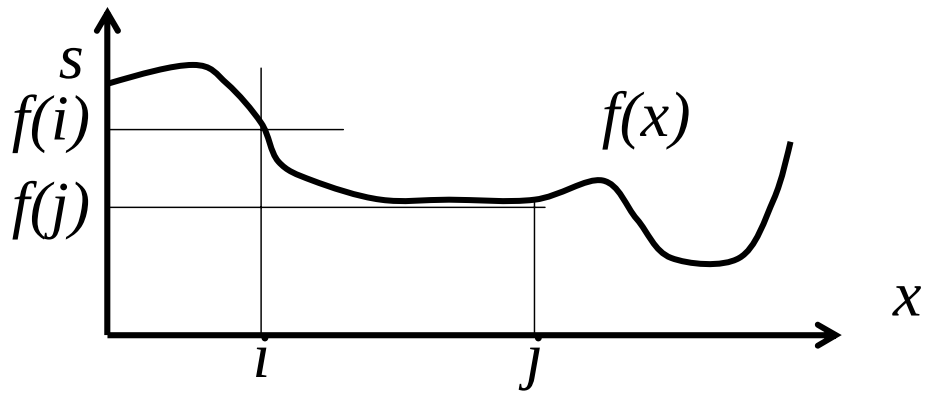
$$s = f(0) = \sum_{i=1}^{t} f(x_i) \prod_{\substack{j=1 \\ j \neq i}}^{t} \frac{x_j}{(x_j - x_i)}$$

- Works the same *mod p*

Using Lagrange Interpolation

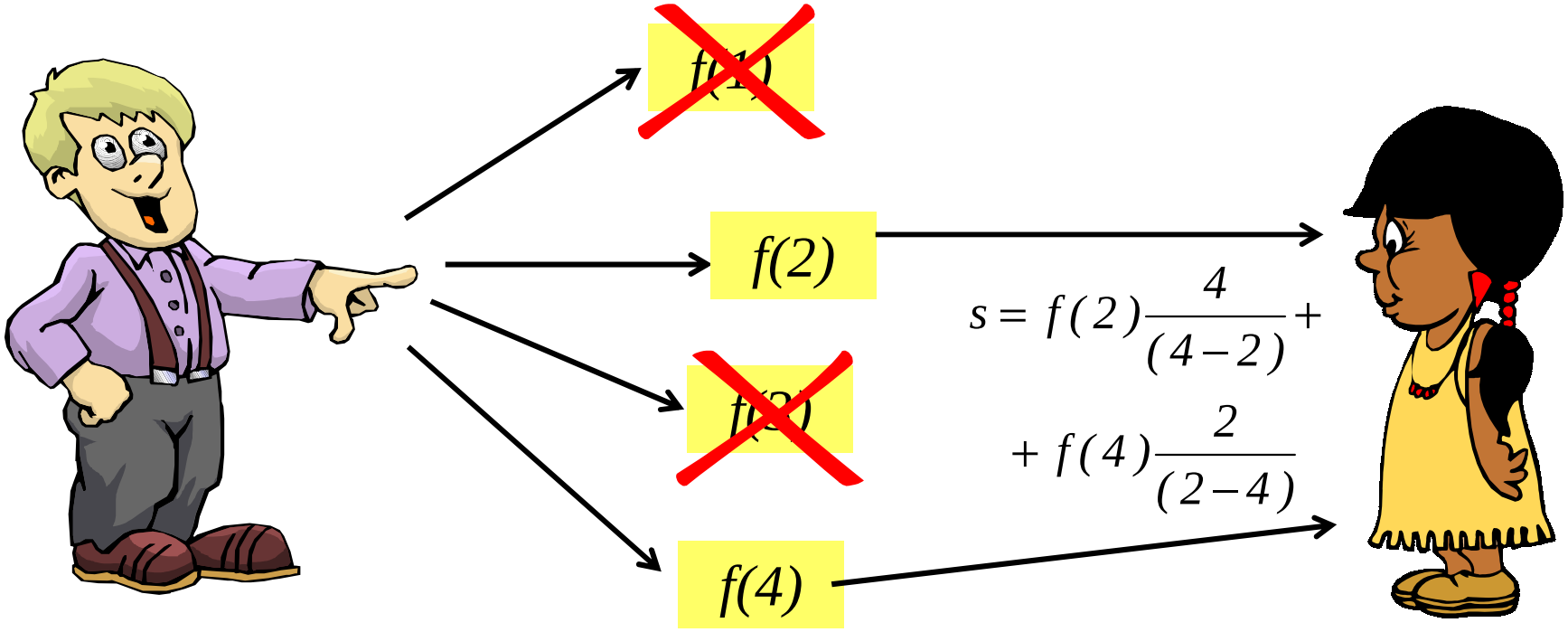
- With polynomial secret sharing, $x_i = i$ and compute $mod\ p$
- Given $\{f(i)\}$ for $i \in T$ s.t. $|T| = t$ and $T \subseteq \{1,\ldots N\}$

$$s = f(0) = \sum_{i \in T} f(i) \prod_{\substack{j \in T \\ j \neq i}} \frac{j}{(j - i)} \ mod\ p$$



Secure communication by sharing

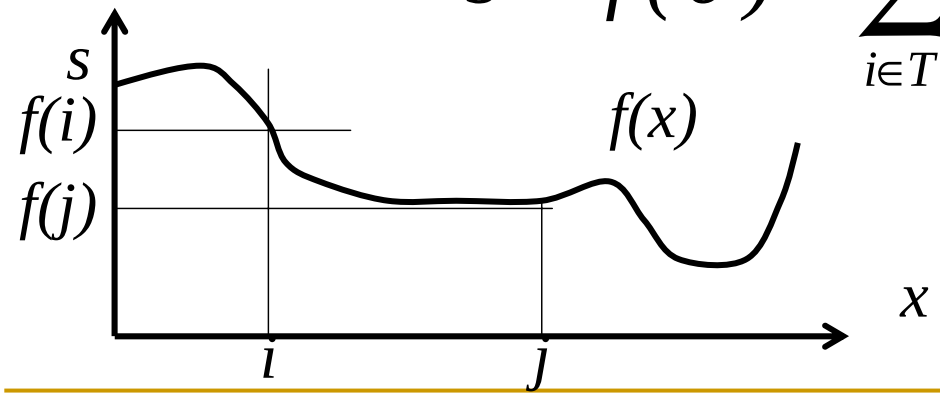
- We can use secret sharing also to secure communication using several partially-trusted channels/messengers…



Polynomial Secret Sharing - Exercise

- Let's use $p=37$
- Select secret s <37
- Select $a_i \in_R \{2\ldots37\}$; $f(i) = a_{t-1}i^{t-1} + a_{t-2}i^{t-2} + \ldots + s \bmod 37$
- Give $f(i)$ to party i
- Receive $f(i)$ from i

$$s = f(0) = \sum_{i \in T} f(i) \prod_{\substack{j \in T \\ j \neq i}} \frac{j}{(j-i)} \bmod p$$



What if some shares are modified??

Share Modification Threats

- Malicious user sends bogus share
  - Undetectable reconstruction of wrong secret
  - Malicious user can recover real secret
- User receives corrupted share
  - In some applications: intentionally corrupted by sender!
  - Undetectable until reconstruction (or at all)
- Solution: VSS - *Verifiable Secret Sharing* [Feldman]

Verifiable Secret Sharing [Feldman]

- Distribute also *public verifiers* $g^{a_1}, g^{a_2}, \ldots g^{a_{t-1}} \bmod p$
- Verify share $f(i) = s + a_1 i + a_2 i^2 + \ldots + a_{t-1} i^{t-1} \bmod p$
- By raising g by both sides:

$$g^{f(i)} \stackrel{?}{=} (g^{s})(g^{a_1})^{i}(g^{a_2})^{i^2} \ldots (g^{a_{t-1}})^{i^{t-1}} \bmod p$$

- This preserves secrecy but allows detection of bogus shares
  - When received by i (in sharing phase)
  - When sent by i (in reconstruction phase)

Polynomial Secret Sharing - Properties

- Perfect (unconditional) security: given less than *t* shares, all values of the shared secret are equally probable
  - No unproven (computational) assumptions
- The size of each share is the same as of the secret
- Shares for new users can be computed without changing existing shares
- Verify shares (VSS) variant to detect bogus shares
  - Computationally-secure
- Symmetric: all users equally trusted / powerful
  - Or: give multiple shares to `highly trusted` users
- Can we trust some users more than others?

Asymmetric Secret Sharing

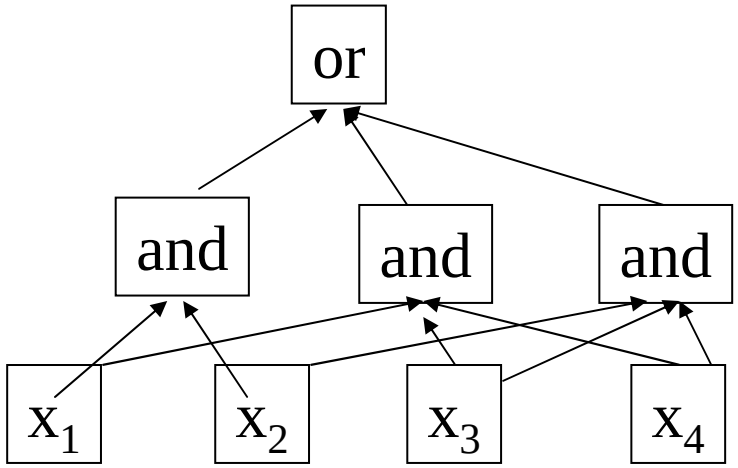
- Not all users equally trusted
- Access function: $f(x_1,\ldots,x_N)$
  - X_I: *True* if user i contributes her share
  - $f(x_1,\ldots,x_N)$ is *True* if $\{x_1,\ldots,x_N\}$ together can recover s
- Monotone Access Function
  - Adding more shares can only help
  - Boolean circuit with only “and” and “or”

$$f\left(x_1, x_2, x_3, x_4\right) = x_1x_2 \vee x_1x_3x_4 \vee x_2x_3x_4$$

View as Monotone Boolean Circuit

$$f(x_1, x_2, x_3, x_4) = x_1x_2 \vee x_1x_3x_4 \vee x_2x_3x_4$$

- Boolean circuit with only “and” and “or”



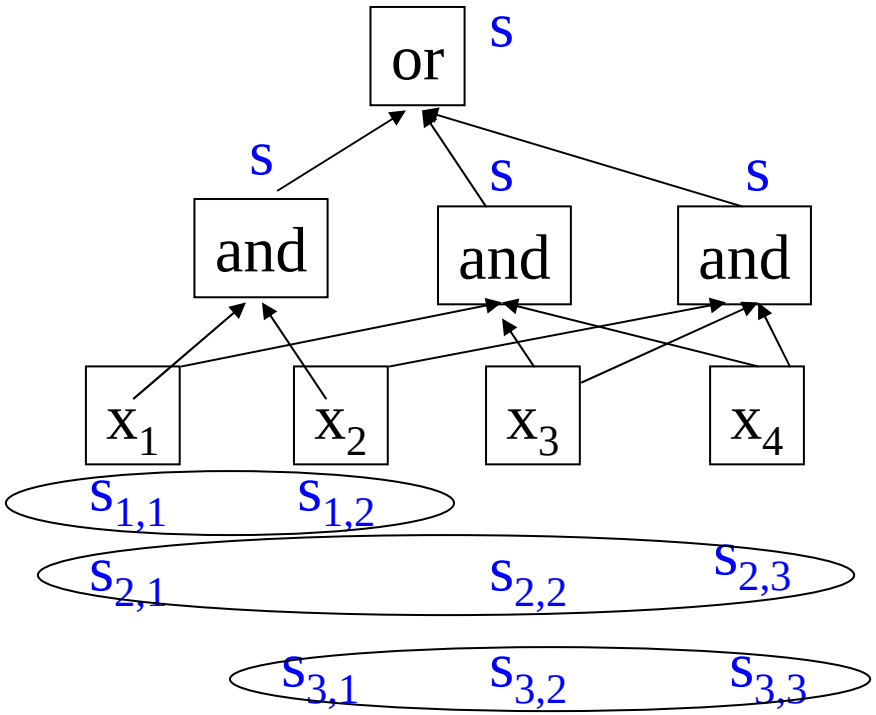
Benaloh and Leichter.

- “or” gate is an $(N,1)$ scheme
- “and” gate is an (N,N) scheme
- Recursively use the simple schemes for $(N,1)$ and (N,N)

Circuit Evaluation

$$f(x_1, x_2, x_3, x_4) = x_1x_2 \vee x_1x_3x_4 \vee x_2x_3x_4$$

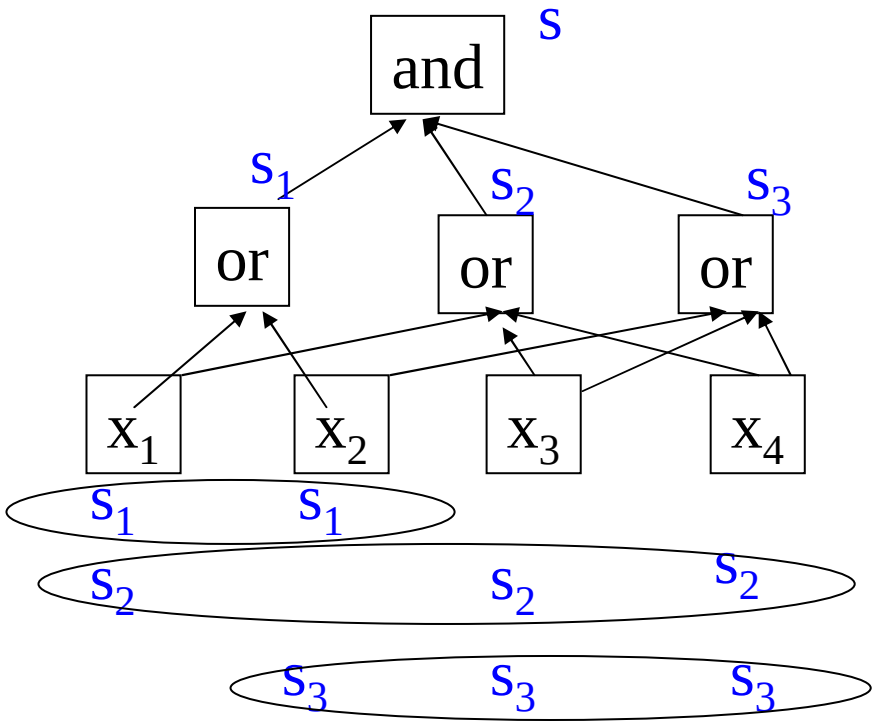
- Boolean circuit with only “and” and “or”



Circuit Evaluation

$$f(x_1, x_2, x_3, x_4) = (x_1 \vee x_2)(x_1 \vee x_3 \vee x_4)(x_2 \vee x_3 \vee x_4)$$

- Boolean circuit with only “and” and “or”

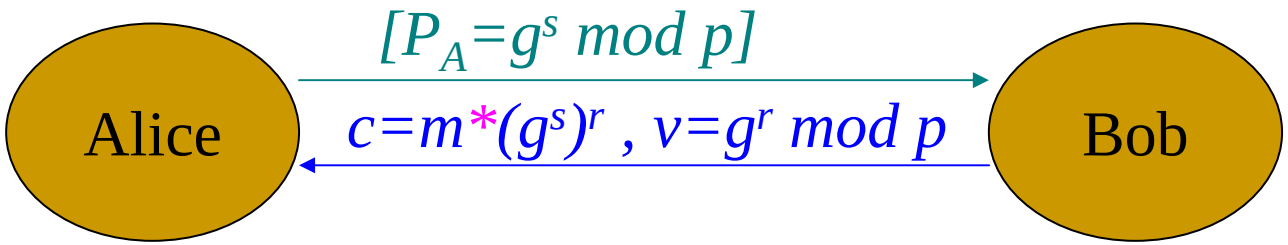


How to use a shared secret?

- Naïve: provide all shares to the user
  - How to authenticate the user?
  - A: user provides secret key / password
  - Why not use this key to encrypt the secret?
- Group operations:
  - Distributed decryption: t servers decrypt $E_{priv}(m)$
  - Distributed signing: t servers sign `together` $Sign_{priv}(m)$
  - Signing allows a distributed secure Certificate Authority (CA)
- How?
  - RSA signatures / encryption
  - DSA signatures
  - El-Gamal signatures / encryption

Recall: El-Gamal PK Encryption

- Let's use secret key s for Alice…
- Bob chooses r and $v = g^r \bmod p$
- Bob encrypts message m using $(g^s)^r$:
  $c = m * g^{sr} \bmod p$
- Bob sends c, v
- Alice uses $v^s = g^{sr} \bmod p$ to decrypt: $m = c / g^{sr} \bmod p$



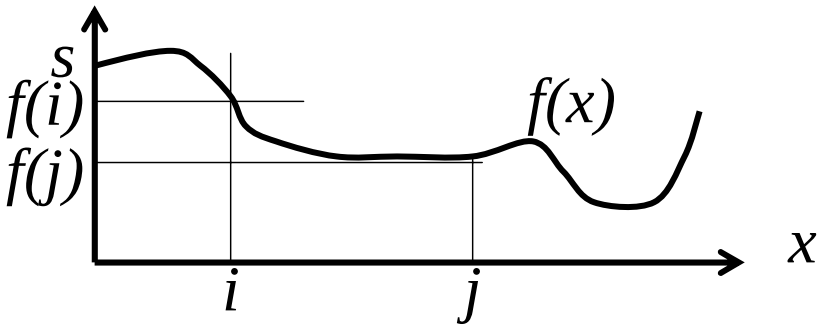
Recall: Polynomial Secret Sharing

- Pick random polynomial

$$f(x) = a_{t-1}x^{t-1} + a_{t-2}x^{t-2} + \ldots + s \bmod M$$

  - $s = f(0)$
  - User i receives $s_i = f(i) \bmod p$
- By Lagrange interpolation:

$$s = f(0) = \sum_{i=1}^{t} f(x_i) \prod_{\substack{j=1 \\ j \neq i}}^{t} \frac{x_j}{(x_j - x_i)}$$



Recall: Polynomial Secret Sharing

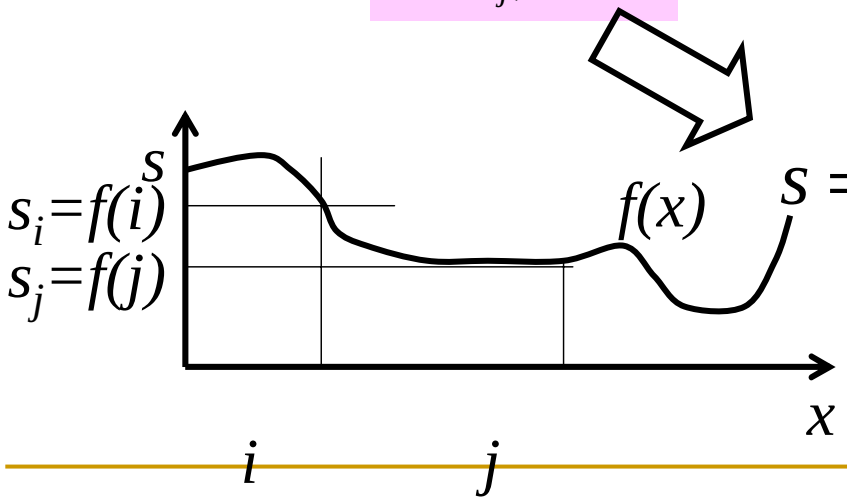
- Pick random polynomial

$$f(x) = a_{t-1}x^{t-1} + a_{t-2}x^{t-2} + \ldots + s \bmod M$$

- In our case $x_i = i$ so:

$$s = f(0) = \sum_{i=1}^{t} f(i) \prod_{\substack{j=1 \\ j \neq i}}^{t} \frac{j}{j-i}$$

- Denote: $\zeta_i = \prod_{\substack{j=1 \\ j \neq i}}^{t} \frac{j}{j-i}$



$$s = f(0) = \sum_{i=1}^{t} f(i)\zeta_i = \sum_{i=1}^{t} s_i \zeta_i$$

Distributed El-Gamal PKC

- Secret key s split between servers $A_1,\ldots A_N$
  - $s_i=f(i);\ s=f(0)$
- Bob operates as usual (unaware key is shared):
  - Chooses r and $v= g^r \bmod p$
  - Encrypts: $c=m*g^{sr} \bmod p$
- Each server A_i computes $v_i = v^{s_i\zeta_i} = g^{r\cdot s_i\zeta_i}$
- Compute $k = \prod_{i=1}^{t} v_i = \prod_{i=1}^{t} g^{rs_i\zeta_i} = g^{r\cdot\sum_{i=1}^{t} s_i\zeta_i} = g^{r\cdot s} \bmod p$
- Decrypt: $m=c / k \bmod p$

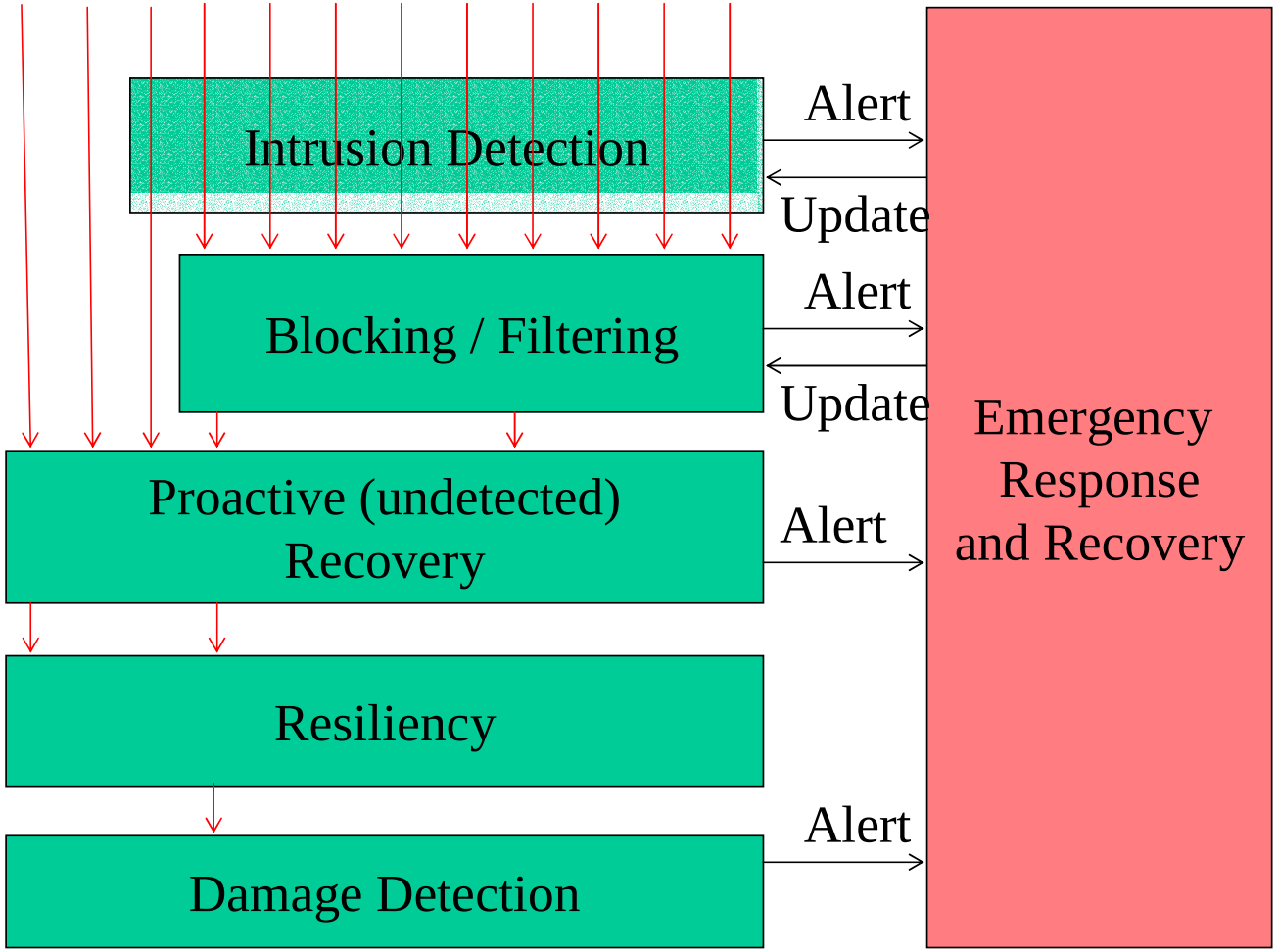
If all servers may be corrupted…

- Can we provide secure services using insecure servers?
- Insecure servers: servers that *may* be corrupted
- We showed how to provide secure services…
  - If server is not corrupted *at time of service*
  - By multiple servers – assuming not *all* are corrupted
  - Assume *enough* are *not* corrupted (One? Majority? 2/3?)
- What if eventually, all servers may be corrupted?
- Assume: *enough* servers are Ok at each period
- Proactive security: periodic process maintains security

Proactive Security: Recovery from corruptions

- Attacker try to avoid detection of attack
  - Once detected, the advantage is with the system administrator
- Multiple defense lines:
  - Prevention of corruption
  - Detection of attack
  - Recovery from detected attack
  - Periodical, *proactive* recovery from *undetected* attack
- Proactive recovery must be inexpensive
  - Automated only – no operator involvement

Defenses against intrusions

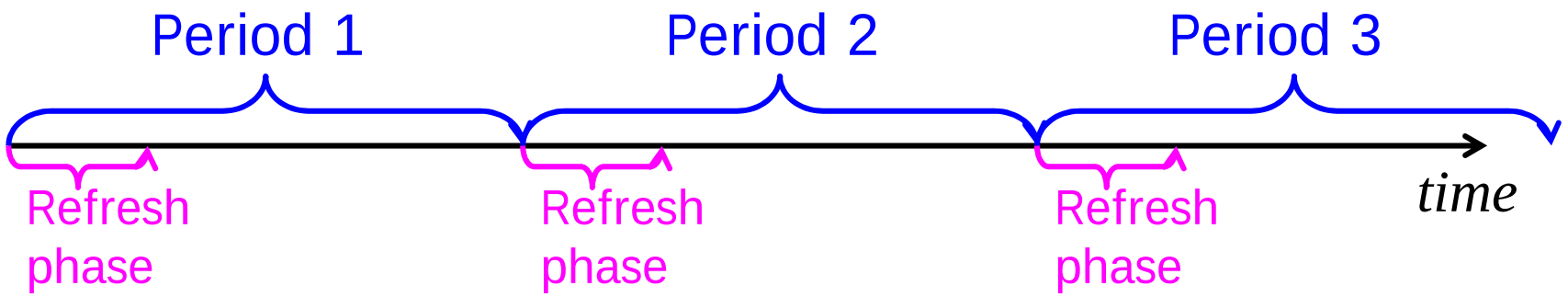


Proactive Password Security

- Simple example of proactive security
- Accepted practice: periodical change of passwords
- Minimal involvement of user, none of operator
- Attacker loses control (if user changes)
- Attacker risks detection (if attacker changes)
- Extreme: one-time passwords (e.g. S/Key)
- This example was centralized – since it involved the user
- Most proactive security mechanisms are *distributed*…

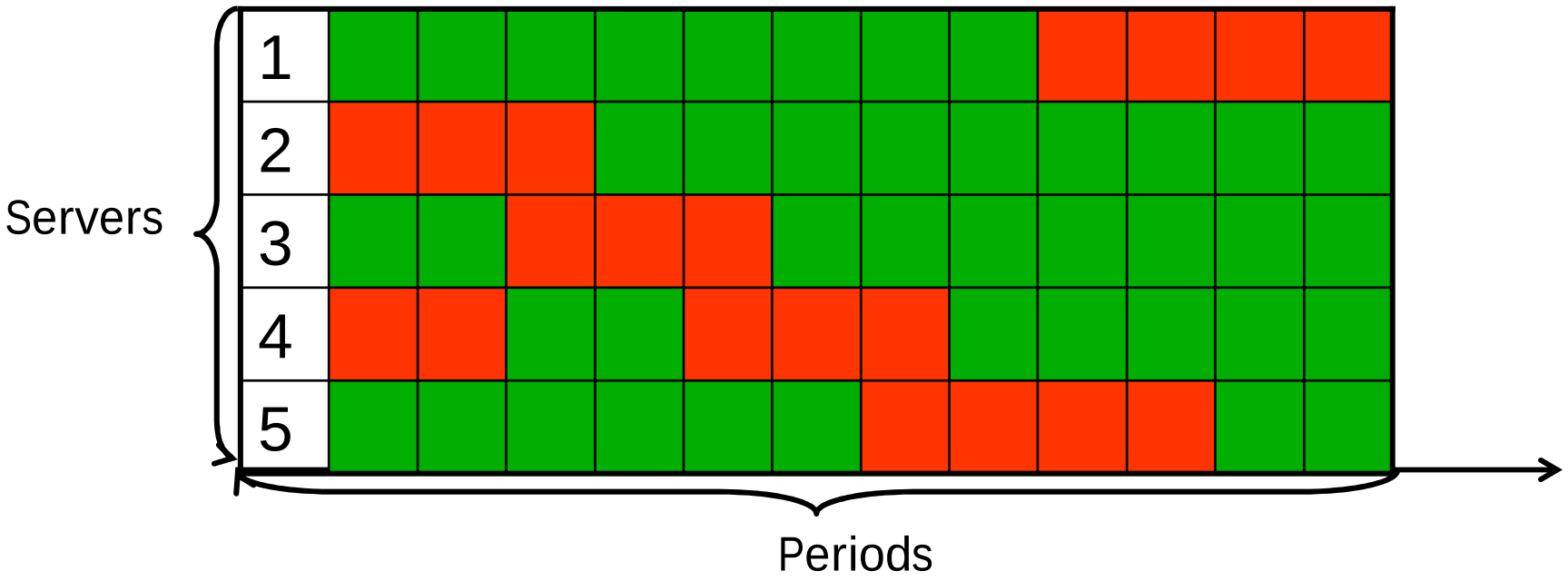
Proactive = Distribution + Refresh

- Proactive security combines:
  - Redundancy – avoid single point of failure
  - Refresh recovery – periodically return servers to secure state (when not controlled by adversary during refresh)
- Secrets exposed by adversary become stale, useless
- Lifetime is divided into *periods*
- Each period begins with *refresh/recovery phase*



Proactive Security Model

- Each period begins with *refresh/recovery phase*
- Assume: *enough* servers are Ok at each period
  - E.g. At least 3 out of 5 (i.e. at most 2 are corrupted per period)
- Proactive security: periodic process maintains security

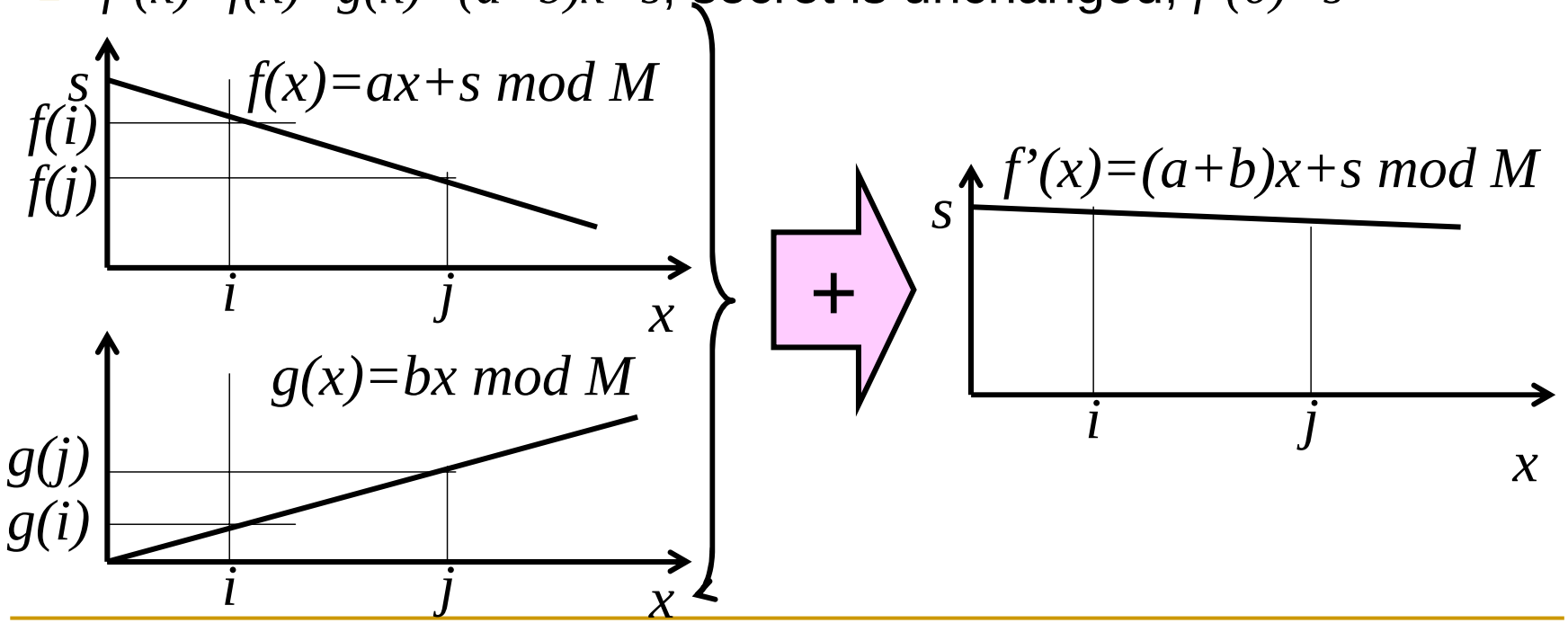


Proactive Secure Log Files

- Another simple proactive-secure mechanism
- Log files are important for intrusion detection
- Hackers try to `erase tracks` by erasing/modifying log
- How to protect log files?? --`Write-once Memory`
  - Simple: use hardware write-once memory (paper, CD-R,…)
- Proactive secure write-once memory:
  - Send every log record to all (or most) servers
  - Servers periodically compare log files (and merge?)
- What about secure storage of *secrets* and *keys*?

Proactive Secret Sharing (t=2)

- Idea: refresh secret shares
- How? Consider sharing by polynomial of degree t-1, e.g. $f(x)=ax+s$
- Refresh shares by adding random $g(x)=bx$; notice $g(0)=0$
- $f'(x)=f(x)+g(x)=(a+b)x+s$; secret is unchanged, $f'(0)=s$



Proactive Secret Sharing (any t>1)

- Given shares as values of polynomial *f()* of degree t-1
  - $f(x) = a_{t-1}x^{t-1} + a_{t-2}x^{t-2} + \ldots + s \bmod M;\ s = f(0)$
  - Server i receives $s_i = f(i) \bmod p$
- Add refresh polynomial of degree t-1: *g()* s.t. *g(0)=0*
- *f'(x)=f(x)+g(x)*; secret is unchanged, *f'(0)=s*
- Each server *i* simply add its own share *f'(i)=f(i)+g(i)*
- Now erase old share *f(i)*
- How to select, distribute refresh polynomial *g()*?
  - Given *g()*, adversary can compute new shares from old shares and old shares from new shares
  - If not properly shared or *g(0)≠0:* reconstruction fails!

Selecting and distributing refresh polynomial $g()$

- Goal: select, distribute refresh polynomial $g()$ s.t. $g(0)=0$
  - Ensure secrecy and integrity of (shares of) $g()$
- Every server i selects, shares polynomial $g_i()$ s.t. $g_i(0)=0$
- Add all of them: $g(x)=g_1(x)+\ldots+g_n(x)$
- ➔ $g(0)=g_1(0)+\ldots+g_n(0)=0$
- Use Verifiable Secret Sharing to verify shares and $g_i(0)=0$
- Actually we verify that we share zero, not a secret…

Verifiable Zero Sharing

- Distribute also *public coefficients* $g^{a_1}, g^{a_2}, \ldots g^{a_{t-1}} \bmod p$
- Verify share $f(i) = 0 + a_1 i + a_2 i^2 + \ldots + a_{t-1} i^{t-1} \bmod p$
- By raising g by both sides:

$$g^{f(i)} \stackrel{?}{=} (g^0)(g^{a_1})^i (g^{a_2})^{i^2} \ldots (g^{a_{t-1}})^{i^{t-1}} \bmod p$$

- Allows detection of bogus shares
- Also verifies these are shares of zero!!

Proactive Signatures and PKC

- Private (signature, decryption) key is shared
- Public (validation, encryption) key is known to all
  - Appears as a `regular` public key
- Shares of secret key refreshed periodically
- Signature/decryption requires *t* out of *N* shares
- Private key is never explicitly reconstructed !
- Known for…
  - RSA (key generation tricky but possible)
  - DSA
  - El-Gamal

Recall: Distributed El-Gamal PKC

- Bob's Secret key b split between servers $Bob_1,\ldots Bob_N$
  - $b_i=f(i)$ s.t.
- Alice (sender) operates as usual (unaware key is shared):
  - Chooses r and $v= g^r \bmod p$
  - Encrypts: $c=m*B^r \bmod p$ where $B=g^b \bmod p$ is Bob
- Each server Bob_i computes $v_i = v^{s_i\zeta_i} = g^{r\cdot s_i\zeta_i}$
- Compute $k = \prod_{i=1}^{t} v_i = \prod_{i=1}^{t} g^{rs_i\zeta_i} = g^{r\cdot\sum_{i=1}^{t} s_i\zeta_i} = g^{r\cdot s} \bmod p$
- Decrypt: $m=c / k \bmod p$

Recall:

$$\left[\zeta_i = \prod_{\substack{j=1 \\ j\neq i}}^{t} \frac{j}{j-i}\right]$$

Proactive PKC, Signature Schemes

- Exactly the same…
- Except secret key shares are updated periodically
- El-Gamal, DSA signature only slightly more complex – jointly pick shared random secret *r*
  - Required for El-Gamal, DSA signatures
  - No server can control shared secret
  - Servers must provide good shares (VSS)
- Proactive RSA a bit more complex (but doable)

Proactive Security: Applications

- Proactive Secure Certificate Authority
  - Secure public key certificates and revocations
  - In particular: automated periodic refresh of private, public keys
  - For recovery from penetration of (certified) key
- To allow use of fixed public key for long time…
  - Use periodic private key, fixed public key
  - Generate periodic private key by multiple servers
  - Each server keeps share of `core` private key
  - Shared of `core` private key refreshed periodically
- Decentralized `Trusted Computing`
  - Secure `vault` for user's data/agents
  - Secure operating system, anti-virus and security programs

Applying Proactive Security

- Existing computers, operating systems are insecure
  - Most have weak security mechanisms, many `holes`
  - Most computers are not managed securely
  - Even patches for known security `holes` not installed
  - Once hacked, attackers plant `trapdoors` inside system code
  - No separation of system code, no `read only` support
- Idea: improve by proactive security
  - Periodically validate operating system, virus DB, etc.
  - Periodically, automatically install security patches, virus DB,…
- Problem: hacker may disable periodical checks…
- How to secure the boot process? – see `extras`

Conclusion

- Key exposure is a major threat
  - Existing operating systems are insecure
  - Insider attacks very common – as well as hacking…
- Solve by:
  - Limiting damage due to penetration
  - Redundancy – attacker must break over threshold
  - Temporal security models:
    - Proactive security – recovery from penetrations
    - Reasonably efficient signing and decryption
    - Standard RSA/DSA Public key operations (validate, encrypt)
    - Forward security – past protected from future exposure

Extras

Proactive Secure Boot & Refresh

- Assume hardware provides:
  - Invokes validation process before boot
  - Periodical boot or validation
  - Read-only memory for unique validation key of processor V_p
  - Matching signature key provided initially S_p
- Each network/organization maintains:
  - Public network/organization certification key V_{cert}
  - Shares of corresponding secret key in several servers
- Each computer should have:
  - Copy of V_{cert}
  - Method for validating V_{cert} at boot / periodical refresh

Validating V_{cert} using local keys

- $M_i = Sign<S_p>(V_{cert})$
- Store M_i in nearby servers for automatic recovery
  - Proactively maintain it valid
- Now erase S_p
  - So attacker cannot install fake V_{cert}
  - Possible hardware mechanism to generate new keys
- Upon periodical refresh or boot…
  - Validate M_i and thereby V_{cert}
  - If invalid, get valid M_i from peers
  - Validate information signed by V_{cert}, e.g. public keys, M_j (of j)
  - Send M_j to peer j