Resilience Despite Malicious Participants

Radia Perlman
radia.perlman@emc.com
EMC

This talk

- I’ll give a few examples to show the wildly different types of problems and solutions

Byzantine Failures

- Fail-stop: Something works perfectly, then halts
- Byzantine: Where something stops doing the right thing, but doesn't halt, for instance
  - Sends incorrect information
  - Computes incorrectly
- The term came from a famous paper where a bunch of processors try to agree on the value of a bit ("attack" or "retreat")
  - Lamport, L., Shostak, R., Pease, M. (1982). "The Byzantine Generals Problem", *ACM Transactions on Programming Languages and System*
- Misbehavior can cause problems even if not consciously malicious (bugs, misconfiguration, hardware errors)

Malicious Participants

- All sorts of things can be subverted with a small number of malicious participants
  - “How a Lone Hacker Shredded the Myth of Crowdsourcing”
    - https://medium.com/backchannel/how-a-lone-hacker-shredded-the-myth-of-crowdsourcing-d9d0534f1731

Malicious Participants

- All sorts of things can be subverted with a small number of malicious participants
  - “How a Lone Hacker Shredded the Myth of Crowdsourcing”
    - https://medium.com/backchannel/how-a-lone-hacker-shredded-the-myth-of-crowdsourcing-d9d0534f1731
- However…Things that shouldn’t work (but do)
  - Wikipedia
  - Ebay

I’ll talk about different examples

- PKI model resilient to malicious CAs
- Networks resilient to malicious switches
- Resilient and nonresilient designs for data storage with assured delete
- Human

Example 1: PKI

What’s PKI?

- “Public Key Infrastructure”
- A way for me to know your public key

Next topic: Trust Models for PKI

- Where damage from dishonest or confused CAs can be limited

Quick review of public keys, certificates, PKI, CAs

- Certification Authority (CA) signs “Certificates”

Alice’s Certificate, signed by CA

Name=Alice
Public key= 489024729

CA’s signature

Communication using certs

Alice Bob

"Alice", [Alice's key is 24789]$_{CA}$

"Bob", [Bob's key is 34975]$_{CA}$

mutual authentication, etc.

What people do think about

- Academics worry about the math
- Standards Bodies worry about the format of the certificate

What people do think about

- Academics worry about the math
- Standards Bodies worry about the format of the certificate
- Both are important, but people should also worry about the trust model
  - I will explain what that means

PKI Models

- Monopoly
- Oligarchy
- Anarchy
- Top-down, name constraints
- Bottom-up

Monopoly

- Choose one organization, for instance, “Monopolist.org”
- Assume Monopolist.org is trusted by all companies, countries, organizations
- Everything is configured to trust Monopolist.org’s public key
- All certificates must be issued by them
- Simple to understand and implement

Monopoly

Alice

Bob

Trust Monopolist.org

[This number is Bob’s key] signed by Monopolist.org

Monopoly: What's wrong with this model?

- No such thing as "universally trusted" organization
- Monopoly pricing
- More widely it's deployed, harder to change the CA key to switch to a different CA, or even to roll-over the key
- That one organization can impersonate everyone

Oligarchy of CAs

- Everything (say browsers) configured with 100 or so trusted CA public keys
- A certificate signed by any of those CAs is trusted
- Eliminates monopoly pricing

Oligarchy

Alice

Bob

Trust any of {CA1, CA2, CA3, …CAn}

[This number is Bob's key] signed by CAi

What’s wrong with oligarchy?

- Less secure!
  - Any of those organizations can impersonate anyone

Important Enhancement: Certificate Chains

- Instead of presenting a certificate signed by a CA Alice knows and trusts, Bob presents a chain of certs, starting with X1, which Alice trusts

Certificate chains

Alice

Trust X1

Bob

[X1 says α is X2's key] signed by X1's key

[X2 says β is X3's key] signed by α

[X3 says δ is Bob's key] signed by β

Certificate chains

Alice

Trust X1

Bob

[X1 says α is X2's key] signed by X1's key

[X2 says β is X3's key] signed by α

[X3 says δ is Bob's key] signed by β

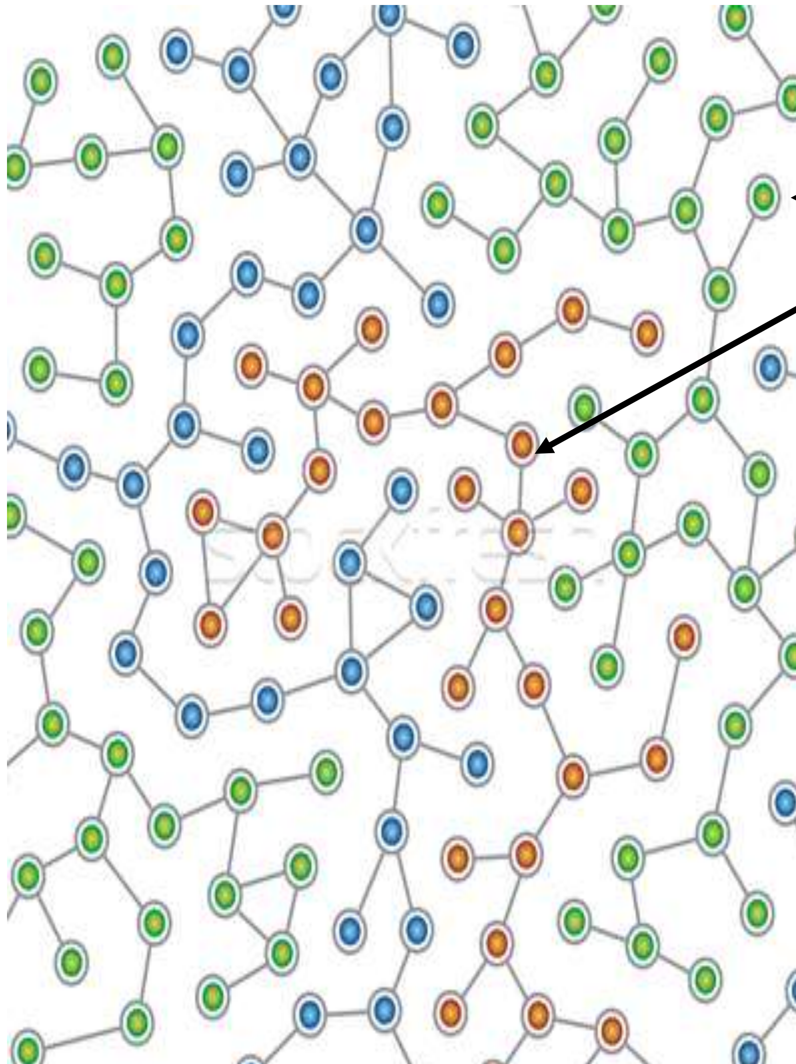
Next model: Anarchy

Anarchy

- User personally configures trust anchors
- Anyone signs certificate for anyone else
- Public databases of certs (read and write)
- Alice tries to find a path from a key her machine knows, to the target name, by piecing together a chain

Unstructured certs, public database

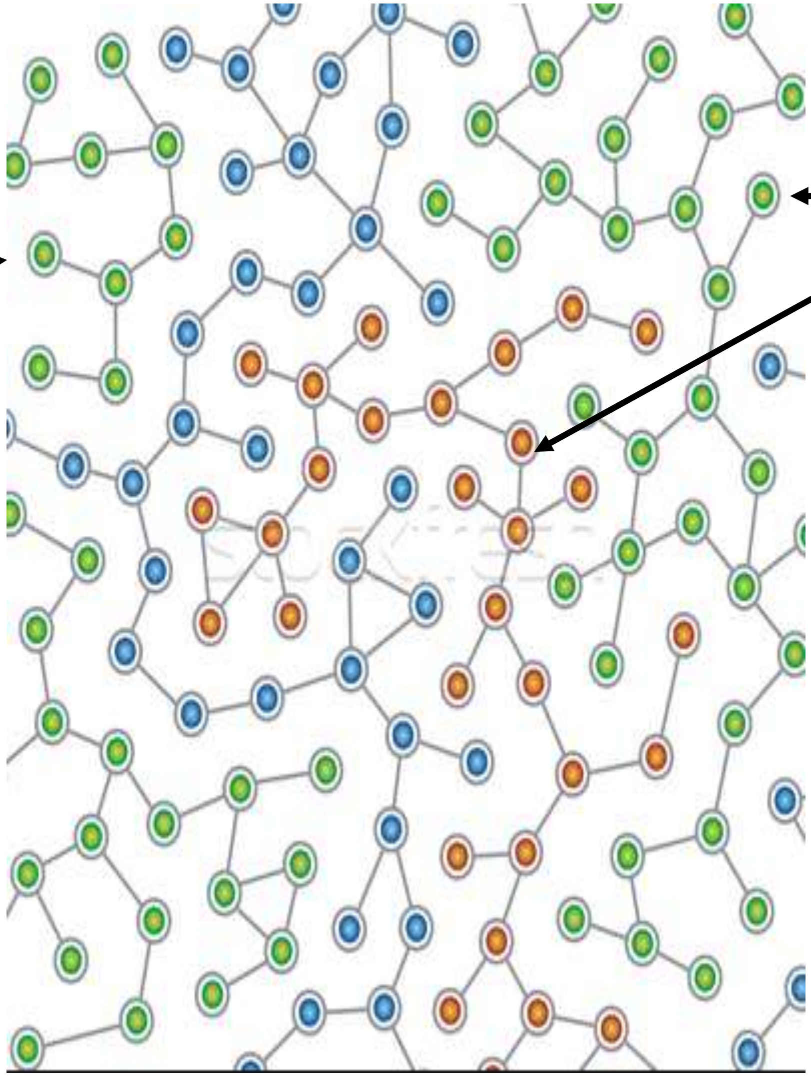
Alice wants Bob’s key



Alice configured with these

Unstructured certs, public database

Alice wants Bob's key



This cert says α is Bob's key

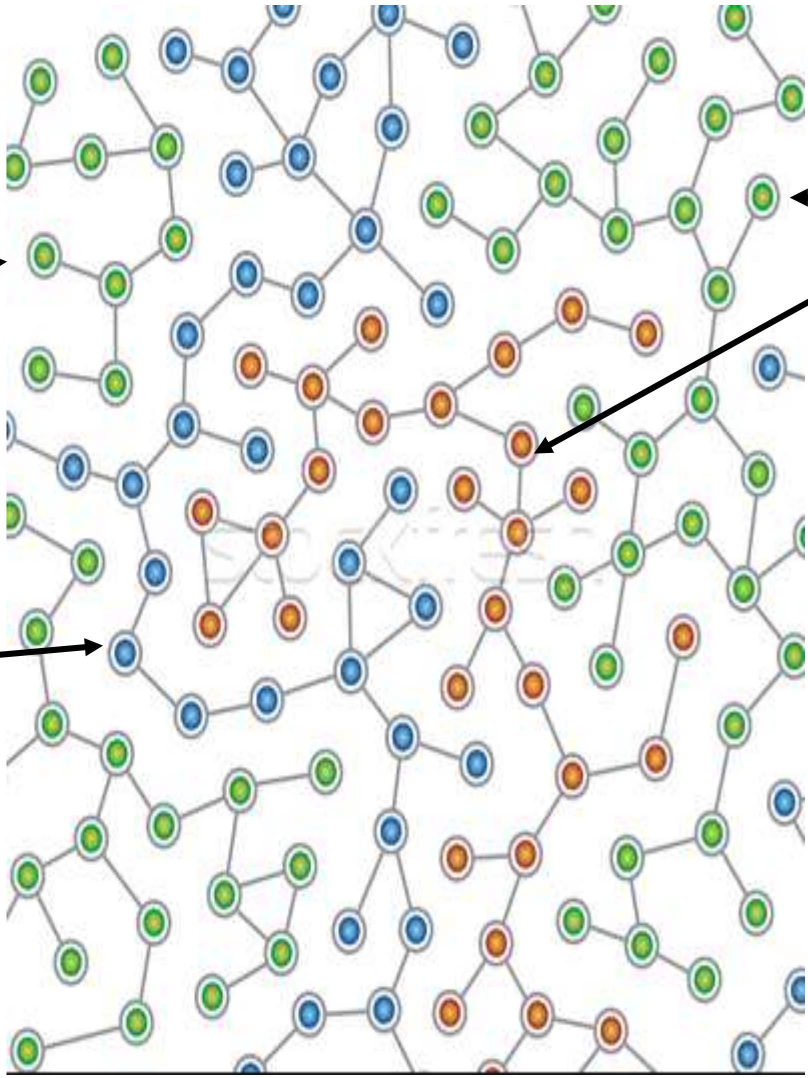
Alice configured with these

Unstructured certs, public database

Alice wants Bob’s key

This cert says α is Bob’s key

This cert says β is Bob’s key



Alice configured with these

Anarchy

- Problems
  - won’t scale (too many certs, computationally too difficult to find path)
  - no practical way to tell if path should be trusted
  - (more or less) anyone can impersonate anyone

Now I’ll talk about how I think it should work

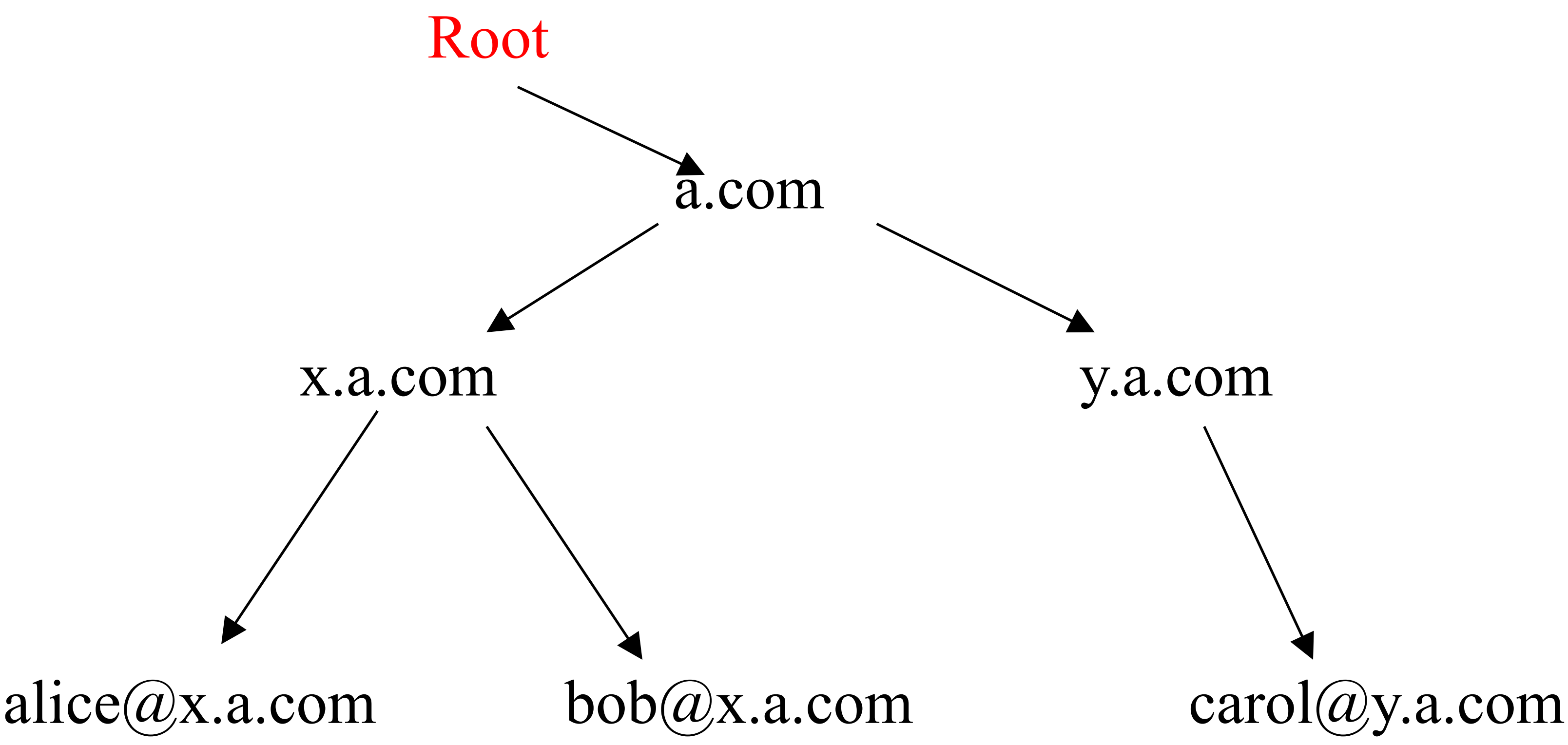
Now getting to recommended model

- Important concept:
  - CA trust isn't binary: "trusted" or "not"
- CA only trusted for a portion of the namespace
  - The name by which you know me implies who you trust to certify my key
    - Radia.perlman.emc.com
    - Roadrunner279.socialnetworksite.com
    - Creditcard#8495839.bigbank.com
  - Whether these identities are the same carbon-based life form is irrelevant

Need hierarchical name space

- Yup! We have it (DNS)
- Each node in namespace represents a CA

Top-down model (almost what we want)



Top-down model

- Everyone configured with root key
- Easy to find someone's public key (just follow namespace)

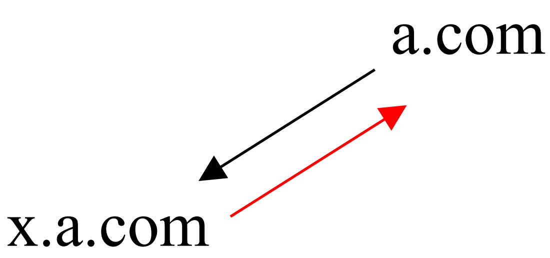
Top-down model

- Everyone configured with root key
- Easy to find someone's public key (just follow namespace)
- Problems:
  - Still monopoly at root
  - Root can impersonate everyone
  - Every CA on path from Root to target can impersonate target node

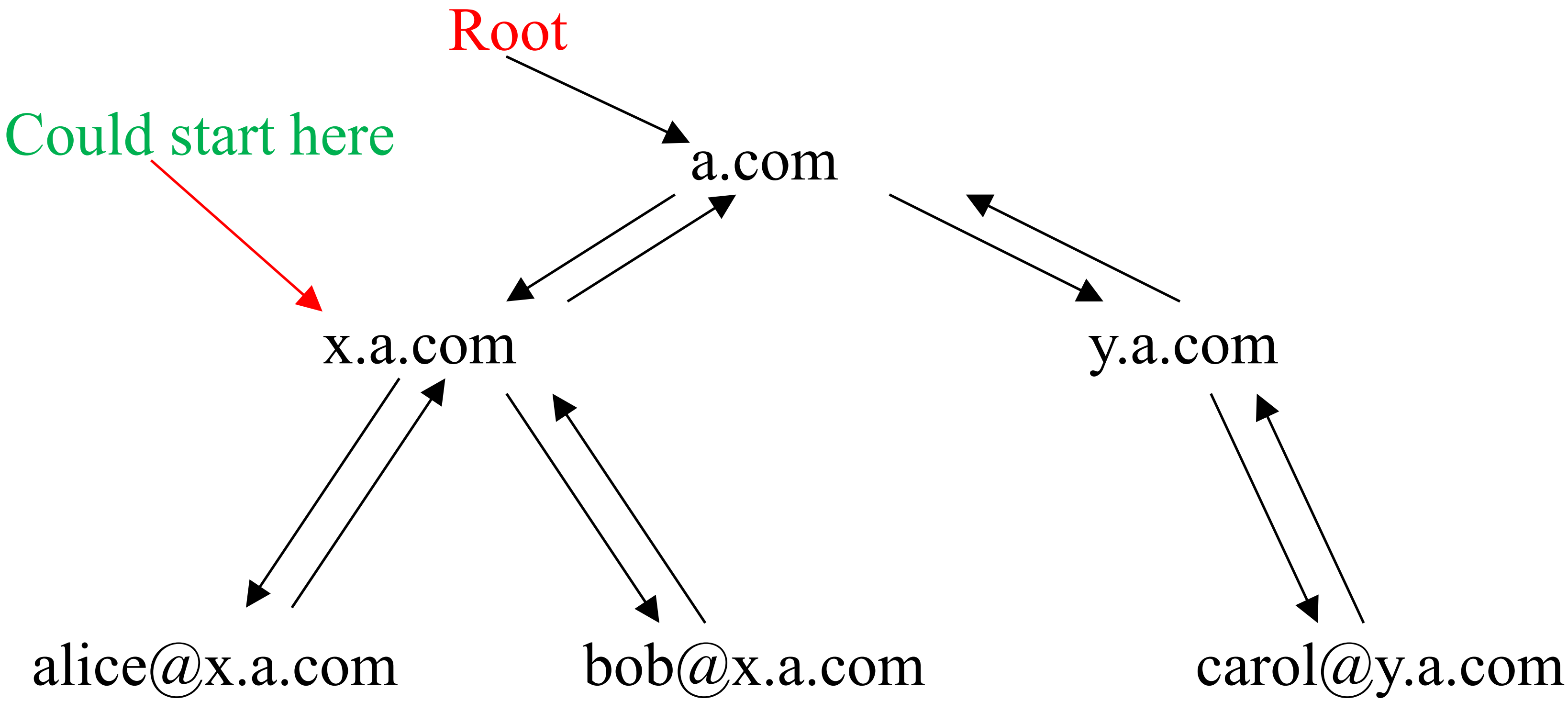
Bottom-Up Model (what I recommend)

Two-way certificates (up and down)

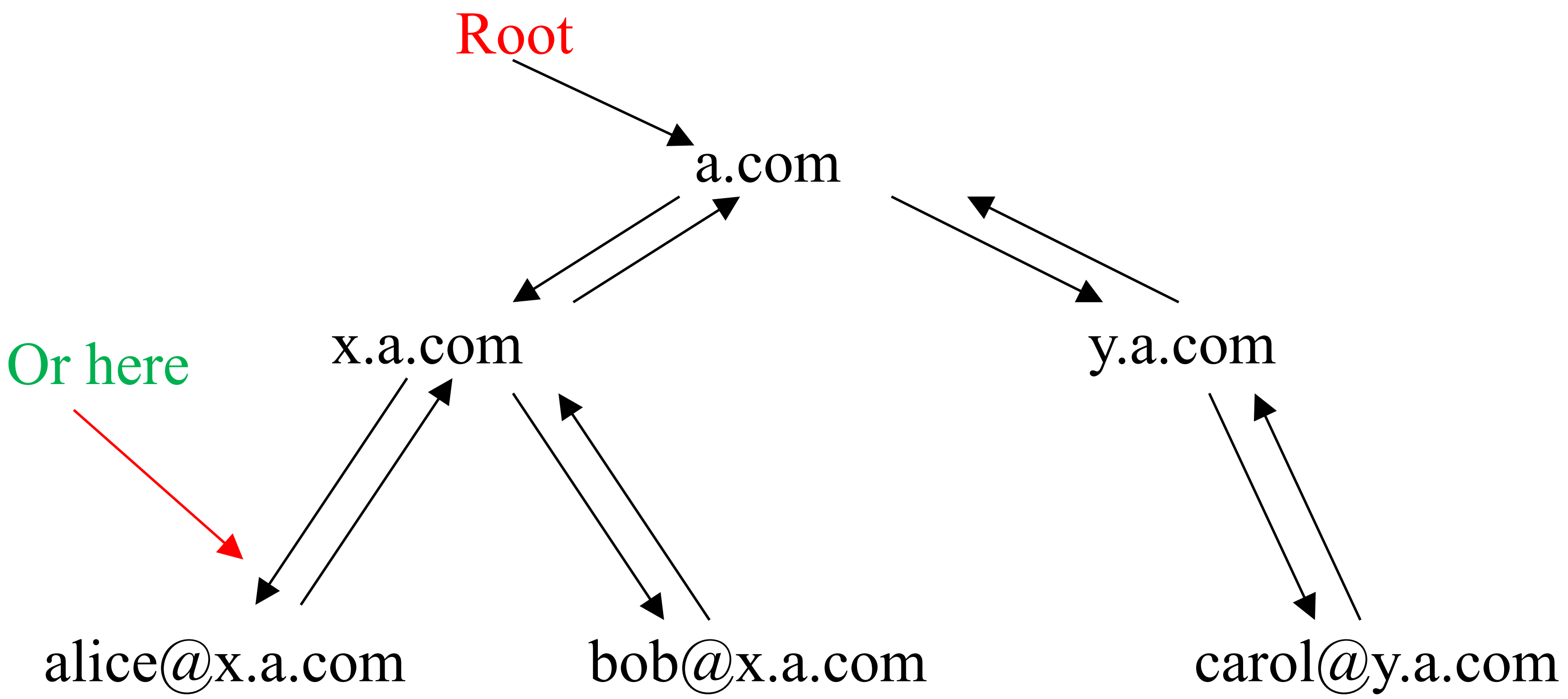
- Each arc in name tree has parent certificate (up) and child certificate (down)



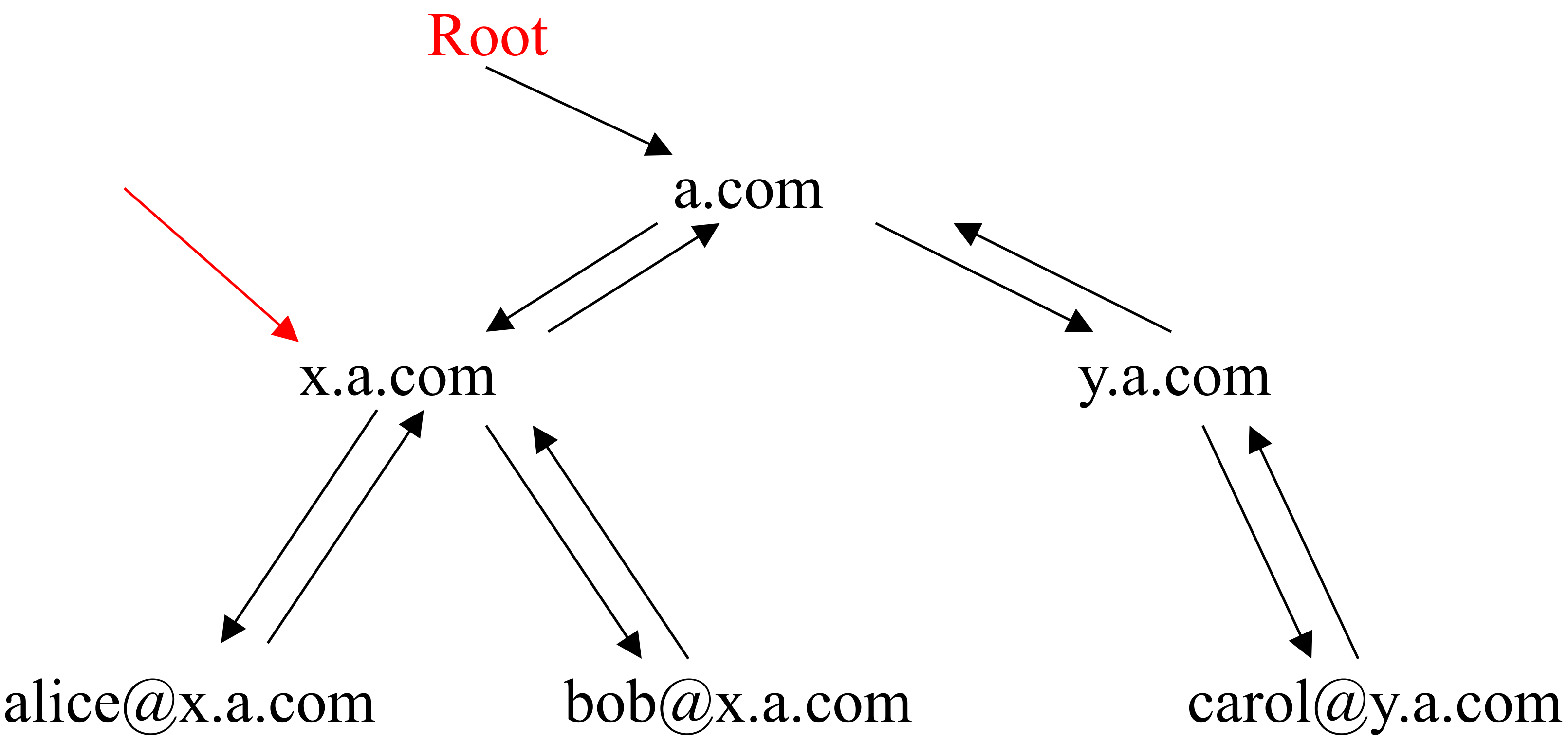
No need to start at the Root



No need to start at the Root



No need to start at the Root

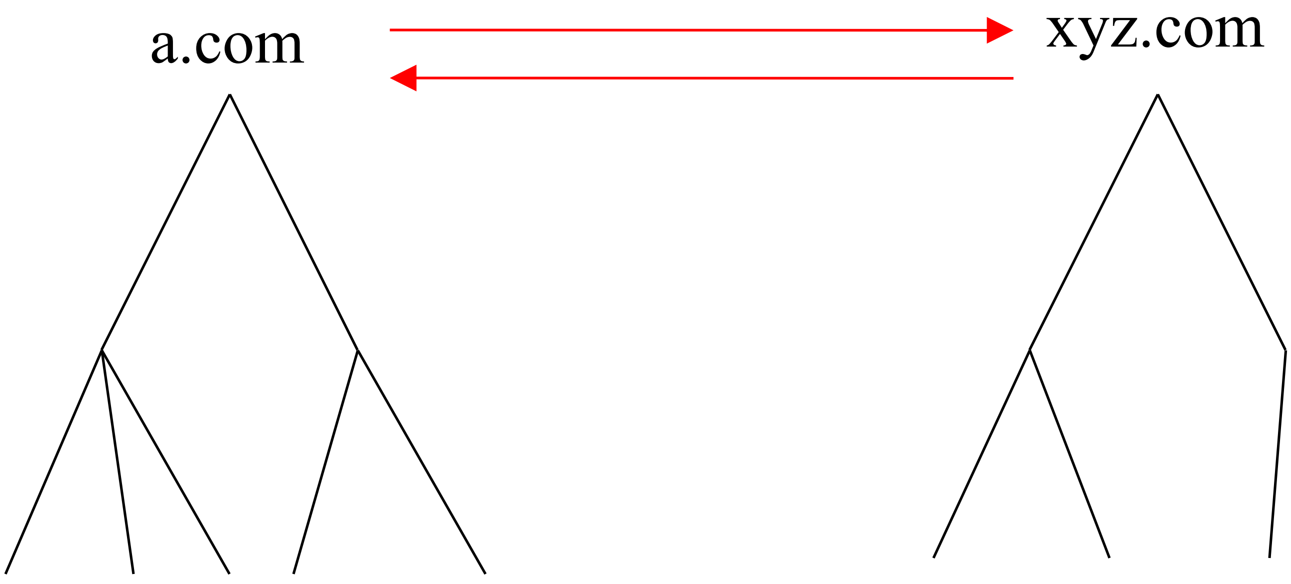


In subtree below x.a.com, fewer CAs to trust
(a.com, and Root, aren’t on path to nodes in subtree)

Another enhancement: “Cross-certificates”

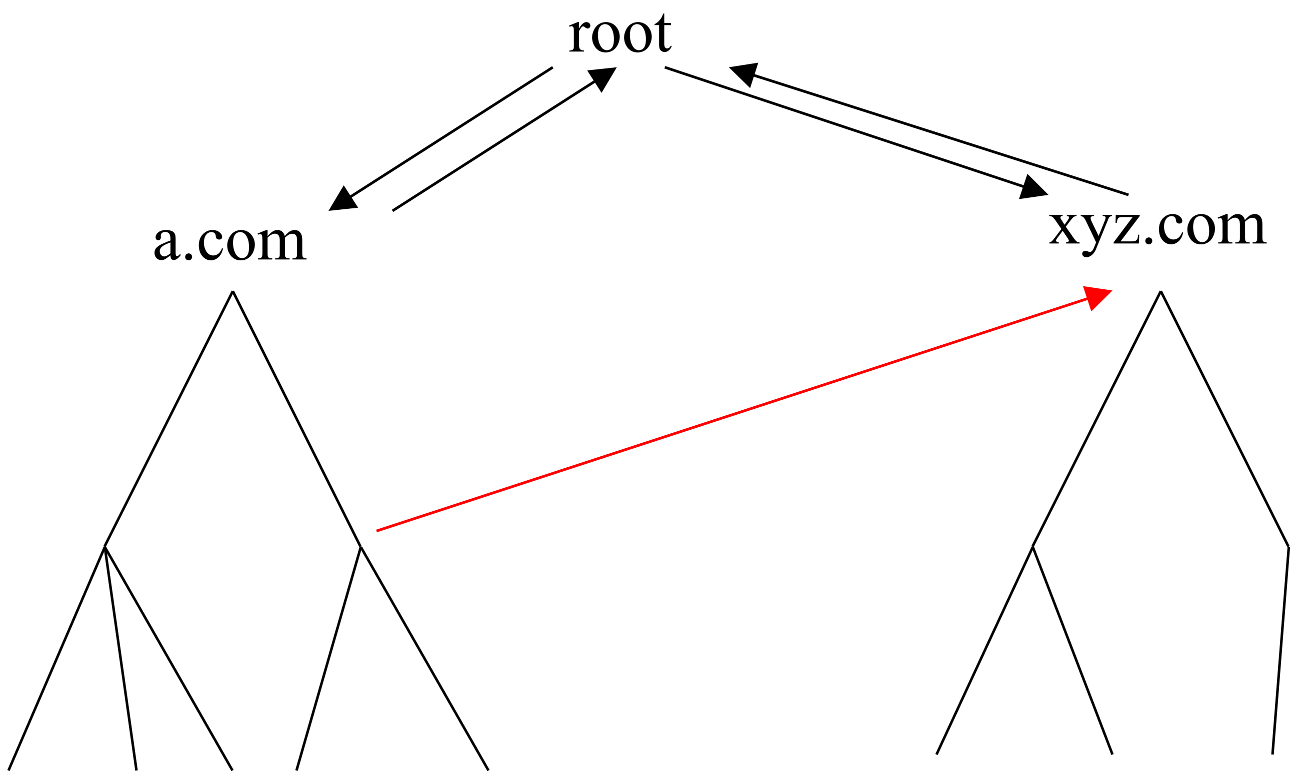
- Cross-cert: Any node can certify any other node’s key
- Two reasons:
  - So you don’t have to wait for PKI for whole world to be created first
  - Can bypass hierarchy for extra security

Cross-links to connect two organizations

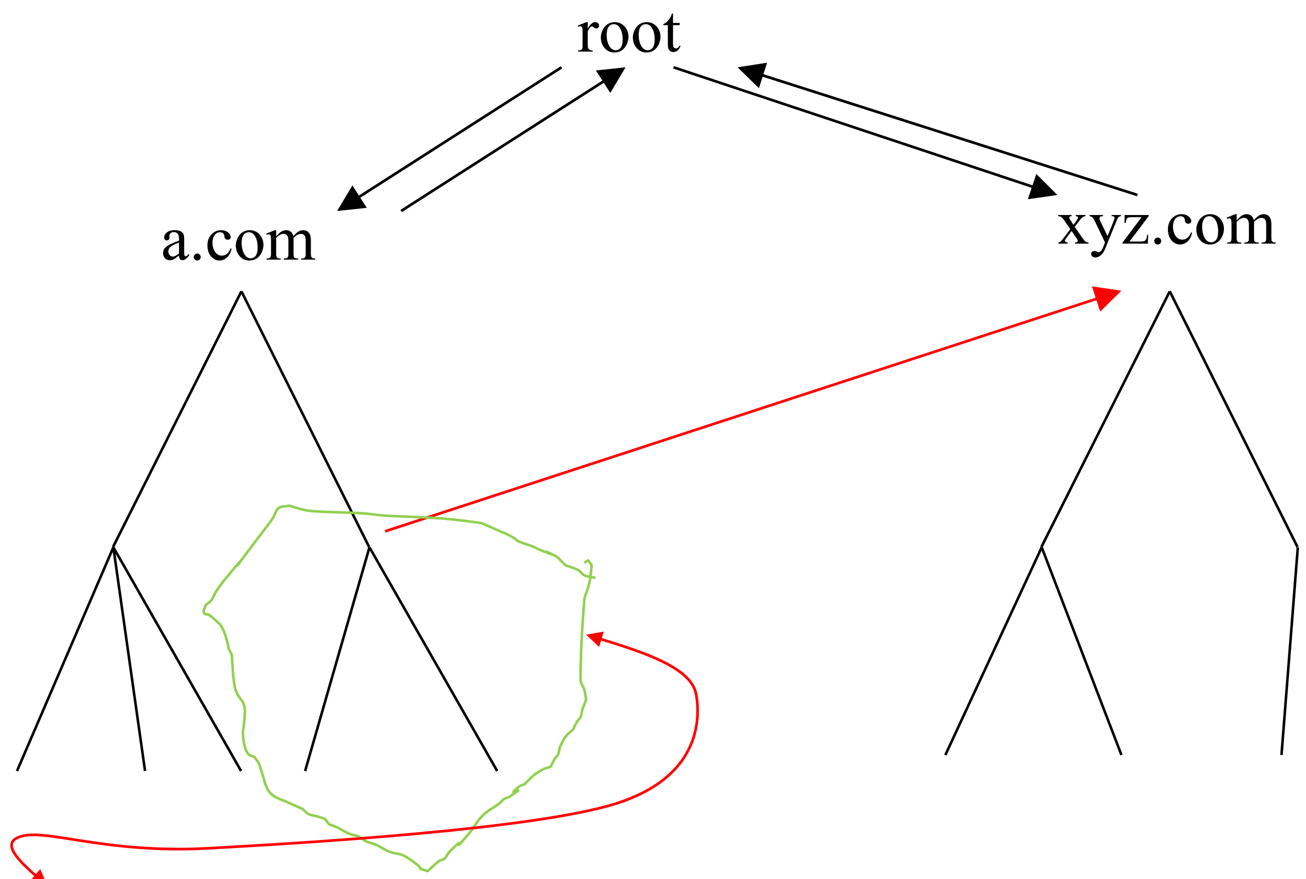


Nodes in a.com and xyz.com subtrees can find each other's key.
No need for Root, or entire connected PKI

Cross-link for added security



Cross-link for added security



Nodes in this subtree can bypass root, and a.com

Navigation Rules

- Start somewhere (your “root of trust” .. could be your own public key)
- If you’re at an ancestor of the target node, follow down-links
- Else, look for cross-link to ancestor, and if so, follow that
- Else, go up a level

Note: Crosslinks do not create anarchy model

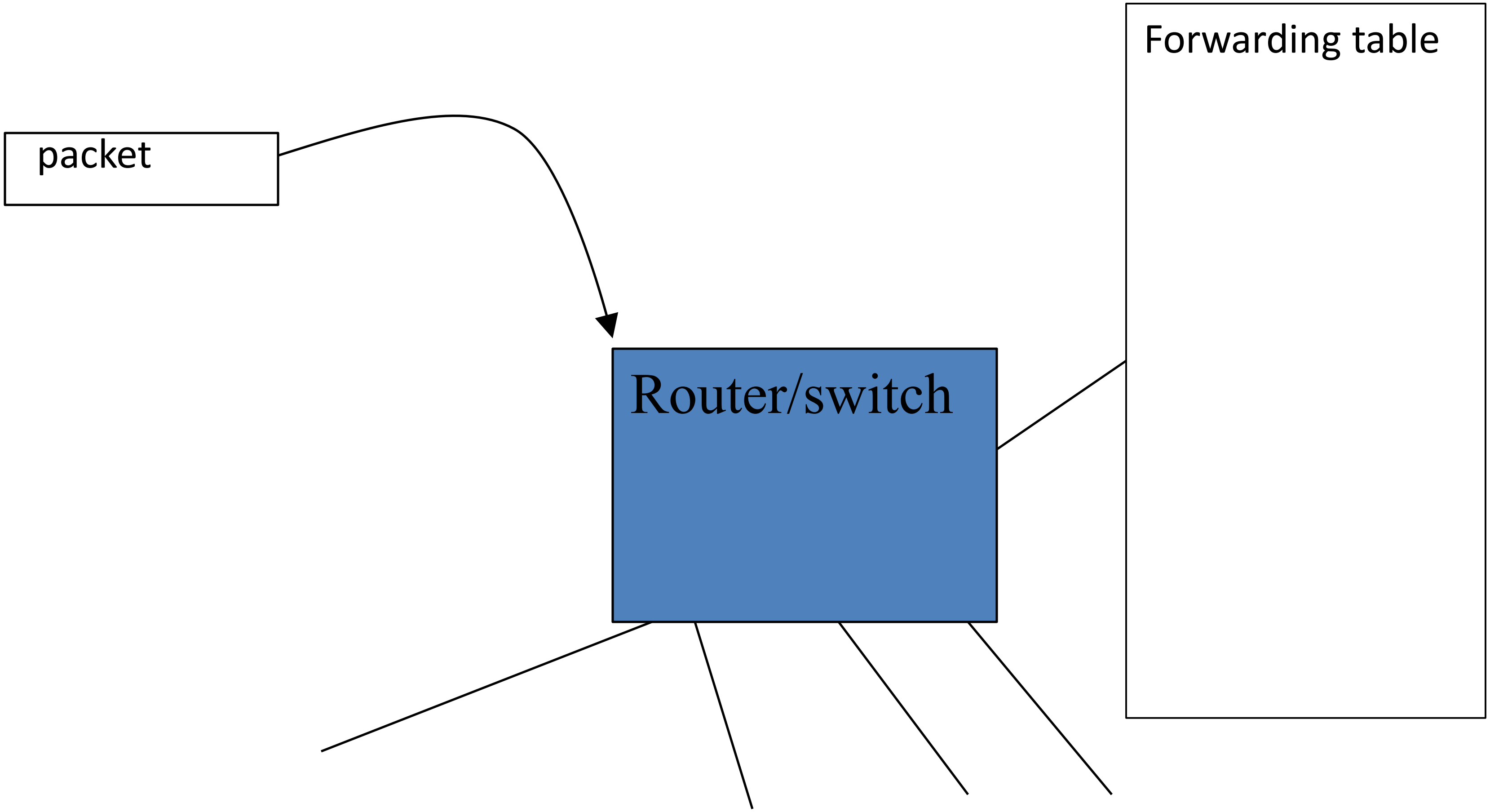
- You only follow a cross-link if it leads to an ancestor of target name

Advantages of Bottom-Up

- Security within your organization is controlled by your organization (CAs on path are all yours)
- No single compromised key requires massive reconfiguration
- Easy to compute paths; trust policy is natural, and makes sense
- Malicious CA’s can be bypassed, and damage contained

Example 2: Network Routing

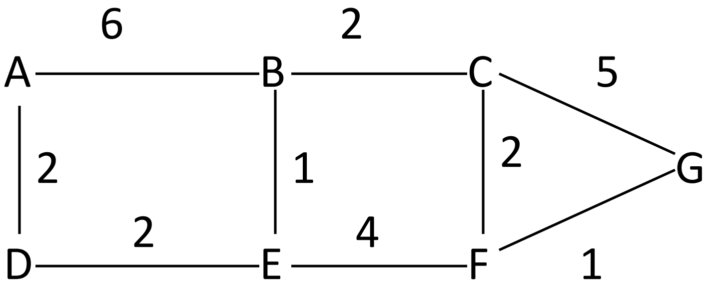
Traditional Router/switch



Computing the Forwarding Table

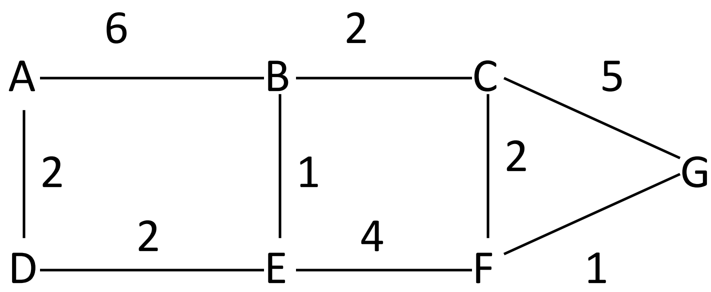
- Distributed computation of forwarding tables with link state protocol

Link State Routing



A network

Link State Routing



A	B	C	D	E	F	G
B/6	A/6	B/2	A/2	B/1	C/2	C/5
D/2	C/2	F/2	E/2	D/2	E/4	F/1
	E/1	G/5		F/4	G/1	

What about malicious switches?

- They can
  - Give false info in the routing protocol
  - Flood the network with garbage data
  - Forward in random directions, resetting the hop count on packets to look new
  - Do everything perfectly, but throw away traffic from a particular source

All sorts of traditional different approaches

- Try to agree who the bad guy(s) are
  – Reputation (problems: who do you believe, bad guys can create arbitrarily many identities, what if bad guy is only bad to one source?)
  – Troubleshooting (can be well-behaved when testing)
- Enforce routing protocol correctness
  – 2-way links
  – S-BGP
  – But that's just routing protocol. Who cares about that? You want your packets delivered.

My thesis (1988)

- Want to guarantee A and B can talk provided at least one honest path connects them
  - With reasonably fair share of bandwidth
  - “Honest path” means all switches on that path are operating properly

Flooding

- Transmit each packet to each neighbor except the one from which it was received
- Have a hop count so packets don't loop infinitely
- This works! Pkts between A and B flow, if there is at least one nonfaulty path…

Flooding

- Transmit each packet to each neighbor except the one from which it was received
- Have a hop count so packets don't loop infinitely
- This works! Pkts between A and B flow, if there is at least one nonfaulty path…
- *If there is infinite bandwidth….whoops!*

So, just a resource allocation problem

- The finite resources are
  - computation in switches
    - assume we can engineer boxes to keep up with wire speeds
  - memory in switches
  - bandwidth on links

Byzantinely Robust Flooding

- Memory
  - reserve a buffer for each source
- Bandwidth
  - round-robin through buffers

Byzantinely Robust Flooding

- Source signs packet
  - (prevent someone occupying source's buffer)
- Put sequence number in packet
  - (prevent old packets reinjected, starving new one)

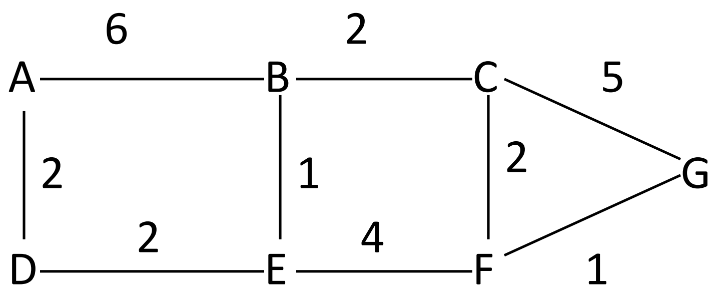
Configuration

- Every node needs other nodes' public keys; would be a lot of configuration
- So instead have "trusted node" TN (similar function to a CA)
  - TN knows all other nodes' public keys
  - All other nodes need their own private key, and the trusted node public key
- Since everyone knows TN's public key, TN can flood
  - Info it floods: all nodes' public keys

Inefficient to send data with flooding

- So, we'll do something else for unicast
- But we will use robust flooding for two things
  - easing configuration (advertising public keys)
  - distributing link state information

Link State Routing



A	B	C	D	E	F	G
B/6	A/6	B/2	A/2	B/1	C/2	C/5
D/2	C/2	F/2	E/2	D/2	E/4	F/1
	E/1	G/5		F/4	G/1	

Data Packets/unicast

- "Traditional" per-destination forwarding won't work
  - Bad guy can keep network in flux by flipping state of a link
  - What do you do if path works for everyone but S?

Data Packets/unicast

- “Traditional” per-destination forwarding won’t work
  - Bad guy can keep network in flux by flipping state of a link
  - What do you do if path works for everyone but S?
- Conclusion: Source has to choose its own path

Data packets

- Source chooses a path
- Sets it up with a cryptographically signed setup packet, specifying the path
- Routers along the path have to keep per (S,D) pair
  - Input port
  - Output port
  - Buffers for data packet fwd'ed on this flow

Unicast Forwarding

- No crypto needed
- Just additional check "is it coming from expected port?"
- As long as path is honest, no malicious switch off the path can disrupt flow

Simple heuristics for S choosing a path that works for S

- If path to D works (end-to-end acks), then have more trust in routers along that path
- If path doesn't work, be suspicious of the routers on that path
- Try to eliminate routers one at a time, but if lots of bad guys, can be really expensive

Note this isn't too scalable

- Since every path requires state
- And requires source seeing entire path (which it can't in hierarchical network)
- More recent work fixes that
  - Perlman, R., Kaufman, C., "Byzantine Robustness, Hierarchy, and Scalability", IEEE Conference on Communications and Network Security, CNS 2013.

Resilience

- Source has fate in its own hands: no need for “agreement”
- Malicious TN: have multiple and vote, or just reserve resources for any public key advertised (and malicious TN can’t use up more than ½ the resources, and will be quickly caught)

Example 3: Data: Making it be there when you want it, and making it go away when you want it gone

Resilient expiring data

- Paper "File System Design with Assured Delete"
  - https://www.isoc.org/isoc/conferences/ndss/07/papers/file_system_assured_delete.pdf

Expiration time

- When create data, put (optional) "expiration date" in metadata
- After expiration, data must be unrecoverable, even though backups will still exist

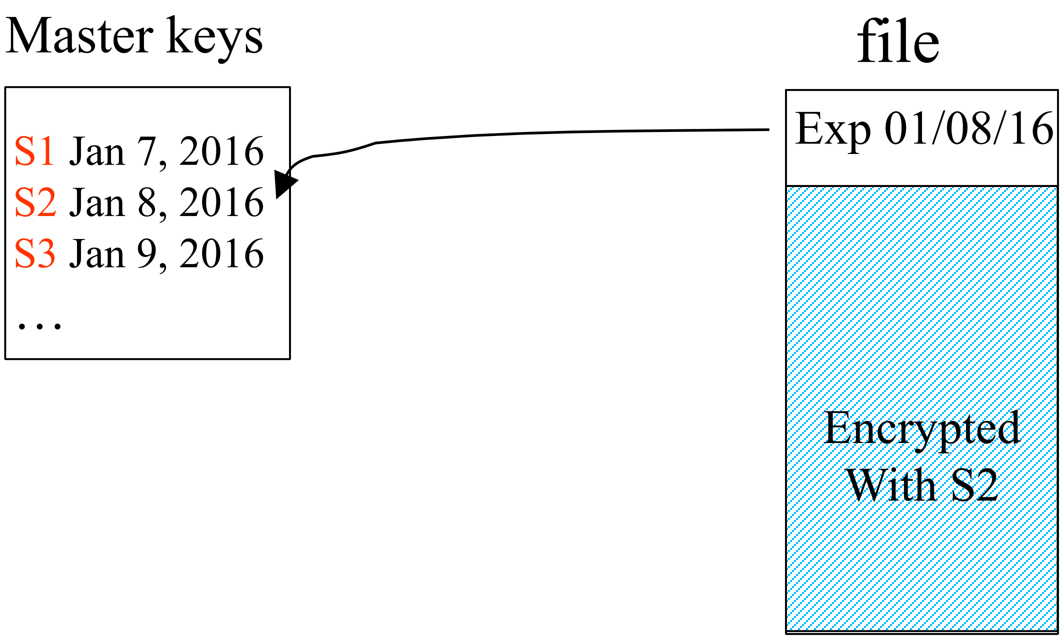
Obvious approach

- Encrypt the data, and then destroy keys
- But to avoid prematurely losing data, you'd have to make lots of copies of the keys
- Which means it will be difficult to ensure all copies of backups of expired keys are destroyed

First concept: Encrypt all files with same expiration date with the same key

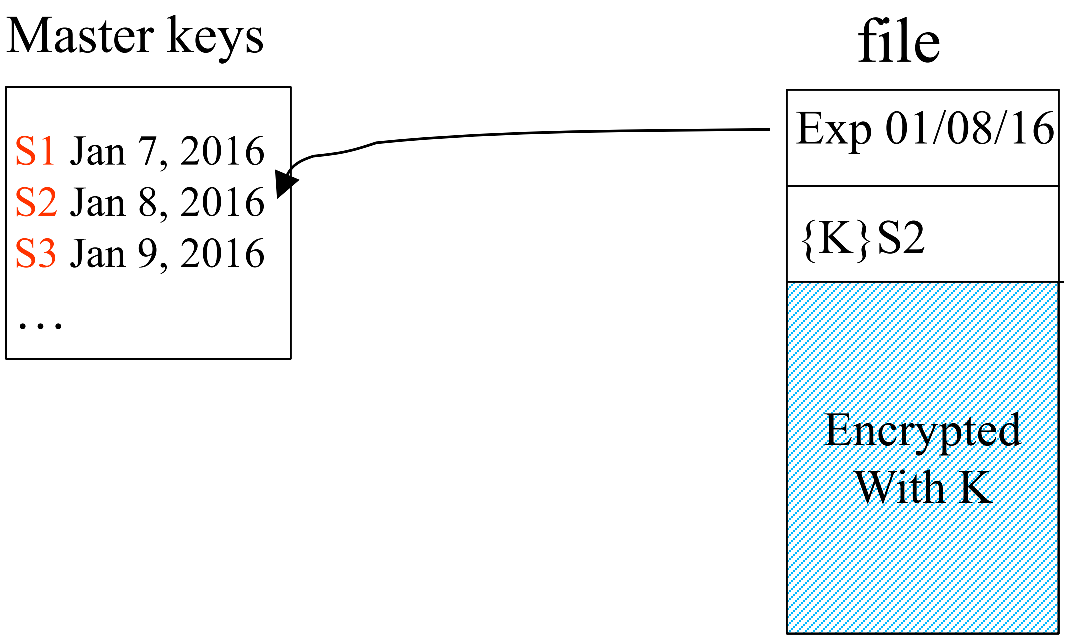
File system with Master keys

Master keys: Secret keys (e.g., AES)
generated by storage system
Delete key upon expiration



File system with Master keys

Master keys: Secret keys (e.g., AES)
generated by storage system
Delete key upon expiration



How many keys?

- If granularity of one per day, and 30 years maximum expiration, 10,000 keys

So…how do you back up the master keys?

Imagine a service: An "ephemerizer"

- creates, advertises, protects, and deletes public keys
- Storage system "ephemerizes" each master key on backup, by encrypting with (same expiration date) ephemerizer public key
- To recover from backup: storage system asks ephemerizer to decrypt

Ephemerizer publicly posts

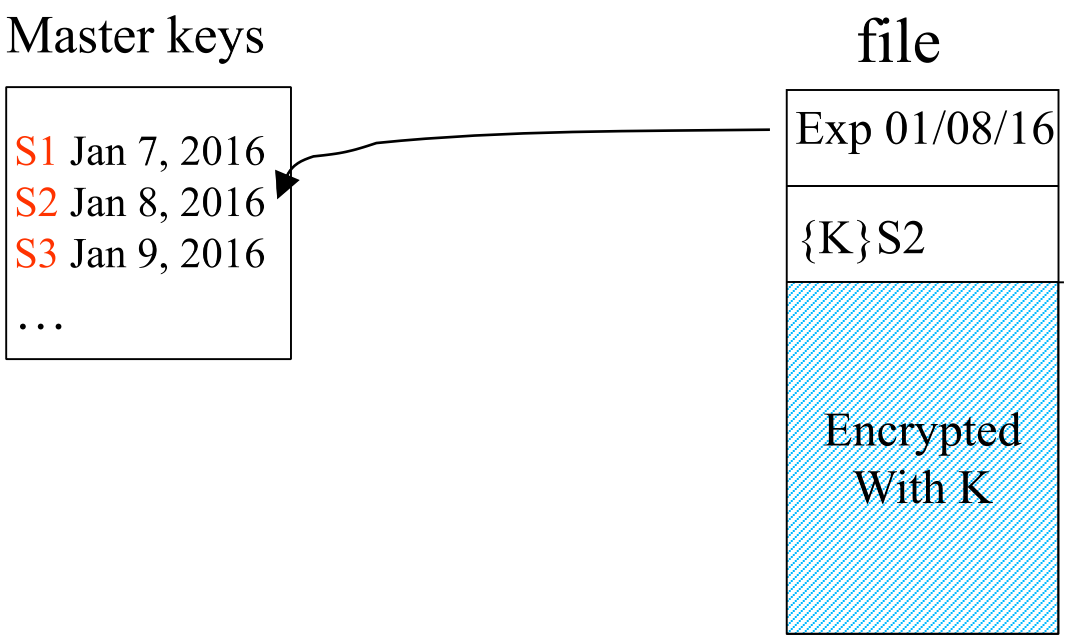
Jan 7, 2016: public key $P_{Jan7of2016}$
Jan 8, 2016: public key $P_{Jan8of2016}$
Jan 9, 2016: public key $P_{Jan9of2016}$
Jan 10, 2016: public key $P_{Jan10of2016}$

etc

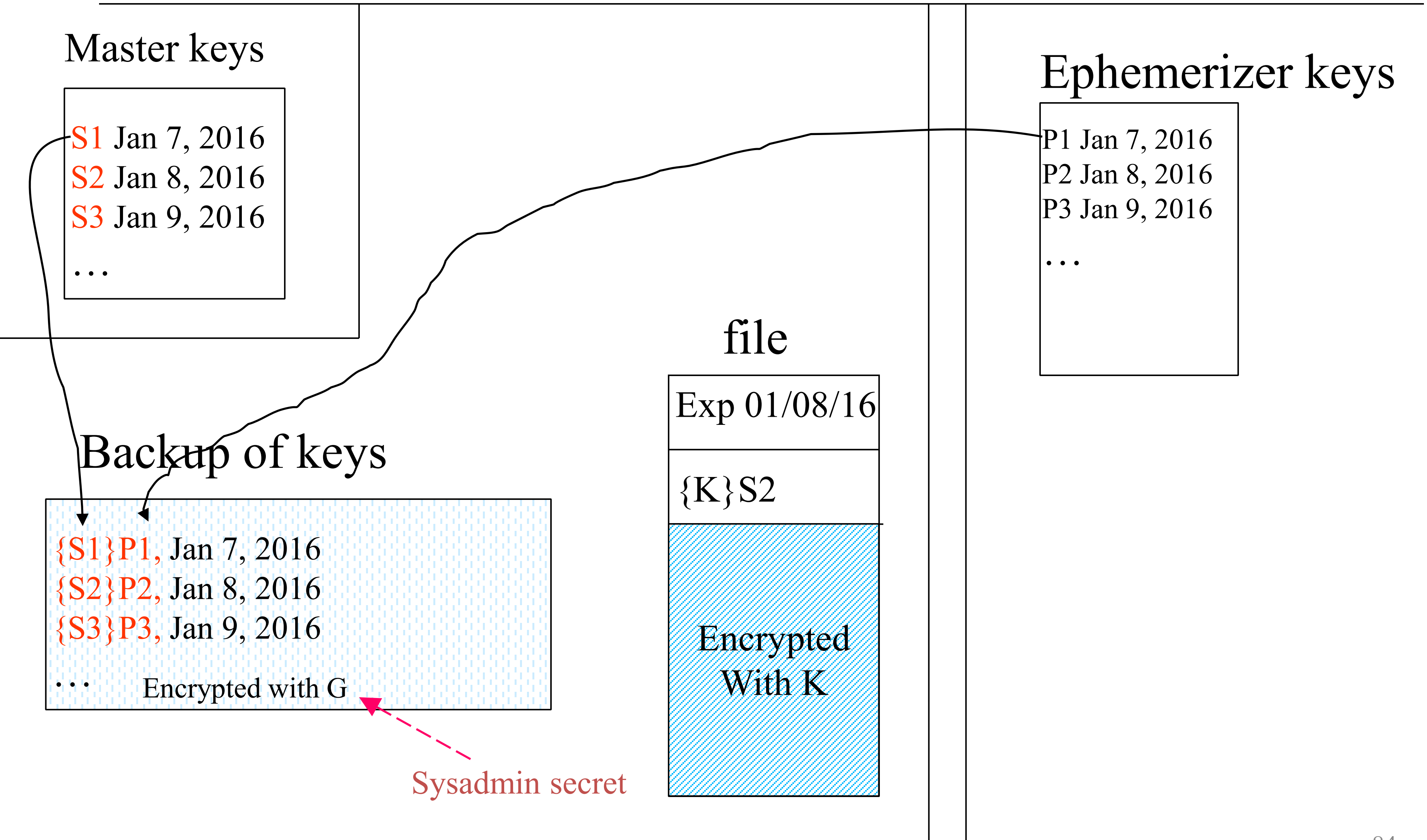
One permanent public key P certified through PKI
Signs the ephemeral keys with P

Storage system with Master keys

Master keys: Secret keys (e.g., AES) generated by storage system



Backup of Master Keys



Notes

- Only talk to the ephemerizer if your hardware with master keys dies, and you need to retrieve master keys from backup
- Ephemerizer really scalable:
  – Same public keys for all customers (10,000 keys for 30 years, one per day)
  – Only talk to a customer perhaps every few years…to unwrap keys being recovered from backup

But you might be a bit annoyed at this point

But you might be a bit annoyed at this point

- Haven’t we simply pushed the problem onto the ephemerizer?
- It has to reliably keep private keys until expiration, and then reliably delete them

Two ways ephemerizer can “fail”

- Prematurely lose private keys
- Fail to forget private keys

Two ways ephemerizer can "fail"

- Prematurely lose private keys
- Fail to forget private keys
- Let's worry about these one at a time…first worry about losing keys prematurely

Losing keys prematurely

- We will allow an ephemerizer to be flaky, and lose keys
- Generate keys, and do decryption, on tamper-proof module
- An honest ephemerizer should not make copies of its ephemeral private keys
- So…wouldn't it be a disaster if it lost its keys when a customer needs to recover from backup?

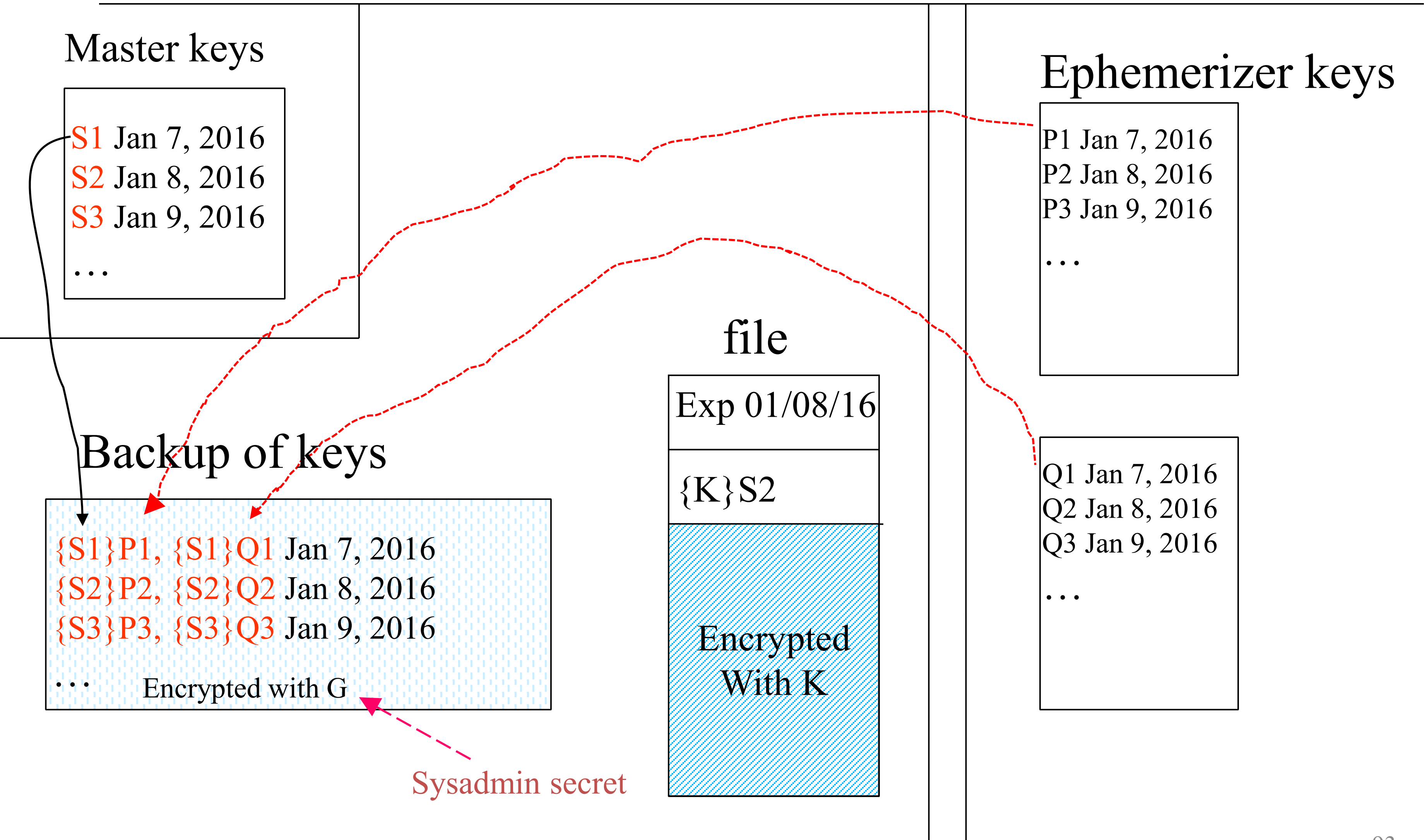
Question: How many copies of private keys should ephemerizer keep so you feel safe?

- Let’s say 20

The reason why it's not just pushing the problem

- You can achieve arbitrary robustness by using enough "flaky" ephemerizers!
  - Independent ephemerizers
    - Different organizations
    - Different countries
    - Different continents
  - Independent public keys

Use multiple ephemerizers!



What if ephemerizer doesn't destroy private key when it should?

- Then the storage system can use a quorum scheme (k out of n ephemerizers)
  - Break master key into n pieces, such that a quorum of k can recover it
  - Encrypt each piece with each of the n ephemerizers' public keys

So, after disaster, 10,000 decryptions

- Not so bad, but we can do better

No reason keys have to be independent random numbers

- We could make day n+1 key one-way hash of day n key (or store it encrypted with day n's key)
- Then we only need to ask for a single decryption after a disaster (the one closest to expiration)
- And we can locally derive the rest of the keys

Ephemerizer decryption protocol

- Protocol for asking ephemerizer to decrypt
  - That doesn’t let the ephemerizer see what it’s decrypting
  - Doesn’t require authentication of either end
  - Super light-weight (can be one IP packet each direction, very little computation)

I’ll skip over how

- Because I’m sure we’ll be short on time
- But I’m leaving the slides there

What we want to accomplish

File system

Ephemerizer

Has $\{S_i\}P_i$

Please decrypt $\{S_i\}P_i$ with key ID i $\rightarrow$

use private key i

$\leftarrow S_i$

What we want to accomplish

File system

Ephemerizer

Has $\{S_i\}P_i$

Please decrypt $\{S_i\}P_i$ with key ID i $\longrightarrow$

use private key i

$\longleftarrow S_i$

But we don't want the Ephemerizer to see S_i

We’ll use “blind decryption”

- FS wants Eph to decrypt {Si}Pi with Eph’s private key #i
  - … Without Eph seeing what it is decrypting
- FS chooses inverse functions
  - “blind/unblind” (B, U)
- encrypts (blinds) with Blind Function, which commutes with Eph’s crypto
- Then FS applies U to unblind

Using Blind Decryption

File system

Ephemerizer

Has $\{S_i\}P_i$

Invents functions (B,U) just for this conversation

Please decrypt **B**$\{\{S_i\}P_i\}$ with key ID i

use private key i

B$\{S_i\}$

File system applies U to get S_i

Ephemerizer only sees B$\{S_i\}$

Non-math fans can take a nap

For you math fans…

Quick review of RSA

- Public key is (e,n). Private key is (d,n), where e and d are "exponentiative inverses mod n"
- That means X^{ed} mod n=X
- Encrypt X with public key (e,n) means computing X^e mod n

Blind Decryption with RSA, Eph's RSA PK=(e,n), msg=M

File System

Ephemerizer

wants to decrypt M^e mod n

chooses R, computes R^e mod n

$M^e \ R^e$ mod n $\longrightarrow$

applies (d,n)

$M^{ed}R^{ed}$

$\longleftarrow$ M R mod n

divides by R mod n to get plaintext M

Properties of our protocol

- Ephemerizer gains no knowledge when it is asked to do a decryption
- Protocol is really efficient: one IP packet request, one IP packet response
- No need to authenticate either side
- Decryption can even be done anonymously

OK, non-math fans can wake up now

Because of blind decryption

- The customer does not need to run its own Ephemerizers, or really trust the Ephemerizers very much
- Ephemeral key management can be outsourced

General philosophy

- Achieve robustness by lots of can-be-flaky components
- Failures are truly independent
  - Different organizations
  - Different administrators
  - Independent clocks

In contrast, a non-resilient solution

People kept wanting “on-demand” delete

- And I kept arguing that it was not useful, and wouldn’t be scalable

People kept wanting "on-demand" delete

- And I kept arguing that it was not useful, and wouldn't be scalable
- But then I realized how to do it

People kept wanting "on-demand" delete

- And I kept arguing that it was not useful, and wouldn't be scalable
- But then I realized how to do it
- And think it's a really bad idea

People kept wanting “on-demand” delete

- And I kept arguing that it was not useful, and wouldn’t be scalable
- But then I realized how to do it
- And think it’s a really bad idea
- And it’s useful to see both how to do it, and why it’s a bad idea

On-demand delete

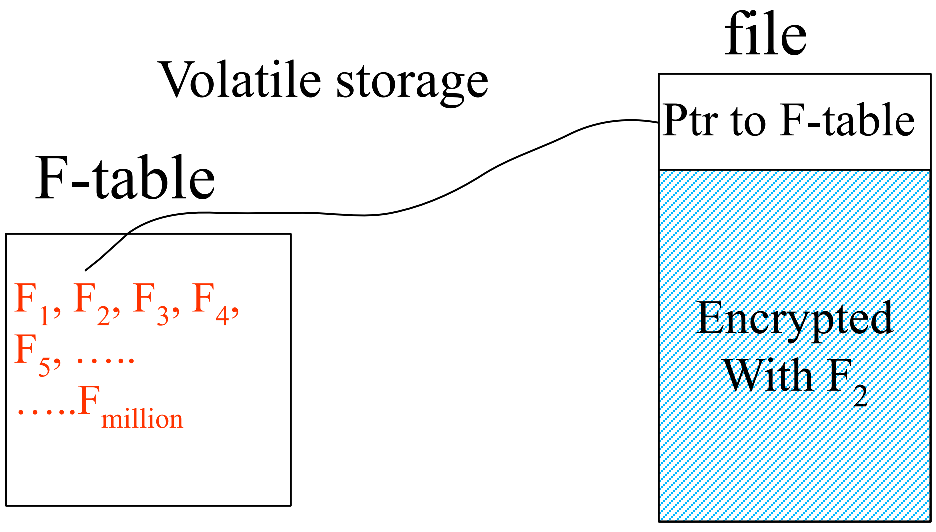
Instead of master keys

- Storage system keeps “F-table”, consisting of a secret key for each (expirable) piece of data
- Adds key to F-table when new (expirable) data stored
- Deletes key from F-table when (expirable) data is assuredly-deleted

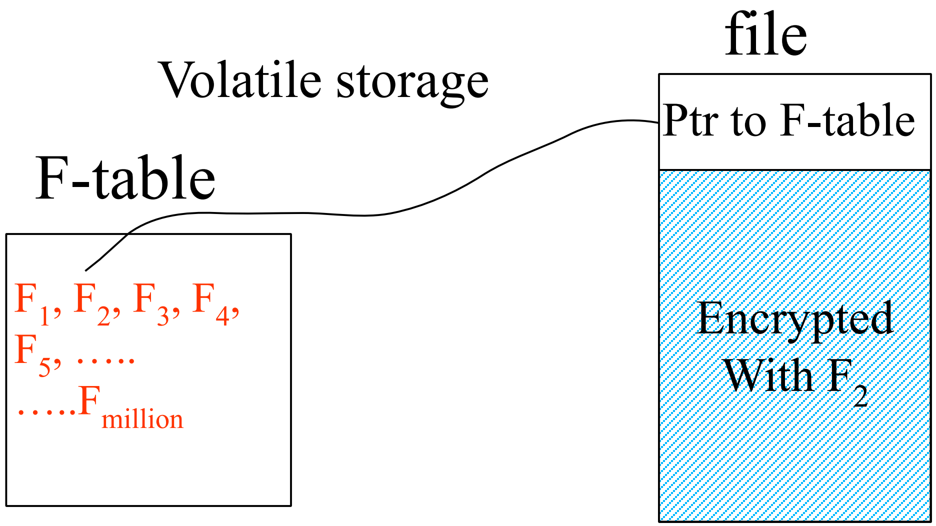
Ephemerizer state

- In time-based system, ephemerizer didn't need to know its customers
- For the on-demand system, ephemerizer needs to keep two public keys for each customer file system
  - current public key (P_n)
  - previous public key (P_{n-1})

File system with F-table

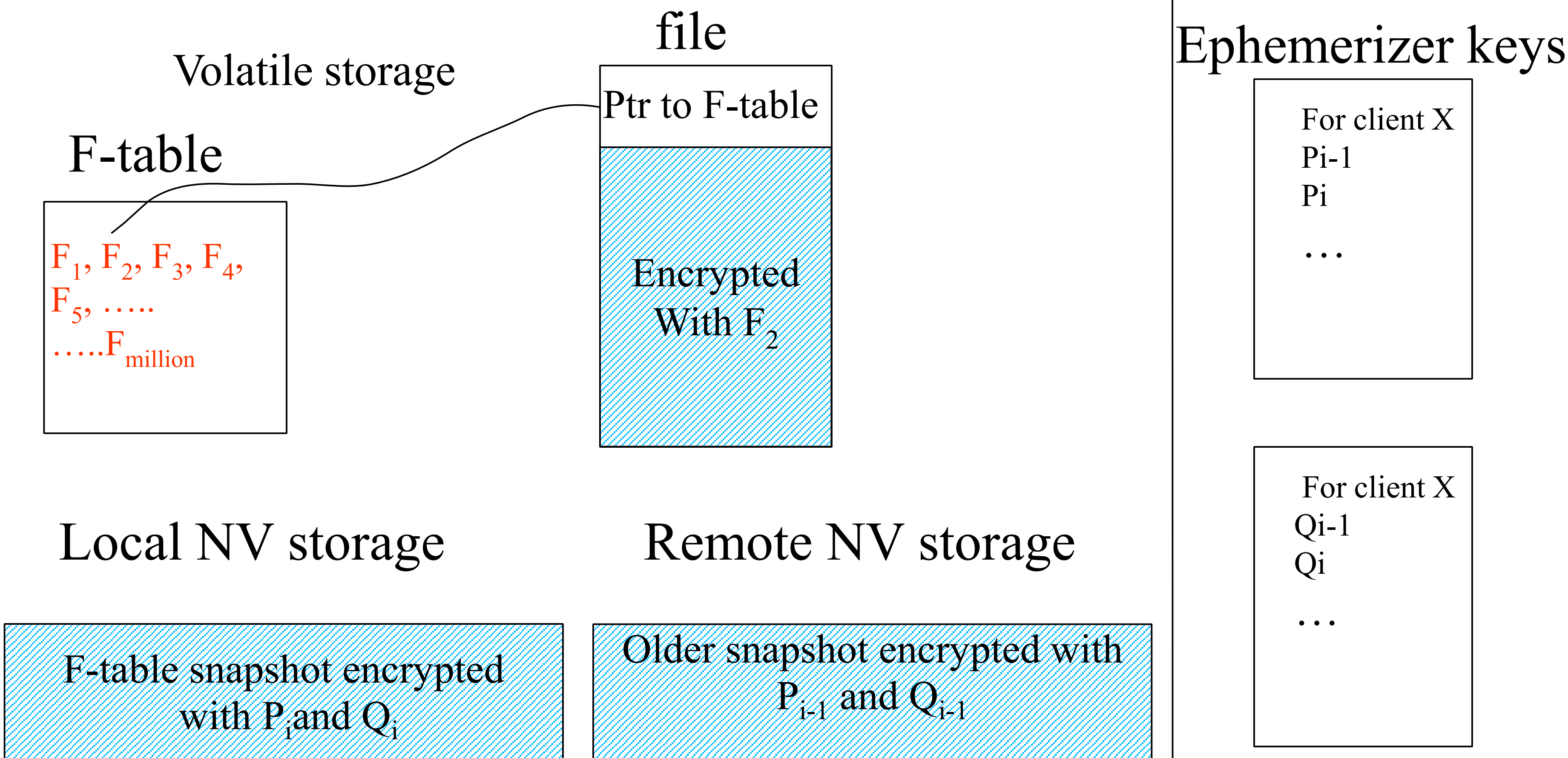


File system with F-table



Modify F-table when you assure-delete a file
Or create a new file
F-table has key for each file…if a million files, a million keys

File system with F-table



Sufficiently replicated for robustness

So what’s wrong?

My concern

- Suppose you change P’s every week
- Suppose you find out that the file system was corrupted a month ago
- And that parts of the F-key database were corrupted, without your knowledge
- You can’t go back

Why isn't pre-determined expiration time as scary?

- If file system is not corrupted when a file is created, and the file is backed up, and the S-table is backed up, you can recover an unexpired file from backup
- Whereas with the on-demand scheme, if the file system gets corrupted, all data can get lost

Note

- I’ve shown 3 very different problems, with very different solutions
- I’m not sure there is any one piece of advice other than “think about the case of misbehaving participants”

Thank you!