# Public key encryption

- **Three randomized algorithms:**
  - *Generate* – $G(1^k) \to K, K^{-1}$
  - *Encrypt* – $E(K, m) \to \{m\}_K$
  - *Decrypt* – $D(K^{-1}, \{m\}_K) \to m$
- **Provides secrecy, like conventional encryption**
  - Can't derive $m$ from $\{m\}_K$ without knowing $K^{-1}$
- **Encryption key $K$ can be made public**
  - Can't derive $K^{-1}$ from $K$
  - Everyone can use the same public key to encrypt messages for one recipient.

# Digital signatures

- **Three (randomized) algorithms:**
  - *Generate* – $G(1^k) \to K, K^{-1}$
  - *Sign* – $S\left(K^{-1}, m\right) \to \{m\}_{K^{-1}}$
  - *Verify* – $V\left(K, \{m\}_{K^{-1}}, m\right) \to \{\mathbf{true}, \mathbf{false}\}$
- **Provides integrity, like a MAC**
  - Cannot produce valid $\langle m, \{m\}_{K^{-1}}\rangle$ pair without $K^{-1}$
- **Many keys support both signing & encryption**
  - But Encrypt/Decrypt and Sign/Verify different algorithms!

# Cost of cryptographic operations

| Operation | msec |
|---|---|
| Encrypt | 1.11 |
| Decrypt | 39.62 |
| Sign | 40.56 |
| Verify | 0.10 |

[1,280-bit Rabin-Williams keys on 550 MHz K6]

- **Cost of public key algorithms significant**
  - Encryption only on small messages ($<$ size of key)
  - Signature cost relatively insensitive to message size
- **In contrast, symmetric algorithms must cheaper**
  - Symmetric can encrypt+MAC faster than 100Mbit/sec LAN

# Hybrid schemes

- **Use public key to encrypt symmetric key**
  - Send message symmetrically encrypted: $\{\mathrm{msg}\}_{K_S}, \{K_S\}_{K_P}$
- **Use PK to negotiate secret session key**
  - E.g., Client sends server $\{K_1, K_2, K_3, K_4\}_{K_P}$
  - Client sends server: $\{m_1, \mathrm{MAC}(K_2, m_1)\}_{K_1}$
  - Server sends client: $\{m_2, \mathrm{MAC}(K_4, m_2)\}_{K_3}$
- **Often want mutual authentication (client & server)**
  - Or more complex, user(s), client, & server

# Case study of successful system: SSH

- **Before 1995: No secure remote login over Internet**
  - Cleartext passwords: sniffed on Ethernet
  - s/key: TCP hijacking, sniffing & off-line password cracking
  - IP-address-based `.rhosts` authentication: spoofable
  - Kerberos: implementation vulnerable to spoofing, weak crypto, no MAC, off-line password cracking, limited deployment
- **Today: Widespread deployment of SSH**
  - Not perfect, but far more secure than what came before
  - Supplanted old tools in many OS distributions
  - Significant and widespread impact on security

# How does SSH work?

- **Similar interface to existing tools (like rlogin/rsh)**

  `% ssh server -l user`

- **Client & server exchange public session keys:**
  - $S \rightarrow C$: $\{K_S$ (server pubkey), $K_t$ (temporary pubkey)$\}$
  - $C \rightarrow S$: $\{\{K_{cs}$ (session key)$\}_{K_t}\}_{K_S}$

- **Client checks $K_S$ if it has talked to server before**

- **Subsequent traffic encrypted with $K_{cs}$**

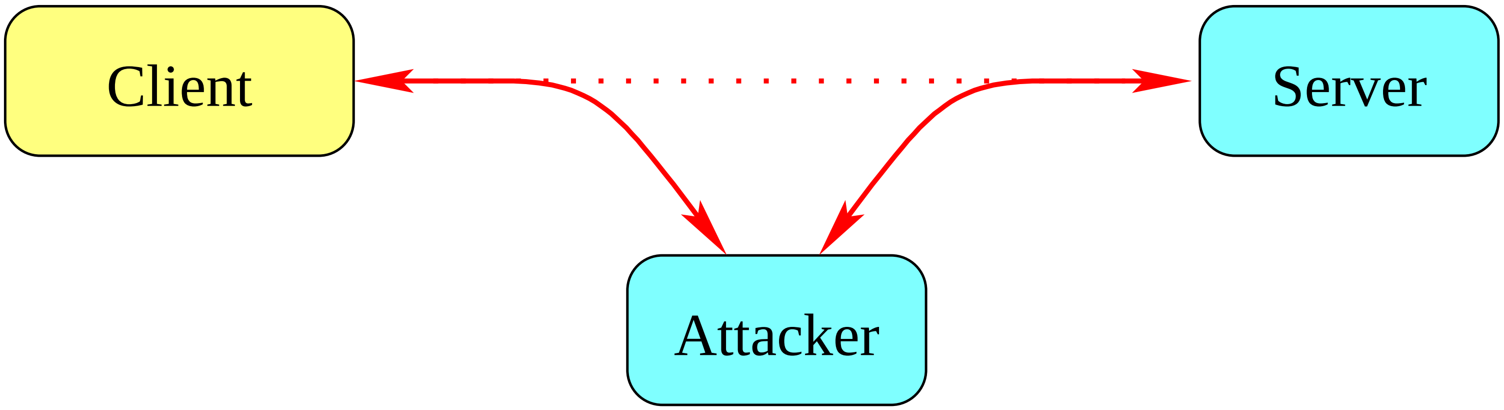
# Why did SSH succeed?

- **Provided better functionality than alternatives**
  - 8-bit clean, sets DISPLAY, accepts passwords (unlike rsh)
- **Had properties conducive to deployment**
  - Simple to install and use
  - Peacefully coexisted with other remote login tools
  - Any client can connect to any server
  - Intuitive to understand given the notion of public keys
- **Provided a highly *composable* abstraction**
  - Encrypted pipes usuful to applications
  - Developers eagerly exploit SSH (CVS, rsync, rdist, ...)

# Limitations of SSH

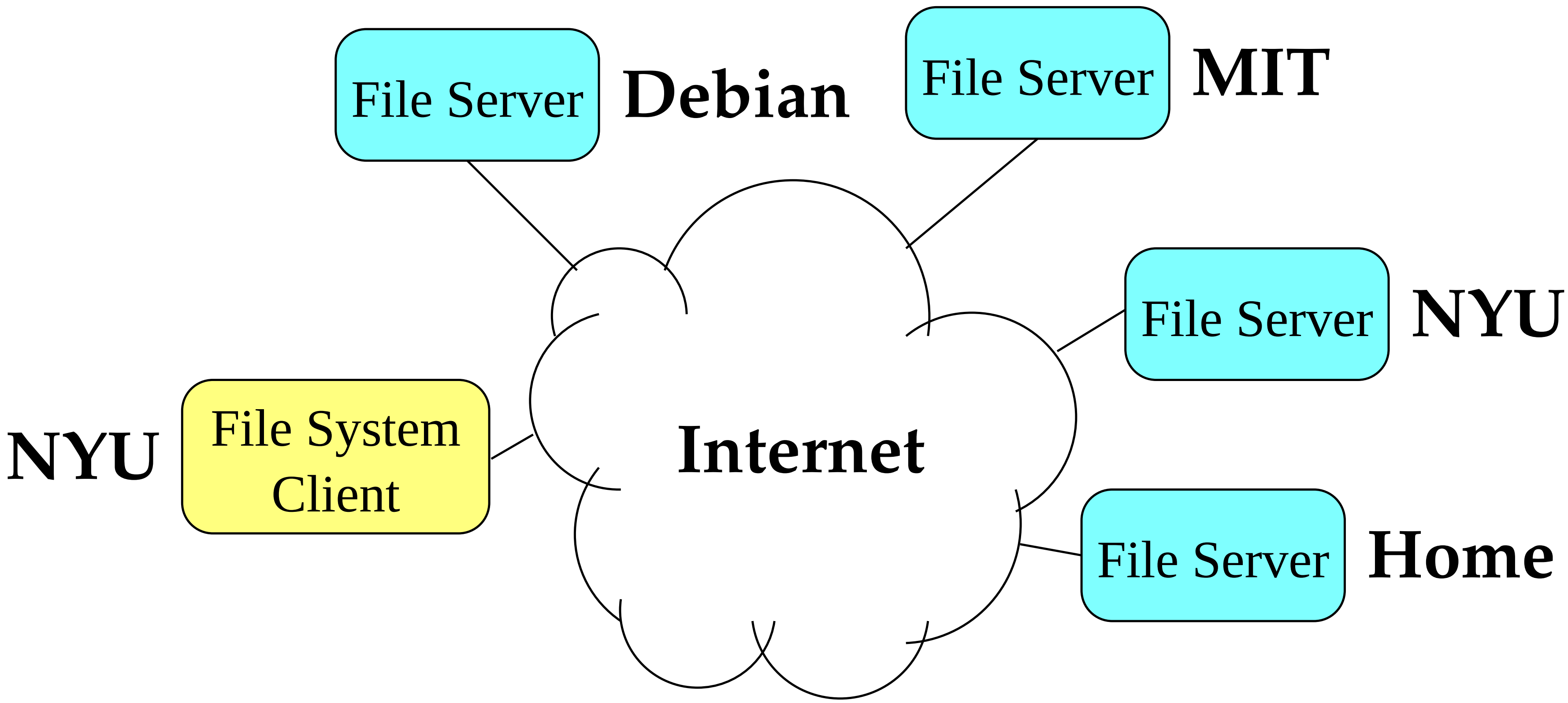
- **Doesn't solve the file system security problem**
  - Many people require network file systems for their work
  - Common protocols cannot easily be composed with SSH
- **Fundamentally provides server authentication**
  - We also need *content* authenticatoin
  - E.g., software distribution and upgrade: OS distributions typically mirrored on untrusted servers
- **Vulnerable to man-in-the-middle attacks**

# Man-in-the-middle attacks




- **Can't trust $K_S$ received first time you talk to server**
  - Attacker might substitute his own key $K_A$
  - Client connects to attacker (thinking it is server)
  - Attacker connects to Server, passes traffic through
  - E.g., terrible if sending credit card #s to merchant

# SFS: A secure global file system

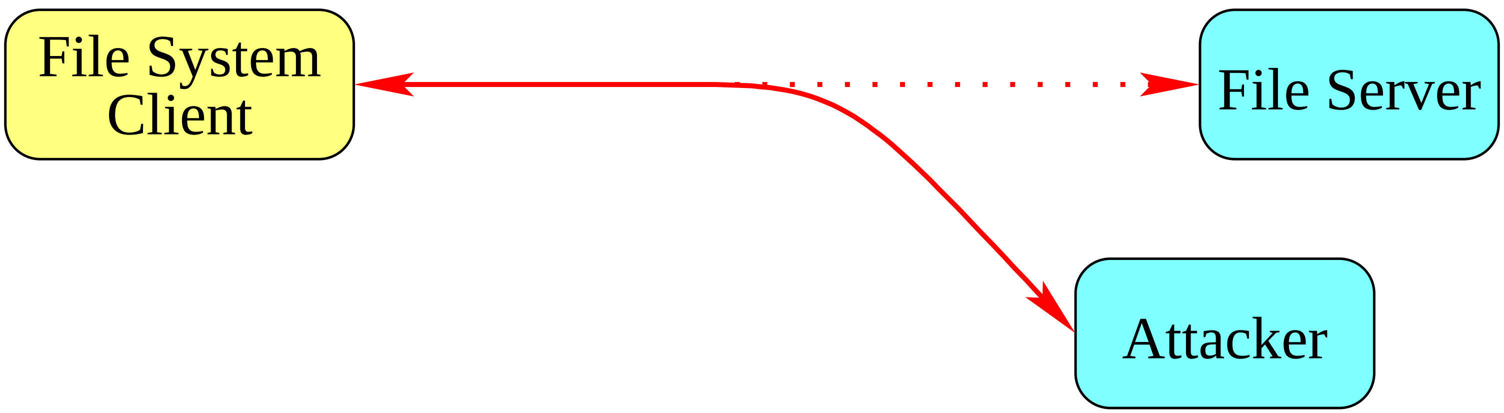



- Goals: Secure, easy to deploy (like SSH)
- A access any file system from anywhere

# The security problem

- **Secure client–server communications**
  - Solution: Use cryptographically *secure channel*
- **Authenticate users to servers**
  - Servers know what classes of users to expect in advance
  - Solution: Store users' passwords or public keys
- **Authenticate servers to clients**
  - Clients don't know about servers in advance
  - A user can potentially access any server in the world
  - Solution: ?

# Implications of man-in-the-middle attacks




- Attacker substitutes modified data for file
- User writes sensitive file to fake server

# Server authentication

- **Can be solved if you have server's public key**
  - E.g., SSH secure once you have server's $K_S$
- **Issue boils down to key management**
  - How to get server's public key?
  - How to know the key is really server's?
  - How to give server key to file system?
- **Problem: Key management has ever scaled to the size of the Internet**

# Possible approaches to key management

- **Put public keys in the phone book**
  - How do you know you have the real phone book?
  - How is a program supposed to use phone book www.phonebook.com? (are you talking to real web server)
- **Exchange keys with people in person**
- **“Web of trust” – get keys from friends you trust**

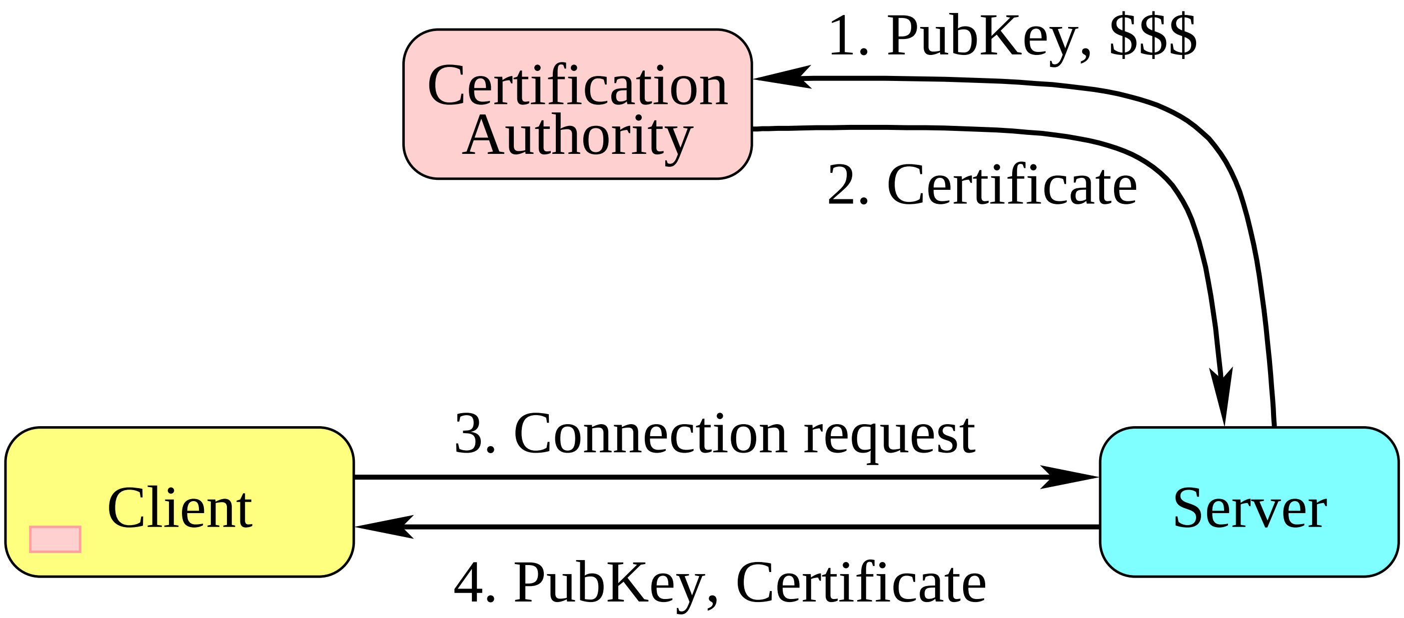
# Hierarchy with local trust

- **All machines in CS department know key for central cs.nyu.edu server**
- **To get from cs.nyu.edu to mit.edu:**
  - cs.nyu.edu knows key for nyu.edu
  - nyu.edu knows key for edu/root
  - root knows key for mit.edu
- **To get within cs.nyu.edu:**
  - No need to trust outside authorities

# Limitations of previous systems

- **Presume hypothetical cooperation of third parties**
  - Echo would have required Internic to manage keys
- **Lack security across administrative boundaries**
  - AFS provides no security to unknown users
- **Penalize the creation of new administrative realms**
  - Kerberos and AFS lead to inconveniently large realms
- **Provide inappropriate security procedures or guarantees**
  - SSL takes “one size fits all” approach to key management

# SSL approach in detail




- Everybody trusts some certification authority
- Trade-off between ease of certification and security
- Precludes other models (passwords, Kerberos, . . . )

# Solution: Self-certifying File System

- **Idea: Make file system security independent of key management**
- **Specify server keys in *self-certifying pathnames*:**

  `/sfs/@sfs.mit.edu,bzcc5hder7cuc86kf6qswyx6yuemnw69/dm/`

  - File name itself certifies server's public key
- **Push key management out of the file system**
  - Problem reduces to finding correct file name

# New approach to key management

- **SFS provides security without key management**
- **Let multiple key management schemes coexist**
- **Make it easy to implement new schemes**
  - *Self-certifying pathnames* managed with standard file utilities
  - *SFS Agents* let external programs manage keys
  - *Secure symbolic links* like web links but secure
  - *SFS itself* allows secure sharing key management data

# User's view of SFS

- **New directory `/sfs` contains global files**
- **Subdirectories of `/sfs` are self-certifying**

  `/sfs/@sfs.mit.edu,bzcc5hder7cuc86kf6qswyx6yuemnw69/`

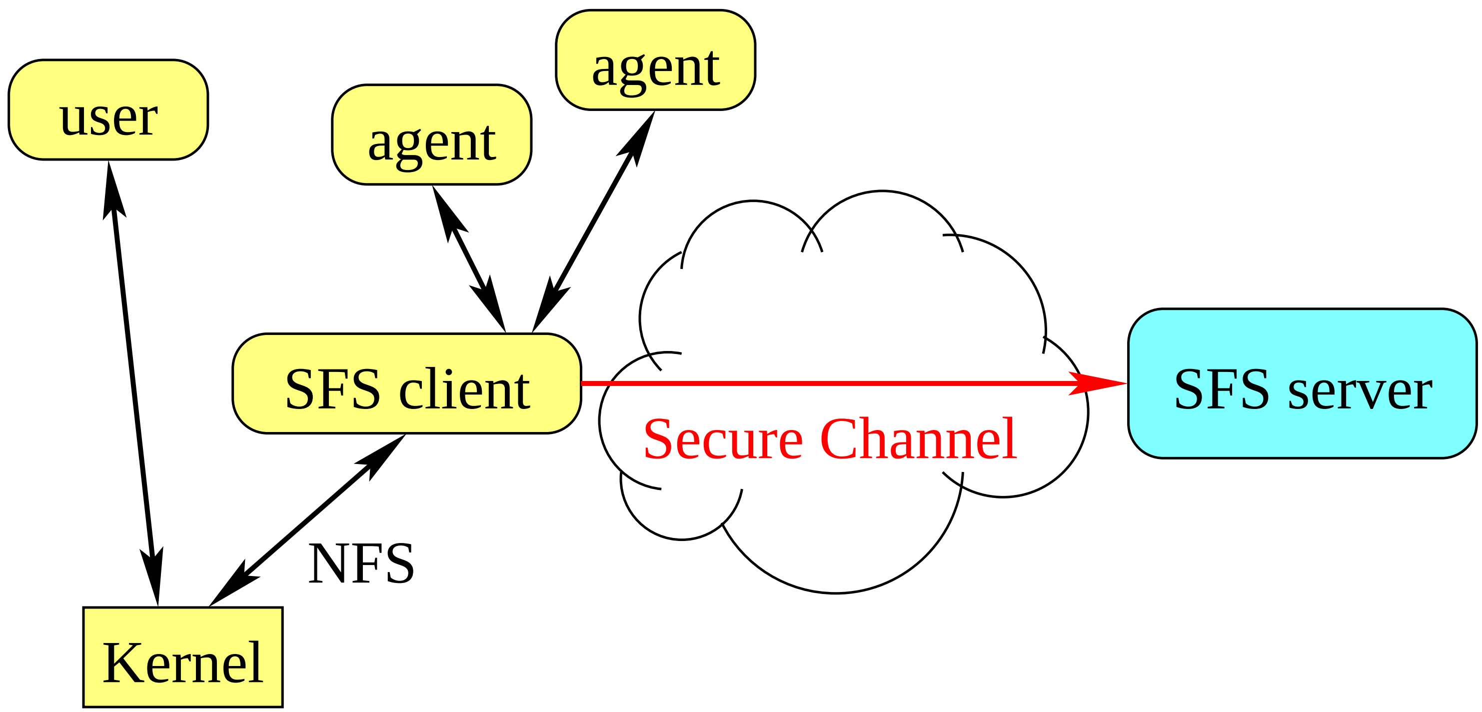
- **Human-readable aliases give names to public keys**

  `/sfs/nyu` → `/sfs/@sfs.nyu.edu,bzcc···nw69`

- **Ordinary naming under self-certifying pathnames**

  `/sfs/@sfs.nyu.edu,bzcc···nw69/usr/dm/mbox`

# System's view of SFS




- **Client appears to system as NFS server for `/sfs`**
- **Interprets requests for self-certifying pathnames**
- **Agents interpret non-self-certifying pathnames**

# Self-certifying pathnames

- **File systems lie under `/sfs/@`*Location*,*HostID***

$$\textbf{\textit{HostID}} = \text{SHA-1}\,(K_S, \ldots)$$

  - *Location* is DNS name or IP address
  - $K_S$ is the server's public key
  - *HostID* is 20 bytes regardless of key length
  - Finding collisions of SHA-1 considered intractable

- ***HostID* effectively equivalent to public key**
  - Client can ask server for key and check against *HostID*
  - *HostID* suffices to connect securely to server

# Self-certifying pathname details

$$/\texttt{sfs}/\texttt{@}\underbrace{\texttt{sfs.fs.net}}_{\textit{Location}}\texttt{,}\overbrace{\texttt{eu4cvv6wcnzscer98yn4qjpjnn9iv6pi}}^{\textit{HostID}\text{ (specifies public key)}}/\underbrace{\texttt{sfswww/index.html}}_{\text{path on remote server}}$$

- **Pathnames transparently created when referenced**
  - Anyone can create a server
  - New servers instantly accessible from any client
- **Client requires server to have *HostID*'s private key**
- **Pathname implies nothing about name of server**
  - e.g., server may not actually be the real `sfs.fs.net`
  - Need key management to produce the correct file name

# Key management through symbolic links

- **Symbolic links assign additional names to paths**
  - *link* $\rightarrow$ *dest* makes *link* another name for *dest*
  - Always interpreted locally on a file system client
- **Link human-readable to self-certifying pathnames**
- **Example: manual key distribution**
  - Install central server's path in root directory of all clients:

    `/nyu` $\rightarrow$ `/sfs/@sfs.nyu.edu,bzcc5hder7cuc86kf6qswyx6yuemnw69`
  - `/nyu/README` designates the pathname:

    `/sfs/@sfs.nyu.edu,bzcc5hder7cuc86kf6qswyx6yuemnw69/README`

    "The file `README` on the server my administrator calls `/nyu`"

# So how to do server key management?

- **SFS separates key management from FS security**
  - Effectively redefines "security" to avoid problem
  - Traditional systems guarantee: "you are talking to server X"
  - With SFS, you are talking to server with pubkey 0x42379…
- **SFS is clearly useful sometimes**
  - E.g., when you already have the key (in a symlink)
- **But goal was for any client to talk to any server**
  - Still need a way to arrive at server public key (pathname) starting from a human's idea of the server
  - This is why other systems all have key menagement built-in

# Ways of getting server public keys

- **Global certification authorities certify keys (SSL)**
  + Works for on-line shopping, banking, etc.
  – No authority/certification procedure suitable for everyone
- **Realm administrators exchange keys (Kerberos)**
  + Good for sharing files between large organizations
  – Need an account/administrative relationship to get security
- **Let individual users manage keys (SSH)**
  + Anyone can run a server
  + Any client can connect to any server
  – Attackers can impersonate servers
- **Right answer: All of the above and more**

# Example: Certification authorities

- **Are simply SFS file systems**
  - Can be named by local symbolic links:

    `/verisign` $\longrightarrow$ `/sfs/@sfs.verisign.com,r6ui9gwucpkz85uvb95cq9hdhpfbz4pe`
  - Name other file systems with symbolic links, e.g.

    `/verisign/NYU` $\longrightarrow$ `/sfs/@sfs.nyu.edu,bzcc5hder7cuc86kf6qswyx6yuemnw69`
- **Have no special privileges or status**
  - Servers reachable from `/verisign` can name other servers

    `/verisign/NYU/cs` might name server for `cs.nyu.edu`
- **Pathnames reflect trust relationships:**

  `/verisign/NYU/README` – "File `README` on the server Verisign calls NYU"
- **Read-only protocol keeps private key off-line**

# Example: Server knows password[a]

- **SRP [Wu98] derives session key from a password**
- **Server proves its identity to user with password**
  - User then securely downloads pathname from server

```
% sfskey add dm@scs.cs.nyu.edu
Passphrase for dm@scs.cs.nyu.edu/1280:
% ls -al /sfs/scs.cs.nyu.edu
lr--r--r--  1 root  sfs  512 May 28 04:16 /sfs/scs.cs.nyu.edu ->
@scs.cs.nyu.edu,85xq6pznt4mgfvj4mb23x6b8adak55ue
```

[*sfskey* also simultaneously handles user authentication.]

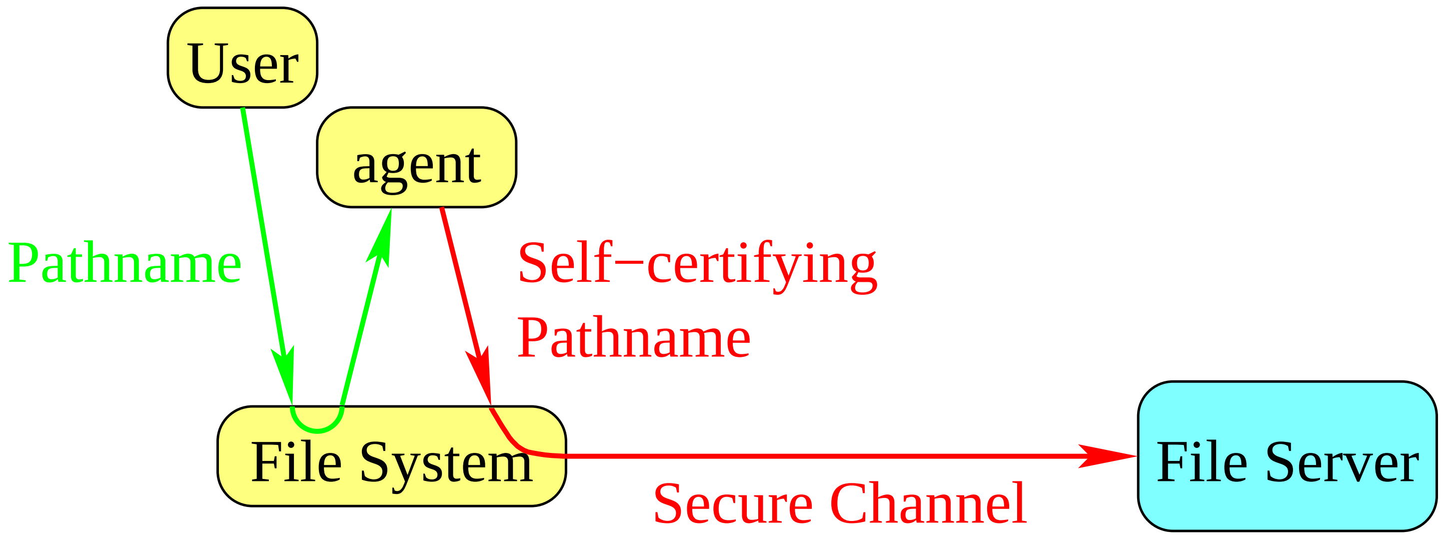
- **Bootstrap security using links on `scs.cs.nyu.edu`**

[a] actually some one-way function of password and server name

# Why authenticate servers with passwords?

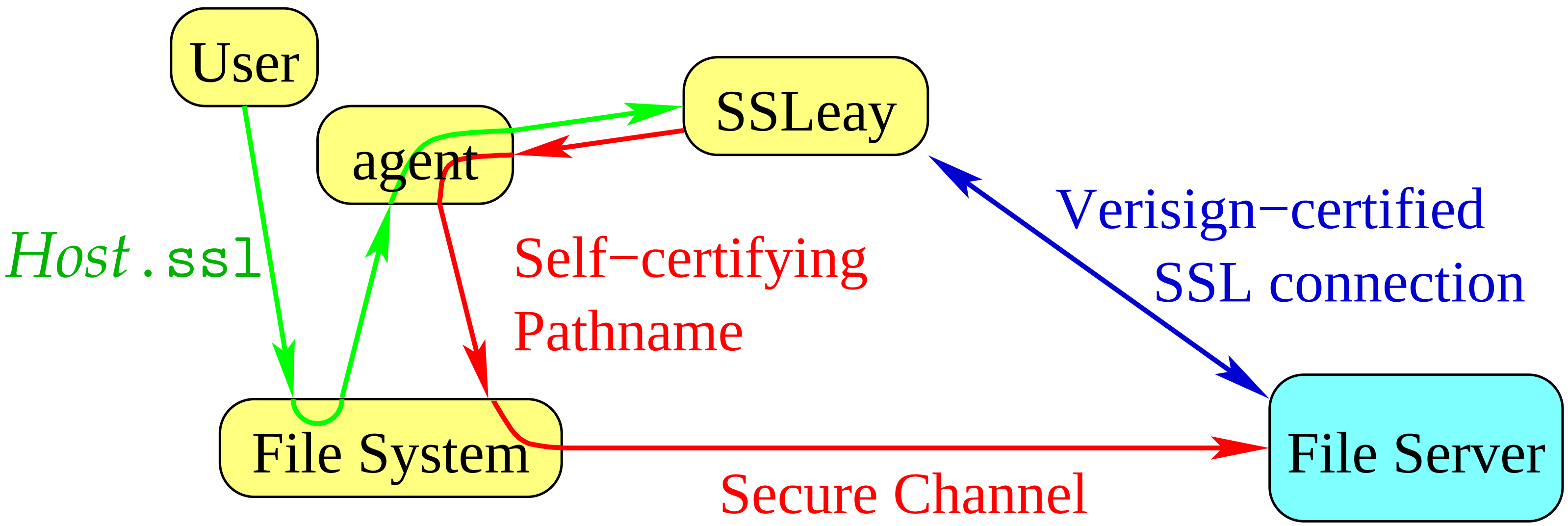
- **The only practical solution for many situations**
  - I don't remember my server's public key or *HostID*
  - No administrative relationship between client and NYU
  - I lack authority at NYU to buy certificates from Verisign
- **Provides exactly the desired security guarantee**
  - The server at which I physically typed my password
  - No need to trust any third parties

# Dynamic server authentication




- **Each user runs an *agent* program to control `/sfs`**
- **Agents can create symbolic links in `/sfs` on-demand**
  - Agent maps names to self-certifying pathnames with arbitrary external programs

# Example: Getting *HostID*s through SSL




- **User references `/sfs/`*Host*`.ssl`**
- **Agent spawns SSL client to get *HostID* securely**
- **Agent links `/sfs/`*Host*`.ssl` → *Host*:*HostID***
  - User's file access transparently redirected

# Implementing key management is trivial!

- **SSL example implemented in two lines**
  - Distribute pathnames from URL `https://`*Host*`/sfspath.txt`
  - Map `/sfs/`*Location*`.ssl` to path retrieved with SSL:

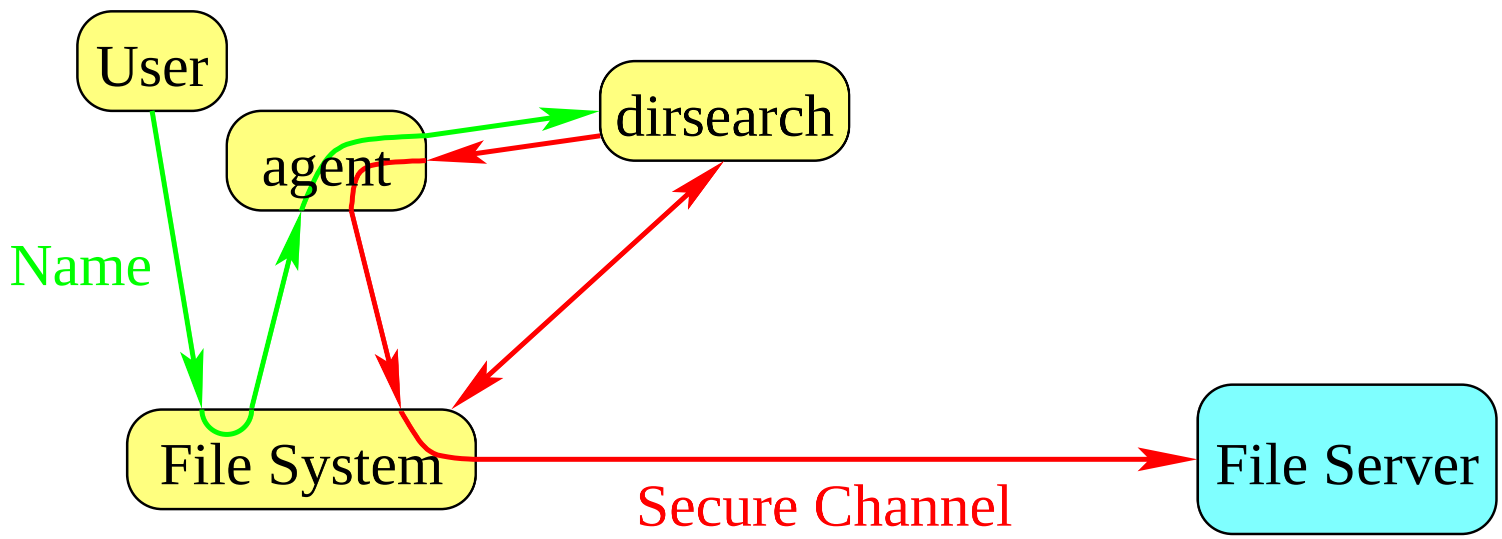
```
% sfskey certprog -s ssl \
    sh -c 'lynx -source https://$0/sfspath.txt'
```

- **Don't like SSL? How about Kerberos?**
  - Map `/sfs/`*Location*`.krb` to path retrieved with Kerberos:

```
% sfskey certprog -s krb \
    sh -c 'rsh -x $0 sfskey hostid -'
```

  - Similar command works for SSH

# Certification paths




- **Combine multiple certification authorities**
  - Merge your own names with those assigned by third parties
- **Make agent search multiple directories for links:**

  `~/.sfs/known_hosts`, `/mit/links`, `/verisign`, `/thawte`

- **Dirsearch implementation easy given file system**

# Revocation


- **Many links may exist to a compromised *HostID***
- **Separate key revocation from key distribution**
  - Announce revocation with self-authenticating certificates

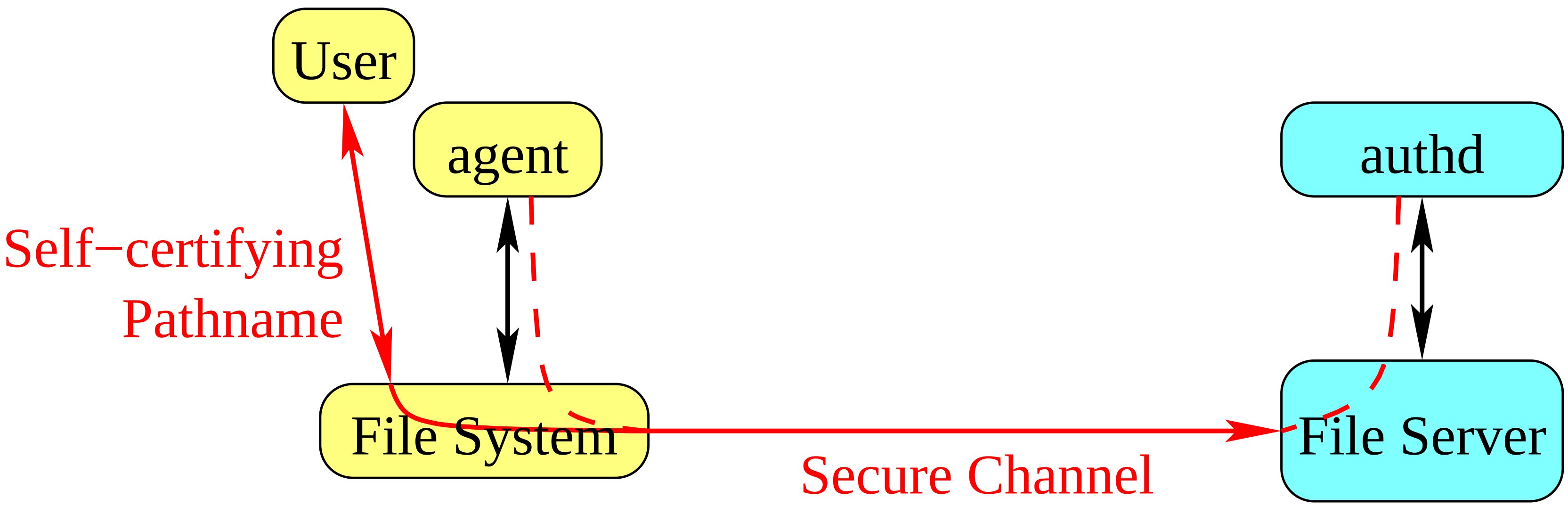
$$\{\text{“Path Revoke”}, \mathit{Location}, K_S, \ldots\}_{K_S^{-1}}$$

  - Let agents search for certificates on-the-fly

# Distributing revocation certificates

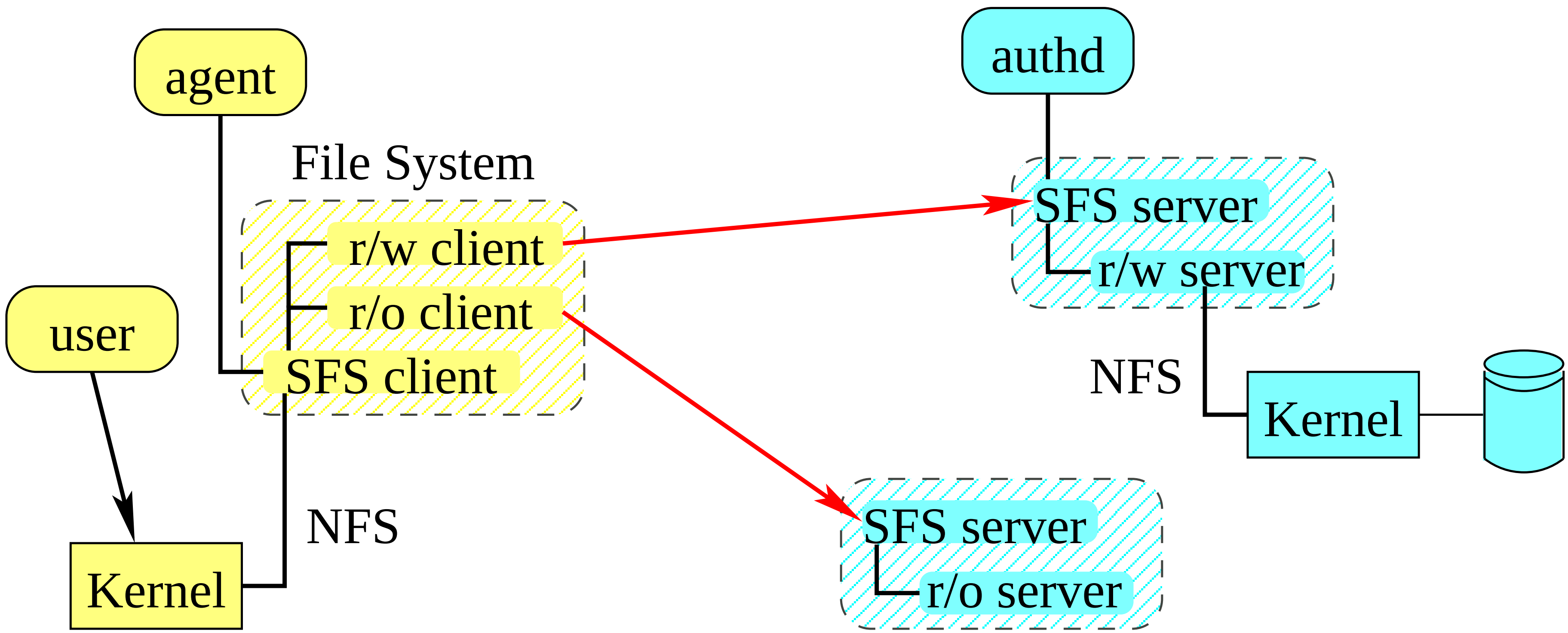
- **Use the file system!**
  - Publish revocation certificates as `/verisign/revoked/`*HostID*
  - *dirsearch* fetches certificates, as with certification paths
- **Benefits of separating revocation from certification:**
  - Revocation certificates require no out-of-band verification
  - No authority necessary to submit a revocation certificate
  - Revocation certificates as secure as *best* CA, not weakest

# User authentication




- **Separate programs handle authentication**
  - User-authentication protocols opaque to file system proper
- **Current authd has simple public-key protocol**
  - No penalty for accessing many administrative realms
  - Use the file system to distribute user keys

# Modular implementation

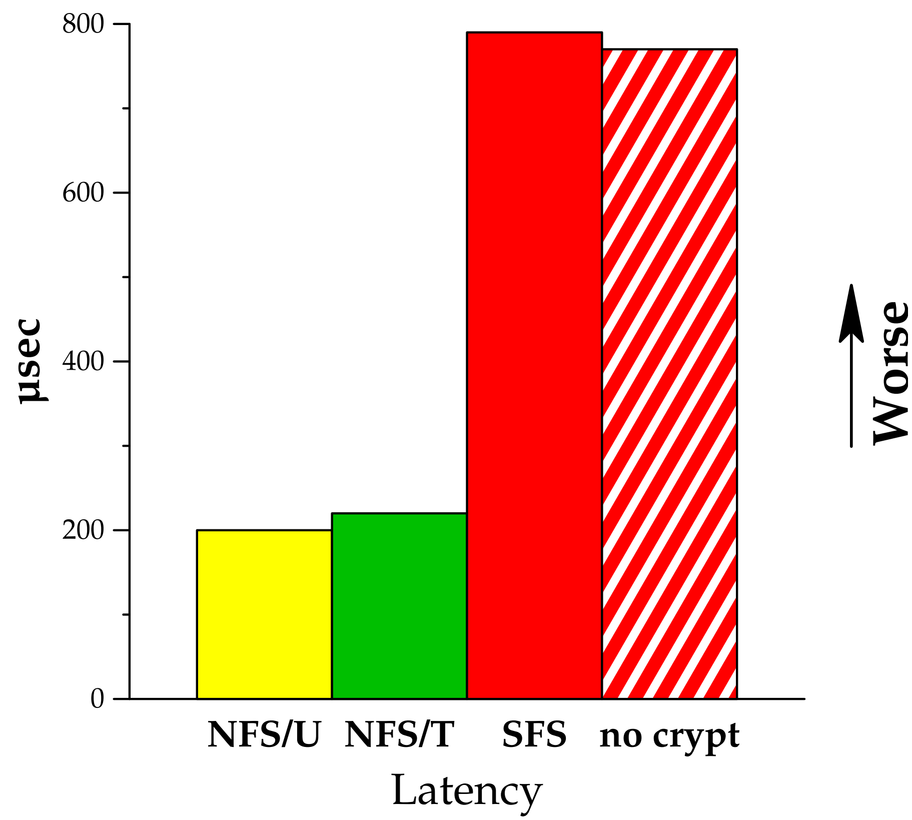



- **Multiple file systems share SFS key management**
- **Solved many problems of user-level NFS servers**
  - Asynchronous I/O libraries for non-blocking applications
  - New "automounter" techniques for mounting in place

# Performance summary

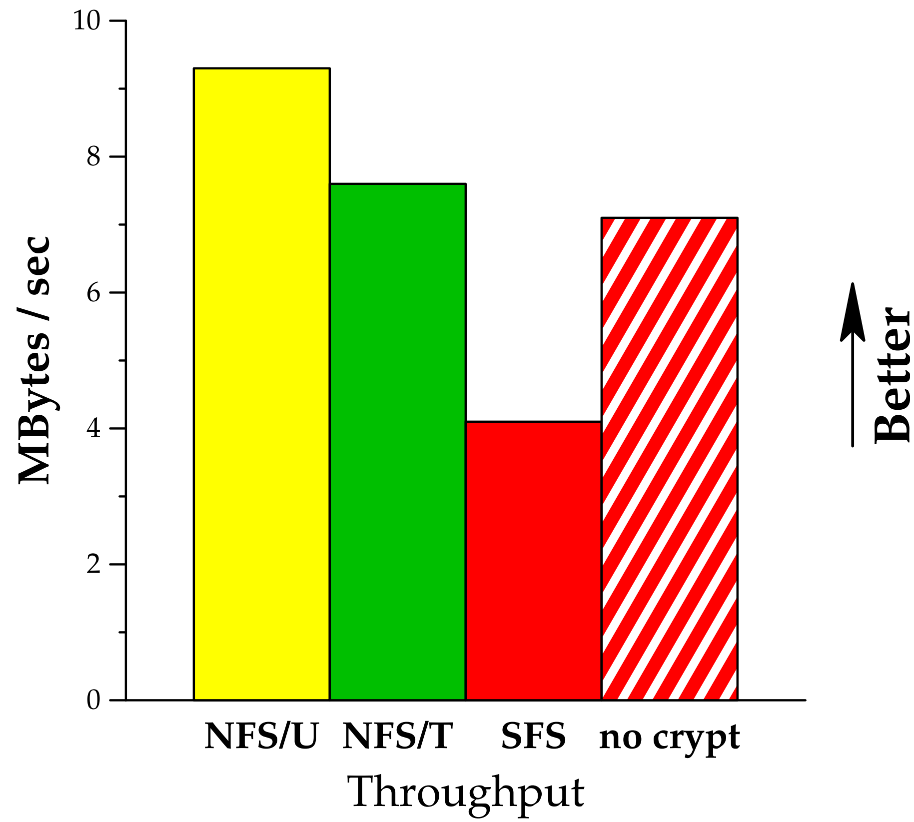
- **Goal: Performance comparable to NFS 3**
- **Three properties of SFS hurt performance**
  - Portable, user-level implementation
  - Software encryption and authentication of session traffic
  - Public key operations during session establishment
- **Performance affected in three places**
  - Latency of RPCs to server increases
  - Maximum data throughput decreases
  - Mounting and user authentication require computation
- **Better caching maintains application performance**

# Performance: Latency



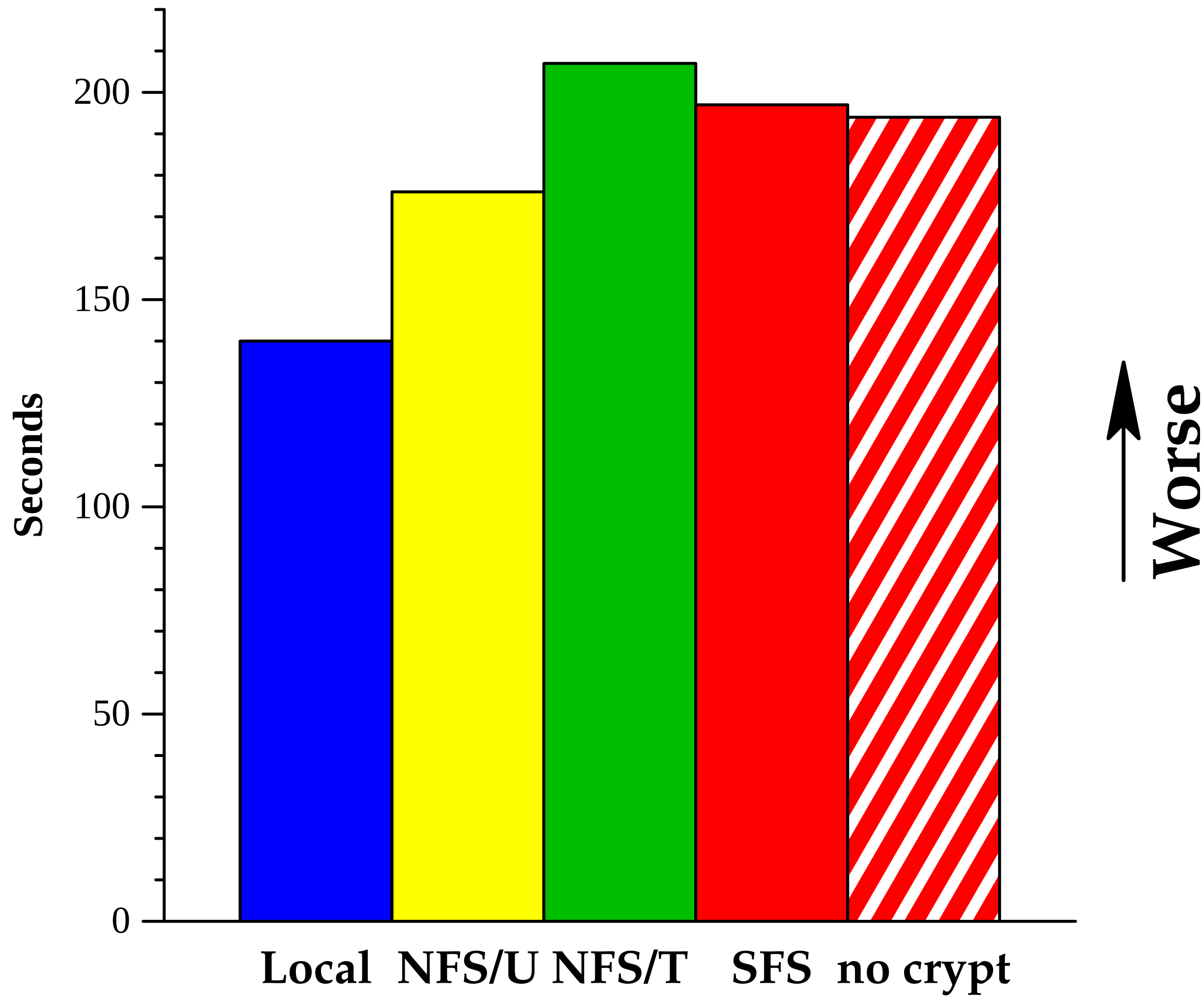

- **Hurt by user-level implementation, not crypto**
- **Mitigated by better protocol with fewer RPCs**

# Performance: Throughput




- **Suffers mostly from cryptography**
- **Effects not visible on workloads with disk seeks**

# Performance: Application




Compile time for FreeBSD 3.3 GENERIC kernel

550 MHz Pentium III, 256 MBytes RAM, 100 Mbit ethernet

# Performance: Mounting file systems

| Operation | msec |
|---|---|
| Encrypt | 1.11 |
| Decrypt | 39.62 |
| Sign | 40.56 |
| Verify | 0.10 |

| Automounter | msec | |
|---|---|---|
| SFS mount | 64 | |
| SFS auth | 49 | |
| SFS both | 109 | |
| NFS amd | 10–1,000 | (unfair) |

[1,280-bit Rabin-Williams keys]

- **No one cares about mount times (*amd* suboptimal)**
- **Latency from public key protocols not noticable**

# Lessons learned

- **Challenge of global security is key management**
- **Global public key management not the answer**
  - Even in a global system, key management often a local issue
- **Don't base system security on key management**
  **…base key management on secure systems**
- **Strip clients of any notion of administrative realm**

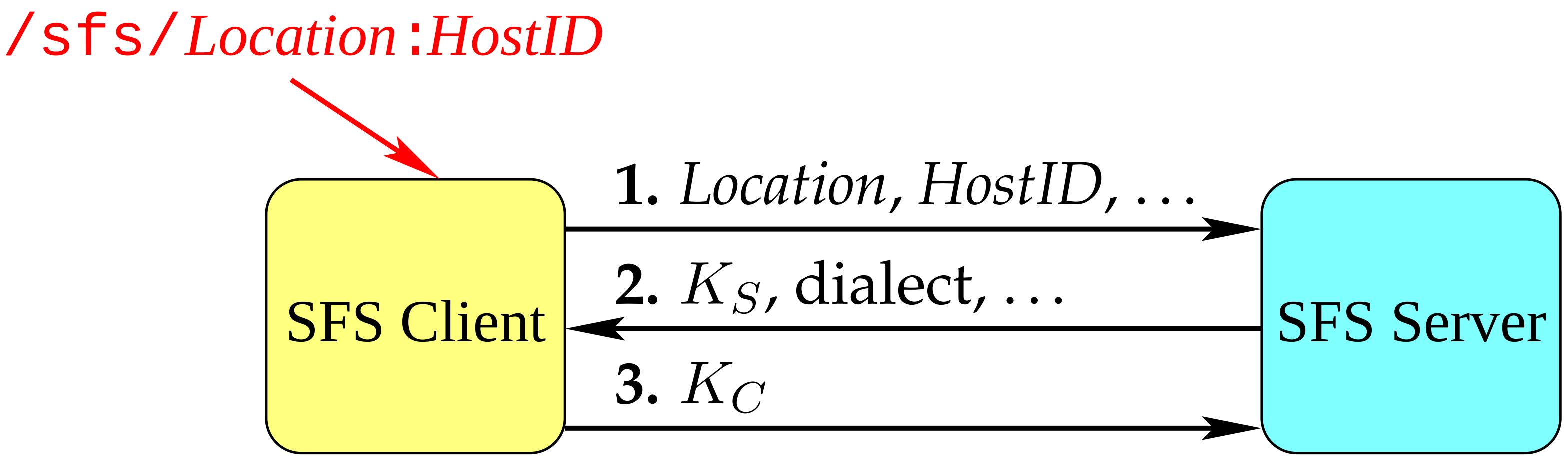
# Conclusions

- **SFS is first web-like system with global security**
  - Provides strong file system security
  - Realistically deployable on a global scale
    (anyone can create a server, any client can access any server)
- **SFS takes a new approach to key management**
  - Provide global security without any key management
  - Let arbitrary key management schemes coexist externally
  - Make it easy to implement new schemes
- **New key management mechanisms**
  - Self-certifying pathnames, Agents, Secure links
- **SFS is its own key management infrastructure**

# Attacking SFS

- **Inherent dangers of a global file system**
  - Attacker's own files visible everywhere—facilitates exploits
  - Symbolic links on bad servers can point to unexpected places
- **SFS may further expose bugs in existing software**
  - Running NFS at all can cause security holes
  - Bugs in NFS may let attackers crash machines (or worse)
- **Attacks on SFS itself**
  - Cause resource exhaustion (e.g. use up all file descriptors)
  - Cut network during non-idempotent operations

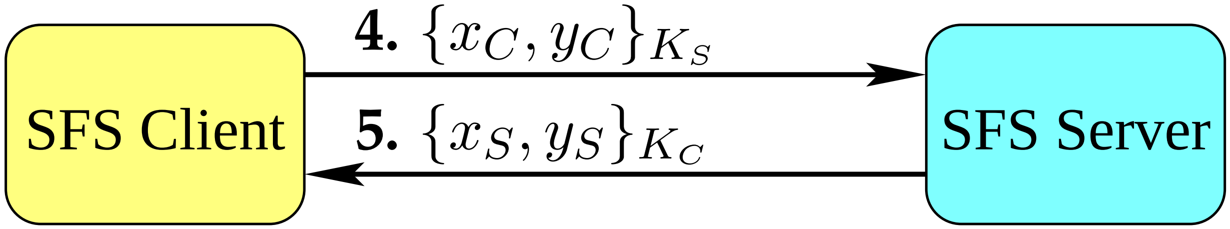
# Connection protocol




**Goal: A secure channel to the server for *HostID***

1. **Client connects to server**
2. **Server returns its public key,** $K_S$
   - Client hashes $K_S$ and verifies it matches *HostID*
   - Client passes connection to appropriate daemon for dialect
3. **Client sends short-lived, anonymous public key,** $K_C$

# Session key negotiation




$$\textbf{6.}\quad \begin{aligned} k_{CS} &= \text{SHA-1}(\text{dialect}, K_S, x_S, K_C, x_C, \ldots) \\ k_{SC} &= \text{SHA-1}(\text{dialect}, K_S, y_S, K_C, y_C, \ldots) \end{aligned}$$

**4. Client encrypts two random key halves with $K_S$**

**5. Server encrypts two random key halves with $K_C$**

**6. Client and server compute shared session keys**

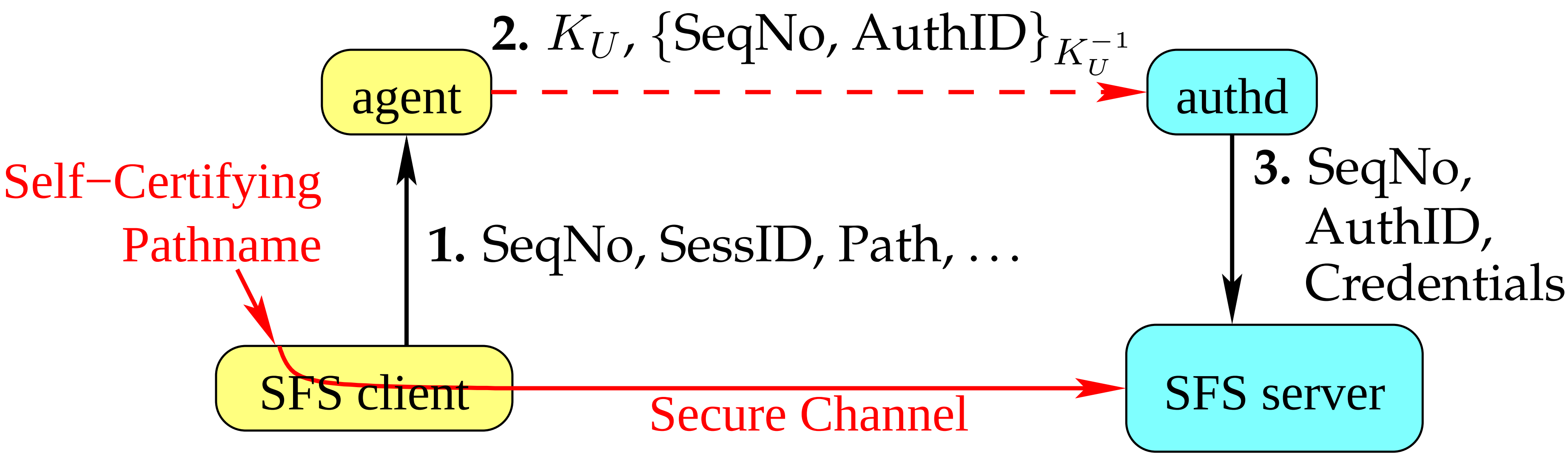
**Important properties of protocol:**

- Efficient: Minimizes server computation, overlaps with client
- Simple: No options, always secure

# User authentication protocol

$$\text{SessID} = \text{SHA-1}(k_{CS}, k_{SC}, \ldots)$$

$$\text{AuthID} = \text{SHA-1}(\text{SessID, Path}, \ldots)$$




1. **Client notifies agent, assigns it SeqNo**
2. **Agent authorizes secure channel to represent user**
3. ***authd* informs file server of user's credentials**