Abstract
This paper describes the design, implementation, and performance of Neo, a wide-area file system whose security architecture is based on the chit. Chits are instantiations of capabilities that capitalize on the strengths of traditional capabilities (flexible granularity, rights tied to naming, non-traditional identities) while mitigating the drawbacks (revocation, accountability, lack of confinement).

Neo allows flexible, fine-grained sharing with differing, chit-defined notions of identity. User-defined policies allow anonymous access, fully authenticated access, and levels in between. We describe the implementation of Neo, and show that the required mechanisms add little overhead to the underlying system.

1 Introduction
Authorization in distributed systems can be performed by one of two archetypal techniques. Accounts represent strong identities once proven to some administrator. They promote accountability by giving each action a responsible actor, and they allow revocation of access by deleting accounts or updating access control lists. Capabilities [29, 19] separate authentication from authorization by realizing access rights in a transferable object. Compared to accounts, capabilities add the ability to control access at fine granularities and to delegate access, but remove accountability and fine-grained revocation.

Neither mechanism suffices because delegation, accountability, and revocation are all important. For example, neither supports collaborative groups well: a would-be collaborator may wish to delegate access to fellow members and track accesses, but might not have the desire or the rights to establish formal accounts on the server for each collaborator. In practice, users share more broadly than they should, sharing more files than needed to more users than they intended, because precise access control often requires administrator support [22]. Consider how much simpler it is to make a file accessible to all other users than to carefully permit access to just one other.

This paper describes the design, implementation, and performance of Neo, a wide-area file system that supports a broad range of user-defined access control policies. We contend that, simultaneously, users should define access control policy, user actions should be accountable, and system-wide policies should be enforceable. Our intent is to support collaboration through user-driven delegation of access rights while providing the accountability required by resource owners. By designing our system around a general mechanism, we hope to have the tug-of-war between users wishing to share, and administrators wanting to maintain control, to be played out using the mechanism, rather than by circumventing it.

Our central contributions are the following:

• We describe scenarios where file systems can be used to support flexible, fine-grained sharing, but are poorly supported by current access control mechanisms.

• We show that a chit-based system like Neo can be crafted to support user-defined access policies that support the above scenarios.

• We show that a file system built on chits can support fine-grained revocation and accountability, properties that have long been difficult for capability-based systems.

Finally we describe Neo’s implementation and performance, showing that the required mechanisms are efficient.

We next describe our model and assumptions, and then a series of motivating scenarios. Section 4 describes the chit in more detail, and we lay out the design of Neo in Section 5. We discuss performance results in Section 6, and then describe related work and conclude.
2 Model

A Neo system consists of a single, globally-trusted metadata server, which stores file metadata and deals with chits, one or more untrusted block servers [12, 28], and a (potentially large) set of mutually distrusting users running client code. All interact over untrusted communication channels (the wide-area Internet). Though we here assume that such a server is a single machine, the server could also be implemented as a set of replicated machines or monitors. Clients can interact simultaneously with multiple servers, but as the servers are independent, we only discuss the single-server case further.

A trusted server in our model means that the server is trusted to correctly verify proffered chits, to correctly serve stored data, and to only grant accesses designated by the chits. In the case where public keys are embedded into the chits, the server must challenge the client to prove possession of the corresponding private key before granting access.

The server does not have to be trusted to see plaintext; file data can be encrypted as long as all potential accessor’s have appropriate keys. Note that we can not derive such keys from the chit, as the server could always recreate anything from the chit.

The server does not have to create, maintain, or even know about mappings between users and public keys, beyond a single user who has a conventional user account on the server, and is given a master chit providing access to storage.

We do not necessarily trust the server with the mappings between users and specific keys or chits, but we do rely on the server to log accesses. Alice should be able to delegate access anonymously with respect to the server; the server authorizes accesses solely based on possession of an appropriate chit.

3 Motivation

We present different scenarios in which Alice, a hypothetical user with a conventional account the server, may wish to share data or resources in a controlled way.\(^1\) We use these examples to motivate the central features of chits.

- **All access for a single principal:** Alice stores her data and has unrestricted access.
- **Share partial views with a small group:** Alice may share some data with other members of a small collaborative group or social network.
- **Limited upload capabilities for a class:** Alice might wish to allow her students to submit their projects by uploading binaries to her server. She gives each the ability to upload a limited number of bytes.
- **Derive equivalent access rights:** Alice derives equivalent, but distinguishable, access rights to allow access from different hosts. Rights for a compromised host could be revoked independently.
- **Derive weaker access for named principals:** Alice gives Bob access to some resources, restricted in scope (which files, directories) and in rights (which operations).
- **Fine-grained revocation:** Alice becomes concerned with Bob’s trustworthiness, and decides to revoke some of the chits she has given him, but not all.
- **Anonymous access to public resources, where misbehavior is accountable when needed:** Alice may authenticate to a trusted entity that allows her to generate anonymous pseudonyms to use online. No pseudonym identifies Alice directly, but under extenuating circumstances like a court order, any pseudonym may be traced back to the original, verified identity.
- **Completely anonymous access to shared resources:** Alice may leave a comment on a blog or update a wiki without any form of authentication.

Of these scenarios, only the first and last are well-supported using current mechanisms. The rest are scenarios in which conventional distributed capabilities are unable to support confinement, accountability, and revocability.

**Confinement** is the ability to control the propagation (delegation) of capabilities. Alice might wish to delegate some rights to Bob (give a capability to Bob), but not allow Bob to delegate these rights to Trudy. Confinement cannot be achieved in conventional distributed capability systems, since a principal may simply copy her capability to another principal.

**Accountability** is the ability to recognize who abused resources and seek redress, perhaps by banning further access. Because capabilities and capability validations do not require identities, mapping an action back to a specific identity is not possible without additional mechanism.

**Revocability** is the ability to deny access once given, particularly in a fine-grained manner (for a particular user or file). Capability revocation is typically a “blunt instrument,” because revoking a capability revokes access for every principal using that capability. Revoking capabilities at a fine granularity requires the ability to distinguish between holders of a capability, or between specific instantiations of a common capability held by more than one principal. Conventional capabilities have neither.

---

\(^1\)Principals may be identified by public keys or another mechanism. Identities need not be certified by a central PKI; however, chits do not preclude a central PKI.
Our insight is that:

**Differentiation can substitute for identities.**

Identity is at the heart of accountability: it represents the principal responsible for an action. Differentiation, through labeling chits with indelible labels, makes it possible to map actions back to the specific chits that authorized them. By using labels in combination with server logs, post-analysis can identify chits used to abuse access privileges, and trusted users who have delegated access to abusive users. Online differentiation makes it possible to prevent unwanted delegation (confine), and to later rescind access (revocability). We believe that each of the sharing scenarios can be supported by a chit abstraction that exploits this insight.

### 4 Chit Fundamentals

We describe the chit through a concrete implementation. Other implementation approaches are possible (Section 7).

A chit is a capability designed for use in distributed environments. Chits are capabilities that grants rights to objects at a specific server. Chits are held by users, who present them to the server when requesting resources. A user mounts a remote file system by presenting a chit to a local client, which communicates with a (potentially remote) Neo server. Note that we use the term “user” to refer to any entity holding a chit. This is a much more lightweight concept than that of users in the typical operating system sense, who must have formally established an account on the server. Neo only requires a single user to have an account on the server. All other “users” are people, programs, or other entities that hold a chit.

Capabilities [29, 19, 25, 23, 31, 35, 33, 2, 7, 19] are unforgeable granting of rights, each implemented by crafting a secret for specific accesses to an object, or a set of objects. On a single machine, capabilities can be made unforgeable through protection by a universally trusted entity, e.g., the operating system kernel. In an untrusted distributed environment, capabilities must be protected through encryption.

A chit may be authenticated or unauthenticated. They differ in that authenticated chits embed public keys, which are used by the server to authenticate users. An authenticated chit is useless without the correct key, and therefore is not a secret and need not be encrypted. By contrast, an unauthenticated chit could be copied and used by other users if it is ever transmitted in clear-text. The entire chit needs to be treated as a secret, and be protected from reading by unauthorized entities at all times.

This functionality must be available to a client without involving the server. Though this approach has ancillary advantages (offloading server burden), the primary reason is that involving the server in all chit derivations would allow the server to see through the various levels of anonymity defined by users. A server in a chit-based system is trusted to be a fair broker in handling data and verifying chits, but not to know the mappings between users and the different identities supported through chits.

#### 4.1 Chit creation and derivation

Data in Neo is initially exported with a **master chit**, which grants unrestricted, anonymous access to the named data, much like conventional capabilities. However, into this chit a user can securely embed indelible **narrowings**: tags that might limit the chit to use only by certain principals, or with specific rights, or in a way that makes it easier to audit and revoke access. Our chits are implemented as XML objects that explicitly list all tags, as well as a **fingerprint**. The fingerprint is derived from a server-held secret and a hash (HMAC) chain that allows the server to efficiently verify that no tags were removed once embedded.

Chits are not actually modified. Instead, new chits are derived from old chits by adding ordered tags. In derivation, the old chit’s tags form a proper prefix of the tags on the new chit. Each of the new chit’s tags is secured into the chit by being added to the XML, as well as being hashed into the new chit’s fingerprint.

Derivation is performed by clients. The alterations to the chit that we describe are not destructive—the previous chit remains—however, the new chit may be delegated to another.

The primary tag types supported by Neo chits are the following:

- **label**: Alice can embed “labels” that will be logged by the server when a chit is used. These log entries can be read to audit accesses and, if necessary, to identify chits to be revoked.
- **narrow**: Alice can embed restrictions on both the portion of the file system reachable through a chit, and the type of accesses: e.g., **read-only** versus **write**.
- **expire**: All chits have an expiration, after which they are not accepted by the server. Alice may set the chit to expire earlier.
- **public**: An unauthenticated chit can be made authenticated by adding a public key, used by the server to authenticate the chit’s holder.
- **delegate**: Anonymous chits can be delegated by copying them and sharing with others. However, authenticated chits require a user to be authenticated via a public key challenge before the server authorizes access. Alice can embed a transfer of rights for an authenticated chit to Bob, so that the server will issue a challenge to Bob’s public key rather than Alice’s. Not all chits require authentication, but to transfer those that do requires embedding the consent.
of the chit holder.

- **nodelegate**: The chit may not be further delegated, allowing Neo to support confinement.
- **group**: Neo supports group certificates. A certifying authority may issue certificates that map between a group key and keys of individual users. Group members use a chit delegated to a group key by adding their certificate to the chit’s XML and fingerprint. Certificates may be issued independently of the chits with which they are used.
- **revoke**: The holder of a chit can create revocation certificates for any chits derived from that chit. Alice can revoke any chit derived from a chit that she holds, allowing revocation responsibilities to be distributed.

These operations can be combined: Alice may narrow the rights of a chit to permit only read-only access to a sub-directory of a file system, label the chit as “Bob’s” to discourage abuse, mark the chit to expire in one week, and delegate to allow the chit to be used only by the holder of Bob’s secret key.

### 4.2 Identities Supported by Chits

The chit allows embedding tags that may be: nothing, labels for local accountability, public keys for requiring specific users, and public keys that represent groups of (potentially anonymous) users. The flexible model of identity is key to making the chit amenable to both loosely- and tightly-controlled services.

**Anonymous** A chit could be completely anonymous, much like a capability, and provide any holder with rights on the named data. When Alice uses an anonymous chit, the server cannot identify that it is Alice and not another user with the same chit. Anonymous chits should not be transmitted in plain-text, since they may be stolen and replayed.Anonymous chits, like traditional capabilities, are most appropriate for single user access to private data, as stronger notions of identity are unnecessary with a single principal.

**Labeled** A chit could be labeled so that the server logs accesses by different chits differently. Alice could label her chits “Alice”; her accesses would be logged with that label. Before sending Bob a chit to grant access, she could embed a label “Bob”; the chit would have the ordered labels {“Alice”, “Bob”}.

When Bob uses his chit to access data, the server will log {“Alice”, “Bob”}. In the server logs, Alice can distinguish her accesses from Bob’s. Bob is able to embed labels of his own, but the new chits will retain the prefix {“Alice”, “Bob”}.

Even though labeled chits provide more accountability than anonymous chits, they are still vulnerable to eavesdropping and replay. Labeled access is useful for small, trusted collaborative groups.

**Authenticated** Chits that embed public keys allow the server to verify user identities, which makes provenance and accountability easier to support, reduces the need to protect chits from eavesdropping, and supports confinement: or control over the delegation of capabilities. Validation of a keyed chit requires first, as always, both the chit integrity check that ensures no embedded labels have been removed and a check that the chit has not been revoked; second, that any certificate chain embedded in the chit is valid; third, that the bearer of the chit possesses the secret key associated with the claimed public key by requesting that a nonce be signed. Note that even an authenticated chit is unusable for anonymous access only if the mapping between a user and the public key-pair described by the chit is known.

Chits with public keys are not vulnerable to eavesdropping (the public key is public) or replay (the server generates a fresh nonce for every access).

**Groups, and role-based access control** The **group** tag allows groups to implement role-based access control by establishing a per-role group key, constructing a chit that encodes access rights appropriate for the role, and delegating the chit to that per-role key. Any client with a certificate for that role, that is, a key signed by the per-role key to signify membership in the role’s group, may use the chit to assume that role by adding his certificate to a copy of the chit.

**Escrowed identities** Chits can create escrowed identities. If the chit includes the public key for group membership, the ability to join the group can be guarded by a separate certificate authority (CA) that grants a certificate based on “true” identity, but which includes a new key that cannot be matched with other accesses.

The certificate proves to the server that Alice was properly identified and authenticated by the CA. However, no information about Alice’s real identity is divulged in the verification process. The server knows that the real identity of this user is registered with the CA, but does not know this identity. The CA would only reveal the mapping between Alice and her public key if compelled, possibly due to a subpoena or a court order.

### 5 Neo

Neo is a wide-area file system built to enable dynamic, flexible access and collaboration across widely-separated machines in distinct administrative domains. Chits allow team leader Alice to create data at a remote server, and to distribute distinct sets of access rights and views of the shared data to collaborators. This is done without arbitration through remote servers, and without establishing per-collaborator accounts on the remote servers.

Nonetheless, labels and public keys embedded in the
chits could allow Alice to distinguish among accesses by distinct collaborators in logs maintained by the server, and thereby enforce accountability.

Neo chits are materializations of rights to access data maintained in server containers. A container identifies a namespace, a (possibly empty) set of data, a cryptographic secret (from which corresponding chits are derived), and a revocation list.

Figure 1 shows an example of data sharing in Neo. The example starts with 1) a client, Alice, requesting a container to be established on the server. The server 2) responds with a master chit that provides full rights and authority over the newly created container. Alice can use the chit to 3) write data to the container on the server. Alice might wish to allow Bob to read a part of the data at the server. Alice shares the data by 4) creating a new chit with a downgraded view and downgraded rights, and then sending the new chit to Bob. Finally, 5) Bob reads the data by presenting his chit to the server. Alice does not need to communicate with the server to create or authorize Bob’s downgraded chit.

Neo supports open-to-close consistency [14], meaning whole files are read from and flushed back to the server. With block servers, however, a file flush consists of dividing the file into fixed size blocks, taking the SHA-1 hash of each, sending the ordered list of hashes to the metadata server, and sending the blocks (if new) to the server.

5.1 Mounting Neo data

A Neo client mounts server data through one of the handshakes shown in Figure 2. Chits name a single server through a combination of the server’s IP address and its public key. Use of the public key as part of the address allows a single physical server to play the role of any number of virtual servers.

The unauthenticated handshake A client mounts a remote directory through an unauthenticated chit by authenticating the server and negotiating a session key. The client creates a session key and encrypts it with the server’s public key. The client then encrypts the chit, together with a random nonce, with the session key. Both ciphertexts are then sent to the server.

The nonce serves as a challenge to the server. The server answers the challenge by returning a plaintext copy of the nonce to the client. Unauthenticated chits do not require the server to authenticate clients.

The server must verify the chit before it responds to the client. Chit verification consists of ensuring that the fingerprint is derived from the server’s initial secret, together with all XML tags, through a proper hash (HMAC) chain (Section 5.3).

Chit derivation never increases the power of the resulting chit; adding a tag at most results in a new chit with equivalent power. Hence, a malicious user can not gain greater rights by adding tags to a chit.

The converse might seem more promising. Given a chit with a narrow tag, a malicious client might leave off that tag when sending the chit to the server, hoping thereby to gain broader rights than intended. However a chit missing a tag that was added to the fingerprint will not verify correctly. Given a chit with tag \( x \), finding a fingerprint that corresponds to a chit without \( x \) would require being able to reverse a cryptographic hash.

The authenticated handshake The authenticated handshake (Figure 2b) differs from the unauthenticated handshake in being based on a chit that contains a public tag, possibly together with multiple delegate tags.

The public tag tells the server that it must authenticate the client by challenging it to prove possession of the corresponding private key. The challenge is again accomplished by sending the client a random nonce, encrypted with the public key from the chit. The client authenticates itself to the server by returning the a plain-text version of the nonce.

Delegation adds more complexity on the server side, though not any additional communication. A delegate tag must be preceded by a public tag. The delegate contains a public key, and a signature of that
(a) Server authentication, unauthenticated chit

\[ (k_{\text{sess}} \text{serverpub}, (N_{C}, \text{chit})k_{\text{sess}}) \]

\[ N_{C} \]

(b) Mutual authentication, authenticated chit

\[ (k_{\text{sess}} \text{serverpub}, (N_{C}, \text{chit})k_{\text{sess}}) \]

\[ N_{C}, (N_{S})\text{clientpub} \]

Figure 2: Mount Handshake

\[
\langle \text{chit} \rangle
\langle \text{id}\rangle42\langle/\text{id}\rangle
\langle \text{serverpub}\rangle8\text{EA771}...\langle/\text{serverpub}\rangle
\langle \text{fingerprint}\rangle59085\text{F}...\langle/\text{fingerprint}\rangle
\langle \text{tags}\rangle
\langle \text{expires}\rangle2013-04-01 00:00:00 GMT\langle/\text{expires}\rangle
\langle \text{label}\rangle\text{bob}\langle/\text{label}\rangle
\langle \text{narrow}\rangle/home/\text{bob}\langle/\text{narrow}\rangle
\langle \text{remove-right}\rangle\text{write}\langle/\text{remove-right}\rangle
\langle \text{limit}\rangle\text{bw1}\langle/\text{limit}\rangle
\langle \text{public}\rangle887...\langle/\text{public}\rangle
\langle \text{delegate to}\rangle"773..."\langle/\text{delegate}\rangle
\langle \text{limit name}\rangle\text{bw1}\langle/\text{limit}\rangle
\langle/\text{tags}\rangle
\langle/\text{chit}\rangle

Figure 3: Simplified Neo chit in XML. The “bob” label can be used to distinguish accesses in the log. The chit allows read-only accesses to /home/bob and below. The profferer of the chit must be able to answer a challenge based on the public key in the “to” field of the delegate tag. The limit tag establishes a named limit that allows at most 1000000 bytes to be read.

Both authenticated and unauthenticated handshakes prevent replay attacks by basing nonces on timestamps, together with a small buffer of prior nonces on the server.

5.2 Sessions

Once the handshakes is complete, a session incurs very few ongoing costs. The server provides the client an access token at the handshake’s conclusion. The token is just the session key if the ongoing connection is encrypted. Otherwise, the token is a secret used to provide integrity through HMACs over message contents.

The server maintains per-client data for the life of a session. This data holds the access token, the final path narrowing, the final access rights, any chit labels or public keys, and any data limits. Each incoming request is integrity-checked or decrypted, and then checked against the client’s narrowing, rights, and data limits. If logging is enabled, the access type, labels, and the public key are logged at each access.

5.3 Fingerprint Creation and Verification

A chit’s fingerprint is derived from a secret specific to the chit’s target object, and the tags contained by the chit. Chit fingerprints are built recursively using HMACs based on SHA-1. Each new tag is embedded into the fingerprint by consecutively creating a new HMAC using the tag type and value as the message, and the previous fingerprint as the key. Hence, to label an existing chit with fingerprint $S$ with the string “Bob”, the tag $\langle \text{label}\rangle\text{Bob}\langle/\text{label}\rangle$ is appended to the chit’s XML representation, and the chit fingerprint is updated to $S' = \text{HMAC}(S, \langle\text{label}\rangle|\langle\text{Bob}\rangle)$.

The one-way nature of HMACs implies that any client with the fingerprint for chit $c$ can verify the fingerprint for any chit derived from $c$, but can not retrieve the fingerprint of any ancestor of $c$. Further, this approach makes the addition of new elements indelible. Removal of an element would entail the computationally infeasible requirement of reversing a cryptographic hash. Combined with the restriction that individual chit operators
only restrict the scope of a chit (or add labels), this irreversibility ensures that any derived chit can only be as powerful as its parent. This weakening relation is transitive: if Alice gives Bob a chit \( c \), Bob can never derive a chit from \( c \) that provides rights specified as lost in \( c \).

This chained HMAC technique is also used for fingerprint verification at a server. The server uses an object’s secret, together with the ordered set of XML tags to recreate the series of HMAC operations that result in the final fingerprint contained in the presented chit. If the result of the server’s computation matches the XML fingerprint, the chit is valid and the access is allowed to proceed. Otherwise, the chit is not valid. Consider a client wishing to modify this chit to encode additional rights. Tags can easily be removed from the chit document (which is just XML). However, any such alteration would be detected during the verification stage, as the result of the server’s fingerprint computation will not match the fingerprint in the chit.

Only the server can verify a chit back to the original secret, as only the server knows the secret. A client can verify that a chit has been properly derived from a second chit, but only if it has possession of that second chit.

5.4 Chit Delegation

Chits can be tagged with public keys to specify that either (a) only the holder of the corresponding secret key or (b) only the holders of keys certified by that public key may use the chit. We use the public tag to represent the first. Delegating to another user involves signing over the right to use that chit, through either the delegate or group tag.

The two types of delegation are implemented slightly differently. When the terminal key (the last key in the delegation chain) is a user’s public key, the user must sign both the new public or group key and the current chit fingerprint. By including the current chit fingerprint, the server can verify that the delegation of rights was made for this chit only.

The second mode, where a certificate is being added to a chit whose terminal key is a group key, the certificate (a public key signed by the group key) alone is required. A certificate can be copied out of a chit and added to other chits, whereas a delegation can not.

5.5 Fine-grained revocation

Fine-grained revocation is difficult with conventional capabilities in that policy must decide who can revoke which capabilities. Further, recall that capabilities are designed to be anonymous, therefore generally not enumerable either at clients or the server. Finally, revocation lists need to be located and maintained.

The structure of a Neo system makes the latter approach a non-issue. All data to be accessed is maintained at a central server, so revocation certificates located at the server are always searchable by the server when chits are verified. Further, both chits and revocation certificates have limited expiration windows, and can be garbage-collected by the server as they expire.

The issue of who can revoke which chits, and how those chits are named, can be solved naturally by leveraging chit structure. All chits for a single server are derived, at least transitively, from a single master chit. The chits so derived can be thought of as a tree rooted at the master chit.

We can characterize a chit by its tag path, which we define as the chit’s sequence of tags, id, expiration, labels, etc. If the tag path of chit \( c_1 \) is a prefix of the tag path of \( c_2 \), \( c_2 \) must have been derived, at least transitively, from \( c_1 \).

We can now say that the holder of any chit \( c \) can revoke any and all chits derived from \( c \). Assume Alice holds a chit \( c_a \) with (simplified) tag path:

\[
\text{label(alice)/public(98E3...)}
\]

and derives a chit \( c_b \) with tag path:

\[
\text{label(alice)/public(98E3...)/label(bob)}
\]

Alice can create a revocation certificate by deriving a new chit, \( c_r \), with a properly formed revokes tag.

\[
\text{label(alice)/public(98E3...)/revoke/label(bob)}
\]

\( c_r \) revokes any certificate whose tag path contains the revocation certificate’s tag path, minus the revoke tag, as a prefix. Hence, \( c_r \) will revoke \( c_b \), together with any chits derived from \( c_b \).

A “properly formed” revokes means there is at least one valid tag after the revokes tag. This requirement allows Alice to revoke any chit derived from \( c_a \), but not \( c_a \) itself.

Alice registers the revocation certificate by sending it to the server. The certificate is valid if it verifies. Revocation certificates are checked when an incoming chit is verified. Certificates are garbage-collected by periodically retiring expired certificates. Note that an expired certificate does not violate correctness, because no chit targeted by the certificate could have an expires time later than the certificate. Hence, any chit matched be an expired certificate will itself be expired, and hence not valid.

5.6 Data limits

Conventional file systems provide per-user write quotas, but Neo has no concept of users. Aligning quotas with specific chits does not work because individual data may be covered by many chits. Instead, we use path-dependent data limits to limit the ability of users to read or write data. A data limit has a
name, a type, a maximal amount, a current amount, and an expiration.

Data limits are defined when seen in a verified chit by the server. The server debits a data limit each time an access of an appropriate time occurs over a session defined by a chit with that limit in its tag path.

We support three types of limits: read limits, write limits, and space limits. The first two refer to bandwidth consumption, whereas the last refers to total disk space consumed.

Chits may contain multiple limits, and all are decremented when an access occurs. Different chits in a derivation chain might share the same write limit; an access through one such chit reduces the data available to all.

As stated, a user could mount a denial of service attack on a specific tag by adding the tag to his chit, and then accessing large amounts of data through it. We prevent this by having the server enforce a requirement that if multiple chits specify the same limit, they have a common ancestor after the limit is first defined. For example, chits with tag paths `label(alice)/limit(pot1)` and `label(bob)/limit(pot1)` would use different instantiations of `pot1`. Chits with `limit(pot1)/label(alice)` and `limit(pot1)/label(bob)` would use the same limit.

5.7 Labels and identity

Section 4 discussed how distinct labels can be used to differentiate among different chits. However, attempting differentiation merely by inclusion of labels allows attacks. If Alice creates a chit with “bob” and another with “charlie”, there is nothing to prevent Charlie from adding “bob” to his chit and attempting to blame Bob for his actions (both chits have the “bob” label!).

One way to prevent this attack is to make label values dependent on their position in the chits tag path. Label “bob” added after “charlie” would therefore be different than label “bob” on a chit where “charlie” was never added. We implement this by appending position information to labels when recording to the log. The positional information could then be ignored, or used to distinguish between chits.

5.8 Automounting and the purse

Automounting is a convenient way to access remote mount points only when needed.

Chits might proliferate, and a single user may have more than one chit that provides access to a specific file. The chits might vary in rights (e.g. “write” versus “read”), narrowings (e.g. “/home/keleher” versus “/home”, when the target is “/home/keleher/foo”), labels, authentication requirements, and data limits. The Neo client provides an encrypted purse that holds chits for a user.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cost (usecs)</th>
<th>Rate (per sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSA keypair creates</td>
<td>86286</td>
<td>11</td>
</tr>
<tr>
<td>RSA signatures</td>
<td>1332</td>
<td>750</td>
</tr>
<tr>
<td>RSA verifies</td>
<td>96</td>
<td>10416</td>
</tr>
<tr>
<td>AES encrypt 10K bytes</td>
<td>76</td>
<td>13157</td>
</tr>
<tr>
<td>HMAC 10K bytes</td>
<td>45</td>
<td>22222</td>
</tr>
<tr>
<td>HMAC 27 bytes</td>
<td>2</td>
<td>500000</td>
</tr>
<tr>
<td>DB insert 100 bytes</td>
<td>22</td>
<td>45454</td>
</tr>
</tbody>
</table>

Table 1: Operation costs on MacBook Pro.

The purse can also select a single chit from a set of alternatives for a specific access, according to some user-defined policy. The purse consists of a database with policies implemented through distinct SQL queries. We have implemented three simple policies. The privacy policy prefer chits that do not require authentication, and chits that have fewer embedded labels. The least-privilege policy prefers fewer rights, and then longer narrowings. The most-privilege policy does the reverse.

5.9 Logging, databases, and permanence

Neo logs access data to the same database that serves file metadata. Synchronous, per-insertion transactions are slow. Our current implementation improves this performance by more than two orders of magnitude by committing data only every thirty seconds. This weakens our guarantees, but they are still on par with other distributed file systems [32, 16, 14]. Further, the resulting data is consistent even if failures happen in mid-operation.

Database operations also become important when checking for revocation certificates at each stage of the server’s chit verification. Database searches can cost twice as much as a single HMAC operation during chit verification. We mitigate this cost by doing a preliminary check at the start of verification. We only do the full per-tag check if revocations exist, and contain at least one tag in common with the incoming chit.

The access log is made visible to the container owner through a read-only portion of the container namespace. As such, Alice can delegate rights to the log just as she would for ordinary data.

6 Performance results

This section describes the performance of Neo with several micro- and macro-benchmarks. Our intent here is not to show that Neo performs better than other systems, but to show that the broader functionality does not come with a penalty in decreased performance. As such, we made several concessions to ease of implementation, each of which does entail some performance
cost. The system is based on the Fuse user-level file system interface, which adds significant overhead in several API traversals, including user/kernel crossings for each upcall into the fuse file system. The fuse interface interferes with kernel-level caching, as numbers make clear in Section 6.3. All servers and clients are currently single-threaded. A large portion of overall latency is caused by single threading when requesting multiple blocks from the block server, and synchronously pushing out blocks when a file is closed. Asynchronous flushed, block caches, and multi-threading the servers would all improve performance.

6.1 Implementation

The Neo prototype, including all chit manipulations, is written in approximately 8,000 lines of C. The prototype consists of three single-threaded servers: the metadata server running on the server, the client (which is a Fuse server) running on the user’s machine, and the block server running anywhere. The prototype uses libtomcrypt for cryptographic operations, ZeroMQ for network communication, Google Protocol Buffers for data serialization, and sqlite for fast access to meta-information on both the client and server. We use 160-bit SHA-1 hashes, 256-bit AES for payload encryption, 2048-bit RSA keys, and 16K data blocks by default.

Chit use and verification does not add network communication, other than a one-time challenge-response RPC. The cost of local operations is dominated by the cost of common cryptographic primitives, such as creating chit fingerprints using HMACs, and verifying delegation signatures with RSA. The least capable machine we use can perform ~750 RSA signature creations or over ~10000 verifications per second, with 2048-bit keys.

We realize a chit in plain-text XML, and embed labels indelibly by including a “fingerprint.” This fingerprint is seeded (and the original for each object remembered) by the application, but is updated with each new tag appended to the XML using an HMAC based on the SHA-1 one-way hash. The one-way HMAC ensures that no removal of rights (by adding a tag) can be undone (by removing a tag) without detection. Additional restrictions can always be added because the new fingerprint is a function of the old fingerprint and the new tag.

Our file system runs on a variety of platforms, but as our primary concern is the cost incurred while starting a session between a single client and a single server, the performance numbers in this section are derived from a client running OS X 10.8 on a Macbook Pro, Intel Core i7 @ 2.6 GHz, and a server running Linux Ubuntu 12.04 on a recent node with an Intel Core i5-3470 @ 3.2 GHz. All single-machine numbers are derived from the least capable of our machines, the Macbook Pro, where Table 1 shows the cost of cryptographic and database operations.

We have two settings. The local setting has all servers on the same local-area Ethernet. RPC’s take several hundred µsecs, and effective bandwidth is 94 Mbit/sec. The remote setting consists of a server at the university and a client at a residence. RPC’s take approximately 13 msecs, and effective bandwidth is 37 Mbit/sec.

All aspects of the system described in Section 5 are implemented, save automounting (though policy-based chit selection is implemented).

6.2 Handshake cost

Neo’s primary overhead is incurred during the mount protocol. Mounting a chit requires the server to be authenticated to the client, and the chit to be verified. The latter includes challenge-response RPCs if there is delegation for strong identities, public-key decryption for each link of a chain of delegations, and a series of lightweight HMAC operations. The client and server also create symmetric session keys for use in either authenticating or encrypting traffic between them.

The unauthenticated case requires only a single RPC,
but authenticated chits require an additional RPC for the client to prove possession of a private key. Nonetheless, Figure 4 shows that overheads are dominated by public-key crypto, in both cases. Reading and parsing chits is non-negligible, but verification is more costly, at least for authenticated chits.

All of these categories can become more expensive as the chits increase in size and complexity. Figure 5 shows chit verification costs for different numbers of label and/or delegations. We expect the usual number of such to be quite small, i.e., < 10. However, there could certainly be cases where the numbers increase. Still, verifying a chit with 100 label tags costs less than a millisecond. Verifying a chit with 100 public-key delegations, which requires verifying RSA signatures at each step, tops out at slightly more than 9 msecs. Such a chit is over 100 KBytes in size, as it must contain 100 complete public keys, and 100 signatures.

There are several ways these figures could be improved. Most usefully, the server could maintain a chit prefix verification cache. The server would place a simplified representation of the chit, together with final hash value, in the cache at each verification step. The cache would be checked for common prefixes for subsequent verifications; hashing would only be needed for the portion of the new chit that is unique. Imagine a work group or classroom where each group member has a chit that only differs from others in the group in the last tag. Verification for all but the first would require only a single step. The cache would work for all chit tags, including delegations.

6.3 Overall performance

Figure 6 shows the overall performance of Neo versus asynchronous NFS v4 and AFS [14] with servers on the same machine, servers on other machines of the remote network, and with servers on a remote network. We have three simple benchmarks. copy-in refers to copying approximately 11 MBytes of data, consisting of the Neo source, plus two sqlite databases, onto a remote mount. compile refers to compiling the Neo source, and copy-out is the cost of copying the directory, complete with objects files and executables, back out (approximately 20 MB). In all cases our results are the average of 10 iterations after caches have been warmed.

The labels can be decoded as follows. Each is the file system name concatenated with one or more letters specifying where the servers are located. An ‘r’ refers to a server on a remote network, an ‘l’ is a server on the local network, and ‘s’ means the server is located on the same machine. NFS and AFS have only a single server, while the Neo labels refer to the metadata server and the block server, respectively. The neo-lle setting is with both servers located elsewhere on the local area network, and all data encrypted.

Looking first at the remote numbers, both NFS and AFS perform much worse than Neo in the copy-in phase. Our only explanation is that NFS and AFS must wait for high-level acknowledgments for a blocks before the next chunk is sent. Neo aggressively pushes blocks into the outgoing ZeroMQ channel and relies on the underlying windowing algorithm to send data efficiently.

The compile phase is very different in that AFS’s default cache size is large enough to fit the entire compiled directory. Object files would be cached locally and
not have to be read over the network, and some data would be cached between runs. As a result, AFS performs more than twice as well as either NFS or Neo with remote servers. However, when Neo’s block server is on the same machine (roughly equivalent to a very large cache) while keeping the metadata server remote, Neo is slightly more efficient than AFS. The copy-out phase is extremely efficient for both NFS and AFS due to caching, whereas a remote metadata server makes Neo relatively slow.

The local network numbers are qualitatively similar, with the exception that AFS is slightly faster than Neo even when the block server is local. Finally, Neo outperforms NFS even on the same machine.

Overall, Neo performs quite well compared on NFS in all environments. Neo does not perform as well as NFS in remote situations, though with a block server on the same machine it outperforms NFS in the remote setting.

The OpenAFS implementation several advantages, including large caches, 32 KByte blocks, being layered on UDP, better kernel integration, and a mature code base. However, the same-machine data server numbers imply that Neo will be competitive, at the very least, when we add a client cache.

7 Related Work

Capabilities have a long history of use in (research) systems [29, 19, 25, 23, 31, 35, 33, 2].

Most systems do not support containment, though monitored communication channels can be used to prevent unauthorized delegation in a single OS [33] or a Java virtual machine. Unfortunately, the multiplicity of communication paths, trust domains, and user policies make a monitored restricted communication model infeasible in large distributed systems.

Kerberos [26, 17] allows users (represented by public keys) to access services across trust “realms,” authenticated through certificate chains. However, all users must have identities known to and certified by the trusted KDC or CA.

Many others systems control access by first authenticating users through trusted certificate authorities [21, 1, 13]. SFS [21] uses self-certifying pathnames to authenticate servers and trusted CAs to authenticate users via public keys. Although SFS is agnostic in how public keys are generated and verified, clients must be mapped to pre-established local identities. Kaminsky [15] extends SFS by allowing ACLs to contain chains of indirection. Indirection allows access to remote users who lack local identities, but requires that security policies be implemented on the server and that, for adequate performance, the system communicate to maintain state proportional to the number of identities in the system.

Recent work in establishing accountability includes tracking the provenance of individual files [24] and establishing verifiable audit trails of distributed applications [36, 6]. Logs of chit labels could contribute to such an audit trail.

The increasing research focus on wide-area environments increases the importance of security. Data can be stored on untrusted servers [20, 8] (with histories secured through signed hash chains), staged at untrusted surrogates [34], and updated consistently despite Byzantine failures [3, 18].

Several file systems have used capabilities as the basis of authentication. Felix [9] and Swallow [30] use randomly-generated file IDs as capabilities. Cryptographically-protected capabilities were introduced in Amoeba’s Bullet file system [35], where they were they were used for authorization.

CapaFS [31] and DisCFS [23] are file systems based on cryptographic capabilities. CapaFS can encrypt access rights and the intended recipient’s public key, within the file name. Delegation and narrowing is achieved by appending to the name new encrypted extensions. DisCFS uses certificate chains based on KeyNote [5], which allow more elaborate graph-based inferences of delegation when compared to CapaFS. DisCFS assumes trusted clients and verifies only a few operations, avoiding the high costs of asymmetric cryptography and the logic inference engine. Ceph [27] uses simple cryptographic capabilities to grant access to extremely large-scale network attached storage.

SharedViews [10] and HomeView [11] are both systems that allow online media to be shared using simple capabilities. SharedViews integrates the capabilities directly into SQL, allowing users to export and share dynamic views.

Persona [4] is a social network based on attribute-based encryption (ABE). ABE allows communication channels to be tailored to logical predicates, and could form the basis of an alternate implementation of chits.

8 Discussion

This paper describes Neo, a wide-area file system based on powerful capabilities called chits. A chit is a unified abstraction that allows disparate access control policies to be easily defined and implemented. Chits are expressive and customizable in the types of identities they support, in the access rights they encode, and in the properties they can be used to maintain. Our intent was to support a broad range of possible policies within a single mechanism, allowing us to build a system that has the following novel abilities:

- Policy is formed and defined by without consulting
administrators. User-defined policies specify who may access which data, and for how long.
- Policy is implemented by a trusted server. However, the server has no need to know the identities behind the policies it executes, nor to map them to local user accounts.
- Fine-grained revocation, support for accounting and accountability, data and space limits, and confinement are all strongly supported.

Neo can be thought of as an end-to-end exploration of chit use in the wide area. The resulting system is extremely expressive, performing well while supporting an entirely novel combination of properties.

Our future work will include exploration of replicated servers, untrusted servers, and other uses for chits. We have already built a Web-accessible shared calendar using chits, and are investigating their use in social networks.

References