Authorizing Applications in Singularity

Ted Wobber
Microsoft Research, Silicon Valley

Aydan Yumerefendi[1]
Duke University

Martín Abadi[2]
Microsoft Research, Silicon Valley

Andrew Birrell
Microsoft Research, Silicon Valley

Daniel R. Simon
Microsoft Research, Redmond

ABSTRACT

We describe a new design for authorization in operating systems in which applications are first-class entities. In this design, principals reflect application identities. Access control lists are patterns that recognize principals. We present a security model that embodies this design in an experimental operating system, and we describe the implementation of our design and its performance in the context of this operating system.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection – *access control, authentication.*

General Terms

Security.

Keywords

Access control, application identity, capabilities, channels, delegation, pattern matching, regular expressions.

1. INTRODUCTION

A central concern in securing a computer system is access control. Classically, access control employs a reference monitor: a trusted component that (at least notionally) makes all access decisions [4]. As input to each access decision, the reference monitor is presented with the identity of a principal, the identity of an object (system resource or data protected by the system), and the specific operation that the principal requests on the object; the reference monitor then produces a Boolean outcome.

In the classic design for this purpose, each principal is identified by an integer identifier (a SID in Windows, a user ID in UNIX-based systems). Reference monitors rely on access control lists (ACLs), kept with each object for each possible operation. Each of these lists enumerates a set whose members are either principals or identifiers for groups. A group, in turn, is defined as a set whose members are either principals or identifiers for further groups. Access is permitted or denied on the basis of the presence of the requesting principal in the closure of the relevant ACL and its constituent groups.

We argue that the classic design is weak in two specific areas:

- In real systems, access requests are made by applications, often on behalf of users. However, most traditional security models do not offer an accurate way of expressing application identity and attributes in principals.
- User identity is often modeled uniformly regardless of how a user is authenticated or what execution precedes a request. As such, it is difficult to consider factors such as authentication strength, location, delegated authority, and application invocation history.

This paper presents a design for addressing these weaknesses. We represent principals as textual names combined in a syntax that describes applications, users, and invocation histories. ACLs are patterns against which principal names are matched. We demonstrate our design in the context of the Singularity operating system [19,20] and present a security model that takes into account the fundamental aspects of Singularity such as the channel abstraction, application manifests, and software isolated processes. Our design enables the deployment of security policies that are difficult or impossible to express in other frameworks. For example, an application can have private access to application-specific data while also maintaining rights based on the identity of the user who invoked it. Access policy can distinguish different applications capable of authenticating users. Moreover, access to a given resource can depend on the presence of only trusted applications in the invocation chain from user login to access request, on the identities of the publishers of these applications and on assertions by those publishers. These assertions should alleviate the task of defining and managing security policies for administrators and users.

Our work is guided by the following design principles:

- **Applications are first-class entities.** As such, they are distinct from users. It is insufficient to treat an application as another class of user. In many cases both application and user identity are needed for making access decisions.

---

[1] Work performed at Microsoft Research, Silicon Valley.
[2] Also affiliated with the University of California, Santa Cruz.

EuroSys'07, March 21-23, 2007, Lisboa, Portugal.

- **Impersonation is deprecated.** In many programming environments, executing code can adopt multiple identities (for example using UNIX `setuid` or impersonation in Windows [25]). By its very nature, this flexibility creates opportunities for attackers to gain unintended authority or for programmers to leave security holes.
- **Authorization decisions are late-bound.** In many systems, intermediaries make security-relevant decisions that cannot be observed by the authorizing agents that control resources. Hence, access decisions can depend on implicit (and often non-obvious) chains of trust. Instead, we attempt to back-load this process by examining as much security-relevant data as practical at authorization time.
- **Applications carry security policy.** Software authors and publishers often know more about an application's security environment than the users or administrators that run it. Published applications should carry annotations to help guide the instantiation of security policy.

In the course of this paper, we describe how these principles are reflected in the design of our system. The remainder of the paper is organized as follows. Section 2 gives a brief overview of Singularity. Section 3 describes our security model for Singularity and Section 4 details its implementation. Section 5 gives some motivating examples of how security policy can be deployed. Section 6 presents benchmark results for access control check performance. Section 7 describes work related to our approach and Section 8 concludes. Portions of this work have appeared earlier in preliminary form [1].

2. SINGULARITY

Singularity is an experimental micro-kernel operating system in which all inter-process communication takes place over bi-directional, typed channels. A channel's contract specifies a state machine that defines all allowable message flows [13]. Each channel has exactly two *endpoints,* an export side and an import side, and each channel endpoint is owned by exactly one process at any given time. Channel endpoints are fundamentally asymmetric. The export side is held by a provider of a contract (a server) and the import side is held by a consumer of that contract.

In Singularity, the *software isolated process* (SIP) is the fundamental unit of execution and isolation. All executable code for a given SIP is known and fixed at process startup. Thus, runtime code extension is not allowed. The OS kernel and all applications are largely written in Sing# [13], a type-safe programming language derived from C# [11]. Although traditional hardware-assisted isolation is possible [3], isolation between SIPs is most often provided by the language type system rather than by hardware protection. Singularity makes use of large parts of the .NET Common Language Runtime, but we do not adopt the .NET security model. Direct memory sharing between SIPs is never allowed. Singularity defines an Application Binary Interface (ABI) through which applications access OS kernel functionality, but all interactions between applications are conducted over channels.

Much of the Singularity design has so far focused on operations within a single machine. Notably, the channel infrastructure currently has no implementation that works across machine boundaries. However, the security model described here should apply equally well to a distributed environment, a question to which we return in Section 3.3.2.

3. SECURITY MODEL

In the following discussion, we describe the security model we define for the Singularity environment. Because Singularity is a micro-kernel system, most important system components such as file systems and device drivers are not part of the kernel. Thus, almost all important security policy boils down to controlling access to resources requested over channels.

3.1 Processes, channels, and endpoints

Each process acts under the authority of one principal and that principal's name is by default associated with all endpoints the process owns. Processes can transfer authority by transferring endpoints or delegating rights to other processes; however, the principal name associated with transferred authority always identifies the holder. One or more processes derive from loading and running an application. An application is described by a *manifest*. The principal name associated with a process derives from its application manifest and the principal name of the process that invoked it. The Singularity kernel allows a process to discover the principal name associated with the partner of any of its endpoints. The resulting principal name is then used to make access decisions pertaining to messages received over channels. Thus, access control in Singularity is discretionary.

Singularity endpoints can move. To determine that a message was sent by a specific process, the partner endpoint must not move between when a message it sent and when its recipient observes the sender. To enforce this, Singularity imposes static restrictions on the movement of endpoints. An endpoint can move only from a process via a message send, and then only if the local endpoint state machine is in a state that allows that process to send. Therefore, when a server receives a request on a channel, the server can know that the partner endpoint has transitioned from send to receive state, and thus the principal associated with the partner endpoint must be the same as when the initial message was sent.

In our model, the universe of protected resources – objects in the access control matrix [23] – is not predefined. Applications act as reference monitors and define the set of objects that they control. For example, a file system application controls the files and directories that it serves, a printer subsystem controls access to the set of printers, and a network stack driver controls access to the specific protocols it supports.

It is common for a contract provider to enforce access controls on the actions of an importer. The import side can impose controls on the server, of course, but in general applications often rely on system services, such as the Singularity Directory Service (described further in Section 5.2), to provide channels trusted to speak for named system resources such as files or hardware devices. (Here, and throughout this paper, we use the phrase "speak for" in the same sense as Lampson et al. [22], but we leave the use of a formal logic of "speaking for" to further work.)

The fact that a request was made by a principal over a channel does not give the receiver the right to act as that principal in subsequent communications. Authority can be passed between processes only through application invocation or through the mechanisms discussed in Section 3.5. Hence, impersonation as practiced in Windows [25] is neither supported nor required. Instead, we offer a precise grammar for principal names that describes arbitrary sequences of application invocations and delegations, where each sequence element is visible to the ultimate reference monitor.

3.2 Application manifests

Singularity applications are described by manifests. Manifests perform a similar function in the Microsoft .NET architecture [9, 24]: they carry metadata that describe code assemblies. In particular a manifest can include *strong names* [26] for the named assembly as well as all assemblies it depends upon. Strong names define not only a module name for a referenced assembly but also a key that can be used for verifying a signed hash of the contents of each assembly. Thus, a manifest can be used to validate the identity of an application and all its dependencies.

Singularity extends the .NET use of manifests in several ways. First, the top-level manifest for an application specifies both a publisher name and a publisher-designated application name. Publisher names are Internet domain names; although our design in this area is not final, a valid publisher name can be mapped to a public key through an X.509 certificate hierarchy. The *manifest name*, which we use to identity the application in our access control grammar, is a concatenation of the application name and publisher name. The application manifest also contains a signature that applies to all its metadata under the publisher key. Hence, we can create a trusted path between the local root certificate(s) for published code and the code assemblies present at application loading time. Only if all the signatures verify is the application granted the specified manifest name. For instance, the *login* application published by *singularity.microsoft.com* would be called *login.singularity.microsoft.com*. For the rest of this paper, we may omit explicit name qualification for brevity, e.g., by writing *login* rather than *login.singularity.microsoft.com.* If allowed by local policy, applications lacking verifiable publishers can be granted manifest names with a default publisher name such as *unknown*.

In Singularity, application manifests have been extended to reflect *privilege assertions* that stem directly from source code annotations. In other words, the author or publisher of an application may assert that it should have a specific privilege at runtime. Local policy dictates which publishers should be allowed to grant which privileges. The use of privileges and privilege assertions is discussed further in Section 3.4.2.

3.3 Principals

There is exactly one principal associated with each Singularity process, and that principal is immutable. This design decision might make certain types of applications more difficult to implement; however, we believe that it will produce a system that is less prone to the misuse of multiple authorities. Many cases in which a computation must execute under more than one identity can be implemented in our design by spawning multiple processes. The reasonably fast process creation in Singularity [18] makes such an approach practical. Nonetheless, Singularity also provides a restricted mechanism for delegating authority that is then associated with a specific channel. Under certain conditions, then, a process can hold channels with authority other than the process default. This mechanism is discussed in Section 3.5.2.

ApplicationName	AN = STRING
PublisherName	PN = DomainName
ManifestName	MN = AN "." PN
RoleName	R = STRING
Application Role-	AR = MN | AR "@" R
Principal	P = AR | P "+" AR

Figure 1: Principal grammar

A principal name is a string constructed from publisher names, application names, role names, and the operators "@" and "+" according to the grammar described in Figure 1. Principals in this grammar are *compound principals* [22]. Application invocation is the primary means of establishing new principals.

As discussed in the previous section, application loading involves checking a manifest's signature and the signatures on all included components. If all signatures are in order, the application can be assigned a manifest name *mn* formed from the specified publisher name and application name. During process invocation, the following method is called.

Process.Create(*mn, role*)

The resultant principal has the form *parent@role + mn*, where *parent* is the principal name of the parent process and *role* is an optional modification of the parent's authority.

In other words, occurrences of the "+" operator within a principal name represent the history of application invocations that resulted in the currently executing application. Occurrences of the "@" operator indicate where an application has decided to adopt a distinguished role. This indication says nothing about whether the role is more or less privileged — that has meaning only to the extent that ACLs grant more or less access to the new principal name. Role names are usually defined by the applications that adopt them and they have no explicit formal structure. However, it will be practical in many cases to qualify role names with a code publisher name or some other global qualification so as to avoid naming conflicts.

Certain applications can truncate the invocation chain. In other words, these applications are sufficiently trusted to act as the head of any invocation chain that contains them. Such applications are often single-function services that run independently from the parties that invoke them, for example file servers, network protocol services, or user-authentication applications. We discuss how we distinguish the applications in this class in Section 5.1. Truncation is essential to the simplicity of principal names and access control policies.

Some systems use integer values to reflect the identities of principals. Doing so requires a significant infrastructure to map between human-readable names and integer IDs. On the other hand, integer IDs offer the simplicity of dealing with fixed-size numbers, and guaranteed uniqueness over time. Despite these differences, one could easily construct data structures that mirror our compound principal names where our textual identifiers are replaced with integer IDs. The details of pattern matching for access control would, of course, be different, but our general approach would still apply.

It is tempting to include more information in principal names. In theory, one could include all sorts of information about a process: its parameters, its initial file system handles, all processes it ever invoked, etc. However, it would rapidly grow difficult to specify useful access control policy. Therefore, our choice to use application invocation lineage as critical marker in principal names reflects a tradeoff between completeness and simplicity.

3.3.1 User principals and application roles

One critical use of roles is to indicate when an application makes an authentication decision. For example, the system might run a console login application that executes as a principal *login*. When the console login application has received a satisfactory user name *andrew* and password, it will use Process.Create to start running a new shell as *login@andrew + shell*.

Similarly, we might run the application *sshd* to listen for incoming SSH connections. After satisfactory authentication through the normal SSH public-key mechanisms it might fork a new shell process under the principal name *sshd@andrew + shell*.

In these two scenarios, if *shell* decides to run the *cat* application and *cat* tries to open a file, we would have an access request to the file system from either the principal *login@andrew + shell + cat* or the principal *sshd@andrew + shell + cat* respectively. The reference monitor for the file system would then consult the ACL on the requested file to decide whether the given principal should be granted access. This ACL might well differentiate the rights of a user who is physically present (e.g. via console login) from those of a remotely logged-in user.

Another example of the utility of roles arises in the context of application installation. Suppose that there is an application *install* that manages the installation of new software. It would be natural for such an application, having checked that it is installing certified Microsoft software, to adopt the role *install@ms*. Acting in this role, the installer might gain permission to update files under `/apps/ms` (as well as other related system resources), but without having rights to resources designated for other publishers.

Nowhere in these scenarios has the system trusted any of the software involved: *login*, *sshd*, *shell*, cat, or *install*. All the system did was to certify the application invocations involved, and that, for example, *login* and *sshd* chose to adopt the role *andrew*. In this design trust occurs only in certifying application manifests (trusting that the applications really deserve their given names) and as a result of the way in which we write ACLs.

Note that the syntax of role names does not preclude user names authenticated in the context of a network authentication system that supports global naming. It would be perfectly acceptable for a globally aware *winlogin* application to adopt a user role such as *wobber.microsoft.com* and thereby to authenticate as *winlogin@wobber.microsoft.com*.

Atom = STRING | "." | "@" | "+" | "!"
Item = Atom | "(" ACL ")" | Item "*" | "{" ExprName "}"
ExprName = STRING
Seq = Item | Seq Item
ACL = Seq | ACL "|" Seq

Figure 3: ACL grammar

3.3.2 Remote principals

As discussed in the last section, remote principals can be acceptably authenticated by applications (such as sshd) that are prepared to handle remote entities. However, this approach violates one of our guiding principles which is to make late authorization decisions whenever possible. If a request comes from a remote host that supports a compatible software stack, there is no reason why application invocation or delegation chains cannot cross machine boundaries.

In this situation, if application instances on the two machines are equally trusted, then machine names are irrelevant to authorization decisions. Therefore, newly launched remote processes or delegated endpoints can receive a principal name exactly as in the local case.

Otherwise (that is, when trust policy is not uniform across machines), the situation gets harder. Our principal grammar can easily be extended to support machine IDs, so that access decisions may depend on these IDs. This extension is not sufficient, however. Our principal naming structure does not name credentials, but credentials (for example cryptographically signed statements) must always be present to accomplish remote authentication. Trusted Platform Modules [31] and attestation [12] can provide strong credentials for certifying the presence of remote software stacks. Fine-grained attestation [27] or attested labels [28] might also fit well with our notion of privileges. However, there still must be a service, perhaps an authentication agent as described in [33] that is capable of offering and checking credentials for named principals as part of remote authentication. When a remote entity is authenticated, it must be up to local policy to determine how the remote principal is named in the local principal grammar. For example, perhaps a certain publisher is trusted on node A but not known on node B. When B interprets a principal that names such a publisher on node A, the publisher may appear on node B as the *unknown* publisher.

3.4 Access control

With complex principal names such as those we propose above, having an ACL be merely a list (or set) of principal names does not give us nearly enough convenience and expressive power. For example, we might want to give access to a user while executing some of a particular set of applications, or when authenticated by some particular set of applications (e.g., *login* or *sshd*, but not *ftpd*); or we might want to give access to an application regardless of its user. While we could perhaps list all allowed principals, that would be awkward at best. Instead we use patterns that recognize principal names.

The exact pattern recognition language that we use is not critical to this idea, although the choice of language will certainly have an impact on the usability of the design, and therefore on the security of the resulting systems. We present here a recognizer for a specialized subset of regular expressions. Obviously, more or less complex recognizers are possible, allowing the expression of more or less complex access control policies. Performance is, of course, a significant factor in this choice. We believe that regular expressions offer a good balance between expressiveness and performance.

An ACL is a string constructed from names, partial names, and special operators from the namespace of applications and roles as depicted in Figure 3. The matching rules are similar to those for conventional regular expressions:

- any Atom matches itself;
- "!" matches any partial name, where:

  PartialName = STRING | STRING "." PartialName;
- "(ACL)" matches ACL;
- "Item *" matches zero or more sequential occurrences of Item (greedily);
- "{ ExprName }" matches whatever is matched by the ACL subexpression named by ExprName;
- "Seq Item" matches Seq followed immediately by Item;
- "ACL | Seq" matches either ACL or Seq.

A principal "P" matches an ACL "A" iff the string P matches the regular expression that is the contents of A. The match must be complete — all of P, not just a substring of it.

We do not currently express negative ACLs in our system. However, matching a negative ACL is not much different from matching a positive one, so there is no reason to expect that negative ACLs would be more difficult in this system than in other implementations of access control.

3.4.1 Permissions

Access control checks often apply to specific access modes, or *permissions*. An application seeking a specific permission to a resource can be thought of as acting in a specific role. For example, the if principal

login@ted + app

requires *read* access to a resource, we can model the requesting principal as the role:

login@ted + app@read

This principal might be matched by the ACL

(!@ted +!@read) | (login@ted +!@write)

which grants read access to *ted* logged in by any authenticator and subsequently running any application. However, write permission is controlled differently and is allowed only to *ted* when authenticated by the *login* application.

3.4.2 Subexpressions

Subexpressions provide a means for sharing ACL fragments among multiple ACLs. Subexpressions are referenced by name in the ACL grammar. When an ACL is interpreted, the subexpression name is resolved into a value and this value is substituted into the containing expression, recursively. We implement two standard mechanisms for mapping subexpression names to values.

In the first mechanism, we interpret names as file paths within the Singularity Directory Service, and the resulting subexpressions are the contents of the named files. Using the file system namespace to store subexpressions permits application installers to define security groups by simply creating files. Protection of the Singularity Directory Service itself is discussed in Section 5.2.

The second mechanism applies to subexpressions whose names begin with the character '$' by convention. These names are treated specially: they are interpreted by the Singularity kernel as follows. System policy defines a collection of names that map to common subexpressions. If the supplied name is in that collection, the corresponding subexpression is returned. If the supplied name is not present, it is deemed to name a privilege as discussed in Section 3.2. For each privilege, system policy defines an ACL that lists the publishers that can grant the privilege to running code. When such a privilege is evaluated as a subexpression name, the kernel consults this system policy and produces a list of all known applications that assert the privilege and have publishers that can grant it.

$user	*{$auth-privilege}@!*
$app	*!|{$user}*
$any	*{$app}(+!)**

Figure 2: Example kernel-defined subexpressions

A few example subexpression mappings are given in Figure 2. In the examples, *$user* resolves to the contents of the subexpression name *$auth-privilege* in an arbitrary role. *$auth-privilege* is a privilege granted to all applications generally trusted to authenticate users, such as the login application. The subexpression name *$app* matches any application or anything that matches *$user,* and *$any* matches anything that matches *$app* plus an arbitrary sequence of subsequent invocations. Hence *$any* matches any sequence of applications headed by either an application or a logged-in user.

An evaluation of a privilege subexpression need only include currently running applications. An application that is not running cannot appear in a principal and therefore need not appear in an access control expression intended to match that principal. This optimization has the potential to reduce the cost of privilege evaluations substantially. To prevent the results of expansions from varying frequently when applications are short-lived, we return matching applications that are running or have recently run.

File-path subexpression resolution and privilege evaluation correspond to what might be called the *push* and *pull* models of authorization [22]. In the push model, privileges associated with principals are gathered *a priori* and presented to the reference monitor for checking. Windows Security [10] and DCE Security [30] both implement this model, with group credentials gathered at login time or at first access to a remote domain. In the pull model, group memberships are gathered at the reference monitor on an as-needed basis with caching and pre-fetching as required for performance. Taos authentication [33], the distributed security model proposed by Kaminsky et al. [20], and the distributed authorization prover of Bauer et al. [7] are examples of this technique. Our authorization mechanism supports both models: expressions defined in the file system are pulled while privileges inferred from application manifests are pushed.

3.4.3 ACL examples

In general, we find that principal names often conform to the following pattern:

firstApp[@user] + middleApps + lastApp

For most access decisions, *firstApp* and its optional *user* role are of primary importance. *lastApp* actually makes the service request mandating an access decision, and is therefore also important, particularly where it is desirable to assert that only a specific application can access a certain class of resources. Specifying particular *middleApps* is useful for restricting access to a limited set of application, but more complicated expressions here probably have diminishing benefit.

It might be profitable to create templates for ACLs with appropriate policy defaults for *firstApp*, *middleApps*, and *lastApp*. Thus, ACLs for common cases could be specified with just a user name or subexpression.

In the rest of this section we briefly discuss some example ACLs and their interpretations.

login@ted + app

> This ACL matches exactly the *login* application in the role *ted* running the `app` application.

*login@ted (+!)**

> This ACL *m*atches the *login* application in the role *ted,* possibly running any application or chain of applications.

*login@ted (+{$trusted-apps})**

> Like the previous example, this ACL matches the *login* application in the role *ted*, but in this case any child application must match the subexpression *$trusted-apps*.

login@ted + script-engine@script + {$script-tools}

> It is often desirable to constrain scripts to run within the context of a specific scripting engine. This ACL matches only when the *login* application in the role *ted* runs *script* under a specific *script-engine,* and the script runs any application in *$script-tools.*

*{$trusted-auth}@ted (+!.adobe.com)**

> This ACL allows access to any application that matches the *$trusted-auth* subexpression, in the role *ted*, and possibility running application(s) published by *adobe.com* (recursively).

3.5 Transfer of authority

We anticipate that requiring every process to speak for exactly one principal might be lead, in some cases, to poor performance and awkward design. Our security model, therefore, supports two different means for transferring authority between processes. The first is a capability-passing mechanism that allows clients to transfer the right to access a given resource to other clients. The second is a delegation mechanism that permits clients to endow delegates with certain aspects of their authority dynamically (e.g., over multiple objects). Other means for authority transfer are easily imaginable.

3.5.1 Capabilities

Channels in Singularity can be viewed as capabilities [5]. Possession of a channel endpoint gives the caller the right to pass messages over a channel. Whether the caller can successfully invoke a service by doing so is at the discretion of the service provider. A channel over which the service provider imposes no further access checks (after initial binding) is essentially a capability. Service providers can and often do perform discretionary access control checks prior to granting capabilities. For example, file descriptors (as in UNIX) can be thought of as capabilities that are granted only after an access control check at file-open time.

One common paradigm for granting capabilities in Singularity is for a client to create an endpoint that entails a desired set of permissions (rights) and then pass that endpoint to a service provider. The service provider performs an access check on the caller's principal name and desired permissions, and if allowed by policy, accepts the channel binding. Our design allows the client to create channel contract subtypes that imply a specific set of permissions by restricting permissible message sends. Thus, a contract for accessing files might specify *read* and *write* messages, but the *read* subtype of that contract would allow only *read* messages, so channel endpoints of the *read* subtype cannot utter *write* messages and the corresponding service provider need not check for *write* messages at runtime.

A channel that entails permissions can be passed to another process. The caller should have reason to trust the recipient since passing a channel in this way entitles the recipient to exercise all the rights inherent in the channel. These rights, however, do not extend to passing new access control checks as the originator of the channel. The recipient process will have (in most cases) a different principal name derived from its invocation, and that name will apply to subsequent access checks on the channel, not the name of the channel originator.

3.5.2 Delegation

The capability-passing mechanism described above has certain drawbacks. In addition to being limited to cases where the service provider does not perform additional permission checks on the channel, it has two shortcomings from a policy management perspective:

- Access policy for a service ends up being split between the service provider (where its ACL is stored) and the client (where its delegation policy is stored). Ascertaining the precise policy in force for the service thus becomes more difficult.
- Capability-passing can cause relevant information to be hidden from both the service provider and the originator of the capability, and hence from their access policy enforcement mechanisms. For example, the recipient of a channel endpoint can pass it in turn to another recipient, and in the absence of further access checks by the service provider, that delegation step (and hence the ultimate user of the capability) will go completely unidentified. Even if further access checks are performed, the path by which the capability passed into the hands of the authorized application that uses it is nevertheless hidden. That path may be significant; it may, for example, include a malicious application that misrepresents the capability's original intended use to the application that ultimately uses it.

In order to support more manageable delegation, Singularity allows a process to delegate authority at runtime. We provide two delegation-types, meant to address different situations. Both involve channel endpoints that have been specially endowed with delegated authority.

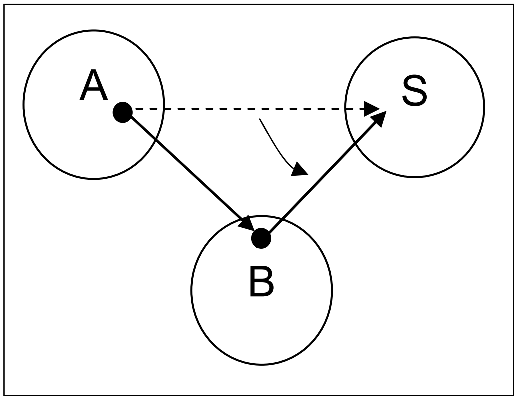

Figure 4: Delegation-by-Reference

- **Delegation-by-reference**. A process can pass a specially-designated endpoint to another process, perhaps referring to some resource, with the intent that the recipient can use the resulting channel to pass subsequent access checks related to that resource.
- **Delegation-by-mediation**. A process may pass a designated endpoint to another process that gives the target, as mediator, the ability to establish new channels with delegated authority. Delegation-by-mediation can be thought of as a special case of delegation-by-reference where the argument endpoint has special rights with respect to making new channels.

The API for delegation is built into the Singularity channel runtime and is further described in Section 4. The delegator explicitly enables delegation with respect to an endpoint, specifying which form of delegation is required. It passes the endpoint through a channel to the delegate process. The resultant endpoint does not speak for the delegate directly. Instead, it speaks for a new compound principal that combines the delegate and the delegator. As with process invocation, we use the "+" operator to join delegator and delegate, where the delegator is akin to the invoking parent-principal, and the delegate application is akin to the invoked child.

In delegation-by-reference, the delegating process knows exactly what resources it intends to pass to the delegate. An example of this sort of delegation is as follows. Imagine that a process wants to delegate the ability to access a specific file system subtree. As in the previous section, passing a directory endpoint to a peer process might allow the peer to perform certain directory operations (enumeration, for example) without additional authority. However, opening a new directory endpoint requires an additional access check at the file system. The first case above allows the delegate to forward, with the endpoint, additional authority needed to pass such an access check. Note that in the file directory example, the delegated authority is meaningful only for operations relative to that endpoint's subtree. This form of delegation is depicted in Figure 4. Client A passes an imp-endpoint, for a resource in server S, to another client B, who can then use it to communicate with S as principal A+B.

Delegation-by-mediation is necessary to support transfer of authority where the delegator cannot know *a priori* what channels the delegate might need. Such mediation is often required as a system structuring tool. Imagine that a service S exists and that a new service S´ offers enhanced service to clients of S by acting as an intermediary. An example mediator is an encrypting file system that is layered between the client and the regular file system.

An endpoint that is enabled for delegation-by-mediation allows the delegate to create new channels with the delegated authority. As depicted in Figure, client A passes an endpoint to an encryption mediator S´, who uses it to create a channel pair, one endpoint of which is subsequently bound to file server S. When S´ speaks on this new channel, S perceives that A+S´ is speaking. Delegation-by-mediation could in theory be implemented through delegation-by-reference: for every channel needed by S´ in the prior example, S´ could ask A to explicitly enable delegation-by-reference on that endpoint. This strategy does not work, however, if A terminates, and it presents an awkward programming model.

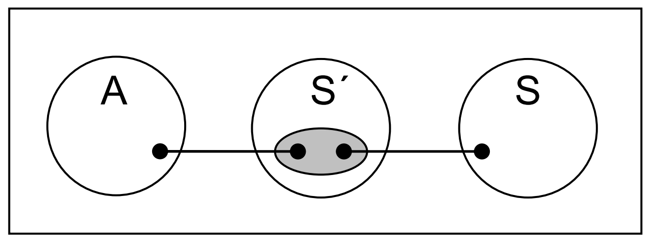

Figure 5: Delegation-by-Mediation

Delegation-by-mediation does not necessarily allow transparent mediation. Because it seems risky to allow the delegate to perform operations without constraint, even if the delegate is named in the resulting principal, we follow the Taos delegation model [33] and require that all delegations be explicitly enabled by the delegator. However, the presence of the delegate name in the resulting principal reduces the risk to the point that we anticipate clients routinely and safely enabling delegation in a large class of scenarios where the service is both trusted to handle delegated access checks correctly, and a potential candidate for transparent mediation. (The file system would be a natural example of such a service.) After all, if the service were simply redesigned to require process invocation, the same form of delegation would occur automatically, by default.

Roles and delegation can be combined to good effect. Much like it is possible to adopt a role before invoking a program it can be useful to alter the rights of the delegator before transferring authority, thus making the principal: *delegator@role + delegate*. This identity is easily expressed in our grammar for principals.

4. IMPLEMENTATION

The Singularity authorization subsystem is implemented by two distinct system components. Figure 5 depicts the overall Singularity software architecture. The microkernel contains a security service that keeps track of the principal names associated with processes and delegations. In-process libraries contain an access control component that contains the pattern matching logic necessary to make access decisions.

4.1 Principals and endpoints

The primary datatype linking the library and kernel notions of principal identity is as follows.

```
struct Principal {
        readonly ulong id;
        static Principal Self();
        string GetName();
        static Principal EndpointPeer(Endpoint ep);
}
```

A principal is represented across the Singularity kernel ABI as a *principal identifier*, an unsigned long integer. Static methods are provided to get the Principal of the currently executing process and to read the Principal associated with the peer partner of a local channel endpoint. Finally, there is a method that corresponds to a Singularity ABI call to resolve a Principal into a textual name..

The API for delegation is built into the Singularity channel runtime. Channel endpoints contain principal identifiers that specify the current owner. When endpoints move across a channel, the owning principal is normally set to the principal identifier for the destination process. If a delegated endpoint crosses a channel, the owning principal is instead set to a principal identifier that corresponds to the compound identity *previousOwner* + *newOwner*. Thus, the client API is simple.

```
ep->EnableDelegation(allowMediation);
```

During endpoint marshalling, the owning principal for *ep* is set to the new principal, and further delegation of *ep* is disabled (although the new owner can subsequently re-enable it). The channel emanating from *ep* now speaks for the new principal. In delegation-by-reference *allowMediation* is *false*, and the delegate has no authority to create further endpoints with the delegated authority. However, if *allowMediation* is *true*, the new owner can create new channel pairs using *ep* for which the owning principal identifiers are set to that of *ep*. The standard Sing#-generated primitive for creating contract-specific channels takes an optional endpoint argument for this purpose.

New principal identifiers are created during process creation and delegated-endpoint marshalling. The kernel Security Service provides two static methods for this purpose.

```
static Principal  NewInvocation(
                      Principal parent,
                      Manifest  manifest, string role);

static Principal  NewDelegation(
                      Principal delegator, Principal delegate);
```

In the former case, new process Principals are stored by the Process Manager for the extent of the associated process lifetime. In the latter case, the delegate Principal must be that of a process and not itself a delegate. Delegation principals are garbage collected when the target process terminates. Our delegation implementation does not currently support role adoption, although this would be a straightforward addition.

4.2 AclCore

Applications that control resources are responsible for using the Singularity access control library to restrict access to these resources as appropriate. So, for example, a file system application is responsible for implementing access control restrictions on the files it supports.

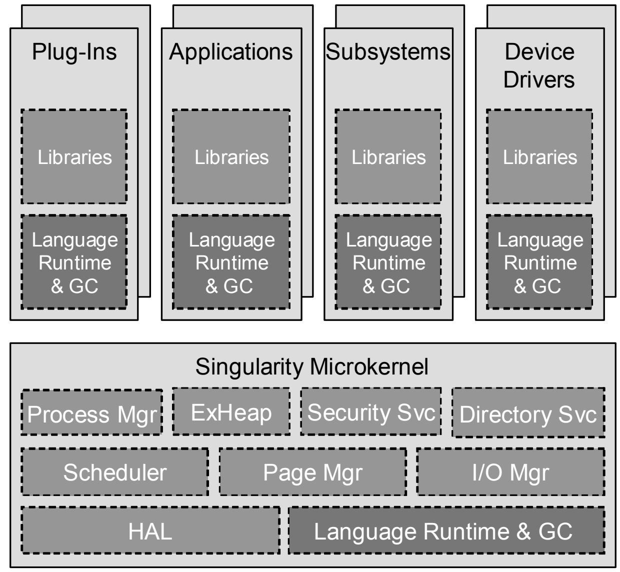

Figure 5: Singularity architecture

An application seeking to make an access control check typically calls EndpointPeer on an in-bound channel at the point where a resource request is extracted from that channel, and the resulting Principal is carried along as part of per-request state until an access check is required. At that time, the application determines the ACL appropriate for the target object and calls into the access control library.

```
bool CheckAccess(string acl, string mode, Principal p);
```

Objects of the AclCore class implement the pattern matching described earlier. The bulk of this work is performed within the context of client processes; the only external dependencies are the ABI call needed to map principals into textual names and the kernel ABI or directory service calls used for resolving subexpressions. The CheckAccess method performs the following steps.

- The input ACL is run through a lexer and a parser that performs a minimal translation between the grammar described in Section 3.4 and the Perl-like syntax expected by the .NET regular expression library. In particular, the translation must interpret wildcard tokens (‘!’), quote characters that have special meanings in regular expression syntax, and resolve subexpressions.
- Subexpressions are resolved recursively, with the lexer/parser applied to each resolution. Infinite recursions are detected here. Subexpression resolution is application-dependent, but the two mechanisms described in Section 3.4.2 are provided by default: an expansion based on SDS file contents and a kernel ABI call to resolve system-maintained subexpressions.
- A completely translated ACL is fed into the C# regular expression library. The library produces a compiled regular expression (e.g. one that has been optimized for fast evaluation).

- Principal.GetName is invoked on the argument principal to produce a textual name.
- If non-null, the access mode argument is appended as a role to the principal's textual name.
- The principal-plus-access-mode string is matched against the compiled regular expression.

Of course, performing all of these steps for each access decision would be prohibitively slow. Thus, the AclCore implementation includes several levels of caching. At the top level, there is a cache of all previously seen ACL for which CheckAccess returned true. For each such ACL, there is a record of which argument principals and access mode have been successfully evaluated. The compiled regular expression for this ACL is cached as well, so principal-mode combinations that have not been tested can easily be evaluated.

If no successful evaluations of a given ACL have been encountered previously, then full evaluation as described above must be performed. However, to optimize the steps above, we also cache resolved subexpressions – the results of expanding subexpressions from component parts. When evaluating an expression that refers (recursively) to other expressions, the full resolved value is cached at every level. Thus, it is usually not necessary to resolve an expression that has been recently visited.

If a successful evaluation is impossible using cached material, the algorithm is re-run from scratch with no caching. Thus, we do not cache negative results: a caller will never be denied access if the ACL allows it. Any access control caching, postitive or negative, involves a tradeoff between the speed of ACL propagation and the cost of cache misses. We assume that successful access control requests far outnumber failures, and optimize the most common case by caching positive results even if doing so doesn't always reflect the most recent ACL state. A consequence of positive-only caching is that the failure case is slow. We are aware that this choice might lead to denial-of-service attacks, but on a single machine we can control the scheduling of processes that produce large numbers of cache misses. Moreover, we can cache frequent misses selectively in order to cut down the rate at which full ACL evaluations occur.

All caches are subject to timeouts and this is the ultimate mechanism for revoking rights granted by previous versions of ACLs and ACL subexpressions. It is worth remembering that several widely-used "push" systems such as UNIX and Windows "cache" group membership for the life of a process, often producing substantial discrepancies between effective and actual access control rights. For example, such systems can sometimes require a user to log off and log on again in order to realize rights. Because we don't cache negative results, we eliminate this specific behavior and expect that timeouts will produce timely enforcement of access control denials.

Credentials that grant privileges to a principal can also become invalid over time because of certificate revocation and timeout. Credentials that become invalid can result in the removal of privileges from a principal's data structures or in the invalidation of a principal name (causing Principal.GetName to raise an exception). Mechanisms for detecting and propagating certificate revocation are beyond the scope of this work.

4.3 Code annotations

We use code annotation to let code authors and publishers specify security properties such as application names, publisher names, and asserted privileges in manifests. Sing# (like C#) enables programmers to specify *custom attributes* that lend themselves to such annotations. These annotations appear directly in the intermediate language (MSIL) output by the Sing# compiler. The Singularity tool set includes a custom tool for creating and populating manifests from MSIL objects. Manifests are XML objects and therefore are simple to manipulate with standard tools.

4.4 Limitations

There are a few remaining areas where our design is not fully implemented. For instance, channel permission subtyping remains a paper design. Other items are more mundane, and require only straightforward coding. In particular, we have not yet ported an X.509 certification mechanism to vouch for publishers, so publishers can be named but not certified. In addition, we have yet to implement manifest signatures and to integrate the code signature support that .NET provides in other contexts.

5. USAGE EXAMPLES

In this section, we describe some examples of how the Singularity security system has been used and how it could be used in the future.

5.1 History truncation

As explained in Section 3.3, certain applications can appear as the first component of a principal name by truncating the execution history. In order to implement this facility, we define a special privilege *$truncate-history-privilege* that, if granted to an invoked application, instructs the process startup mechanism to discard prior invocation history. For example in Singularity the console driver process *tty* is always parent to the *login* process. However, when it is granted *$truncate-history-privilege, login* appears at the head of any principal chain it spawns. Specifically, the kernel security service takes special note of this privilege at application invocation time and truncates the principal name accordingly. Thus it is easy to identify the primary (leftmost) component of an invocation chain stemming from *login*, or another application with *$truncate-history-privilege.*

As mentioned in Section 3.4.2, the publisher of an application must have authority to grant named privileges. Authority is checked at application invocation time by treating the candidate publisher name as a principal to be matched against the kernel-maintained ACL that determines acceptable grantors. The decision to assert a specific privilege is up to the author or publisher of the application, as those parties best understand. Alternatively, such assertions could in some cases be added by programming tools in an automated fashion.

5.2 Directory service

Like most operating systems, Singularity supports a naming hierarchy. Our hierarchy is implemented by a service called the Singularity Directory Service (SDS), an early version of which is described in the Singularity Technical Overview [19]. A Singularity node can support a hierarchy of SDS providers, with each such provider in a separate process. The Singularity kernel implements the local root provider. (The hierarchy is not currently distributed, although it could be made so.) The namespace contains files and directories, but also allows channel exporters to register entries through which they can accept new channel bindings from client processes. Thus, clients typically identify exporters by name.

Using the AclCore library described in Section 4.2, we inserted access control checks into the code paths of all security-relevant operations in the kernel instance of SDS. One problem in doing so is to determine what ACLs should apply at each point in the naming hierarchy. We solve this problem in a fashion that should be applicable to any SDS instance, not just the kernel instance. Rather than storing the access control information for all nodes in the SDS tree itself, we create a parallel data structure that maps path prefixes to ACLs. A lookup of a path name returns the entry for the maximal matching path prefix, creating a natural inheritance structure. We assume that the number of distinct ACLs in most file systems is small; otherwise, access control policy would become unmanageable. Hence, we expect our parallel data structure to fit easily into memory.

The Singularity kernel SDS instance is not persistent. However, we are currently retrofitting persistent access control into our Fat32 SDS provider. Persistence for the in-memory structure can be accomplished with an ACL update log. We must be careful to maintain consistency between the ACL log and the file system itself. There are three actions that are relevant to consistency. When a file or directory is created, no log action is required because it suffices for the new entry to inherit access control state from its parent. Changing an ACL requires an update to the ACL log, but doesn't change the state of the file system. However, deletion from the file system must be correctly reflected in the ACL log lest an incorrect ACL be assigned if the directory entry is recreated. Our implementation uses persistent "intent to delete" and "deletion complete" log entries to ensure that consistent state is recovered if a crash occurs when a deletion is in progress. The Fat32 file system is rather simplistic in that it doesn't support atomic rename. File systems that offer this feature require additional intent logging to guarantee consistency between file system and access control persistent state.

In our SDS implementation, the entries on the right-hand-side of our prefix table are {*nodePolicy, inheritedPolicy*} pairs, where both elements are ACL strings. The *nodePolicy* applies to pathnames that match exactly while the optional *inheritedPolicy* applies to path children. *nodePolicy* is inherited if no *inheritedPolicy* is specified. Lookups on the access control structure return handles that can be cached by callers to avoid subsequent lookups. A handle is invalidated if a subsequent ACL modification alters the corresponding cached result.

We implement a utility that is similar in function to UNIX `chmod`. It sets either or both ACL strings for an SDS path and uses a distinguished SetACL message in the SDS channel contract for this purpose. The code that handles this message checks that the sender has *setAcl* permissions for the named path. This kind of utility can be used to enforce system policy on ACL structure and hide the full complexity of regular expressions from users. Note that we can ensure that all ACL updates pass through such a utility by setting an appropriate default on *setAcl* permissions.

Because SDS supports the ability to register channel exporters, it naturally supports registration of application-space sub-instances of itself. One of the first difficulties we encountered with our security model revolved around handling requests to non-kernel SDS instances. In particular, if all calls to an SDS instance are relayed through the kernel instance and perhaps through intermediary instances, then the identity of the calling channel is lost. Instead, we follow the lead of systems like the Domain Name Service (in non-recursive mode), Echo [8], and GNS [21] and re-direct clients to the correct SDS instances, so clients talk directly to the service that holds the target objects.

Because the binding of names to exporters has serious security consequences in this architecture, we control the ability of applications to register names in the namespace. We write default ACL rules that require that applications be granted *$rg-privilege* in order to register service names in SDS.

5.3 Application private data

The fact that user identity and application identity are conflated is problematical for many systems. It is not easy for an application to manage data that is private to the application and not specific to any given user. In UNIX-like systems, the application can be assigned a UID and operate under that authority, but this is rare, probably because it is subsequently difficult or risky (or both) to allow such applications to adopt a user identity.

In Singularity, user and application identity cannot be confused. Operations to be accessible only to an application `app` should be protected with an ACL such as:

$$((!\ |\ !@!)+)^*\ app$$

In other words, access is granted to `app` as invoked by any other application or role of an application. As suggested in Section 3.3.1 granting such specific access can be useful for applications that need to manage protected state such as installers or system administration utilities. A computer game that needs to manage a high-score list presents a similar scenario, as does a service that wishes to restrict usage to calls made through a specific front-end utility. Moreover, restricting access on a per-application basis allows the same principal to match both user-specific and application-specific controls when appropriate, so there is no need to switch principals in mid-execution.

We speculate that current practice for protecting executable binaries and system files suffers from the lack of application identity. Since it is not suitable in most cases to allow users full access to such files and application-only principals are awkward for the reasons suggested above, most systems use some sort of *administrator* authority for this purpose and often this authority is overused. Overuse of administrator authority, in turn, makes it difficult to enforce any notion of least privilege when managing application state and resources across multiple applications.

Thus, Singularity compound principals offer the possibility of self-management, where installation and manipulation of protected application state is performed by the application itself or through utility software written by the software publisher explicitly for that purpose. Update access to all application-relevant files can be restricted to this software. Such restrictions would not prevent an attacker from misusing management software, but they would confine the attacker to actions possible through the management software. At the very least, self-management of application resources constitutes a step back from a world in which a single principal has unlimited authority to modify executables and other security-relevant state.

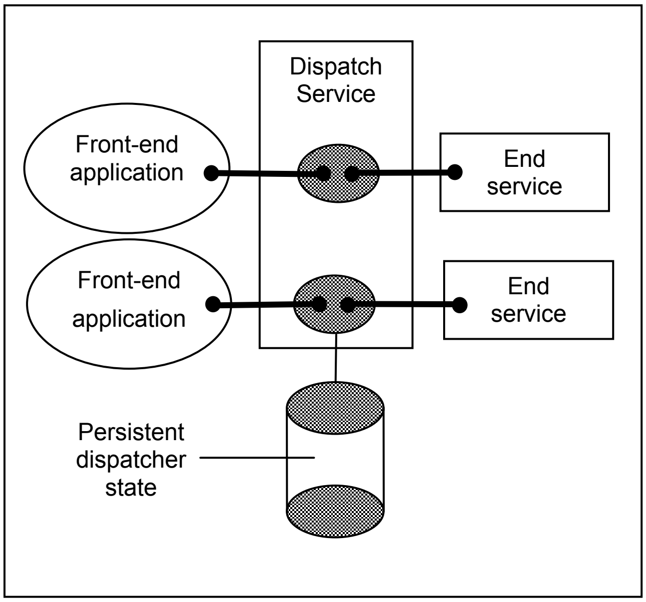

Figure 6: A dispatch service

5.4 Web service

Applications that run in the context of web servers can present difficult authorization tradeoffs. If, as in Windows IIS, a web server can authenticate an inbound connection, the web server can then choose to impersonate an authenticated user. However, this impersonation gives the ensuing computation the full authority of the authenticated user and doing so might not be desirable. Alternatively, the web server can act as a separate reference monitor and perform operations under its own authority or perhaps that of a child application. Doing so risks incurring the cost and complexity of re-implementing authorization infrastructure. For example, a web application might maintain files for different users but store them under its own identity. If so, the application would need to implement its own access control mechanism to differentiate rights of different users.

In Singularity, a web server can authenticate users, in other words it can spawn a principal identity such as

webserver@dan + webapp

where *webapp* is a script or other application meant to run in the specific authorization context of *webserver@dan*. As in this example, any application can authenticate its users and adopt a user-specific role. To reflect this, resource ACLs must differentiate the rights of, say, *webserver@dan* versus those of *login@dan.* It might be that the form

*webserver@dan (+!)**

appears on ACLs for resources controlled by *webserve*r. Here, *webserver* is acting as a reference monitor, but because both *webserver* and *dan* appear explicitly in the ACL, and there is no need for *webserver* to maintain a separate authorization mechanism. *webserver@dan* might also appear on ACLs of files owned by *dan.* In this way, the web server and its applications can be granted limited rights, rather than all of *dan's* authority.

5.5 Layered services

It is often convenient to deploy long-running, intermediary processes that manage state across multiple users. Many three-tier web applications that use a back-end database or file store fit the model. The encrypting file system example of Section 3.5.2 does as well. We now present another example that can arise in the deployment of service management software.

Figure 6 depicts *dispatcher*, a hypothetical service that manages service instances and mediates requests for those services. `inetd` is an example of such as service in the UNIX world, although it does not permit reconnection to existing service instances. Suppose that we employ *dispatcher* to field requests, start up end-services on demand, and re-issue the original requests on behalf of clients. *dispatcher* might also ensure that requests are re-issued in the event of end-service failure. (For the sake of argument, we assume that it would be impractical to spawn a new *dispatcher* instance per request.)

Singularity delegation provides some options for handling authorization in this scenario. As in the private reference monitor example of the previous section, *dispatcher* can forego delegation and operate as its own reference monitor and issue requests under the principal:

dispatcher@user

However, end-services might prefer to see direct evidence of the user's participation as in:

login@user + front-end + dispatcher

Here an ACL maintained at the end-service can distinguish the *front-end* application, the *user*, and *dispatcher*. For example, such an ACL might allow equivalent access rights to different front-end applications with different intermediary managers, or the ACL might deny access to intermediaries altogether. Note that *dispatcher* can use its default process principal to manage persistent state relevant to its own operation.

6. PERFORMANCE

In this section we present performance measurements for our authorization implementation. All experiments were run on an AMD Athlon 64 3000+ (2.0 GHz) CPU with an NVIDIA nForce4 Ultra chipset, and 1GB RAM.

In our test, we matched the principal associated with the test tool with a series of ACLs. For each ACL, we obtained average timings for four different ACL cache configurations. These configurations were: full caching; forced re-evaluation of cached regular expressions; forced recompilation of regular expressions (with subexpression cache); and no caching. The timings are averaged over three runs of 1000 access checks. Results are in thousands of machine cycles; divide by 2 for real time in microseconds.

Figure 7 details the subexpressions referenced by the ACLs in our timing test. *$test-privilege* is a privilege associated with the test tool itself. *$auth-privilege* is granted to all applications trusted to authenticate users. *$rg-privilege*, as above, is granted to applications that can register channel listeners in the SDS. *$dsanyr* and *$dsanyrw* give read and read-write access respectively to any principal. Thus, *{$dsanyrw}|{$dsregister}* gives read-write access to all, but register access only to holders of *$rg-privilege.* The last four tests approximate the conventional file access pattern where read access is granted to all, but write access is granted only to one user or a group of users (in this case possibly running applications published by *microsoft.com*). *$login* is shorthand for *$auth-privilege. $grp5, $grp10,* and *$grp20* are disjunctions of names (e.g., user groups). In some of the example ACLs, the leading *{$dsanyr}* term is elided.

The name of the test tool is SecBVT and it is run by the shell, so the principal associated with the test tool is:

login@ted + shell + SecBVT

For this test, *singularity.microsoft.com* has authority for all privileges and is the publisher of all applications. Note that domain name qualifications are omitted from the principal name above for brevity, but are present in the execution environment.

Table 1 shows access control check timings for various ACLs. Authority to exercise *write* access is checked for each. Timings are expressed in processor cycles. The "Size" column shows the size in characters of the regular expression just prior to compilation by the regular expression package. The number of subexpression visited to evaluate each ACL is also indicated.

The first thing to note in Table 1 is that ACL checks with cached results are very fast, usually in the sub-microsecond range, and their costs vary with the textual length of the ACL. This result is to be expected since there are only two hash table lookups involved. Note that we use a general purpose hash table implementation, so these fast-path numbers are probably larger than they need to be.

The cost of evaluating a cached regular expression ("Eval Regex" column) is in the tens to hundreds of microseconds. This time doesn't vary strictly according to the size of the regular expression. As one might expect, it is a function of both the size and complexity of the regular expression, and the candidate text being matched. Not surprisingly, the time required to compile a regular expression varies with the size of the expression, but the complexity of the expression is clearly also a factor. Both of these costs seem hard to predict without a full understanding of the regular expression compiler.

The difference between no caching and regular expression recompilation is highly dependent on the number of subexpressions to be parsed. This difference is the cost of the translation between

$app	*!|{$user}*
$login	*{$auth-privilege}*
$user	*{$auth-privilege}@!*
$any	*{$app}(+!)**
$anyuser	*{$user}(+!)**
$anyuserall	*{$anyuser}@!*
$dsregister	*(({$any}+)*{$rg-privilege})@register*
$dsanyr	*{$any}@(read)*
$dsanyrw	*{$any}@(read|write|notify)*
$grpN	*<a disjunction of N names>*

Figure 7: Subexpressions used in benchmark

our ACL grammar and regular expression syntax. We note that our very simple lexer performs too many allocations for the task at hand, and that there is considerable room from improvement. Even so, this relatively rare event (evaluating an entirely new ACL) still executes in the sub-millisecond range. We expect that ACLs will be widely shared across resources, and that therefore we expect relatively few distinct ACLs and high ACL-cache hit ratios.

7. RELATED WORK

The issues of identifying principals, and of authorizing or rejecting access requests purportedly made on their behalf, have been explored continually since the very first uses of shared computer system; the bibliography is extensive, and we cannot come close to representing it all here. Most of our opinions on the comparison of our proposals with the most relevant earlier proposals have been included point-by-point throughout the paper. In this section we add some final thoughts on how the present work compares to the most closely related prior work.

Many authors, including some of the present ones, have proposed authentication schemes that try to support a more complex notion of principal than merely "the logged-in user" [6,16,29,33]. Most commonly, such schemes allow a principal to adopt a "role" or "restricted context" with the intention of reducing or enhancing

Table 1: Access control check timings (in 1000s of machine cycles)

ACL	Exprs	Size	Full Cache	Eval Regex	Compile Regex	No caching
{$anyuserall}	4	152	1.3	35	299	498
{$any}+{$test-privilege}@write	5	205	1.6	210	723	876
{$any}(+!.microsoft.com)@!*	5	201	1.6	144	704	853
{$dsanyrw}	5	174	1.4	194	524	784
{$dsanyrw}\|{$dsregister}	11	563	1.4	119	1022	1738
{$dsanyr}\|{$login}@ted(+!.microsoft.com)@write*	7	266	1.9	320	759	1382
...|{$login}@{$grp5}(+!.microsoft.com)@write*	8	289	2.0	319	1124	1469
...|{$login}@{$grp10}(+!.microsoft.com)@write*	10	322	1.9	311	1212	1568
...|{$login}@{$grp20}(+!.microsoft.com)@write*	12	385	2.0	386	1220	1789

the principal's privileges. Some designs for expressive principals are quite elaborate, including in the resulting compound principals such details as the principal that signed the certificate proving the identity of an executing application. Such designs provide great power, but with a lot of complexity. The complexity takes two forms. First is the nature of the compound principal itself, and of the certification mechanisms used by the system to construct the principal. Second is the issue of how to decide whether to grant access to that principal for a particular operation. We believe that the current design provides an attractive compromise in both respects. First, the components of our compound principals are derived directly from the causal chain of process invocations that preceded the access request. Second, our use of a pattern-matching language for access decisions allows an administrator to represent the intention of the corresponding policy.

Our definition of "role", e.g., a simple modifier on principal names, differs considerably from the definition used in role-based access control systems (RBAC) [14]. In such systems, a role is akin to a job function in an organization. Roles (and only roles) are authorized to perform transactions, and system policy controls which roles a user can adopt. It might be possible to use a variant of our scheme to authorize role adoption in the context of RBAC.

Several recent systems and proposals have included security mechanisms that take account of execution history. These include Java [17] and Microsoft's .NET environment [9]. However, these systems take a quite different approach to ours. They achieve their security by inspecting the dynamic state of the computation at the access decision points (in particular through stack inspection [32]). They are complex, partly because of the goal of modeling fine-grained object-oriented interactions. We focus, in contrast, on coarser-grain processes and channel-based communication. We record the state of computations and we specify ACLs in compact strings. We believe that the resulting simplicity is attractive.

One of the novel aspects of the current design is that is easily expresses distinctions about *how* a user was authenticated. While many systems, including Windows and most current UNIX-based systems, support extensibility in their authentication mechanisms, this extensibility does not get reflected in the resulting principal name in any systematic way. Consequently, in Windows and UNIX, all of the authentication mechanism becomes part of the overall trusted computing base. In contrast, in our design there is a single, simple way to reflect authentication mechanisms in principal names, and the decision about whether to trust a particular authentication mechanism is made by the reference monitor, not as part of the overall trusted computing base.

Other designs that involve compound principals have also resulted in revisions to the design of access control lists, though in somewhat different ways than the present design. Abadi et al. [2] present a theoretical access control calculus for compound principals. The subsequent Taos work focuses on practical mechanisms for distributed authentication, but doesn't fully address authorization with the resultant compound principals [33]. Moreover, neither design offers a scheme for practical authorization where applications, or chains of applications, are the active principal elements.

8. CONCLUSION

In this paper we describe a security model for the Singularity operating system and its implementation. In this model, principals are represented by expressions in a grammar whose elements are applications and roles, and access control is done by pattern matching. As a result, in Singularity, applications are first-class entities. Impersonation is not required; so many access decisions made by intermediaries in traditional systems are visible, in Singularity, to the parties that ultimately decide access to resources. Furthermore, application writers and publishers can contribute to the definition of security policies, both by asserting application privileges and by defining access control expressions.

Our design makes it feasible to express complex access control decisions precisely. Complexity is often the enemy of security. We believe that our underlying access control design will allow us to create many, if not most, ACLs mechanically from higher-level access control policies. The end goal would be a system where the administrator can express global policies, instead of tweaking individual ACLs, thus removing many opportunities for inconsistency, neglect, and other forms of error. We have made some small steps in this direction with our access control inheritance model for the Singularity Directory Service, however this is a clearly a fruitful avenue for future work.

As programming tools evolve, attacks that exploit low-level memory corruption should become less common. Unfortunately, these attacks will probably be replaced by others. In particular, memory safety will not protect applications from being invoked in unintended or malicious ways. Some attacks of this form (e.g., on scripting engines) have already been demonstrated. Our principles and system constitute a step towards thwarting such attacks.

9. ACKNOWLEDGEMENTS

We would like to thank Mark Aiken, Úlfar Erlingsson, Manuel Fähndrich, John DeTreville, and Galen Hunt for their contributions to design discussions. Tim Harris and Fang Yu provided helpful comments on early drafts of this paper. We also thank all of the members of the Singularity Research Team for their help in building, running, and debugging the Singularity prototype.

10. REFERENCES

[1] M. Abadi, A. Birrell, and T. Wobber. Access Control in a World of Software Diversity. In Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS X), Santa Fe, NM, pp. 127—132, June 2005.

[2] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A Calculus for Access Control in Distributed Systems. ACM Transactions on Programming Languages and Systems, 15(4): 706–734, September 1993.

[3] M. Aiken, M. Fähndrich, C. Hawblitzel, G. Hunt, J. Larus. Deconstructing Process Isolation. In Proceedings of the 2006 Workshop on Memory System Performance and Correctness, San Jose, CA, pp. 1—10, October 2006.

[4] J. Anderson. Computer Security Technology Planning Study Volume II. ESD-TR-73-51, Air Force Systems Command, Oct. 1972.

[5] R. Anderson. Security Engineering. John Wiley & Sons, Chapter 4: Access Control, pp. 58–59. 2001 (also http://www.cl.cam.ac.uk/~rja14/book.html).

[6] L. Badger. A Domain and Type Enforcement UNIX Prototype. USENIX Comp. Sys., 9(1): 47–83, Winter 1996.

[7] L. Bauer, S. Garriss, M. Reiter. Distributed Proving in Access-Control Systems. In Proceedings of IEEE Symposium on Security and Privacy, pp. 81–85, May 2005,

[8] A. Birrell, A. Hisgen, C. Jerian, T. Mann, and G. Swart. The Echo Distributed File System. DEC SRC Technical Report 111. October 1993.

[9] D. Box. Essential .NET, Volume I: The Common Language Runtime. Addison-Wesley Professional (2002).

[10] K. Brown. Programming Windows Security. Addison-Wesley Professional (2000).

[11] ECMA International. C# Language Specification. ECMA Standard ECMA-334. June 2006. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf.

[12] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman. A Trusted Open Platform. IEEE Computer, 36(7): 55–62, 2003.

[13] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. Larus, and S. Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In Proceedings of EuroSys 2006, Leuven, Belgium, pp. 177–190, April 2006.

[14] D. Ferraiolo and D. Kuhn, Role-Based Access Control, In Proceedings of the15th National Computer Security Conference, pp. 554–563, 1992.

[15] B. Fried, A. Lowry, and M. Stanley. "BigDog: Hierarchical Authentication, Session Control, and Authorization for the Web". USENIX Second Workshop on Electronic Commerce, Nov. 1996.

[16] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson. The Digital Distributed System Security Architecture. In Proceedings of the National Computer Security Conference, pp. 305–319, 1989

[17] L. Gong , G. Ellison, M. Dageforde. Inside Java 2 Platform Security, Second Edition. Addison-Wesley (May 2003).

[18] G. Hunt, M. Aiken, P. Barham, M. Fähndrich, C. Hawblitzel, O. Hodson, J. Larus, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. Sealing OS Processes to Improve Dependability and Security. To appear, EuroSys'07, Lisboa, Portugal.

[19] G. Hunt, J. Larus, M. Abadi, M. Aiken, P. Barham, M. Fähndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An Overview of the Singularity Project. Microsoft Research Technical Report MSR-TR-2005-135.

[20] M. Kaminsky, G. Savvides, D. Mazières, and F. Kaashoek. Decentralized User Authentication in a Global File System. In Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03), Bolton Landing, NY, pp. 60–73, October 2003.

[21] B. Lampson. Designing a global name service. In Proceedings of the 5th ACM Symposium on Principals of Distributed Computing, pp. 1–10, 1986.

[22] B. Lampson, M. Abadi, M. Burrows and E. Wobber. Authentication in Distributed Systems: Theory and Practice. ACM Trans. Comp. Sys., 10(4):265–310, Nov. 1992

[23] B. Lampson. Protection. ACM Operating Systems Review, 8(1): 18–24, January 1974.

[24] Microsoft Corporation. Assembly Manifest. .NET Framework Development Guide. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconAssemblyManifest.asp.

[25] Microsoft Corporation. Client Impersonation. Win32 and COM Development. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/secauthz/security/client_impersonation.asp.

[26] Microsoft Corporation. Strong Named Assemblies. .NET Framework Development Guide. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconStrong-NamedAssemblies.asp.

[27] E. Shi, A. Perrig and L. Van Doorn. BIND: A Fine-grained Attestation Service for Secure Distributed Systems. In Proceedings of IEEE Symposium on Security and Privacy, pp. 154–168, May 2005.

[28] A. Shieh, D. Williams, E. Sirer, F. Schnieder. Nexus: A New Operating System for Trustworthy Computing. Work in progress session – SOSP 2005, Brighton, UK, October 2005, http://doi.acm.org/10.1145/1095810.1118613.

[29] M. Swift, J. Trostle, J. Brezak, and B. Gossman. Improving the Granularity of Access Control for Windows 2000. ACM Trans. Info. and Sys. Security, 5(4): 398–437, Nov. 2002.

[30] The Open Group. DCE 1.1: Authentication and Security. Catalog number C311, August 1997. http://www.opengroup.org/pubs/catalog/c311.htm.

[31] Trusted Computing Group. Trusted Platform Module (TPM) Specifications. https://www.trustedcomputinggroup.org/specs/TPM.

[32] D. Wallach, A. Appel, and E. Felten. "SAFKASI: A Security Mechanism for Language-based Systems". ACM Trans. Soft. Eng. and Meth., 9(4): 341–378, Oct. 2000.

[33] E. Wobber, M. Abadi, M. Burrows and B. Lampson. Authentication in the Taos Operating System. ACM Trans.Comp. Sys., 12(1): 3–32, Feb. 1994.