Singularity Design Note 9

Security Model

1. Introduction

This document presents the current security model for Singularity. In keeping with the mandate of the Singularity Security Group's charter, this design has the following goals:

- To meet the security needs of a general operating system—but with a particular focus on Singularity's proposed application space (the "in-the-closet" home server), as discussed in [SDN8]. This design places a premium on simple, effective mechanisms for hands-off administration and security.
- To be as simple and as transparent as possible, in keeping with the "Singularity razor" approach to design.
- To facilitate the formulation and expression of clear, understandable security policies by developers, administrators, and users.

In brief, this document describes how *isolation* is accomplished in Singularity using processes, type-safety, and channels. Section 3 discusses how processes are grouped into *applications,* with rights derived from application identity. Section 4 presents the composition of *security principals,* while Sections 5 and 6 discuss how *access rights* to resources are determined in relation to the overall system policy. Section 7 lists some possible directions for future work.

Singularity is not a distributed operating system, but it is intended to support the construction of distributed systems via federation. This document describes an initial design for the current, very early version of Singularity. Much of the design is intentionally simplistic in order to allow for forward progress. The intent is to lay an appropriate groundwork for further research, not to solve all the problems at once. In particular, many of the difficult distributed systems issues will not be tackled in the initial implementation.

2. Overview

Singularity is a small, modular operating system whose basic tenets should support the construction of a secure system.

- Singularity is programmed in a statically-typed language that restricts access to information and state held in objects and classes. Static and dynamic analyses can ensure that accesses to information and state are properly restricted.

- The unit of code isolation is the process. The Singularity language does not allow a process to directly access information or state outside it. In particular, direct inter-process sharing of state is disallowed. Isolation helps us achieve our integrity and secrecy goals.
- Processes execute with a fixed set of code components, which can be statically determined. New code components cannot be added to processes during their execution. System extension is achieved by creating and communicating with new, isolated processes, not by adding code to existing processes.
    - All application code is declared by precise manifests [see SDN12]. All manifests are in turn declared within a system configuration [see SDN13a]. The system configuration determines where, when, and how application code is run.
- Singularity rigorously restricts the interactions between processes.
    - Processes communicate exclusively via channels.
    - Channels have two bidirectional endpoints whose movement is constrained. The passage of data through channels is statically constrained by type declarations and behavioral channel contracts.
    - Likewise, most kernel services are provided to processes via channels between the process and the kernel.

This set of characteristics leads naturally to a security design with the following features:

- Security policies are enforced by regulating the establishment, transport, and use of channels between processes.
- Security policies are based on the well-defined concept of the identity of the code running in a process, as well as other attributes such as the user identity that initiated the process, or the identity of the machine on which it runs.
- Security policies are also based on the well-defined concept of application identity.

The Singularity security design emphasizes system integrity and simplicity of administration and management. The security component of Singularity must operate autonomously, with little or no direct human assistance. We must therefore construct and install applications that are highly-resistant to viruses, worms, and other malware. It should be straightforward to run less-trusted applications in restricted environments.

For each Singularity machine, there is an explicit "system configuration" that acts as the local root of trust for that machine and its connections with the external world. The system configuration defines which applications can run, as well as the trust policy (e.g., for external machines and publishers) and the default access control policy.

The system configuration establishes a naming hierarchy. This hierarchy contains configuration information such as known users, groups, trust relationships with networked entities, trusted publishers, and installed applications. Local configurations can be extended with policy rules downloaded from trusted network machines or domains. Although local or remote *administrators* might manage such policies, users need not be administrators. Singularity supports remote administration through paid or contractual services.

Singularity applications are declared through persistent manifests. Manifests declare not only the code segments that embody each application, but also include the application publisher and any certificates pertaining to the application. When an application is installed on a Singularity machine, its manifest is incorporated into the local naming hierarchy, forming part of the local system configuration. As part of installation, each manifest is decorated with parameters that describe how it should be instantiated in the running system. This includes information for binding the resources required by the application (e.g., service providers, hardware abstractions, and file systems).

In Singularity, application identity is fundamental. Each application is identified by its manifest and named relative to the position of the manifest in a local naming hierarchy. Applications can be access resources as controlled through conventional access control mechanisms. The rights associated with individual Singularity application instances can depend on machine identity, user identity, code identity, and application identity. Singularity facilitates the categorization of applications and attribution of rights according to various forms of code identity, including code provenance. In doing so, Singularity integrates execution history (as also found in the Java and CLR security models) with traditional OS-supported identities and access control primitives.

Applications are invoked via their manifests. An application instance has one or more processes. Each process executes with the authority of exactly one (compound) security principal. This coarse-grain approach is not intended to preclude subsequent research into finer-grain techniques such as data flow analysis; this design is intentionally minimal and will be expanded through future research.

Systems such as NGSCB offer hardware support for application identity and isolation. Singularity does not require these hardware features, but future may use them if present. They may be useful for administering a distributed environment. There will not be a Digital Rights Management subsystem in the initial design. If DRM is to be necessary in the context of Singularity, there may be many ticklish issues of standards compatibility to be worked out. Innovation in this area might therefore be difficult.

This document deals only with native Singularity applications. It does not propose any model for non-native applications such as those to be supported by the Singularity virtual machine monitor.

Singularity will not initially offer any generic mechanism for preventing or mitigating denial-of-service attacks. This is left as a direction of possible future research.

3. Processes and Application Instances

In Singularity, the process is the fundamental unit of execution and isolation. Isolation between processes is provided by the language type system, so that information or state in one process is normally unavailable to others, and is augmented through channel contracts. Singularity depends on the correctness of the native language type-safety abstractions, allowing all code to execute in hardware “ring 0” without remapping virtual addresses. MSIL (or a variant of it) is the standard interchange format for executable code.

Each process can call a small, fixed amount of privileged code, the *privileged runtime*, that is logically part of the operating system. The privileged runtime provides core kernel functionality such as garbage collection, IPC, and scheduler primitives. This runtime component, which may need to execute unsafe code or access raw hardware devices, is accessible only through a narrow

interface. Singularity relies on (the safe subset of) MSIL for reference safety and scoping, and perhaps for protection of privileged object fields and methods. As the security abstractions for MSIL don't always correspond closely to those of our language, getting the privileged runtime right requires considerable care.

When a process is invoked, the kernel loads the process's (immutable) code into memory. Processes communicate with each other and with the OS kernel (except for the privileged runtime) through typed channels with behavioral contracts. A process can create a new channel, with a pair of bidirectional channel endpoints and pass one or both of these endpoints across existing channels to other processes. A mechanism for authentication of channels is discussed in Section 4.5.

Processes may be invoked explicitly by other processes, but we expect it to be more common for the system to invoke them and bind them at system startup, as defined in the system configuration.

The instantaneous execution environment on a Singularity machine can be described as the set of running processes (with the kernel as the initial process). One or more processes can arise from each application invocation. For each process, the OS maintains a reference to the manifest from which the process was created; a unique application instance identifier that binds all the processes stemming from a specific application invocation; and a security principal name that characterizes the authority that the process can exercise. Processes can have only a single, immutable principal name. The intent is to avoid programming errors that result from explicitly managing multiple identities. There is some danger that the performance ramifications or other effects of this decision will outweigh the benefits.

When a process receives a request on a channel, it must usually make an access-control decision based on the identity of the other process on the channel. For example, a PVR application might be allowed to write the raw screen but not the raw disk. Singularity supports three limited forms of delegation of authority: a) a channel endpoint, with its identity, can be passed between processes; b) a channel endpoint acquired from another process can be used to exercise new rights due to a combination of the authority of its original owner and its new owner; and c) a channel exporter can bind an inbound channel to a newly created *linked channel*, over which he can exercise his authority combined with that of his importer. These mechanisms are further described in Section 4. Note that in no case is information lost, allowing a process to directly impersonate another.

3.1. Process Invocation

A freshly started process receives a channel for core OS services such as naming, channel creation, etc. This lets us interpose on naming and channel creation, and therefore indirectly control the behavior of a child process. Possible future work in this area is discussed in Section 7.1.

When an application is invoked, by default its identity automatically derives from its invoker, but its authority does not. The new application instance receives a new compound identity (as described in Section 4) that names its invoker, its user identity (if any), its machine identity, its own static identity, etc., but it may have more rights, fewer rights, or (most commonly) just plain different rights from its invoker. This enables the implementation of traditional protected subsystems that own all the resources they manage. An administrative option in the application

manifest can indicates that the target application should not inherit any part of the identity of the invoker.

The static identity of an application is part of its identity, and can (in whole or in part) determine its rights. Of course, the same application running on two different machines will have two different identities (and two instances on the same machine can also have different identities). The mapping from an application's identity to its rights can change over time, due to administrative actions.

Application invocation returns a channel to the services that it provides. If the target application does not derive its identity from its invoker, multiple invocations may share the same running instance. Otherwise, a new running instance is created. It might be desirable to invoke a new (dedicated) process instance even if identity is not inherited. Per-invoker instances promote isolation by eliminating the need for shared state. Note also that existing per-invoker instances can be reused so as to avoid forking a process per invocation.

It might be practical for an application publisher to package multiple different applications within the same manifest. Thus, application manifests will support multiple invocation points (e.g., verbs) to enable this functionality, as well as invocation parameters.

An application manifest must be installed (into the system configuration) before it can be invoked. The policies imposed on applications at installation time and at execution time are described in Section 6. For example, the start-up environment for each application is precisely specified in the system configuration to minimize the possibility of undefined or unspecified behavior, such as when an application instance might attempt to gain improper access by starting a privileged application in an unexpected environment. Application extensions that require different privileges are typically implemented as separate applications.

4. Principals

Processes receive a principal name at creation, and this name is immutable. Singularity principals are compound entities that take the following form.

Manifest Name:	MN = "/" Arc | MN "/" Arc
Role:	R = "/" Arc | R "/" Arc
Manifest Role:	MR = MN | MR "@" R
Process Name:	PN = MR | PN "+" MR
Delegation:	D = PN | D "%" PN
Principal:	P = PN | D

The system provides three operations that affect principals and delegations:

- Process Invocation
- Forking a Role
- Delegation

These will be discussed in detail below.

A Process Name is an ordered list of applications, each of which can act in specific roles. As discussed below, human users are modeled as application roles. The list ordering is intended to reflect an immutable log of application invocations. Each process is named with a specific Process Name.

Compound principals arise during application invocation. If an invoked program inherits authority from the invoker, that is reflected in the resultant compound. So, for example if /Apps/Microsoft/IE invokes /Apps/Adobe/AcroRd the resultant principal might be /Apps/Microsoft/IE + /Apps/Adobe/AcroRd. Note, however, that as previously described, application invocation does not necessarily involve inheritance of authority. This might be modeled as /Apps/Microsoft/IE@Nothing + /Apps/Adobe/AcroRd, or simply as /Apps/Adobe/AcroRd, each of which has its advantages.

Process invocation is actually a specialized form of delegation. The invoker allows, in most cases, the invokee to inherit a constrained form of its authority. This form of delegation always results in a child process that acts entirely for the invoker.

The more general Delegation principal can appear in cases that involve delegation of authority where the target process might not be specific to the delegator. A process can never have delegated authority as its primary name; however it can exercise delegated authority in a constrained fashion through the use of delegated channels.

4.1. Trust Hierarchy

Each Singularity machine will have a trust hierarchy that is rooted in its system configuration, and that is therefore subject to the policies defined there. Nodes in the trust hierarchy name security principals or groups. (Groups are discussed in Section 5.) Principal entities are represented by arcs in the hierarchy; each such name is valid within the namespace of its parent node in the tree. At the root of the trust hierarchy is the local configuration, which is the authority of the local running system. A Path in the grammar above enables the holder to trace back through local names for all the intermediate trusted entities to the local system root.

Nodes in the trust hierarchy can reference supporting information in the system configuration. For example, a node naming a user role might use a name in the system configuration to reference configuration information for authentication of that user. A group node might reference that group's name in the configuration and from that name its membership information. An application node might reference the corresponding application manifest, as represented in the system configuration.

Intermediate nodes in the trust hierarchy can represent delegation of trust. For example, the system root might trust the publisher Microsoft to specify applications that it publishes. Thus, Microsoft could specify the meaning of the name /Apps/Microsoft/IE, although the rights given to that application could still be determined locally. Or, the system might trust a domain authentication service (e.g. /com/microsoft/corp/northamerica) to authenticate user names within that domain, with the meaning of such authentication defined elsewhere. There might be symbolic links in the hierarchy to give an application its own local namespace, and to enable local renaming of principals and groups in this namespace. Again, the shape of the trust hierarchy is always subject to the policies (e.g. rules and certificates) specified in the system configuration, because the system configuration ties trust declarations to their effects.

Imagine that every application has a subtree of the trust hierarchy that reflects context specific to that application. It would be extremely desirable for applications to make most security decisions based on this sort of application-specific context. This helps enforce the principle of least-privilege since each application's configuration can constrain the elements of the trust hierarchy that it must itself trust. In Singularity, the majority of configuration information is supplied off-line by application publishers. Since application publishers presumably understand their

applications, it makes sense to allow publisher-supplied configurations to effectively express security policy, although such configurations must of course be localized within the context of the importing configuration.

The relationship between the system configuration and the trust hierarchy is not yet understood. The trust hierarchy might best be embedded within the system configuration, but the details have not yet been worked out. For example, it is unclear at this time how configuration data that can change dynamically should be represented in the system configuration (such as group membership).

4.2. Roles

Roles represent a strict diminution of authority for applications. As expressed in the grammar, the authority of an application in a role can be further diminished through a sub-role, and so on. Roles of a single application do not commute. In other words, an application running without a role has access to resources granted to any possible role of that application. However, an application in a specific role can only access resources granted under that role and any possible subroles. Roles exist only in the trust namespace of an application that might adopt them. Symbolic links in the trust namespace allow roles to be shared between applications.

Roles allow applications to pass on some subset of their authority to the applications they invoke. This can be a valuable tool for implementing protected subsystems that must manage resources accessed by mutually distrusting entities. Roles can be adopted only by creating new processes. This will maintain the invariant that process principals are static for the life of a process.

When considering the adoption of roles during process creation, there are two distinct cases. In the first, a process invokes code from the same application. In this case the transition of authority might be.

```
/Apps/App1 -> /Apps/App1@WeakRole
```

However, the situation is different when the target application is not the same application. Suppose App1 wants to start App2, with inheritance, but in a weaker role.

```
/Apps/App1 -> /Apps/App1@WeakRole + /Apps/App2
```

This dichotomy seems intrinsic, and must be reflected in the process creation API.

4.3. User Authentication

Human users are expressed as roles of applications. So, for example, there might exist a variety of system login-shell programs that perform user authentication one such shell that authenticates via user passwords; another for remote authentication via RSA keys; and still another for biometric methods. In each case, the original application, which might run as /Sys/PwdLogin will take a role upon successful user login so as to run as /Sys/PwdLogin@ThisUser. Alternatively, the new process might run as /Sys/PwdLogin@/Users/ThisUser where users is a link to a system-wide set of user-roles.

Given that Singularity is targeted at home appliance applications, it might prove more suitable for certain end-user applications to perform authentication directly. This could be useful to avoid losing application context while switching users. Imagine a PVR application that wants to access files private to Ted. In order to do so, it might run as the principal /Apps/PVR@/Users/Ted.

4.4. Remote Authentication

In some cases, such as for compatibility with legacy (non-Singularity) systems, it will be necessary to authenticate communications channels (as opposed to Singularity channels) and associate them with principals in the local trust hierarchy. So, for example, a physically remote administrator of a local application must be able to establish an authenticated channel with credentials that map to a local identity with administrative privileges for the target application.

When a remote computation is authenticated over a network, the authenticating party is in practice evaluating a policy that takes into account the administration of the remote machine, the trust hierarchy linking local and remote machines, and perhaps the software that is running on the remote machine. The policy required for this authentication is embodied in the local trust hierarchy, that is to say that the local trust hierarchy only links in remote namespaces that are trusted under local policy, and the system configuration specifies the parameters for any required authentication protocols. This is one reason for retaining a log of selected security-related actions in compound principals, since the eventual meaning of these actions cannot always determined at the time and place that they occur.

Suppose a remote shell client seeks to authenticate channels from users holding pre-shared public keys (as in ssh). Such keys might be named in the trust hierarchy by a hash of the public key (e.g. Keys/ab456cefa). The remote shell namespace would then link to the key namespace and, after carrying out the appropriate authentication protocol, produce principals such as /Apps/RShell/Keys/ab456cefa. However, using the key hash as a principal might be inconvenient. Instead, the actual pre-shared public key, stored in the system configuration, might be named so as to attribute the user holding it: /Apps/Sshd@/Users/Fred.

Similar indirection tricks in the trust hierarchy can be used to link large hierarchical naming authorities into the local namespace.

Compound Singularity principals might ultimately contain remote host components to allow seamless remote invocations using native Singularity naming. This will not be considered for the initial implementation.

4.5. Principals and Channels

In Singularity, processes send messages over channels. Singularity provides a mechanism that allows a process to obtain the principal corresponding to the owner of the distant endpoint for a given channel. The resulting principal name can then be used to make access control decisions pertaining to messages received on that channel. However, the fact that a request was made by a principal over a channel does not give the receiver the right to act as the sender in subsequent communications. Authority can only be inherited by application invocation as described in Section 3 or by delegation as described below.

In Singularity, endpoints are fundamentally asymmetric. For every channel, there is a separate export-side and import-side endpoint. The export side is held by a provider of a contract and the import side is held by a consumer of that contract. The contract itself specifies the order of communication on the channel. Authentication in Singularity is largely about the server side of a channel determining the identity of the principal holding the distant (import) endpoint. The import side can authenticate the server, of course, but in general applications trust the system namepace (which controls dynamic binding) or the system installer (which controls installation-

time binding) to create bindings to entities entitled to speak for a specific name in the namespace (e.g. dynamic binding) or as specified by system policy (installation-time binding).

Singularity channel endpoints can move from process to process. In order to determine that a distant endpoint is bound to a particular process, it must first be determined that the endpoint is stationary. In order to avoid race conditions, Singularity imposes static restrictions on the movement of endpoints. An endpoint cannot be moved unless the state machine at that endpoint is in a state where a send is possible. When a channel is newly-created, the state machine for the import-endpoint is guaranteed to be in a state where a send is impossible. Thus, authentication of newly-created channels will never suffer from the aforementioned race. Furthermore, it is possible to check channel protocols to determine that certain states, such as when a requestor is waiting for a response, are also race-free. Thus, it is possible to correctly determine the principal associated with a peer endpoint in those states.

As of this writing, Singularity allows for two different types of authentication on channels. Newly-formed channels that appear as arguments on existing channels can be authenticated before first use (e.g. during binding). Often, the principal backing such a channel will be checked using an access control list and, if it is acceptable, the channel can then be used as a capability, as described in the next section. However, it may also be the case that an existing channel requires an access control check for certain types of operation. Such checks will be possible subject to the aforementioned constraints on the channel contract.

Credentials that grant authority to a principal can become invalid over time due to certificate revocation and timeouts. Rather than exposing this fact through the GetPrincipal interface, invalidation of credentials will be handled in background by the security subsystem. The detailed design for doing this is TBD.

It seems unlikely, at this point, that intra-machine channels as defined in Singularity will extend unchanged to inter-machine operation. However, there are ample examples in the literature of systems that provide authenticated and encrypted channels over which inter-machine messages can be transmitted. If Singularity adopts one of these schemes, the security architecture offered in this document might then allow a receiver to model the authenticated name of a remote message originator just as in the local case.

4.6. Capabilities

Channels in Singularity can be viewed as a simple form of capabilities. 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 of course at the discretion of the service provider. Hence, such access controls are ultimately discretionary.

Capabilities and access control lists have long been discussed as alternative methods of implementing an access control matrix. In some sense, the two schemes are duals of each other. Both have their benefits and drawbacks. Since Singularity is a research OS, it seems acceptable to offer both. In practice, this means that a process can give a channel endpoint to a second process and infer that messages received over the channel are from that process or from a third process to whom the endpoint was intentionally given. This train of logic might be used in lieu of an explicit access control check on messages received over a channel. Usually, a process will also perform some discretionary access control check prior to granting a capability. 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 standard paradigm in Singularity for obtaining a capability is 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 the desired permissions, and if allowed by access control policy, grants the permissions to the caller. SDN26 describes a proposed method for describing permissions in Singularity channel contracts. This proposal makes it possible to create endpoints that represent subsets of permissions, and it does so in a way that does not require the programmer to issue multiple contracts for this purpose.

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. The authority invoked when the recipient makes requests on the received channel that require access control checks will be a combination of the originator and delegate name, as described below.

4.7. Delegation

Singularity will allow delegation of authority in two forms. Both involve access to delegated authority through the transfer of channel endpoints that have been in some sense specially enabled with delegated authority.

1) Processes may pass a specially-designated import-endpoint to another process with the intent that the recipient can use the resulting channel with delegated authority. The import-endpoint may reference a specific resource at the exporter with the understanding that the exporter will allow limited delegated access only with respect to that resource.
2) Processes may establish an endpoint to a mediator process that allows the mediator to establish new channels (or pass additional access control checks) with delegated authority.

These two delegation tools are meant to address different situations. In the first, the delegating process knows exactly what resources it intends to pass to the delegate. A good example of this sort of delegation is as follows. Imagine that a process want to delegate the ability to access a specific FS subtree (and also suppose that the FS contract allows the holder of a director to open a child directory.) Passing a directory handle to a peer process would allow the peer to access the directory operation as described as in the section on capabilities. However, opening a new directory handle would most likely require another access check. Delegation case (1) supports this. Note also that it is easy for the directory contract to make sure that the client cannot open directories outside of the scope of the original subtree.

The second delegation mechanism is necessary to allow for mediation as a system structuring tool. Imagine that a service A exists and that a new service A' is offered with the same API as A but with additional functionality. It is in many cases desirable to allow system policy to dictate that A' will mediate on A, in other words requests to A will be handled transparently by A' and then passed on to A. An example of this is an encrypting file system that is layered between the client and the regular file system.

In both cases, endpoints resulting from the delegation will not speak for the delegate directly. Instead, they will speak for a compound principal that combines the delegate and the delegator. This compound principal will take the form of the Delegation construction specified in the principal grammar. We currently expect to use only one operator to specify delegation, although it is possible to have two different operators to reflect the two different forms of delegation.

The APIs to enable these two forms of delegation are still under discussion. However, most likely the first form of delegation will be realized by the delegator "blessing" an import-endpoint directly before passing it to the delegate. When the delegate uses the endpoint, any access control checks enforced on the delegate channel will see the principal delegator % delegate. Note that any new channels created through the delegate channel will also bear this principal.

The second form of delegation is more problematical. If the client is forced to take a special action to enable delegation, and this is not the default, the benefit of using mediation as a system structuring tool will be lost. On the other hand, it seems risky to allow the delegate to perform operations without constraint, even if the delegate is named in the resulting principal.

It will never be allowed for a process to delegate to a delegation. This greatly simplifies the principal grammar, and hopefully eliminates some potential for misuse of delegation.

5. Access Control

There are two fundamental mechanisms for access control in Singularity. The first derives from the fact that application instances acquire all access to resources from configuration information required in their manifests. The Singularity installer controls exactly which applications obtain which resources and the system doesn't provide tools to find others. There is no need to allow the generic access to services and resources that can be obtained in other systems, for example, by enumerating the system namespace. Because of this, Singularity can make static guarantees on the accessibility of a wide class of resources.

However, Singularity must also provide for dynamic access control of shared resources. The set of resources that might be controlled by access control policy in Singularity is diverse. Here is a non-comprehensive sample of some possible resources that might be controlled or encoded through traditional access control mechanisms:

- user files in one or more file stores;
- access to hardware devices by driver software;
- access to device drivers from application code;
- special rights for privileged programs (for example authentication modules);
- ability to register services in the name server space; and
- permission to install programs.

Singularity will provide conventional *access control lists* and *access control groups* as a primary method for controlling runtime access to protected resources.

5.1. Access Control Expressions

The Singularity system configuration subsystem will offer strong mechanisms to help ensure the integrity of system and application code, static data, and parameters. This means that many of

many system and application-relevant objects might be protected without resorting to traditional access control. There will certainly be resources that fall outside of the scope of the system configuration, especially those that require confidentiality. The hope is that the number of these resources will be relatively few.

Access control in Singularity seeks to control access based on three specific sorts of rights:

- *rights propagating from the head of a principal chain*. These rights will typically be associated with a specific user or application that allows inheritance of rights through multiple invocations.
- *rights to access a specific type of protected resource*. These rights will typically involve the most recent invocation in a principal. For example, only the Word application might be allowed access to .doc files.
- *rights of intermediate principal components.* These rights can be used to specify what intermediate applications are allowable on an invocation chain.

Access control expressions in Singularity are patterns. A principal requesting access to a protected object must match such a pattern to gain access to the object. It is likely that there will be several standard paradigms for constructing access control patterns corresponding to common usage scenarios.

An ACE is a string constructed from arcs in the naming tree and operators, as follows:

Atom = Arc | "/" | "@" | "+" | '%"

Item = Atom | "." | "(" ACE ")" | Item "*" | "{" GroupName "}"

GroupName = "/" Arc | GroupName "/" Arc

Seq = Item | Seq Item

ACE = Seq | ACE "|" Seq

The matching rules are similar to those for conventional regular expressions:

- any Atom matches itself;
- "." matches any single Arc (explicitly excluding "/", "@", "+", and "%");
- "(ACE)" matches ACE;
- "Item *" matches zero or more sequential occurrences of Item (greedily);
- "{ GroupName }" matches whatever is matched by the ACE that is the contents of the node GroupName in the naming tree;
- "Seq Item" matches Seq followed immediately by Item;
- "ACE | Seq" matches either ACE or Seq.

A principal "P" matches an ACE "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 sub-string of it.

Singularity supports groups in order represent common shared components of ACEs. Groups are named in the trust hierarchy and represented in the system configuration.

Singularity will include a standard runtime component that allows for the maintenance and encoding of ACEs, and provides the ability to check whether a principal is a granted a privilege by an ACE. Stable storage for encoded ACEs will be handled by the code responsible for the resource, for example the file system service might store ACEs in the file system itself. (Alternatively, the file system might define all ACEs as groups and store only group names along with file data.) This facility, taken together with the ability of any Singularity process to reliably determine the sender of a message, provides sufficient mechanism to perform discretionary access control on requests that flow from other processes, or even within the same process.

6. Policy Management

One serious problem with existing systems is that security policy is maintained in a hundred different places. It's very hard to enumerate all the elements of a machine's security policy, or to ensure its consistency. This can be different in Singularity, since it offers a single place to stand, the system configuration, where all security policy is rooted. It will be critical that the system configuration presents a coherent model for representing security policy.

It would be a great improvement if most access control policies could be expressed through a small number of well-defined rules, with explicit exceptions as appropriate. For the file system, for example, it might be more tractable to define a set of hierarchical policies and then allow overrides, rather than specifying an ACE for every file. The following sections describe some of the policy statements that need to be expressed in any Singularity machine.

It is at this time unclear how best to implement the enforcement of policy in all cases. There will certainly be a discretionary access control system governing most runtime resources. The system configuration will certainly embody much of the installation-time policy mentioned below, and perhaps other mechanisms will also exist. The correct blend of mechanisms for the initial version of Singularity is not yet well understood.

6.1. Credentials and Groups

The system configuration must contain sufficient information to support the authentication of machines, users, and applications. In general, authentication requires the validation of credentials offered by the authenticating party in the context of system policy. Successful authentication in Singularity will result in a name for the authenticated entity that is unique in the context of the local trust hierarchy.

The system policy needed for authentication of users and machines is well understood in the distributed systems literature. Singularity will adopt some collection of existing techniques according to the needs of its emerging prototype applications.

Application credentials are carried in the application manifest (see Section 6.4). They describe the code that is permissible for the OS to load under a given application identity. These credentials also include cryptographic keys and signatures that certify the code hashes in the manifest and vouch for the provenance of the application.

An application identity is a name in the local trust hierarchy. The leaf arcs in this name are typically defined by the application publisher. The universe of publisher names is specific to the local trust hierarchy, but will probably be standardized to correspond with some more global notion of publisher certification.

6.2. Application Installation

The installation of an application is in general a somewhat sensitive task, since it allows the application access to resources—in some cases, critical ones. Installation must therefore be authorized.

Installation can only be performed by an application with the specific permission to install applications. Initially, a specific administration utility (the *installation utility*) that is part of the system has this permission, although other applications may eventually be granted it. Installation is controlled by policy represented in the system configuration and involves decisions based upon the meta-data provided in the application manifest (see Section 6.4.)

Singularity will maintain two types of information related to application installation:

- An *administration policy* for establishing a set of privileged administrators (perhaps per application);
- An *installation policy* for evaluating and acting upon requests to install applications.

Administration policy determines what user credentials are necessary to run the installation utility. Note that the administrator might be physically present and operating a prompted user interface, or might be a remote entity communicating over a secured channel.

Installation policy determines which application credentials represent permissible applications for the local machine, and it maps application identities into groups of applications that can be managed in a similar fashion (see the next section).

Installation policy is represented in the form of trusted public keys and certificate-chains, simple predicates on certain information in manifests (application name, version number, publication date, and so on), and groups of public keys and applications identities that represent the fundamental trust relationships in the system. The system configuration subsystem will specify a language for defining policy.

6.3. Application Groups

The key to allowing more manageable control of application permissions is to define a few categories (*standard application groups*) of application to which new applications can be assigned on installation, according to a set policy. Possible standard application groups include:

- *Normal applications.* This would be the default category for an installed, trusted application.
- *Applets.* Untrusted "network clients" would be installed with very limited privileges, equivalent to those granted today's Java applets.
- *System utilities.* Highly trusted applications installed for system administration functions (such as backup—although backup utilities have a specific enough set of privileges that they might get their own category) would get privileges equivalent to those associated with kernel code.
- *Sandboxed applications.* "Guest applications" (say, borrowing cycles and disk space for grid computing applications), or applications performing some function that can be completely separated from the rest of the computer, get permissions equivalent to normal applications—except that they are confined to a VM-like shadow version of the rest of

the computer, in which all the persistent state associated with other applications is off-limits.

When an application is installed, it is assigned, based on the installation policy, to one or more standard application groups. These groups can then be listed in ACEs to define access rights to resources. For example, the group, "normal applications" might be included on the ACEs for most data files, as well as standard system resources (printer, network, and so on). "Applets", on installation, would be given access to their "home" network address, as well as only those resources associated with applet operation (perhaps only UI devices).

Because many of the trust policies in a Singularity configuration will be encoded in the naming hierarchy, we expect that wildcard access control patterns with common prefixes will be used to describe sets of equally trusted programs.

Note that once installed, applications can then create new application groups, own resources, and set access to those resources to include itself or other applications, groups (included those it created), or standard application groups. However, the application cannot change its own membership or non-membership in any of the standard application groups or change any of the trust decision derived from publisher-certificates that are encoded in the naming hierarchy.

6.4. Application Manifests

The Singularity application manifest will contain a wealth of information about an application, some of which reflects on security policy. The definition of the application manifest is the domain of the Singularity Application Abstraction [SDN12]. Application manifests should support at least the following properties:

- Application name;
- identity of and hashes for code components;
- signatures and public keys to certify code hashes;
- shape of process sub-tree (e.g. subprocess/application creation rights);
- versioning information – equivalence classes of component versions;
- intended installation policy relevant to this application.

And, of course, any or all of the above information should be attestable by signatures, and public key data so as to establish a trust relationship between the executable code and a set of software publishers trusted by system policy.

A code component can be associated with one or more applications. However, since code components are only loaded into new processes with fresh and isolated process state, there will be no runtime sharing of state between applications due to sharing of code components

7. Future Work

7.1. Installer

Substantial parts of the Singularity security model will be embodied by how applications are installed on a given machine and the constraints under which applications are instantiated. Precisely how this will be done is TBD, however the following aspects of the problem will be critical.

Many security problems are caused by code that allows attackers to exploit rights unexpected and unintended by the author. This will probably be the case for a long time to come, but Singularity can offer the possibility of tools that make secure code easier to write. Some of the ideas mentioned here represent speculative, but perhaps promising research directions.

Channels in Singularity are typed. Because the Singularity runtime (through the installation manifest) has knowledge of all the runtime types statically known to an application, it is impossible for an application to obtain a channel endpoint for a type that is not included in the runtime scope of the program.

Resources and application parameters can also be scoped statically in the manifest. For example, manifests can name the set of resource to which an application can gain access. Singularity will proceed with the goal of statically declaring as much security policy as possible.

7.2. Namespace

The contracts that make up the Namespace are currently problematical. These contracts are fundamental to the system and to system security. We will soon need to formulate a set of correct contracts for this functionality, and agree as to the desired security semantics.

7.3. Interposition

As discussed in Section 3, every new process receives an "initial channel" from which it can obtain core OS services such as the ability to communicate with other services. This architecture suggests that it might be possible to allow a parent to *interpose* on a newly started child process by totally controlling its interactions with other processes (and the OS). Although this has yet to be proven, such interposition should make it easy to implement sandboxing environments in which untrusted child processes or applications can be run without risk of corrupting the rest of the system. In this context, the child process (or application) is encapsulated by the interposing agent and is dependent on the parent's continued operation.

Interposition services can be designed explicitly to allow sandboxing of portions of applications or encapsulated applications. When a parent process chooses to interpose, it can then proxy requests from the new process to the kernel or other services, possibly with modifications enroute. Arbitrary interposition policies can be implemented by proxying communications in this manner, but certain standard interposition policies might become popular. Examples include "read-only", forbidding changes to the file system or other persistent state beyond the application, or "virtual machine", defining some sandboxed collection of state (including some persistent state) outside of which the application has no access. Since objects (including channel endpoints) can only be transported via channels, controlling a process' ability to obtain a channel endpoint determines to which resources (including other processes) it has access.

7.4. Delegation Constraints

Delegation, especially to a mediator as described above, is a risky proposition. We expect that there are rules, static or otherwise, that might be applied to constrain the possible activities of the delegate. This could be specified in the form of constraints applied in the channel contract that transfers authority, or it might be specified in system or application-specific policy. Because Singularity runs in an environment where security is often enforced by strong typing and language-based mechanisms, we might be able to enforce controls on delegation within the delegate process itself by means of inline reference monitors.

