

Coordinating Desired Accessibility versus Desired Restrictions in Distributed Object Systems

Arnon Rosenthal
The MITRE Corporation
arnie@mitre.org 781-271-7577

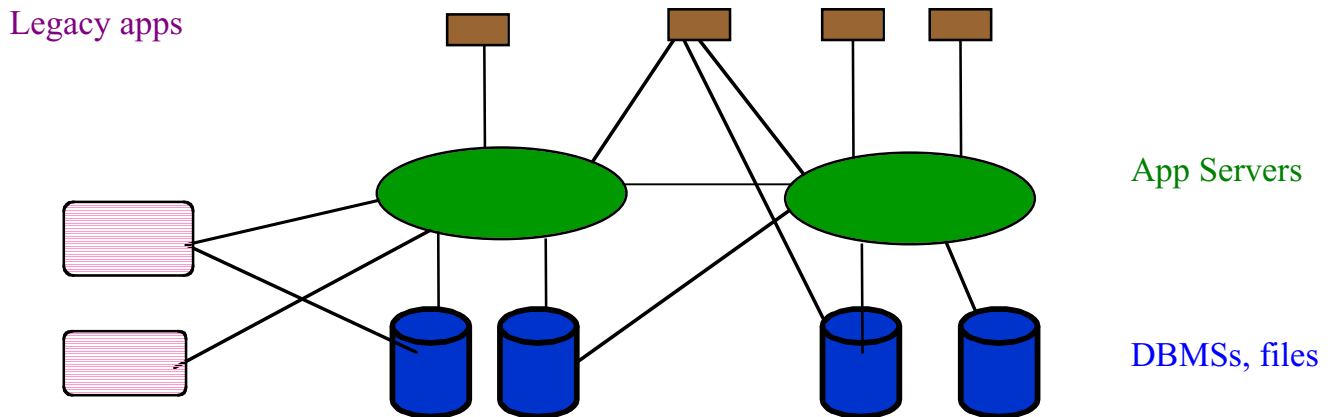
Abstract

This work aims to provide administrators with services for managing permissions in a distributed object system, by connecting business-level tasks to access controls on low level functions. Specifically, the techniques connect *abilities* (to complete externally-invoked functions) to the access controls on individual functions, across all servers. Our main results are the problem formalization, plus algorithms to synthesize least privilege permissions for a given set of desired abilities. Desirable extensions and numerous research issues are identified.

Keywords: Access controls, distributed object management, security, business tasks

1 The Problem

This work aims at providing administrators with services for coordinating accessibility versus protection in a distributed object system, shown in Figure 1. Such systems can be seen as collections of diverse types of servers — object middleware, database managers, and specialized application systems. Each server hosts one or more interfaces, consisting of functions that can be invoked. The code that implements the function may invoke other functions, in the same or other servers. We consider only requests made through these server-controlled interfaces — we do not look under the covers to see other information flows among programs running within a server.



**Figure 1: A Typical Distributed Object System:
Many Interfaces, Many Stakeholders**

1.1 Goals

Our aim is to reduce the labor and the skills for security administration in distributed object computing (*d. o. c.*). It is part of a grand challenge for distributed systems security —to make security administration so easy that ordinary organizations will do it well. The specific goal is to produce theoretical underpinning that can guide development of a security administrator's assistant. Our theory (if implemented in tools) would allow administrators to see how access controls on function invocation connect to abilities to involve all the functions needed to complete a work task. Tools could then provide automated analysis and synthesis.

The need for *abilities* to accomplish work is modeled as running functions to completion, unhindered by access controls. The process of balancing task accessibility versus resource protection is particularly difficult, because ability needs span many invocations, and access controls deal with single ones. Distributed object systems introduce further difficulties, with different server characteristics and spans of control.

We first discuss some strategies for automated synthesis, based on the execution model and the principle of least privilege. Our synthesis takes the desired abilities as a constraint. It then seeks to impose the tightest access controls on each function consistent with 1) those abilities and 2) the capabilities of each server. In work not presented here, we provide a theory for analysis, to determine the abilities that stem from an arbitrary set of access permissions.

1.2 The Problem and the Services to Be Provided

Our work provides models that automatically maintain connections between



- *Servers access control policies* — predicates that limit the incoming requests that a server will invoke (i.e., begin to execute). Functions and data that are subject to such limitations are called *protected* by this access control system. (Isolation mechanisms in programming languages and operating systems are outside our scope.)
- *Abilities to complete work*: These describe the ability to complete a function that represents the automated processing needed for some business task. Completion requires that each onward call satisfy the access policy of the servers that execute it.

Administrators will supply ability predicates that answer questions like the following:

- Who can hire an employee into the Engineering department?
- Who can mark financial software as tested ?
- Who can update Accounts Payable with amounts over \$1000?
- Who can discharge which patients, for what reasons (i.e., can run Discharge(Patient_Id, Date, Reason, Bill_Id, DoctorName))?
- Who can issue a particular SQL request to the database information about Boston employees?

Ability and access control predicates will typically be defined over request arguments, request context (user, time issued), and database contents. Access control predicates might also reference the call stack. For example, the database might grant access to EMP and DEPT to certain users only for requests made in the course of executing the function HireEmployee(Name, Dept, Salary) with Salary<\$100,000.

Figure 2 shows a system s security specifications. For various functions within the servers, administrators specify required abilities (giving lower and perhaps upper bounds); they also may specify upper bounds on the permissions that may be granted. Our theory aims to support tools that would specify the desired access controls on invocations — of a form the function s server can enforce, and such that abilities and controls respect the appropriate bounds.

-  Specify desired *abilities* (lower bounds for business; upper bounds for security)
-  Access controls to be enforced on invocation: Can invoke a method or table only if a predicate holds. (*Abilities require onward calls to succeed also*).

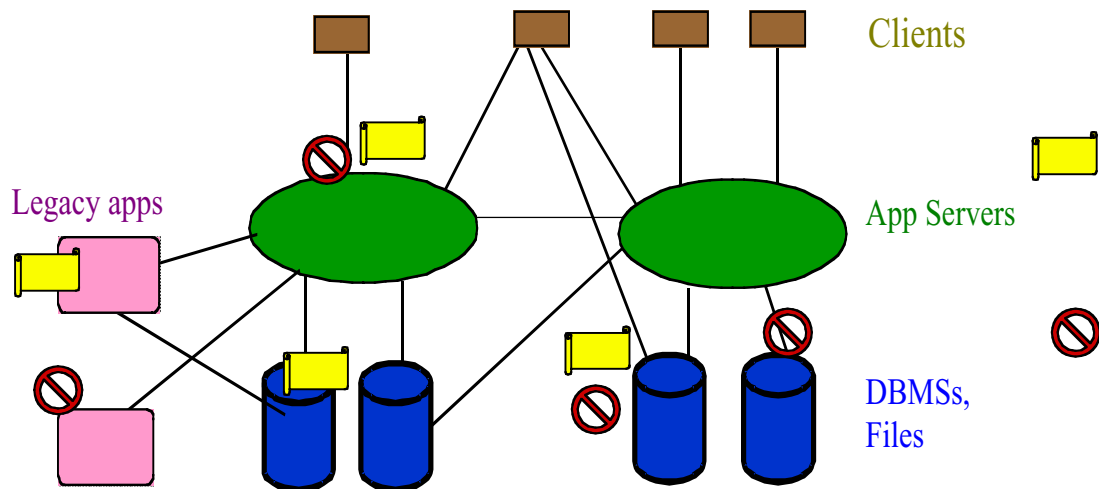


Figure 2: Specify where convenient. Enforce where convenient

A policy specified in terms of one interface may be enforced by access controls on other interfaces. For example, a policy that allows Compensation Analysts to run the Analyze Salaries application on Permanent Employees could potentially be implemented in at least three ways. One could enforce the policy only on the call to AnalyzeSalaries; one could have the database restrict access to views of the underlying data; or one could enforce in both places.

We impose no restrictions on how an organization distributes the authority to make policies. Administration tools should be able to handle several modes: one global administrator; one administrator for each server; or one administrator for all functions in each business area (e.g., Finance), regardless of server boundaries. For controls over the act of delegation, see [Sand99, Rose00, Glad97].

1.3 What Do Systems Do Now?

Today's multi-tier systems do enforce security, but in ways that are far from adequate. None of them (to our knowledge) has tools that can determine what abilities would stem from access restrictions on all the servers involved in a user transaction. Instead, one often sees reliance on just one of the servers. Or else separate administration within each server, and no effective means of coordinating their policies. Below, we describe two common practices today, and explain why they are often not satisfactory.

1.3.1 Middleware Provides All Controls

In many systems, the database is told to accept any request that the middleware has approved. The middleware security may be basic (e.g., predicates just on the user and the function name) or may include a rule engine that provides a powerful language for expressing access controls [Net00]. These engines can be close to the original point of invocation, and can be replicated relatively cheaply.

A prerequisite for this approach is that other resource owners be willing to trust the middleware; it is less likely to suit architectures where middleware spans several enterprises. Even within an enterprise, this approach has serious limitations, which make it rather low assurance. Failure modes include:

- Careless administration: The middleware policy maker may not be very concerned to protect the database's resources.
- Unanticipated behaviors. Administrators may misjudge the accesses a function may make. The cause may be rare circumstances, or Trojan horses maliciously inserted.
- Spoofing: Someone may tamper with the approved request en route to the database. (Many techniques are available to reduce this threat, but some risk remains.)

For all these reasons, it may be preferable to have multiple lines of defense. Instead of give free rein to middleware-approved functions, we want the tightest permissions that (1) allow function executions that implement the approved abilities, and (2) that the DBMS server can enforce.

1.3.2 Database Provides All Controls

Another typical pattern is to rely on database access controls. Testing values of stored data can be easier and more efficient in the DBMS. Also, data owners can enforce their own restrictions locally, if they do not trust the middleware, or they wish to continue using controls that are already specified. But giving the DBMS full responsibility has three categories of disadvantages:

- § One sometimes wants to give extra privileges to trusted functions. Business functions increasingly run in the middleware, but today's DBMSs recognize trusted functions only if they run in the database (i.e., database view or stored procedures). Fortunately, the credentials/PKI features being added to DBMSs for authenticating users also will apply to authenticating functions [Oracle00].
- § Enforcement at the DBMS is not on the request the client generated, but rather on a request descended from the client method. This distance makes it harder to frame an understandable error message.
- § One wants to push processing to stateless middleware objects, rather than to the database. As workload increases, one can easily add processors that hold copies of stateless middleware services. A database is not so cheaply replicated, because one must constantly ship updates among the copies.

1.4 Scope Limitations

It can be impossible to predict a function's behavior. If one cannot bound the calls a function will make; hence, one cannot guarantee that any set of permissions provide the ability to run the function to completion. We focus, though, on tractable cases that skilled administrators routinely handle.

Due to time constraints, we make some serious simplifying assumptions. We believe that most of them can be removed (and that they delineate an important research program).

- We treat the system as a *static* collection of interfaces, functions, and access control predicates. Later researchers will need to add mechanisms for change. Since many distributed object systems are 24 x 7, changes will occur even while functions run (and not every function is enclosed in a transaction).
- We consider that it's not a security administration problem if the application fails (e.g., if the application voluntarily decides to signal failure, or its code crashes). For us, ability is taken just as adequate access privileges.
- *Fault tolerance* (i.e., a function's ability to do its job despite failure of some of its onward calls) will not be considered. However, one could get better bounds if one understood which exceptions could be tolerated.
- We have not yet established *upper* bounds that guarantee that a resource cannot be too widely accessed. We believe they can be obtained by reversing the direction of bounds used in guaranteeing abilities.
- Our permissions aim to provide the least privileges, i.e., to be the tightest possible subject to having the desired abilities. However, definitions and theorems remain to be formalized.
- Enforcement strategies assurance and performance require detailed attention.
- While our theory allows arbitrary calling patterns, our algorithms assume that the function call graph is acyclic, numbered so that f_i can call f_j only if $i < j$.

Despite the limitations, the work still is rather general. We make no assumptions about the programming model used within a function's implementation, nor about argument-passing mechanisms, nor whether function calls must return before the caller resumes operation. The model does not constrain concurrency or isolation; it tacitly places all isolation and coordination logic (e.g., private versus shared storage, transaction management) as part of the function's semantics.

1.5 Contributions of this Work

In Section 1 we motivate the problem, and in section 2 we formalize the Execution model, which describes how requests execute as part of a distributed object computing system. Together, these sections extract the problem from the morass of real world issues, and describe the services desired.

Our aim is to open up the multifaceted problem to work by specialists in each facet. Security policy experts can describe the forms of predicates that are most needed for interesting policies. Experts in code and data flow analysis can find predicates that relate the states of a function and its onward calls. Datalog experts can provide techniques for

handling graphs with cycles [Ull88]. Experts in security services of middleware and DBMSs can implement additional predicate types, e.g., on the call history. And security management experts can build tools that exploit synthesis and analysis to simplify administration.

Next, Section 3 provides algorithms to synthesize minimal permissions that will guarantee the desired abilities. For now, we synthesize based on the principle of least privilege. That is, we impose the tightest access permission predicates from which we can infer that the desired abilities will be present. We conjecture several lemmas about tightness of the synthesis. The final section sketches an agenda for the necessary additional research.

2 Execution Model

2.1 The Basic Models: Machine and Function Call

The entire system (including its input stream) is seen as a deterministic state machine. Its foundation rests with low level (atomic) operations. Synthesis and analysis use a higher level of abstraction (function execution). Access controls are specified as Boolean predicates to be applied at function invocation.

The system *state* is the product of the states of all the memory locations, including ordinary memory, control memory (e.g., call stacks, return codes, scheduling information), security memory (e.g., access predicates, user credentials), and the sequence of future input values. The machine executes sequentially and deterministically, consuming one input (usually *null*) at each step. The state describes the system so thoroughly that it fully determines future execution.

Within this machine, we will next define a higher layer of abstractions for function calls and returns. System software will give programmers additional abstractions (e.g., concurrently-executing threads, transactions, user credentials) that guide execution, but these are not explicit in our model.

The function layer views execution in terms of functions described in the server interfaces. *Callstates* are states of the machine that correspond to a function call; a callstate S that invokes function f will often be denoted S_f . The model shows callstates, access predicate evaluation, function execution, and function return states. We are particularly interested in the (onward) callstates generated during call execution, and in whether the call execution is able to invoke them. Beyond these glimpses, we will abstract away the execution logic.

The semantics are: When the function f is invoked, its access predicate¹ is evaluated instantaneously, using data in the callstate. If the result is *true*, the server creates a *call*

¹ The predicate has no side effects on the system state. An extended model might allow actions to write to the log, or to attach information to the user's request, as in [Net00].

execution and begins executing the function; if *false*, the next (and only) step from this call is the return event with return code `failed`. Useful work is assumed to correspond to completion of a call execution.

The function layer connects to the basic machine as follows: Machine states are partitioned into three categories: ordinary, function callstates, and function return. Calls and returns refer only to server-managed functions, not to internal code. A call is active beginning at its call event and up through its return (if the return exists). At call and return, the argument values are just parts of the system state that the functions may reference. A unique outermost call-execution encloses everything; the call `System()` has one instance, with neither `Invoke` nor `Return` steps.

Each machine execution step (including each invocation and completion) is part of exactly one call-execution. Steps from different active executions can interleave, scheduled by mechanisms outside our model.

In the function execution trace illustrated below, execution begins with a *system* call to function f_1 (call execution C_a), and a later call from *system* to f_2 (call execution C_b), which invokes f_3 (execution C_d). C_b continues executing and interleaves its calls with callee C_d . The code of the various executing functions determines (below the level of our model) what the next state shall be, and what function's execution is associated with that step. Ordinary operations which are not distinguished by the function call model are all denoted *o*.

Our algorithms traverse a function *call graph*, which has a node for each function, and an edge from f to g if f can call g . For lower bounds, it suffices to work with f a superset of the actual call graph. For example, simple tools might identify calls to functions g_1, g_2 in the code that implements f . This paper considers only acyclic graphs. Nodes with no input edges will be called *sources*, and nodes without outputs will be called *sinks*.

Call-Execution to which this step belongs	Operation Executed	Identifier of Call- Execution Created by This Step
[sys]	f ₁	<C _a >
[C _a]	o	
[sys]	o	
[sys]	f ₂	<C _b >
[C _b]	f ₃	<C _d >
[C _d]	o	
[C _a]	return	
[C _b]	o /* model lets execution interleave with C _d */	
[C _d]	o	
[C _d]	return	
[C _b]	return	

The next definition is crucial: Given a callstate S_f we define its *onward callset* (or just *callset*(S_f)) to be {call states associated with the execution of f, when f is invoked in S_f }

We note several important properties of these definitions:

- The callset includes only direct calls from the code of f, not calls from f's callees. One obtains it by understanding the code of f, not the callees.
- S_f contains *all* information relevant to the execution of f.
- The callstates in the onward callset(S_f) are equally well defined, so behavior can be examined transitively.

The administration process confronts function code in just one place -- to describe (or bound) the onward callset.² The means of obtaining such bounds are outside the model, left to experts who understand the reverse engineering, the language used to implement each function, and the mechanisms (private storage, transactions) used to separate different function executions.

2.2 The Model for Deriving Abilities

We focus on abilities, and their derivation from permissions. The key question is Do the access controls permit all onward calls associated with this call's execution? This leads us to the basic rule, which will drive the rest of the paper.

Basic Rule: If you have the permission to invoke f and have the ability to complete everything directly called by f, then you have the ability to complete f. (And vice versa)

² Callset bounds are useful for purposes beyond security. For example, one may wish to load a mobile platform with all the resources that will be accessed by critical services running on it.

The Basic Rule simply restated the definitions and the execution model. From the verbal statement, we get a formal recursion, a Boolean equality for each callstate.

$$(1) \text{ ability}(S) = \text{accessPredicate}(S) \wedge (\wedge \{ \text{ability}(X^j) \mid X^j \text{ in } \text{callset}(S) \})$$

This formula can be interpreted as describing the ability to complete from a particular state S . But a more interesting interpretation is that it defines a predicate on callstates. Next, we add the externally specified ability requirements that drive synthesis. Let $\text{externallyDesiredAbility}(S)$ denote a predicate that tells the minimum ability desired for each callstate.³ Then

$$(2) \text{ ability}(S) \geq \text{externallyDesiredAbility}(S)$$

Theorem: Expressions 1-2 have a unique least fixed point (which we call their *solution*.)

Proof Sketch: A theoretical construction of the fixed point will be presented. Since $\text{externallyDesiredAbility}(S) \leq \text{ability}(S)$ and (from the definition, or from (1)) $\text{ability}(S) \leq \text{accessPredicate}(S)$, we initialize accessPredicate and ability to $\text{externallyDesiredAbility}$, for every state. Then repeatedly apply (1) to derive additional desired abilities, for each state, *ad infinitum*. To apply the expressions, derive additional abilities for each callstate S , and OR those with the previously known ones. At each step, the ability predicates and access predicates are nondecreasing. The union of all iterations results gives a fixed point. To show uniqueness, we intersect all predicates that are fixed points. This is a fixed point, and is the unique least possible one.

QED

Our algorithms will often yield access predicates that are an upper bound on the solution. Suppose we know the access predicate assigned to each function (referred to as *accPreds*). Equation 1 plus *accPreds* again has a least fixed point, called *derivedAbilities(accPreds)*.

A similar treatment appears possible if one wants to protect resources by imposing *upper* bounds on what callstates might be invoked, or might be able to complete. (If partial execution can damage a resource, then one can forbid invocation). That is, one would create upper bound inequalities for *externallyLimitedAbility* and *externallyLimitedPermissions*. Now there appears to be a greatest fixed point, showing the greatest abilities consistent with the protections. An extension would be to allow mixes of both types of inequalities; in such situations, administrators will need help in detecting and resolving inconsistencies.

We now know how to analyze any single state, by traversing the call graph onward from that state. But administrators cannot examine all states individually, nor can they iterate *ad infinitum*. The next section chooses new predicates to describe collections of states, and uses whatever bounds on callsets are available. To avoid being mired in complex algorithms, we handle only the acyclic case.

³ To avoid self-referential definitions, we assume that *externallyDesiredAbility* and *accessPermission* predicates do not reference {access predicates}. A weaker condition, monotonicity, would suffice.

3 Synthesizing Access Permissions to Provide Abilities

We now exploit the Basic Rule to synthesize access permissions based on the *principle of least privilege*. For each computational step, we grant the least access permissions (tightest predicates), for which we can demonstrate that externally-desired call executions will complete. For example, we do not give blanket privileges to all middleware-approved function calls; we would give permission only on the database objects which the functions ought to access, and only for calls onward from the approved middleware functions.

The subsections below synthesize permission predicates of different sorts, to meet the goals above. Section 3.1 presents the general algorithm, and Section 3.2 discusses the simplest special case. Section 3.3 allows more information about functions behavior. Section 3.4 raises the general problem of producing predicates within the capabilities that a server is offering.

3.1 Synthesizing Permissions to Guarantee Abilities

A little theory will be needed before the algorithm can be presented. Consider any function g with an externally desired ability predicate p_g . We treat a predicate as synonymous with a set of states, using whichever notation is locally more convenient. Suppose that there exists a callstate S_g satisfying p_g such that g or one of its callees generates a call to f (i.e., a call to f is a descendent of the call from S_g). Then we say f is *needed for* $g[p_g]$.

Suppose f can call f_i . Define the *call-mapping function*, denoted $cm_{\langle f, f_i \rangle}(S_f)$ and abbreviated $cm(S_f)$, to be the mapping that yields the callstate S_{f_i} produced from S_f .⁴ Let p_f denote the predicate that identifies all the abilities desired for f .

The synthesis algorithm lets desired abilities propagate along an edge, deriving a predicate that accepts every f_i callstate that is reached from a desired callstate for f . This can be expressed as saying that f_i must complete for $\{\text{states in the image } cm_{\langle f, f_i \rangle}(p_f)\} \equiv \{S_{f_i} \mid \exists \text{ a callstate } S_f \text{ that satisfies } p_f \text{ and such that } S_{f_i} \text{ is in } callstates(S_f)\}$. This set (or its indicator predicate) is denoted *exactPropagatedDesires*(f, f_i).

Unfortunately, the call mapping function may not be known in a tractable form that we can apply to the desired set of states. Fortunately, any predicate that is an upper bound on the exact propagated desires will lead to sufficient permissions. By definition, for each $f_i \in callstates(f)$, the predicate (f is parent(f_i) in callstack) holds. (Recall that the callstack is part of the state.) We can thus seek upper bound predicates of the form (f is parent(f_i) in callstack) \wedge (any other upper bound). We call the second conjunct the *specific propagated desires*, denoted *sprop*.

⁴ If f_i is not called by the call executed from S_f , then $cm_{\langle f, f_i \rangle}(S_f)$ is undefined. If there are multiple calls to f_i , then we add superscripts to distinguish them; for readability, we do not show this case.

Pragmatically, we want $sprop(f, f_i)$ to be as tight as possible, to be simple to manipulate in larger expressions, and enforceable by servers. *True* is always a legal choice, if we cannot infer a tighter bound. Later subsections will explore situations where helpful upper bounds (other than *true*) can be inferred for *sprop*. Servers' limited enforcement abilities are considered after all desired predicates have been calculated.

The synthesis algorithm traverses the call graph from sources to sinks, always respecting the calling order. Administrators express application requirements by specifying a lower bound predicate, denoted *externallyDesiredAbility*(f), for ability to complete each function in the call graph. The (total) desired ability to complete f_i is computed by OR-ing its *externallyDesiredAbility* with the needs propagated from all its parents.

Simplified Synthesis Algorithm: Determine sufficient permissions

In externallyDesiredAbility(f): An ability predicate for each function f
Out accessPermissions(f): A permission predicate for each function f, such that the system with these permissions will exhibit all the externally desired abilities.
Out desiredAbility(f): The ability predicate obtained with the above permissions.

Postcondition: For all f, the access predicates are set such that
the ability to complete f \geq externallyDesiredAbility(f)

For each function x /* Initialize each node*/
desiredAbility(x) = externallyDesiredAbility(x)

/* traverse top down, from sources to sinks */

Visit each non-source node f in graph order

/* Create permissions so the accumulated desired calls can be invoked */
accessPermission(f) = desiredAbility(f)

For each f_i in fnCallset(f) /* f_i permissions must allow f to complete */

/* propagate the abilities that the parent requires */

Determine a specific propagated predicate, denoted *sprop*(f, f_i)

desiredAbility(f_i) = [(f is parent(f_i) in callstack) and *sprop*(f, f_i)] OR
desiredAbility(f_i)

For each f

Round up accessPermission to a predicate that the server for f can enforce.

Main Theorem The above algorithm yields sufficient access permissions. That is, for each f , $\text{externallyDesiredAbility}(f) \subseteq \text{derivedAbilities}(\text{accPreds})$

Proof:

It will be sufficient to prove (by induction) two hypotheses about the algorithm's results:

- *Invocation:* For each needed call, we have permission to *invoke* the call.
- *Completion:* For each needed call, we can *complete* f .

Proof of Invocation hypothesis: To see that invocations will succeed, perform induction top down (from calls to source nodes).

For every node, the *DesiredAbility* is initialized to its own externally desired ability, and changed only to make it less restrictive, by OR-ing in additional predicates⁵. Source nodes receive only external calls, so when the *accessPredicate* receives this value, it suffices. To complete the induction, we now prove that the *access predicate* will allow invocations needed for *other* functions to complete.

Consider a non-source node f_i that is needed $v[p_v]$ for some predecessor v . Let f denote f_i 's immediate predecessor on a call path from v . f precedes f_i in the traversal, so (by inductive hypothesis) its *access predicate* allows f to be invoked on any state needed for $v[p_v]$. When the algorithm traversed the edge from f to f_i , the term OR'd into *desiredAbility* (and from there, to *accessPermission*) accepted all callstates of f_i reached from the desired states of f . QED

Proof of Completion hypothesis: To see that execution will succeed in completing, we now do induction in the reverse direction. First consider sink nodes, which have no onward calls. The previous induction proved that they could indeed be invoked, on any needed calls. Since they have no onward calls, they can complete. The base case holds.

Now consider a needed invocation for a non-sink node f . By the invocation hypothesis established above, f can be invoked. By inductive hypothesis, each of its callees is later in the graph, and hence can complete. Thus, the call to f is able to complete. QED

Discussion: The algorithm is based on determining callstates and, based on them, choosing an upper bound *sprop*. The conjectures below state first, that tightening the analysis helps reduce the unnecessary abilities and access permissions, and second, that the algorithm does as well as is possible, based on the analysis.

We say that the code analysis1 is *tighter* than analysis2 if $\text{callsets}(\text{analysis}_1) \subseteq \text{callsets}(\text{analysis}_2)$ and $\text{sprop}(\text{analysis}_1) \subseteq \text{sprop}(\text{analysis}_2)$. Let $\text{accPreds}(\text{analysis}_i)$ and $\text{derivedAbilities}(\text{accPreds}(\text{analysis}_i))$ denote the results of the algorithm using analysis $_i$.

⁵ Since *externallyDesiredAbility* predicates do not reference descriptions of access predicates, establishing an access permission does not cause other predicates suddenly to fail.

Conjectured Theorem: If analysis₁ is tighter than analysis₂, then the system is at least as secure, in that for all f , $\text{accessPredicates}(\text{analysis}_1) \subseteq \text{accessPredicates}(\text{analysis}_2)$ and $\text{derivedAbilities}(\text{accPreds}(\text{analysis}_1)) \subseteq \text{derivedAbilities}(\text{accPreds}(\text{analysis}_2))$.

Conjectured Theorem Assuming that the analysis results are the tightest possible, then no tighter set of access predicates can be found. (That is, for any tighter set of access predicates, there are functions with the indicated callsets whose needs would not be met.)

3.2 A Simple Tractable Case

We say a desired ability predicate p is *preserved*(f, f_i) if whenever f calls f_i , $p(S_f) \Rightarrow p(\text{cm}(S_f))$. That is, the steps between $\text{callstate}(f)$ and the step invoking f_i do not reduce the truth of p . Predicates such as user identity, and time of day are typically preserved. In such cases, we assign $\text{desiredAbility}(f)$ to $\text{sprop}(f_i)$

Observation: If desiredAbility is *preserved*(f, f_i), then sprop has indeed been assigned an upper bound, i.e., $\text{exactPropagatedDesires}(f, f_i) \subseteq \text{sprop} = \text{desiredAbility}(f)$

The next theorem, proved by a simple induction, may be helpful in verifying that more complex desiredAbility expressions are preserved.

Theorem: Suppose each of p_1, \dots, p_k is (f, f) -preserved. Let $B(x_1, \dots, x_k)$ be a (nonnegative) Boolean expression using just \wedge and \vee . Then $B(p_1, \dots, p_k)$ is (f, f) -preserved.

3.3 Synthesis Ideas for More Difficult Cases

This section identifies some properties that are less restrictive than preserving all desired abilities, but still justify usefully restrictive sprop predicates. Proofs are omitted for lack of space. We suspect that we have just scratched the surface of exploitable cases.

Exploiting Conjuncts. First, suppose we can express the desired ability predicate $p_f = p^{\text{easy}} \wedge p^{\text{hard}}$, where the first conjunct is *preserved*(f, f_i). We can then set $\text{sprop} = p^{\text{easy}}$.

Exploiting Disjuncts. Suppose $\text{callstates}(f)$ can be partitioned as $q \vee q'$ where both q and q' are *preserved*(f, f_i). Suppose that p_f is *preserved*(f, f_i) on executions from states satisfying q . Then we can set $\text{sprop} = (q \wedge p_f) \vee q'$.

Motivating Example: Let ApproveCredit denote the function below. The predicate q is $\text{Amount} \leq 1000$; q' is $\text{Amount} > 1000$; both are preserved by ApproveCredit .

Function $\text{ApproveCredit}(\text{Customer}, \text{Amount}, \text{customerDesirability})$
 If $\text{Amount} > 1000$ then $\text{unanalyzable_update_to_customerDesirability}$

CreditDecision(Customer, customerDesirability)
<additional code, including function calls>

Suppose the desiredAbility predicate for ApproveCredit is $p_f = (\text{user is creditAnalyst and customerDesirability} \leq 17)$. This is preserved when $\text{Amount} \leq 1000$. The resulting actual ?? predicate propagated for CreditDecision is

$$((p_f \wedge \text{Amount} \leq 1000) \vee \text{Amount} > 1000) \wedge (\text{ApproveCredit is parent in callstack}).$$

Exploiting Easy, Localized Transformations: Suppose that preservation does not hold, but we understand how $\text{cm}_{\langle f, f_i \rangle}$ alters the portion of memory referenced in p_f . Specifically, we require that the call mapping from f to each child f_i be invertible, and that cm^{-1} be known.

To handle such transformations, we set *sprop* to be:

$$(\text{some member of } \text{cm}^{-1}(S_{f_i}) \text{ satisfies desiredAbility}(f))$$

Proof: Suppose S_f satisfies $\text{desiredAbility}(f)$ and S_{f_i} is in $\text{cm}(S_f)$. Then by definition, $S_f \varepsilon \text{cm}^{-1}(S_{f_i})$. Hence the disjunct is satisfied for calls from f to f_i . QED

Motivating Examples:

- Replace a name by a code, and use it as an argument to a subsequent call. (For example, Massachusetts becomes MA, as a 1:1 function).
- Convert arguments to UPPERCASE and use the converted form as an argument to a subsequent call. (a many:1 function)
- A Name is mapped to one of several social security numbers, based on factors that we cannot predict (a 1:many relation)
- A value is held constant (the trivial case).

We conjecture that the technique can be extended further, e.g., to conditional mappings and to deriving a cell of S_{f_i} from a different cell value in S_f .

3.4 Adapting Synthesis to the Servers' Limitations

We now recall our high level picture that approximates how distributed object systems (including DBMSs, webservers, object servers,) are often organized. Functions are grouped into *interfaces*; each interface resides in some *server*. An *invocation request* (abbreviated *request*) can be sent from any running process inside or outside the protected system.

Sometimes we cannot directly impose the predicates we want. Difficulties may stem from limitations within a server (e.g., a limited language for expressing predicates, or refusal to make the callstack available), limitations on interserver communication (e.g., inability to pass credentials), or efficiency concerns (manifested as a policy that forbids remote calls).

When it is not possible for a server to enforce the desired permissions, then one selects some predicate enforceable by the server. That is, one rounds up. It is easy to prove:

Theorem: If one rounds up the permissions produced by the synthesis algorithm, the derived Abilities do not decrease, and hence still suffice.

4 Summary and Conclusions

Permission administration will become increasingly important, as organizations deploy multi-tier and peer-to-peer distributed systems. Their cross-organizational issues will greatly increase the need for detailed security administration. Metaphorically, the desired control requires gauges that provide information, knobs to turn, and intelligence to choose settings for the knobs. Unfortunately, the gauges and knobs in such systems will soon outstrip administrators' capacity to use them well. Our research aims to give them automated tools that increase this capacity.

In this report, we have formalized several of the important issues, and provided partial solution techniques. The area seems theoretically rich, with room for major improvement in what we have done. Assurance and performance seem major open issues.

Market demand lies in the future, but there will be several years' lead-time in developing theory and then tools. It seems the right time to begin building a research base so the tools can be principled and powerful.

5 References

[ISO99] ISO X3H2, *SQL 99 Standard*, section 4.35.

[Oracle00] Oracle 8i DBMS Reference Manual,
http://technet.oracle.com/docs/products/oracle8i/doc_index.htm

[Glad97] H. Gladney, Access Control for Large Collections, *ACM Trans. Information Systems*, Vol. 15, No. 2, April 1997, pp. 154-194.

[Net00] Netegrity Site Minder <http://www.netegrity.com/>

[Ros00] A. Rosenthal, E. Sciore, View Security as the Basis for Data Warehouse Security, CAiSE Workshop on Design and Management of Data Warehouses, Stockholm, 2000. Also available at <http://www.mitre.org/resources/centers/it/staffpages/arnie/>

[Sand99] R. Sandhu, V. Bhamidipati, Q. Munawer, The ARBAC97 Model for Role-Based Administration of Roles, *ACM Trans. Information and System Security*, Vol. 2, No. 1, Feb. 1999, p 105-135.

[Ull88] J. Ullman, *Principles of Database and Knowledge-Base Systems*, vol 1. Computer Science Press, Rockville Md.