To appear in the Proceedings of Eurosys 2010

Policy-based Access Control for Weakly Consistent Replication

Ted Wobber Thomas L. Rodeheffer Douglas B. Terry
Microsoft Research, Silicon Valley
{wobber,tomr,terry}@microsoft.com

Abstract

Combining access control with weakly consistent replication presents a challenge if the resulting system is to support eventual consistency. If authorization policy can be temporarily inconsistent, any given operation may be permitted at one node and yet denied at another. This is especially troublesome when the operation in question involves a change in policy. Without a careful design, permanently divergent state can result.

We describe and evaluate the design and implementation of an access control system for weakly consistent replication where peers are not uniformly trusted. Our system allows for the specification of fine-grained access control policy over a collection of replicated items. Policies are expressed using a logical assertion framework and access control decisions are logical proofs. Policy can grow to encompass new nodes through fine-grain delegation of authority. Eventual consistency of the replicated data is preserved despite the fact that access control policy can be temporarily inconsistent.

Categories and Subject Descriptors C.2.4 [*Computer-Communication Networks*]: Distributed Systems; D.4.5 [*Operating Systems*]: Security and Protection

General Terms Algorithms, design, security

Keywords Eventual consistency, replication, security logic

1. Introduction

The availability of cheap and portable computing has resulted in a proliferation of computing devices with a profound effect on our personal and professional lives. Although data communications technology has served to connect many such devices at the network level, users continue to be faced with the task of managing their data across multiple devices, and the problem increases with each new device. While some applications succeed in managing data in a centralized or well-synchronized fashion, there remain many which do not. As a result, users routinely deal with data that is replicated across multiple computing devices with (at best) weak guarantees about consistency.

Numerous protocols and applications have been proposed for creating order from distributed disorder. Although techniques for constructing tightly synchronized systems such as state-machine replication [28] are well understood, many deployed systems have adopted loose synchronization semantics as these have fewer operational constraints. For example, tools such as Groove [19] provide file replication between directories located on multiple machines. Microsoft Sync Framework [20] offers a general platform for providing multi-node synchronization of arbitrary datatypes. Directory services such as Grapevine [8] and Active Directory [18] support lazy propagation of updates between distributed servers. Even in the data center, strong consistency can be an expensive proposition in terms of performance and geo-location, and more relaxed strategies are often employed [31]. Many of these systems can be thought of as *peer-to-peer* in the sense that updates to replicated state can propagate through peer nodes; point-to-point links between all communicants are not required. The systems that we consider are weakly consistent, with no guarantees as to the temporal replicated state at each node. However, these systems do support *eventual consistency*: nodes are guaranteed to eventually converge to identical states if updates cease.

In this paper, we consider the problem of providing access control in the context of weakly consistent replication. We assume that it is desirable for nodes to have different rights in terms of what data they can receive, change, and propagate. More succinctly, nodes are not equally trusted. While there are numerous examples of systems that use replication to provide a distributed service with restricted access to clients (including some of the aforementioned systems), we know of little work that discusses how to support differing levels of access privilege within such a service so that some peers have only limited authority to read, author, and propagate

This is the author's version of this work. It is posted here by permission of the ACM for your personal use.
Copyright c 2010 ACM 978-1-60558-577-2/10/04...$10.00

updates. This is an important problem because in real life, collections of cooperating devices are not homogeneous. A user may give more rights to his home machine than a cloud server, or fewer to his web server than the computer where his finances are kept, or choose to deny most modification privileges to his cell phone. In a distributed directory service, a node serving a particular naming domain may be trusted to update names only within that domain. There may be good reason to give a photo-sharing service access to only a subset of a user's replicated photo collection. And, of course, with portable devices becoming such an important part of the digital lifestyle, we must realize that such devices are easily lost and hence open to compromise. Even as these devices join in replicated systems, we need techniques to allow trust in them to be constrained and revoked.

We use the term *replica* to refer to an active distributed system element that holds replicated content. We use the term *policy* to refer to the shared state that defines the set of actions replicas can perform during the replication process. Policy also defines the set of data items to which a replica may make local modifications. As such, it constrains the replicas on which data can originate or be modified, and the paths through which it can flow. Policy must be mutable, and there is no guarantee that each replica will see the same policy at any given time.

Providing policy-based access control where there is only weak consistency presents three primary challenges.

- Policy specification must not depend on implementation detail.
- Policy must evolve seamlessly as devices and data are added or deleted.
- Access control must not prevent eventual consistency.

First, it is always desirable to be able to reason about access control policy without considering the details of a given implementation, and it's even more important to do so in a distributed environment where reasoning about security is complex to begin with. Implementation-independence also permits policy enforcement to be audited after the fact and facilitates caching of evaluation results and offline pre-authorization. Second, policy must be able to evolve: by adding rights for new replicas, by revoking old rights, and by delegating the right to do both.

The nature of our third challenge about preserving eventual consistency can be seen in Figure 1. In a system in which all nodes trust each other, the access control policy for allowing an update U can be enforced independently by nodes A, B, and C, even though there might be transient variations in policy at the three nodes. Because the nodes trust each other to enforce policy, they will never permanently disagree about which data items have been accepted. Once an update is admitted (as in Figure 1a), no further checks are required. Note that there is no guarantee that the most recent policy is used for any one access control decision.

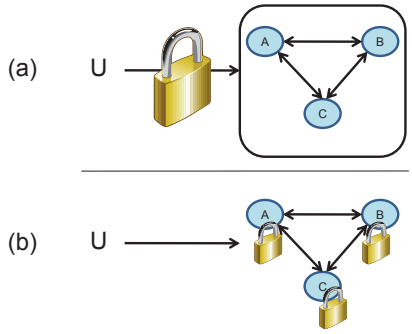


Figure 1. Distributed access control with (a) trustful and (b) distrustful nodes

If, however, A, B, and C are mutually distrustful (as in Figure 1b), each must evaluate policy on every update to make sure that updates are compliant with policy. As we discuss in Section 2.2, the possibility of transient differences in policy risks permanent divergence in state between nodes. This would violate eventual consistency.

To further illustrate the problem, in Figure 2, we depict two replicas that we use to examine what can happen when nodes are both inconsistent and mutually suspicious. The solid lines represent update flows. Replication state is depicted at four different times. At T_0 both replicas initially have policy P and data item U. Suppose at T_1 two concurrent updates are introduced into the system: a policy change P and a data update U , where U is valid under policy P but not P . (A similar problem would arise if U were disallowed by P but allowed by P .) Since our system places no constraints on update ordering, B might attempt to propagate U at T_2. In a framework of mutual trust, U would be subsequently accepted at A because the critical access control check takes place when the update enters the system, at B. No other access checks are needed. However, in a mutually distrustful system, there must be an access check at every hop and U would be rejected at A although it was accepted at B. Finally, A sends the new policy P to B at time T_3. B keeps U since it was accepted under the old policy. Thus, the replica state at A and B now differs, and lacking other action, the divergence will be permanent.

This paper presents a new system that, to our knowledge, is the first to provide an authorization framework for weakly consistent replication without uniform trust. We specify security policy using the SecPAL logical policy assertion language [7]. Thus, policy is separate from data and policy is enforced by requiring logical proofs of authorization. Our logic expressions provide not only a solid foundation for assignment and delegation of authority between existing and new peers, but also a succinct mechanism for propagating policy and dealing with any potential inconsistencies that might lead to divergence. We have implemented our system in the context of a general-purpose replication framework, Cimbiosys [24]. However, we expect the methods we de-

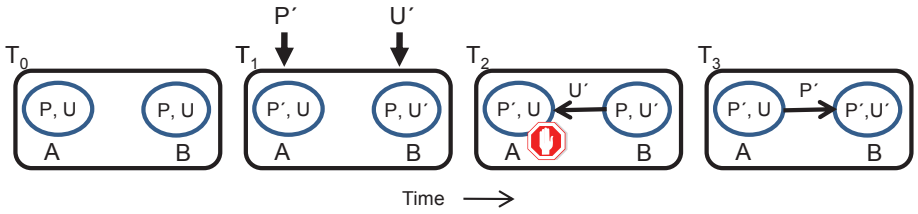


Figure 2. Concurrent policy (P) and data (U) updates lead to permanent inconsistency

scribe here to be applicable to weakly consistent replication protocols in general.

The remainder of the paper is organized as follows. Section 2 lays out our system model and the threat environment it tolerates. Section 3 presents our system design and how it satisfies the first two challenges above. In Section 4 gives a detailed explanation for how we deal with the third challenge of maintaining eventual consistency. Section 5 motivates our design and cites some potential limitations. Section 6 provides implementation specifics as well as a brief performance evaluation. Section 7 describes related work, and Section 8 concludes.

2. System and threat model

We refer to a dataset subject to replication as a *collection*. In practice, a collection might be a file system subtree, a SQL database, a digital calendar, a list of personal contacts, some other type of shared data, or a combination of the above. Collections contain sets of *items*. These sets can appear, in whole or in part, on one or more collection *replicas*. Because of access controls, all data may not appear on all replicas, thus we rely on the presence of partial replicas, for example as implemented in Cimbiosys [24]. In our system, updates operate on whole items and items are independent in that there are no constraints on replication order between items. Updates can originate at any replica. Replicas synchronize periodically, that is, a destination replica requests the enumeration and download of items from a source. We do not constrain the period or distribution of synchronization events and we expect arbitrary communication patterns between replicas. However, we do assume that the communication pattern is rich enough so that all updates will reach all authorized replicas.

We assume that any update to data in a collection takes place either through direct invocation on the replication infrastructure or is observed by a helper application (such as a file system watcher) that can then alert the replication infrastructure. Updates can arrive out of order or be superseded en route by newer updates to the same data. Concurrent updates to a single item, which we call *replication conflicts*, may occur and conflict resolution need not be automatic as long as conflicts are resolved identically at every replica. Communications failures are tolerated as long as network partition is not permanent.

2.1 Principals and rights

The security principals in our access control system are replicas, not users. Several of the applications we are considering (e.g., productivity and portability applications) typically involve a single human user per replica. Often a single user will own all the replicas (e.g., home computers, home servers, and phones), but also desire these entities to have different security privileges. However, a multi-user model is also possible. For example, a collection's policy might grant authority to a replica hosted by a server that hosts replicas for many users, trusting that the server will provide adequate isolation between them. Alternatively, an application might build user-level access control on top of our framework.

Since replicas are principals in our system, we associate a public key with each. Every creation of or modification to an item at a replica results in a local update that is then propagated to other replicas. We call this originating replica the *author* of an update. Each such update bears a public key signature that guarantees data integrity and identifies the author replica. The purpose of security policy is to grant appropriate rights to replicas by granting authority to their public keys. In the case of updates, an update is not valid unless policy states that an update bearing a signature by a given public key is valid. Since updates are signed, a replica can forward an update authored elsewhere even if it lacks authorship privileges itself. We do not implement cryptographic privacy, although replica keys could potentially be used for that purpose.

We insert access control checks into the replication protocol implementation at the point where replicas synchronize with one another. This mechanism can therefore restrict the flow of data between replicas: it guards against insertion of unauthorized updates by malicious replicas, unauthorized reading of data items, and corrupt operation of the replication protocol. So in general, we seek to control the ability of less-trusted replicas to perform certain replication actions, and thus construct a system that supports weak consistency without uniform trust.

Figure 3 depicts a possible deployment scenario that applies to network infrastructure. We consider a naming service, for example, Active Directory [18], that implements a hierarchical name to resource map. In this case, an item might comprise the map elements corresponding to a single name, and a collection might comprise all such items.

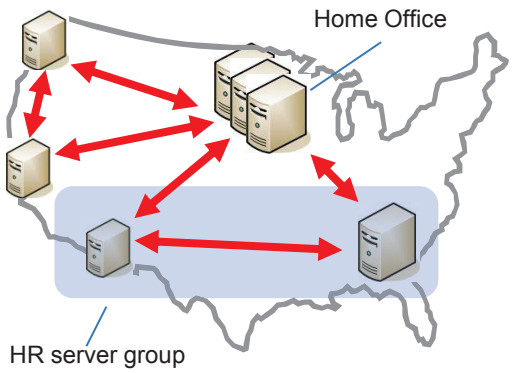


Figure 3. An example distributed name service

Systems of this form often make use of weak consistency to improve performance and availability. However, it is also often the case that certain replicas are naturally associated with specific parts of the naming hierarchy. In our example, it may constitute a prudent defense-in-depth strategy to restrict update authorship within the *HomeOffice* domain to servers at the home office's geographic location. Or, it may prove valuable to restrict data elements in the *HR* domain to those replicas (and thus their clients) that actually require them. Our system is designed to support just these sorts of fine-grained security policies.

Our experience is primarily with state-based replication protocols: those protocols that implement replication by representing update propagation as transitions on shared state. As is common with such protocols, our controls on item read access and protocol operation depend transitively on the correct implementation of the guards at each replica through which protected data can pass or that influence protocol state. We do detect any unauthorized update by a malicious replica when it produces an update contrary to policy.

We do not explicitly deal with denial-of-service attacks.

2.2 Policy replication and consistency

As suggested earlier, one difficulty that arises when implementing access control under weakly consistent replication is that the local view of policy can vary between replicas at any given time. As we describe later, our system distributes policy in such a way that it cannot permanently diverge. However, temporary divergence can occur.

One familiar model of access control posits a *reference monitor* that guards security-relevant operations. The reference monitor checks the requesting principal's rights by inspecting an *access control matrix*. In a strongly consistent system, each replica sees the same access control matrix and thus makes the same decision on every update. However, in a weakly consistent system each replica's view of the matrix can differ at any given time. Thus, access control decisions related to the same update may differ depending on time or location.

In our system, policy is just a representation of such an access control matrix and the guards we add to our protocol implementation are reference monitors that enforce policy. As described in the next section, the design of our policy framework ensures that system policy, which is the aggregation of statements by different principals, is never ambiguous, and this is a prerequisite for distributed consistency. We address the eventual consistency of both policy and data in Section 4.

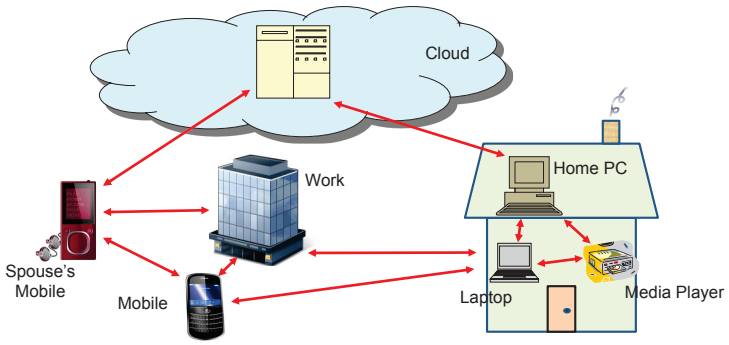
3. System design

In this section we describe the design of our access control system. We first present another scenario to illustrate our design, and we use it throughout the remainder of the section.

Figure 4 depicts a small collection with several replicas in the form of personal computers, business servers, and consumer electronics. In this scenario, we imagine a household network that shares photos and contacts with various other devices, a cloud storage facility, a corporate datacenter, and two cell phones. Elements of the scenario, such as weakly consistent contact sharing between laptops and cell phones, and photo sharing between cloud and home, are already commercially available. The fact that these services are currently implemented by a disjoint set of device features and protocols is perhaps just an historical accident. Certainly if communications services become better integrated in the fashion implied here, assigning different access control rights to different devices will become a priority.

The figure also suggests an informal authorization policy that might govern such a collection. The columns in this chart represent rights that can be assigned to the principals represented in the table rows. We consider two classes of items in this policy: *photos* and *contacts*. Rights can apply to a specific class of items, or to all items. Policy enforcement is carried out during the synchronization operations implied by the inter-replica arrows.

We control three primitive operations during replica synchronization: *write*, *read*, and *sync*. When a replica receives an item during synchronization, we treat it as a *write* operation. If the authoring replica is not entitled to write the item, the operation is denied and the received item is not considered valid. Given that authorship is conveyed through a public key signature on the item, the sending replica need not have write access for the item to be valid. When a replica is considering whether to send an item during synchronization, we treat it as a *read* operation. If the receiving replica is not entitled to read the item, the operation is denied and the item will not be sent. Finally, as we discuss in Section 5, replication protocols often require the distribution of certain protocol-specific metadata both for performance and correctness. In our case, metadata applies to the collection as a whole. Thus, when a replica receives protocol metadata during synchronization, we call it as a *sync* operation. If the sending replica does not have sync rights on *all* the collec-



Replica	*Read*	*Write*	*Sync*	*Control*	*Own*
HomePC	*all*	*all*	*all*	*all*	*all*
Laptop	*all*	*all*	*all*	*contacts*	*contacts*
MediaPlayer	*photos*	—	—	—	—
Cloud	*all*	—	*all*	—	—
Work	*contacts*	*contacts*	—	—	—
Mobile	*contacts*	*contacts*	—	*contacts*	—
SpouseMobile	*contacts*	—	—	—	—

Figure 4. An example replicated collection and policy

tion, the operation is denied and the received metadata is not considered valid.

The *control* and *own* columns in Figure 4 refer to rights that govern changes to policy. The *control* column indicates the right to grant *read* or *write* permissions for a given item class. The *own* column represents the right to add new rights to any of the columns (including the *own* column) for the designated item class, as well specifying that the principal has *read*, *write*, *sync*, and *control* rights over the class.

In the example, we initially create our collection at the *HomePC* and give it rights over all our data. The *Laptop* is given read and write permissions on everything, but permissions to set access rights on only the *contacts* subset of the collection. Only the *HomePC*, the *Laptop*, and the *Cloud* are trusted to act as sources of protocol metadata. Other replicas are delegated specific rights as noted. In the following section, we describe how such a policy can be encoded.

3.1 Labels as protected resources

The resources subject to access control in our system are policy *labels*. Labels represent explicit classes of items (for example *photos* and *contacts* in our scenario) and every item bears an immutable label. Our policy language is composed of statements that grant replicas rights over labels and hence over the items bound to those labels. Since we expect large clusters of items with the same access policy, defining policy for labels is much more efficient than defining policy individually for each item.

Binding an item to a policy label requires care. If it is possible for a replica to change an item's policy label, the replica might thereby be able to obtain more rights to the item than was intended. We solve this problem by naming items with identifiers that incorporate the item's policy label. Specifically, item identifiers are of the form *uniqueId$hash*, where *hash* is a cryptographic hash of the *uniqueId* component and the item's policy label. Item identifiers are the primary means for naming replicated items (both on the wire and in local storage), so if an identifier changes, this constitutes the creation of a new item not an update to an old one. Although an item, once created, cannot be re-bound to a different policy label, the set of policy claims that apply to a given label can and do change, as discussed in the next section.

It is convenient to structure policy labels into a hierarchical namespace. We use the empty string as the root path and '**.**' as the path arc separator. Rights over a name in the hierarchy imply rights over its children. We use the special label *all* as a synonym for the root path. For example, rights on *all* connotes the same rights on all labels, and rights on *contacts* connotes rights on *contacts.private*, but not vice versa.

Our principals are replicas identified by their public keys. Collection items are signed with the public key of the replica at which the most recent change occured. That is, when a user or application issues an update to an item either directly through the replication framework API or as observed by the replication software, a new version of the item is generated and signed with the key of the local replica. The numbering of item versions and the mechanism for determining dominance between versions is assumed to be a property of the replication framework, as in Cimbiosys [24]. Thus, item version numbers do not appear in access control policy (except for revocation as discussed in Section 3.6.2).

By signing a labeled item, a replica asserts that it has *write* permission on the associated label. A correctly imple-

mented replica will not permit writes if policy prohibits it. But even if a malicious replica does permit such a write, the violation will be detected during update propagation.

3.2 Policy claims

We use logic to formalize access control policy and to construct logical proofs that correspond to access control decisions. Numerous logical frameworks have been proposed in the literature that would suffice for our purposes. We chose to use the SecPAL framework because it offers particular flexibility with respect to delegation of authority, and because there is a publicly available SecPAL toolkit. This toolkit translates policy and queries written in SecPAL, or its corresponding XML object model, into Constrained Datalog [25] expressions. In general, access control queries represent assertions to be proved. Policy consists of a set of logical statements from which SecPAL's evaluator must prove the target assertion. The evaluator has been shown to be both sound and complete, and it always terminates [7].

Claims are policy statements made by principals. Collection policy is the sum of all claims made by all replicas that support a collection. A principal is a public key or a special hardcoded entity. There are two special principals: *LA* and *Anonymous*. The first is the universally-trusted local authority (e.g., the ground truth). Any direct claim by *LA* is believed, and all such claims are hard-coded policy axioms. Every access control decision is an attempt to deduce, by chaining together claims, that *LA* says that the desired action is permitted. *Anonymous* is the principal without credentials. It can be used for giving limited rights to "everyone".

A claim is a statement by a principal about a *fact*. A *fact* is a statement about a principal, often inferring the right to perform an action, usually concerning a policy label. Consider the following claim.

A says *B* can *write contacts*

This claim indicates that the principal *A* says that the principal *B* can *write* items that are bound to the *contacts* label.

Facts can be conditional, and such facts contain unbound variables for either principals or labels. In the following claim, *A* indicates that a unbound principal (*%p*) can write *contacts* if that principal can write *photos*.

A says *%p* can *write contacts* if *%p* can *write photos*

Each claim bears an optional, author-relative *claimId* (denoted below in brackets below) to permit subsequent revocation. Facts come in four forms and those forms connote: grant of authority ("can"); delegation of authority ("can say"); revocation ("revokes"); and delegation of revocation authority ("can say ... revokes").

This is a monotone logic in which new statements, other than revocations, cannot cause previously true statements to become false. It is possible for a claim author, or a named delegate, to revoke a claim, but neither revocations nor delegation of revocation rights can be revoked. Since only positive rights can be added, and the right to revoke cannot be revoked, policy cannot be ambiguous. Any set of claims that act as input to an evaluation produces the same yes-or-no result, and the result does not depend on the order of the claims. So, for example, it is not possible in our logic to issue negative (denial) rights since the order in which rights are granted or denied then becomes important. Similarly, it is not possible to revoke a revocation or a revocation right, and for the same reason.

Above, we defined rights associated with the *read*, *write*, and *sync* operations. Here we axiomatically define *control* and *own*. We use the {x,y,z} syntax to connote iteration here. It is not part of SecPAL.

LA says *%p* can say %q can {*read*, *write*} *%l*
    if *%p* can *control %l*

LA says *%p* can {*read*, *write*, *sync*, *control*} *%l*
    if *%p* can *own %l*
LA says *%p* can say
    %q can {*own*, *read*, *write*, *sync*, *control*} *%l*
        if *%p* can *own %l*

On other words, if a principal *controls* a label, he can say that another principal can *read* or *write* it. Similarly, if a principal *owns* a label, he can also *read*, *write*, *sync*, or *control*. Furthermore, he can say that another principal can do any of the above (including *own*) on the label.

SecPAL offers an extensive claim syntax, some of which is applicable to our system. In brief, it includes a means for specifying groups of principals, a constraint grammar that can represent abstractions such as time, pattern matching, and hierarchical path comparison, and a means for limiting recursion when delegating. The SecPAL reference manual [21] describes this syntax in detail.

3.3 The collection manager

As the root of authority on a collection, we introduce a collection manager *CM* and make an axiomatic claim that grants it complete authority. The collection is named by the collection manager public key, and, by axiom, every replica believes this key to have complete authority.

LA says *CM* can *own all*

The collection manager secret key can be maintained either offline or online. The collection manager is a replica, but perhaps often a partial replica holding only replicated policy. In the offline case, the collection manager state resides on storage media (such as a flash key) that can be secured offline. Initial replica bootstrap (as described in Section 3.6.1) is performed by a single program that reads both the collection manager state and the new replica state into memory. In the online case, the collection manager takes the form of

an online server that accepts requests for bootstrap. In either case, sufficient authority can be delegated to online entities to assure progress in the collection manager's absence.

The collection manager, whether online or offline, can then write claims concerning collection policy. Following the example in Figure 4, it gives all rights to the *HomePC* by writing:

CM says *HomePC* can *own all* [CM.1]

The collection manager can now be taken out of the picture if desired since it has delegated all its authority. We use public key cryptography, so the collection manager and its delegates don't need to be online services that present a single-point of failure. However, if a fault-tolerant online service is needed, the cryptography we require could also be implemented using secret sharing [29]. Or, more simply, a single offline collection manager could delegate to multiple online servers.

3.4 Delegating specific rights

To continue the example, *HomePC* then mints a new replica *Laptop* and gives it control over *contacts* items. It also creates replicas for the *Cloud* store and for the *MediaPlayer*, and gives them distinguished rights.

HomePC says *Laptop* can {*read,write,sync*} *all* [PC.1]
HomePC says *Laptop* can *own contacts* [PC.2]
HomePC says *Cloud* can {*read,sync*} *all* [PC.3]
HomePC says *MediaPlayer* can *read photos* [PC.4]

At a later time, we let the *Laptop* create the *Work* and *Mobile* replicas and delegate some of its rights. Note that the delegation from *Laptop* to *Mobile* is an example of a more limited form of delegation than ownership. In this example, *Mobile* is given the right to to change read and write policy, but not the (recursive) right to delegate ownership.

Laptop says *Work* can {*read,write*} *contacts* [L.1]
Laptop says *Mobile* can {*read,write*} *contacts* [L.2]
Laptop says *Mobile* can *control contacts* [L.3]

Summarizing, each replica can issue policy claims granting authority to other replicas over shared resources. Claims are signed with the key of the issuing replica and can thus be verified. Collection policy is the union of all replica claims. Note that our scenario is but one example. The framework can specify a wide range of axioms, rights, and policies.

3.4.1 External representation

Policy claims are encoded as XML and stored in collection items that we call *policy items*. In fact, these items are no different from other collection items and are exchanged between replicas during the normal synchronization process. Thus, no explicit protocol for propagating policy is needed. For simplicity, we write all policy claims for a given replica into a single item. Each replica (that writes claims) is given its own policy label that is then bound to its policy item. To continue our example from Figure 4, the collection manager might enable such labels with the following claims.

CM says *Anonymous* can *read policy* [CM.2]
HomePC says *Laptop* can *write policy.homepc.laptop* [PC.5]

Here we invent a label called *policy*. This label is used as a prefix for all labels that apply to policy items. The first statement allows any replica to read any policy. The hierarchical relationship between labels is used to make children of the policy root replica-specific. *HomePC* already has *own* rights on *all*. However, *Laptop* needs to be able to write a piece of the policy namespace if it is also to issue policy statements. The claim above allows *HomePC* to grant *Laptop* such rights. Note that only *Laptop*, and the replicas through which *Laptop* gained authority (such as *HomePC*) can write policy items for *Laptop*. Note also that *HomePC* could instead grant *own* rights which would allow *Laptop* to authorize another replica with its own policy namespace.

Policy claims are signed by their issuers and the items that contain claims are signed as well. Because the claims themselves are signed, the outer item signature on policy items serves only to detect attempts to overwrite legitimate policy with garbage or old policy.

3.5 Policy enforcement

When a replica makes an access control decision, it must produce a proof indicating that the requested action is allowed by policy. Such proofs are mechanically generated (or a decision is made that no proof is possible) using the collection policy currently available at the authorizing replica. There are three situations in which we perform access control checks. The logical assertions that represent the conditions we want to prove can be expressed as follows.

LA says *R* can *write Label(Item)*
LA says *R* can *read Label(Item)*
LA says *R* can *sync all*

In the first case, the proof context is that of a replica's synchronization engine checking the validity of an updated item. *R* is the key that signed the update and the target is the policy label bound to the item. In the second case, the prover is a replica attempting to validate that a partner replica can download an item during synchronization. *R* is the key that authenticated the synchronization request, and again the target is the policy label of the item being considered. In the last case, the proof context is that of a replica deciding whether the synchronization partner can be trusted to supply protocol metadata. The well-known label *all* is used, since all of our current examples of protocol metadata apply to an entire collection, not a specific item.

In all these cases, the party performing the access control check must prove the corresponding assertion using the cur-

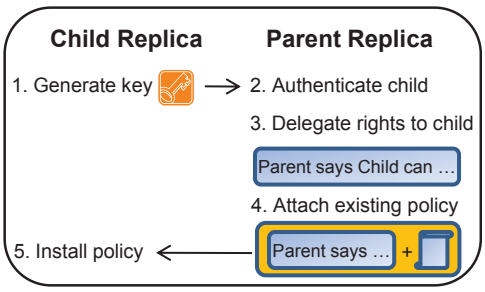


Figure 5. Replica bootstrap

rent policy, initially believing that only *LA* can be trusted. Policy will not include a direct claim by *LA* that allows the desired result since all such claims are axioms, not dynamic policy. So, the logical prover must search the extant collection policy to find a set of claims from which the result can be deduced. The Appendix contains an example proof graph that lends insight into how the prover operates.

3.6 Policy evolution

Policy claims for a collection will accumulate as new replicas and datatypes are added, and as claims are revoked. Hence, we must deal appropriately with the bootstrap of new replicas and revocation of existing policy.

3.6.1 Replica bootstrap

The collection manager defines the initial policy for a collection. We now describe the process of establishing and installing policy in new replicas. The important steps are depicted in Figure 5. We call the replica with existing policy the *parent replica* and the new replica to be endowed with rights the *child replica*. As we've seen, arbitrary replicas can be given the authority to bootstrap new child replicas. The same process is used for all bootstraps, whether the first delegation by the collection manager or a subsequent delegation by a one of its descendants.

When a child replica is created, it cannot perform synchronization without an initial policy since there will be no policy from which to make access control decisions during synchronization. Therefore, we provide an API call that serializes the parent replica's policy such that it can be injected into the child. Before this can be accomplished, a key identifying the new replica must be known to the parent. Once the new replica key is known, appropriate claims for the new replica are written to the parent's policy item.

To complete the example from Figure 4, suppose the replica *Mobile* creates a new replica *SpouseMobile* and gives it (or rather its replica key) read access on *contacts*.

Mobile says *SpouseMobile* can *read contacts* [M.1]

As in Figure 5, all of the existing policy items as well as the new claim are then injected into the new replica as its bootstrap policy.

Secure transport of the child's public key to the parent replica is handled out-of-band and is not addressed by our work. However, there are numerous well-known mechanisms that can be used. In the consumer environment, transfers can often be trusted due to physical proximity, for example by sharing the same physical medium. Over a network, the situation is similar to PKI certificate request/response [1]; steps can be taken to authenticate the new key material. The nature of the authentication protocol can be deployment-specific (for example, it can be based on an existing PKI, payment for service, pre-shared passwords, or knowledge of personal information).

3.6.2 Revocation

Revocation claims are handled in the same fashion as other policy updates. Imagine that the owner of the collection in Figure 4 is away from home and the mobile phone to which it previously delegated authority has been stolen. If that person has possession of *Laptop*, he can use it to write:

Laptop says *Laptop* revokes {*L.2,L.3*}

Where L.2 and L.3 are identifiers for the claims which delegated authority to *Mobile* in the first place. The revocations are believed in the logic because they are uttered by the same principal that made the original statement.

However, such claims are tricky in that they can result in the invalidation of previously valid items. Although it is sometimes useful to invalidate all items that rely on a claim, it is more often appropriate to honor historical claims and disallow only new dependencies on revoked claims. As with similar replication protocols, Cimbiosys maintains a monotonically increasing version number per replica. In order to allow items that depend on historical claims to remain valid, revocation claims can be issued with respect to a vector of version numbers, one element per replica. Hence, a revocation claim need only apply to those versions newer than the associated version vector.

Although we propagate revocation claims like any other policy, the effect of discovering a new revocation is purely local. We invalidate local item versions that are now invalid according to locally visible policy. Other replicas will eventually learn of the revocation and do the same. More generally, revocations only apply to rights associated with replicas, not items. If a revocation claim includes the issuing replica's current version vector, then all versions of items that the replica currently stores will remain valid, but future updates to those items from replicas whose rights have been revoked will be refused.

Our system does not track tainted versions when processing revocations, for example item versions based on revoked versions, but not authored by a revoked replica. Neither does it guarantee to restore previous content when an item version is revoked. These topics are covered in detail in prior work, specifically Mahajan et al. [16]. The mechanisms for

provenance tracking and archiving proposed there should be equally appropriate here.

3.7 Design summary

Recalling the challenges described in the Introduction, our design succeeds in providing policy specification that is independent from implementation detail by encoding access control policy as logical statements that combine without ambiguity. Not only can we reason about access control decisions in abstract, but we can potentially construct and distribute access control proofs ahead of time. Secondly, our design exposition has demonstrated that the delegation primitives available in our logic are sufficient to allow graceful policy evolution. We discuss eventual consistency next.

4. Maintaining eventual consistency

Let us review what has been said to this point. We assume an existing system that supports weak replication of a collection of independent data items, and that replication is eventually consistent. We have posited a system with the following additional properties.

- Policy is encoded in a logical language that provides a means to specify principals, resources (in the form of labels that apply to data items), and delegations of authority involving these, as well as a limited ability to revoke previous statements.
- Authority flows from a single root; policy statements combine without ambiguity. Two replicas with identical sets of policy claims will make identical access control decisions regardless of the order in which they learned of these claims.
- Access control checks are performed within the operations that implement data replication. These access control checks are enforced according to the local policy state present at the time of enforcement.
- Encoded security policy is replicated as data by the existing replication framework.

Our stated goal from the Introduction is that adding access control will preserve eventual consistency. These first steps in doing so ensure that policy propagation is eventually consistent.

1. All policy-relevant state uttered by a given replica propagates in a replica-specific item. Therefore, even if multiple replicas update policy simultaneously, these updates cannot result in a replication conflict.
2. Read access controls do not apply to policy items, and therefore cannot prevent policy from propagating.

In a weakly consistent environment, local policy may differ between replicas at any given time. Thus, we risk the possibility that access control decisions may have different results depending on when and where they are performed. If a failure occurs on a read or sync operation, this doesn't pose a problem. Synchronization is periodic. Eventually policy propagation will complete and subsequent operations will succeed and produce the same result. However, what happens if an update fails? If an update succeeds at one replica but fails at another, we cannot permit the situation to become permanent. Since items are independent, it is sufficient to study the single item depicted in Figure 2: if we can show eventual consistency is preserved for one item in the presence of access control, the same will be true for all items.

Consider an original version U and an updated version U' of an item. Eventual consistency is violated if U becomes persistent at one replica while U' persists at another. Divergence cannot occur if the same policy is in place at all replicas when the update U' is applied: the same access control decisions will be made in all cases. However, suppose policy P is in place at one replica while an updated policy P' prevails at a different replica. Divergence can occur if P' allows U' when P does not, or if P allows U' when P' does not. The update U' enters a replica either by being created there or via a synchronization.

3. If U' is not authorized at a authoring replica, then the update will not be admitted and there will be no divergence.
4. If U' is not authorized at a receiving replica, then transmission of U' is retried periodically until P' arrives at the destination, allowing U' to succeed on the next attempt.
5. Suppose U' is inappropriately authorized at the receiving replica because updated policy P' has not arrived yet. If P' allows fewer rights than P there must have been a revocation. To handle this case, the replica must re-evaluate the validity of all previously received items that might now be invalid.

If the steps above are followed, we argue, eventual consistency will be maintained. If retries of failed updates are necessary as in Step 4, this situation will not persist. If the receiver's policy is up-to-date, then the sender's policy must be out-of-date. Retries will cease as soon as the sender's policy is updated.

5. Design motivation and limitations

We now discuss several issues that motivate our design choices and suggest potential limitations.

Single root of authority. The root of authority at each node of our system is the special principal *LA*, the local authority. *LA* axiomatically believes everything the collection manager says, thus each replica of a collection believes the same root of authority. Furthermore, our logic is monotone with limited revocation, and thus absent of any ambiguity when policy statements are combined. Without these properties, replicas can engage in policy wars where multiple authorities issue policy statements that are logically inconsistent. For example, one root might choose to revoke the

authority of a second root, and vice versa. The lazy propagation of policy that our system assumes would exacerbate the effects of such inconsistencies.

Similarly, distributed trust protocols like SDSI [23] make it possible to combine disjoint, local roots of authority. These protocols allow non-transitive patterns such as *A* trusts *B* and *B* trusts *C*, but *A* doesn't trust *A*. This sort of system produces what might be thought of as intentional divergence, and it runs contrary to our goal of eventual consistency.

Public key infrastructure. In our prototype, we make statements directly about replica keys. We do so to avoid a dependency on a specific authentication infrastructure, but we could just as well add a layer of indirection and name principals with strings that are authenticated elsewhere (for example in a PKI or shared-key infrastructure). It is easy, however, to confuse authentication and authorization. Existing PKI certificates are not sufficient to express the authorization relationships we provide. For example, PKIs support only a limited form of delegation, namely the right to make certificates involving a specific name prefix. Our "can say" expression is more flexible in what the delegate can say, and also can grant the ability to make new policy.

Binding of labels and data. We chose to closely bind labels and data items. In the types of applications we imagine, it is easy enough to create a new item under a different label if reclassification is necessary. However, using the same framework we could have allowed labels to vary and set policy limiting such modifications.

Policy as a first class object. In many systems, such as file systems, access control policy is deeply encoded in system metadata, for example in file system ACLs and group membership structures. This can make it difficult to determine or evaluate access control policy out of context. In contrast, our policy statements are enumerable and independent from the details of access-control enforcement. This independence makes it easy to transport policy in the replication protocol it protects, and simple to add constraints that, for example, ensure that policy propagation will converge. Moreover, our design for policy representation also allows us to quickly determine the effect of a revocation on a replica's state and thus perform invalidation efficiently.

Item longevity. In our system, items are only valid for as long as policy permits: a change in policy can result in invalidation. Public key certificate stores usually operate in this manner; file systems do not (the deletion of a user account does not usually invalidate that user's files.) There is a legitimate question about where dynamic data invalidation is appropriate. However, in protocols that attempt to replicate state, rather than an log of operations, invalidation is the only plausible option if revocation is supported at all, since there is no notion of history to help gain consensus about when an update might have been valid. Even log-based protocols without global ordering suffer from some of the same problems, moreover logs must ultimately be truncated.

Replication topology. We want our techniques to apply to eventually consistent systems in general, and so we wish to avoid constraints on network topology and update periodicity in our design. However, it is certainly possible to add read access controls to an existing topology that interfere with the intended propagation of updates. Eventual consistency depends on the correct operation of at least one trusted, transitive propagation path between any pair of replicas that share updates. We leave it future work to prevent misconfiguration of access controls. Existing proposals for compromise recovery [16] can be employed to ensure that damage caused by a malfunctioning replica can be recovered if detected, so the temporary absence of a propagation path can be tolerated.

Protocol metadata. Compromised replicas cannot forge the signature of other replicas and, if item version identifiers increase monotonically, they cannot pass off old content as new. However, a replica that incorrectly implements the replication protocol can inhibit update propagation by hiding new versions of items or passing bogus replica metadata to confuse its sync partners. *Learned knowledge*, from the Cimbiosys protocol, is an example of this kind of metadata. We introduced access controls on the *sync* operation to specify whether a replica should be trusted to act as the source of this kind of metadata. This limits, but does not eliminate, the impact of protocol corruption.

Complex policy. We have not fully explored the range of policies that can be implemented using our system. However, our framework is quite flexible. For example, in order to model access control on a replicated hierarchical file system where a principal's rights on a directory might differ from those on the directory's files, we can easily create new primitive rights describing actions on a directory. However, in order to maintain item independence, any ordering dependencies between file system updates must also be eliminated.

6. Implementation

As discussed in Section 2, access control policy is enforced by monitoring replication protocol operations. We implemented our system within the Cimbiosys [24] replication framework. A high-level component diagram of Cimbiosys is given in Figure 6. The Cimbiosys API associates items and policy and allows applications to access a persistent *item store*. The synchronization engine drives the *communication* component which implements the replication protocol.

We augmented the existing Cimbiosys framework by adding the security guards and policy evaluator depicted in Figure 6. These components monitor the replication protocol as well as making sure the application cannot violate security policy. Our modifications consist of roughly 1000 lines of C# code. Access control policy is created and maintained in the *PolicyMgr* component on each replica. This component is responsible for encoding policy in the form of SecPAL logical expressions and bootstrapping new repli-

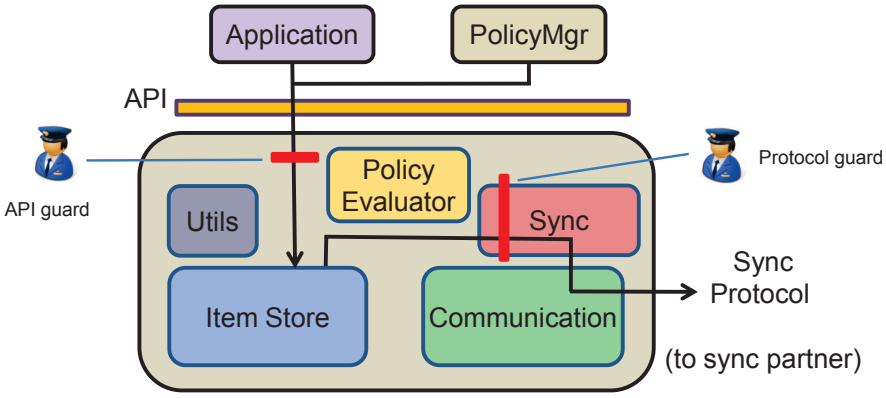


Figure 6. Cimbiosys components with security guards in sync protocol and API

cas. Furthermore, the *PolicyMgr* stores policy statements as replicable items and retrieves those items from stable storage when the system restarts.

Because per-replica policy is stored in identifiable items, it is easy for the replication machinery to tell when policy changes. This proves useful for checking if a received update has become invalid and for caching of access control results. The *PolicyMgr* can be viewed as just another client of the Cimbiosys API. In particular, the API contains a notification mechanism that calls clients when an item changes. This allows the *PolicyMgr* to be notified when policy items change so that it can, in turn, inform the security component of new policy.

We depend heavily on the Microsoft Research SecPAL release which is a .NET library that we use without modification. The language parser included in this release can evaluate statements in the SecPAL grammar, however it does not support encoding of actual cryptographic keys or signatures on claims. Instead, the SecPAL library provides an XML object model in which all the claims represented in the grammar can be expressed. We employ this model since it provides serialization (to XML) that handles RSA signature creation, as well as signature checking on deserialization. Although it should be possible to build a component that bridges the gap between the SecPAL language grammar and the XML model, for simplicity we chose to use the XML model directly. Thus, we offer a procedural interface to policy creation, rather than direct access to the grammar.

6.1 Performance

Other work has evaluated the performance of Cimbiosys [24]. Here we attempt to justify that adding a logical policy checker does not add excessive overhead. Furthermore, in many scenarios, evaluation results can be effectively cached, thus requiring checker overhead only when policy changes. We discuss this opportunity in the following section.

Assuming policy changes are relatively infrequent, the most important metric is the cost of deriving an access control proof, which is almost entirely spent in the SecPAL library trying to prove logical assertions. We measured the cost of evaluating various assertions using the policy from Figure 4. The policy contains 23 claims and we give results for queries that require different sets of claims. For each query, Table 1 shows the number of steps in the resulting proof, the length of the delegation chain from the collection root, and the average latency over 1000 tries. Our tests were run under Windows Vista on a HP xw4400 Workstation with an Intel Core 2 processor at 2.40 GHz.

On modern hardware, 56 ms. is a very long time. However, there is room for optimism. First, the timings from the *Released* column of Table 1 correspond to the released SecPAL implementation which is completely unoptimized. A later, unreleased version of this code base, for which timings appear in the *Optimized* column, includes a collection of performance improvements. These represent more realistic performance expectations. Most significantly, the new library contains a claim indexing framework that enables the solver to make better choices about the set of candidate claims for consideration during proof generation. This gives at least a factor of 5 improvement in operation speed.

However, at least one important optimization is still untried. As mentioned earlier, there is a conversion from our policy representation to Datalog. A better implementation would perform that conversion once for any given policy. However, the SecPAL release we are using does not cache the Datalog representation. Nor does it try to avoid redundant transformations on policy in the process of deriving the Datalog representation (such as those required to implement hierarchical resources). Code profiling shows three major components of proving overhead: transformations on policy, conversion of transformed policy to Datalog, and proof resolution of the Datalog representation. The first two of these consume 40-50% of the overhead for the examples tested

Query	*Proof steps*	*Delegations*	*Released (ms.)*	*Optimized (ms.)*
HomePC can *write all*	10	1	42	7.5
MediaPlayer can *read photos*	13	2	43	7.7
Mobile can *write contacts*	15	3	56	9.2
SpouseMobile can *read contacts*	20	4	56	9.4

Table 1. Prover performance

here. Thus, an implementation that caches the Datalog representation gains at least a further factor of two in performance. There are undoubtedly other possible optimizations.

6.1.1 Evaluation caches

Given the nature of our application, we can easily cache not only Datalog conversions, but entire access control evaluations. Since changes to policy are clearly identifiable when an update to policy arrives, any cache of previous results can be accurately invalidated when a claim is revoked. The prover can be made to output the set of claims involved in any proof, thus cached results can be indexed by constituent claims making it easy to identify which results rely on a revoked claim. With such accurate cache invalidations, the only reason to perform a new evaluation is to accomodate a new replica/action/label tuple in an access control request. Any repeat requests can be resolved from the cache, and in some scenarios the hit rate can be very large.

A cache of negative results is somewhat more difficult to manage since the prover cannot tell us which claims it lacks. Since recovery from spurious access denial due to stale policy state relies on periodic retry, any cache of negative results can be invalidated entirely based on time. We are not sure if a negative cache can be justified.

An evaluation cache will clearly be most effective where there are relatively few subjects and objects of access control decisions. Our system was designed with the intent that policy labels would be relatively few. As more labels are used, the number of claims that must be represented (and searched) in replica policy increase and performance will suffer. Nevertheless, we believe that in any tractable access control system, the overall size and scope of policy is bounded by complexity, which limits the useful number of claims. Most of the applications we have modeled use only a handful of labels to represent policy. Similarly, we target applications with relatively few replicas such as home networks and collections of devices used within a family, small social network, or small business. We also target collections where many replicas can gain *Anonymous* access without needing independent credentials such as large distribution networks where the integrity of the source is important, and privacy is not a concern.

7. Related work

There is a wealth of related work in the literature. Much of this work breaks down into two categories: access control in distributed systems and logic-based access control. We discuss the most directly relevant examples.

Grapevine [8] and Bayou [30] are examples of distributed systems with eventually consistent replication. Microsoft's Active Directory [18] is a commercial example of such a system in widespread deployment. These systems enforce access controls on clients, however all replicas are equally trusted as in Figure 1a. In a similar vein, Samarati [26] studies how access control updates compose when subject to misordering under weak consistency. As above, in this setting each node's updates are equally trusted.

Most distributed file systems, such as AFS [27], FARSITE [2], Taos [32], provide a model of a centralized system, even though their implementation can involve multiple servers and replication of data. Their network servers are equally trusted, or in the case of FARSITE, untrusted. Peer-to-peer file systems such as Chord/CFS [10] and Ivy [22] provide distributed or replicated data storage over peer-to-peer networks, but the replica servers aren't themselves principals in the corresponding access control scheme. Self-certifying file names [17] underlie both CFS and Ivy, and also inspire our method for binding policy labels to items.

Our system is perhaps most closely related to UIA [12]. UIA addresses the similar problem of joining cooperating devices into an ad hoc naming network. It uses a per-device log to encode, merge, and ultimately gain agreement on naming across devices. There is no root of authority in this system, so disputes with revoked principals have to be resolved manually. Furthermore, UIA currently only manages naming elements such as groups, names, and links, but does not extend to arbitrary data.

There has been much prior work concerning the use of logic for policy enforcement. Early logical frameworks for distributed system security [14, 32] used logic to rationalize system security design. Later work by Appel and Felten [3] demonstrated that a logical proof checker can be employed to automate the process of validating encoded credentials. PolicyMaker [9] showed the value of expressing system policy, not just authentication credentials, in a precise fashion. SD3 [13] and Binder [11] joined these threads by expressing policy in a logic-based language that can be evaluated. Subsequent systems [4, 6, 7, 15], have extended the performance and expressibility inherent in logical policy frameworks.

Like the Grey system [5] which deployed a logic-based physical access control system using networks, custom door-locks and cell phones, our work is a demonstration of the

applicability of logic-based security in a distributed setting. Although the problem domains are different, by way of comparison we are able to take advantage of the more-powerful logic that SecPAL offers, for example, giving richer control over delegation of authority. Furthermore, in Grey, considerable work must be done to piece together assertions scattered about a network, while our system is concerned with propagation and consistency.

8. Conclusion

Replicated systems that offer only weak consistency are increasingly common in home settings, the cloud, and mobile environments. Most obviously, they add value when connectivity is imperfect, but they also can benefit environments where there are many devices but no management infrastructure to coordinate them. Similarly, loosely-organized or ad hoc systems can be useful for spanning administrative domains when no formal arrangements exist (for example, where home computing interacts with work-related infrastructure). Data replication with eventual consistency is a fundamental tool for such applications.

In this paper we have discussed an access control framework for weakly consistent replication in which replicas are not equally trusted. We demonstrate that there are serious difficulties in building such systems due to ordering of updates and the need to maintain eventual consistency. To address these difficulties, we encode our security policy in a logical framework where access control decisions correspond to automatically-generated, logical proofs. This framework creates a portable and extensible substrate for expressing security policy in a distributed system. Our policy statements are self-contained and enumerable items, replicated in the protocol they protect. Rather than enforce a one-time guard on the admission of valid items, we accept that policy can be temporarily inconsistent and ensure that the set of valid items always corresponds to the current policy.

We have shown that our framework of logic-based policy distribution works well for handling the unpredictable semantics of weakly consistent replication, and that using it we have offered solutions to the three-fold challenges of policy specification, evolution, and consistency.

A. An example SecPAL proof

In this Appendix, we list some example diagnostic output from the SecPAL prover using the example security policy from Section 3. The outermost leaves of the proof are statements made by principals, and these are joined into proof deductions going upwards. The top line is the desired query result.

```
LA says SpouseMobile can read "contacts"
|
+-LA says Laptop can say SpouseMobile
  | |       can read "contacts"
  | |
  | +-LA says %p can say %q can read %r if
  | |       %p can own %r
  | |
  | +-LA says Laptop can own "contacts"
  |   |
  |   +-LA says HomePC can say Laptop
  |   | |       can own "contacts"
  |   | |
  |   | +-LA says %p can say %q can own %r if
  |   | |       %p can own %r
  |   | |
  |   | +-LA says HomePC can own "contacts"
  |   |   |
  |   |   +-LA says CM can say HomePC
  |   |   | |       can own "contacts"
  |   |   | |
  |   |   | +-LA says %p can say %q
  |   |   | |       can own %r if %p can own %r
  |   |   | |
  |   |   | +-LA says CM can own "contacts"
  |   |   |   |
  |   |   |   +-LA says CM can own ""
  |   |   |
  |   |   +-CM says HomePC can own "contacts"
  |   |     |
  |   |     +-CM says HomePC can own ""
  |   |
  |   +-HomePC says Laptop can own "contacts"
  |     |
  |     +-HomePC says Laptop can own "contacts"
  |
  +-Laptop says SpouseMobile can read "contacts"
    |
    +-Laptop says Mobile can say SpouseMobile
    | |         can read "contacts"
    | |
    | +-Laptop says Mobile can say
    |           %p can read "contacts"
    |
    +-Mobile says SpouseMobile can read "contacts"
```


Acknowledgments

We would like to thank our shepherds, Marc Shapiro and Ashvin Goel for their help in improving this paper. We also thank the anonymous reviewers for their insightful comments. Martín Abadi made significant contributions to this work. Rama Kotla, Cathy Marshall, Rama Ramasubramanian, and Úlfar Erlingsson gave us valuable feedback on early drafts. We also thank Moritz Becker, Jason Mackay, and the SecPAL team for their patience in helping us understand SecPAL and its prototype implementation.

References

[1] C. Adams, S. Farrell, T. Kause, and T. Mononen. Internet X.509 Public Key Infrastructure Certificate Management Protocol (CMP). `http://tools.ietf.org/html/rfc4210`.

[2] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceeding of the 5th Symposium on Operating Systems Design and Implementation, Boston, MA, USA*, pages 1–14, 2002.

[3] A. W. Appel and E. W. Felten. Proof-carrying authentication. In *Proceedings of the 6th Conference on Computer and Communications Security, Singapore*, pages 52–62, 1999.

[4] L. Bauer, M. A. Schneider, , and E. W. Felten. A general and flexible access-control system for the web. In *Proceedings of the 11th USENIX Security Symposium, San Francisco, CA, USA*, 2002.

[5] L. Bauer, S. Garriss, J. M. McCune, M. K. Reiter, J. Rouse, and P. Rutenbar. Device-enabled authorization in the Grey system. In *Proceedings of the 8th Information Security Conference, Singapore*, pages 431–445. Springer Verlag LNCS 3650, 2005.

[6] L. Bauer, S. Garriss, and M. K. Reiter. Distributed proving in access-control systems. In *Proceedings of 2005 IEEE Symposium on Security and Privacy, Oakland, CA, USA*, pages 81–95, 2005.

[7] M. Y. Becker, C. Fournet, and A. D. Gordon. Design and semantics of a decentralized authorization language. In *Proceedings of 20th IEEE Computer Security Foundations Symposium, Venice, Italy*, pages 3–16, 2007.

[8] A. D. Birrell, R. Levin, M. Schroeder, and R. Needham. Grapevine: an exercise in distributed computing. *Communications of the ACM*, 25(4):260–274, 1982.

[9] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy, Oakland, CA, USA*, pages 164–173, 1996.

[10] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles, Banff, Canada*, pages 202–215, 2001.

[11] J. DeTreville. Binder, a logic-based security language. In *Proceedings of 2002 IEEE Symposium on Security and Privacy, Oakland, CA, USA*, pages 105–113, 2002.

[12] B. Ford, J. Strauss, C. Lesniewski-Laas, S. Rhea, F. Kaashoek, and R. Morris. Persistent personal names for globally connected mobile devices. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation, Seattle, WA, USA*, pages 233–248, 2006.

[13] T. Jim. SD3: a Trust Management System with Certificate Evaluation. In *Proceedings of 2001 IEEE Symposium on Security and Privacy, Oakland, CA, USA*, pages 106–115, 2001.

[14] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, 1992.

[15] C. Lesniewski-laas, B. Ford, J. Strauss, R. Morris, and M. F. Kaashoek. Alpaca: extensible authorization for distributed services. In *Proceedings of the 14th ACM Conference on Computer and Communications Security, Alexandria, VA, USA*, pages 432–444, 2007.

[16] P. Mahajan, R. Kotla, C. Marshall, V. Ramasubramanian, T. Rodeheffer, D. Terry, and T. Wobber. Effective and Efficient Compromise Recovery for Weakly Consistent Replication. In *Proceedings of the Fourth EuroSys Conference, Nuremburg, Germany*, pages 131–144, 2009.

[17] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proceedings of 17th ACM Symposium on Operating Systems Principles, Kiawah Island Resort, SC, USA*, pages 124–139, 1999.

[18] Microsoft Corporation. About Active Directory Domain Services. `http://msdn.microsoft.com/en-us/library/aa772142(VS.85).aspx`.

[19] Microsoft Corporation. Office Groove 2007 Developer Portal. `http://msdn.microsoft.com/en-us/office/bb308957.aspx`.

[20] Microsoft Corporation. Microsoft Sync Framework. `http://code.msdn.microsoft.com/sync`.

[21] Microsoft Corporation. Security Policy Assertion Language (SecPAL), .NET Developer Documentation. `http://research.microsoft.com/projects/SecPAL/`, 2007.

[22] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation, Boston, MA, USA*, pages 31–44, 2002.

[23] R. Rivest and B. Lampson. A Simple Distributed Security Infrastructure (SDSI). `http://groups.csail.mit.edu/cis/sdsi.html`, 1996.

[24] V. Ramasubramanian, T. Rodeheffer, D. Terry, M. Walraed-Sullivan, T. Wobber, C. Marshall, and A. Vahdat. Cimbiosys: A platform for content-based partial replication. In *Proceedings of 6th USENIX Symposium on Networked Systems Design and Implementation, Boston, MA, USA*, 2009.

[25] P. Z. Revesz. Constraint databases: a survey. *Semantics and Databases*, 1358:209–246, 1995.

[26] P. Samarati, P. Ammann, and S. Jajodia. Maintaining replicated authorizations in distributed database systems. *Data and Knowledge Engineering*, 1(18):55–84, 1996.

[27] M. Satyanarayanan. Integrating security in a large distributed system. *ACM Transactions on Computer Systems*, 7(3):247–280, 1989.

[28] F. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(299), 1990.

[29] A. Shamir. How to share a secret. *Communications of the ACM*, 11(22):612–613, 1979.

[30] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles, Copper Mountain Resort, CO, USA*, pages 172–182, 1995.

[31] W. Vogels. Eventually consistent. *ACM Queue*, 6(6):14–19, 2008.

[32] E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems*, 12(1):3–32, 1994.