Microsoft Research Technical Report MSR-TR-2009-15
Published February 2009, revised July 2009.

Policy-based Access Control for Weakly Consistent Replication

Ted Wobber, Thomas L. Rodeheffer, and Douglas B. Terry
Microsoft Research, Silicon Valley

Abstract. Enforcing authorization policy for operations that read and write distributed datasets can be tricky under the simplest of circumstances. Enforcement is too often dependent on implementation specifics and on policy detail that is inextricable from the data under management. When datasets are distributed across replicas in a weakly-consistent fashion, for example when updates to policy and data propagate lazily, the problem becomes substantially harder. Specifically, if disjoint replicas can make different decisions about the permissibility of a potential modification due to temporary policy inconsistencies, then permanently divergent state can result. In this paper, we describe and evaluate the design and implementation of an access-control system for weakly consistent replication where peer replicas 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 organically to encompass new replicas through delegation. Eventual consistency is preserved despite the fact that access control policy can be temporarily inconsistent.

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, intentionally or unintentionally, 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 [1] are well understood, we choose to examine systems with loose synchronization semantics as these have fewer operational constraints. For example, tools such as Groove [2] provide file replication between directories located on multiple machines. Microsoft's Sync Framework [3] offers a general plat-

form for providing multi-node synchronization of arbitrary datatypes. Directory services such as Grapevine [4] and Active Directory [5] support lazy propagation of updates between distributed servers. All of these systems can be categorized as *peer-to-peer* in the sense that updates to replicated state can propagate through peer *replicas*; 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 equality of replicas. However, these systems do support *eventual consistency*: replicas eventually converge to identical states and are guaranteed to do so if updates cease.

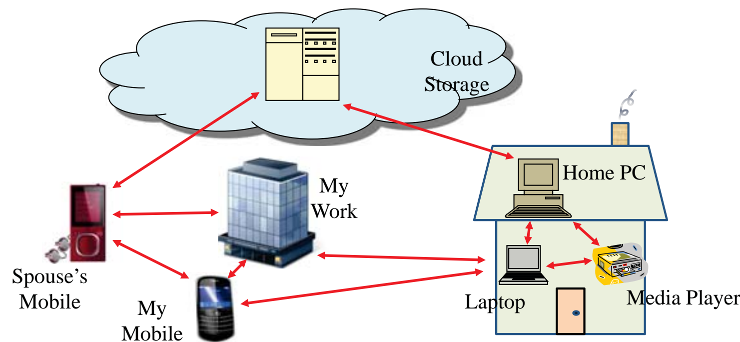
In this paper, we consider the problem of providing access control in the context of weakly consistent replication where replicas are not equally trusted. While there are numerous examples of systems that use replication to provide a distributed service with restricted access to clients (including some of the aforementioned systems), we know of little work that discusses how to support differing levels of trust within such a service so that some peers have only limited authority to read, author, and propagate updates. This is an important problem because in real life collections of cooperating devices are not homogeneous. A user might trust his home machine more than a cloud server, or his web server less than the machine where his finances are kept, or his cell phone less than his laptop. A replica serving a particular naming domain might be trusted only to update names within that domain. There might 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.

Deploying meaningful access control policy where there is only weak consistency presents three primary challenges.

- Policy must be distinguishable, and enforcement implementation-independent.
- Policy must evolve seamlessly as new replicas and data are added.
- Access control must not prevent eventual consistency.

The first challenge requires additional explanation. In many systems, for example file systems, policy is not stated explicitly but rather encoded in a multiplicity of access control lists (ACLs) that are not easily separable from ordinary data. Moreover, ACL enforcement is highly implementation dependent. As we will see, neither of these properties works well in a system in which there are temporal policy inconsistencies.

This paper presents a new system that addresses these challenges in the context of weakly-consistent replication. To our knowledge, this is the first system to provide an authorization framework for weakly-consistent replication without uniform trust. We specify security policy using the SecPAL logical policy assertion language [6]. Thus, policy is separate from data and enforcement amounts to proof-generation. 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 access control state and dealing with any potential inconsistencies that might lead to data divergence. We have implemented our system in the context of a general-purpose replication framework, Cimbiosys [7]. However, we expect the methods we describe here to be applicable to weakly-consistent replication protocols in general.



Replica	*Read*	*Write*	*Own*	*Sync*
HomePC	all	all	all	all
Laptop	all	all	contacts	all
MediaPlayer	photos			
Cloud	all			all
Work	contacts	contacts		
Mobile	contacts	contacts	contacts	
Spouse-Mobile	contacts			

Fig. 1. An example replicated collection and policy

The remainder of the paper is organized as follows. Section 2 lays out our system model and the threat environment our system is designed to tolerate. Section 3 describes the problem at hand in more detail. Section 4 presents our system design. Section 5 provides additional implementation specifics as well as a brief performance evaluation and discussion. Section 6 describes related work, and Section 7 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, or some combination of shared data. Figure 1 shows an example collection. Collections contain sets of *items* and can appear on one or more *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 [7]. Updates to items 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 all replicas will ultimately synchronize (in both directions) with at least one peer.

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 that then alerts the replication infrastructure. We place only a few constraints on replication mechanism itself (discussed in Section 4.3). As mentioned earlier, we assume eventual

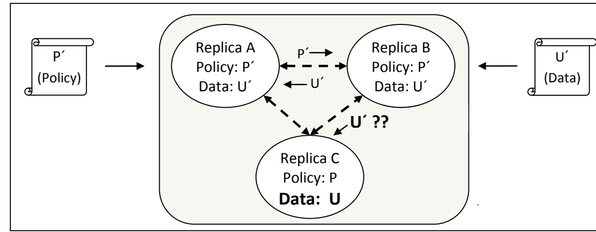


Fig. 2. Concurrent policy and data updates

consistency. Updates can arrive out of order or be superseded en route by newer updates to the same data. Conflicts can occur due to simultaneous updates authored at different replicas, and conflict resolution need not be automatic. Communications failures are tolerated as long as network partition is not permanent.

The security principals in our access control system are replicas. The sorts of client applications we are considering (e.g., productivity and portability apps) typically involve a single human user per replica. Devices or servers that hold replicas for multiple users are assumed to be capable of restricting replica access to the replica owner. Alternatively, an application might build user-level access control on top of our framework.

Since replicas are unequally trusted, our aim is to control access over the replication protocol operating between them. Figure 1 depicts a small collection with several replicas. Two types of items are held in this collection: contacts and photos. The figure also suggests an informal authorization policy that might govern such a collection. In this policy, *own* access implies the right to set policy, while *write* access implies the right to create or modify. All replicas are allowed to sync with other replicas, however a *sync* value of *all* implies that the named replica should be trusted to provide collection metadata, for example an accurate enumeration of collection items. The access control decisions that enforce this policy are carried out during the synchronization operations implied by the inter-replica arrows.

We assume that items are signed with public keys associated with replicas. The purpose of security policy is to grant appropriate rights to these replica keys. Note that since data items bear the authoring replica's signature, a replica can provide a new item version to a sync partner without *write* permission as long as the item bears the signature of an authoring replica that does. Similarly, we can detect attackers who forge items or author them with insufficient rights. Our controls on reading items and providing collection metadata depend transitively on the correctness of each replica through which data can pass. We do not implement cryptographic privacy, although replica keys could potentially be used for that purpose. In another paper, we describe an archive-based technique to permit distributed state recovery after replica compromise within this framework [8]. We do not explicitly deal with denial-of-service attacks.

3 Policy and replication

One difficulty that arises when implementing access control under weakly-consistent replication is that the access control state at different replicas can vary at any given

time. Inconsistency due to re-ordering of updates can clearly occur since we make no assumptions about update ordering or global clocks. We must ensure that such temporary policy inconsistency does not result in permanent inconsistency of replicated state.

In Figure 2, we depict a three-node collection that we use to examine what can happen when replicas are both inconsistent and mutually suspicious. The dashed lines represent the normal synchronization relationships between these nodes. The solid lines represent update flows. The collection initially has policy P. Suppose two 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 allowed by P but disallowed by P'.) Suppose further that P' has not yet propagated to node C. Since our system places no constraints on update ordering, U' can in fact arrive at node C prior to P'. In a framework of mutual trust, U' will be accepted at C because the critical access control check takes place where 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' can be rejected at C although it was accepted at other nodes. If so, the replica state at C will differ from other replicas, and lacking other action, the divergence will be permanent.

The situation becomes more complicated if access control policy is not separable from data under management. For example, if an access control list carried in one item can reference group names whose membership is represented by other items, then a reordering in the update flow of such items can again cause unintended states, incorrect access decisions, and violation of eventual consistency. As more collection-specific state becomes involved, and as the actual policy to be enforced depends more on implementation detail, determining and producing a consistent state can become difficult.

A final scenario can also cause trouble. It is sometimes necessary in replicated systems to download an entire replica from another, possibly due to failure or to the creation of a new replica. What should happen during such a download if the policy that allowed for the authorship of an item has been changed and the item is thus no longer approved by the current policy? In other words, do we enforce policy exactly once when the item is first encountered, or check whether items are valid for only as long as the policy that validates them is extant?

We address these concerns in Section 4.3 below.

4 System design

In this section we describe the design of our access control system. We refer the three challenges introduced in the Introduction and organize this material to reflect how our system handles these separate challenges. We conclude the section with a discussion that motivates our design.

4.1 Policy encoding and enforcement

Collections, items, and labels. Our principals are replicas and we identify them with public keys. We call the initial replica of a collection its *collection manager*, and the collection is named with its key. The collection manager is the de-facto trusted root

authority for the collection. The collection manager secret key can be maintained either offline or online. 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 4.2) 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.

Collection items are signed with the public key of the authoring replica. 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 handled by the replication framework. Thus, item version numbers do not appear in access control policy (except for revocation).

We control three specific operations during replica synchronization: *write*, *read*, and *sync*. An author replica must be able to prove (to other replicas) that it is entitled to *write* an item in order for such an item to be considered valid. Each replica must be able to prove during synchronization that it is entitled to *read* items from the requested collection (or subset). Finally, in order for a replica to trust collection metadata from its peer, for example an accurate enumeration of items, that replica must have *sync* rights on the collection.

The resources subject to access control in our system are *policy labels*. Labels represent explicit classes of items. Policy grants replicas access rights over labels and hence over the items bound to these labels. Policy labels are defined in a hierarchical namespace where the empty string is the root path and '`.`' is the path arc separator. Rights over a parent in the hierarchy imply rights over its children. We use the special label *all* as a synonym for the root path.

Policy binding requires care. If an untrusted replica can change a policy label on an item, it can assign a policy label that gives itself access rights, and this is tantamount to forging an item. Hence, we bind a policy label into each item by creating a secure item identifier that contains a cryptographic hash of the item's identifier and its policy label. The hash value is checked as part of every access control decision. Thus, an attacker cannot change the item policy without creating either an invalid update or a new item. The cost of this technique is that once created, an item cannot be re-bound to a different policy label. However, as discussed next, the set of policy claims that apply to a given label can and do change.

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 choose to use the SecPAL framework because it offers particular flexibility with respect to constraints on delegation of authority. Furthermore, Microsoft has released a free public toolkit that can represent and evaluate a substantial subset of SecPAL. This toolkit translates policy and queries written in SecPAL, or its corresponding XML object model, into Constrained

Datalog [9] expressions. In general, queries represent assertions to be proved, and policy represents a set of evidentiary claims. SecPAL's evaluator attempts to construct a logical proof using the Datalog expressions derived from these entities. This evaluator has been shown to be both sound and complete, and it always terminates [6].

Claims are policy statements made by principals. 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 most if not 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".

Claims can be conditional on either other facts or constraints. A fact is a statement about a principal, often inferring the right to perform an action, usually concerning a policy label. Facts can contain unbound variables for both principals and labels (e.g., *%p* and *%l* which link direct and conditional facts in a claim. Each claim can bear an optional, author-relative *claimId* 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"). Hence, this is a monotone logic in which rights can only be added, but that supports revocation of specific claims by the claim author or a delegate. Facts, in general, are not revokable.

We defined the action verbs *read*, *write*, and *sync* in the previous section. Here we axiomatically define the verb *own* which connotes full authority over a label including the right to delegate all rights. [1]

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

In other words, if a principal *owns* a label, he can *read*, *write*, or *sync*. Furthermore, he can say that another principal can do any of the above. We will not present the full claim syntax here. 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.

As the root of authority, we introduce a collection manager *CM* and write a claim that grants it authority over the entire collection.

LA says *CM* can *own all*

The collection manager, whether online or offline, can then write claims concerning collection policy. Following the example in Figure 1, it gives all rights to the *HomePC* by writing:

CM says *HomePC* can *own all*

[1] The {x,y,z} syntax is not part of SecPAL. We use it as an abbreviation for an iterated expression formed by substituting each of the elements within the braces into the encapsulating text line.

The collection manager can now be taken out of the picture if desired since it has delegated all its authority. 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. Similarly, the new *Laptop* replica can delegate some of its rights, and so on. Note that the delegation from *Laptop* to *Mobile* is an example of a more limited form of delegation than ownership.

HomePC says *Laptop* can {*read,write,sync*} *all*
HomePC says *Laptop* can *own contacts*
HomePC says *Cloud* can {*read,sync*} *all*
HomePC says *MediaPlayer* can *read photos*
Laptop says *Work* can {*read,write*} *contacts*
Laptop says *Mobile* can {*read,write*} *contacts*
Laptop says *Mobile* can say *%p* can {*read,write*} *contacts*
Mobile says *Spouse-Mobile* can *read contacts*

So, each replica can issue policy claims determining the authority of other replicas over resources in the collection. Such claims are signed with the key of issuing replica and can thus be verified. The claims above complete the example policy from Figure 1. The overall collection policy is the union of all the replica claims.

Authorization. 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 collection metadata. Such metadata might be used, for example, to implement local deletion of items that have been expunged from the collection. *R* is a key authenticating the partner. The well-known label *all* is used, since all of our current examples of collection metadata apply to an entire replica, not a specific item.

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

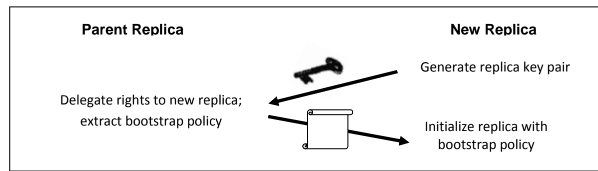


Fig. 3. Replica bootstrap

Encoding. Policy claims are encoded as XML and stored in collection items and these items are distributed as are all others. Thus, no explicit protocol for propagating policy is needed. For simplicity, we write all policy claims for a given replica into a single item, but that is not a requirement. Each replica controls a private policy label that is bound to the items into which it writes policy. Since only one replica (and the universally-trusted collection manager) can write items under such labels, there is no possibility of conflicting updates on these items. To continue our example from Figure 1, the collection manager might enable private labels with the following claims.

CM says *Anonymous* can *read policy*
CM says *HomePC* can *own policy.homepc*

Here we invent a world-readable label called *policy* that represents collection items that hold policy claims. The hierarchical relationship between labels is used to control naming. The claim above gives *HomePC* ownership of a subspace of all policy items. Delegates of *HomePC* can then be given control over a more restrictive subspace (for example, *policy.homepc.laptop*).

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. Thus, although malformed claims can never pass for believable policy, the replication-level item validity check removes the need for the policy management logic to recover from attacks that try to install old or corrupt policy.

4.2 Policy evolution

In the previous section, we established a set of policy claims over a collection in an *a priori* fashion. In general, this will not be possible. Instead, 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.

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 3. 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. All of the existing policy items at the parent are then injected into the child.

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 [10]); 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 existing PKIs, payment for service, pre-shared passwords, and knowledge of personal information). Regardless of the details, bootstrapping can, but doesn't have to, rely on security by physical proximity or simple network authentication. A PKI can be used, but is not required.

Revocation. Revocation claims are handled in the same fashion as other policy updates. 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 the affected local items. Other replicas will eventually learn of the revocation and do the same.

4.3 Convergence

Because of policy propagation delays, access control failures may arise due to incomplete policy replication. If a failure occurs on a read or sync operation, this doesn't pose a problem. Synchronization is periodic. Eventually propagation will complete and subsequent operations will succeed. However, what happens if an update fails? The failed update cannot be incorporated as valid state since most replication systems maintain only the most recent version at every replica, and that should mean the most recent valid version. But, the failed update may later succeed when the destination receives updated policy. Consequently, failed update operations must be retried as long as they remain valid at the sender. These operations will either eventually succeed, or the sender

will receive new policy that invalidates the pending updates locally. Either way, eventual consistency is maintained.

Invalidation of existing items in the face of new policy represents the other side of the coin. The race condition depicted in Figure 2 can produce an spurious access control failure when the policy update P' newly allows data update U', but can also require an invalidation of U' when P' contains a revocation. We must be prepared to perform such invalidations at any time and on any data that is in our store.

Our rationale for why these steps give eventual consistency is as follows. We assume that eventual consistency would be achieved in the absence of access control. Each replica issues its own policy items, and all policy is readable by all replicas. Due to the monotonicity of the logic and the single root of trust, policy updates cannot conflict. Thus, if the system would converge in the absence of access control, it follows that system policy converges in the presence of access control as well. In the security-enhanced system, if replica state is consistently re-evaluated when new policy arrives (e.g., by requiring retransmission or revocation), then that state will also converge.

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 construct access control topologies that prevent propagation of updates. Reliable update propagation also depends on the correct functioning of replicas in the propagation path. Malfunctioning or compromised replicas cannot forge the signature of other replicas and, if protocols require item version identifiers to 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. This is why we introduced access controls on the *sync* operation to specify whether a replica should be trusted to act as the source of a synchronization operation. Correct update propagation depends on the correct operation of at least one trusted, transitive sync path between any pair of replicas that share updates. Our design for compromise recovery [8] can be used to ensure that damage caused by a malfunctioning replica can be recovered if detected, so the temporary absence of a trusted path can be tolerated.

4.4 Discussion

The collection manager represents a single root of authority over a collection. Without a single root, replicas can engage in policy wars where multiple authorities issue conflicting policy statements. The lazy propagation of policy that our system assumes exacerbates such conflicts, and so we opt to avoid them.

In our prototype, we make statements directly about replica keys. We chose to do so to reduce our dependence on external authentication services, 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. In addition, most existing authorization systems are not capable of expressing the delegation relationships we need to allow policy evolution. Furthermore, we benefit from using a declarative policy specification

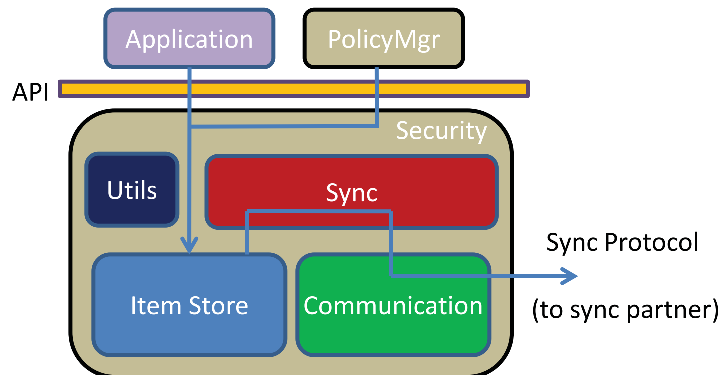


Fig. 4. Cimbiosys components

that boils down to logic that can be evaluated outside of the context of the system implementation. Not only can we reason about access control decisions in abstract, but we can potentially construct and distribute access control proofs ahead of time.

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.

In our system, items are only valid for as long as policy permits: a change in policy can result in the invalidation of existing items. In protocols that attempt to replicate state rather than an log of operations, this is the only plausible option 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. Our policy statements are self-contained, enumerable, and independent from the details of access-control enforcement. We can quickly determine the effect of a revocation on a replica's state and thus perform invalidation efficiently.

5 Implementation

As discussed in Section 2, access control policy is enforced by monitoring replication protocol operations. We implement our system within the Cimbiosys [7] replication framework. A high-level component diagram of Cimbiosys is given in Figure 4. The Cimbiosys API associates items and policy and provides clients 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* component depicted in Figure 4. It monitors both the item store and the replication protocol and consists 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 replicas. 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 released in binary form 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.

5.1 Performance

Other work has evaluated the performance of Cimbiosys [7]. Here we attempt to justify that adding a logical policy checker does not add excessive overhead. 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 1 as elaborated in Section 4.1. The policy contains 23 claims and we give results for queries that require different sets of claims. For each query, Table 1 shows 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.

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
Spouse-Mobile can *read contacts*	20	4	56	9.4

Table 1. Prover performance

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. 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. An optimized 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 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.

Finally, 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, 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.

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. Nevertheless, we believe that in any tractable access control system, the overall number of policies must be small because otherwise the system will become unmanageable. 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.

6 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 and do not attempt completeness.

Grapevine [4] and Bayou [11] are examples of distributed systems with eventually consistent replication. Microsoft's Active Directory [5] is a commercial example of such a system in widespread deployment. These systems enforce access controls on

clients, however all replicas are equally trusted. In a similar vein, Samarati [12] 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 [13], FARSITE [14], Taos [15], 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 [16] and Ivy [17] 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 [18] 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 [19]. 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 [15, 20] used logic to rationalize system security design. Later work by Appel and Felten [21] demonstrated that a logical proof checker can be employed to automate the process of validating encoded credentials. PolicyMaker [22] showed the value of expressing system policy, not just authentication credentials, in a precise fashion. SD3 [23] and Binder [24] joined these threads by expressing policy in a logic-based language that can be evaluated. Subsequent systems [6, 25–27], have extended the performance and expressibility inherent in logical policy frameworks.

Like the Grey system [28] 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.

7 Conclusion

Replicated systems that offer only weak consistency are valuable for solving a number of problems. 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.

References

1. Schneider, F.: Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Computing Surveys **22**(299) (1990)
2. Microsoft Corporation: Office Groove 2007 Developer Portal. `http://msdn.microsoft.com/en-us/office/bb308957.aspx`
3. Microsoft Corporation: Microsoft Sync Framework. `http://code.msdn.microsoft.com/sync`
4. Birrell, A.D., Levin, R., Schroeder, M., Needham, R.: Grapevine: an exercise in distributed computing. Communications of the ACM **25**(4) (1982) 260–274
5. Microsoft Corporation: About Active Directory Domain Services. `http://msdn.microsoft.com/en-us/library/aa772142(VS.85).aspx`
6. Becker, M.Y., Fournet, C., Gordon, A.D.: Design and semantics of a decentralized authorization language. In: Proceedings of 20th IEEE Computer Security Foundations Symposium. (2007) 3–16
7. Ramasubramanian, V., Rodeheffer, T., Terry, D., Walraed-Sullivan, M., Wobber, T., Marshall, C., Vahdat, A.: Cimbiosys: A platform for content-based partial replication. In: Proceedings of 6th USENIX Symposium on Networked Systems Design and Implementation. (2009)
8. Mahajan, P., Kotla, R., Marshall, C., Ramasubramanian, V., Rodeheffer, T., Terry, D., Wobber, T.: Effective and Efficient Compromise Recovery for Weakly Consistent Replication. In: Proceedings of the Fourth EuroSys Conference. (2009) 131–144
9. Revesz, P.Z.: Constraint databases: a survey. Semantics and Databases **1358** (1995) 209–246
10. Adams, C., Farrell, S., Kause, T., Mononen, T.: Internet X.509 Public Key Infrastructure Certificate Management Protocol (CMP). `http://tools.ietf.org/html/rfc4210`
11. Terry, D.B., Theimer, M.M., Petersen, K., Demers, A.J., Spreitzer, M.J., Hauser, C.H.: Managing update conflicts in Bayou, a weakly connected replicated storage system. In: Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles. (1995) 172–182
12. Samarati, P., Ammann, P., Jajodia, S.: Maintaining replicated authorizations in distributed database systems. Data and Knowledge Engineering **1**(18) (1996) 55–84
13. Satyanarayanan, M.: Integrating security in a large distributed system. ACM Transactions on Computer Systems **7**(3) (1989) 247–280

14. Adya, A., Bolosky, W.J., Castro, M., Cermak, G., Chaiken, R., Douceur, J.R., Howell, J., Lorch, J.R., Theimer, M., Wattenhofer, R.P.: FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In: Proceeding of the 5th Symposium on Operating Systems Design and Implementation. (2002) 1–14
15. Wobber, E., Abadi, M., Burrows, M., Lampson, B.: Authentication in the Taos operating system. ACM Transactions on Computer Systems **12**(1) (1994) 3–32
16. Dabek, F., Kaashoek, M.F., Karger, D., Morris, R., Stoica, I.: Wide-area cooperative storage with CFS. In: Proceedings of the 18th ACM Symposium on Operating Systems Principles. (2001) 202–215
17. Muthitacharoen, A., Morris, R., Gil, T.M., Chen, B.: Ivy: A read/write peer-to-peer file system. In: Proceedings of the 5th Symposium on Operating Systems Design and Implementation. (2002) 31–44
18. Mazières, D., Kaminsky, M., Kaashoek, M.F., Witchel, E.: Separating key management from file system security. In: Proceedings of 17th ACM Symposium on Operating Systems Principles. (1999) 124–139
19. Ford, B., Strauss, J., Lesniewski-Laas, C., Rhea, S., Kaashoek, F., Morris, R.: Persistent personal names for globally connected mobile devices. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation. (2006) 233–248
20. Lampson, B., Abadi, M., Burrows, M., Wobber, E.: Authentication in distributed systems: Theory and practice. ACM Transactions on Computer Systems **10**(4) (1992) 265–310
21. Appel, A.W., Felten, E.W.: Proof-carrying authentication. In: Proceedings of the 6th Conference on Computer and Communications Security. (1999) 52–62
22. Blaze, M., Feigenbaum, J., Lacy, J.: Decentralized trust management. In: Proceedings of the 1996 IEEE Symposium on Security and Privacy. (1996) 164–173
23. Jim, T.: SD3: a Trust Management System with Certificate Evaluation. In: Proceedings of IEEE Symposium on Security and Privacy. (2001) 106–115
24. DeTreville, J.: Binder, a logic-based security language. In: Proceedings of IEEE Symposium on Security and Privacy. (2002) 105–113
25. Bauer, L., Garriss, S., Reiter, M.K.: Distributed proving in access-control systems. In: Proceedings of 2005 IEEE Symposium on Security and Privacy. (2005) 81–95
26. Bauer, L., Schneider, M.A., , Felten, E.W.: A general and flexible access-control system for the web. In: Proceedings of the 11th USENIX Security Symposium. (2002)
27. Lesniewski-laas, C., Ford, B., Strauss, J., Morris, R., Kaashoek, M.F.: Alpaca: extensible authorization for distributed services. In: Proceedings of the 14th ACM Conference on Computer and Communications Security. (2007) 432–444
28. Bauer, L., Garriss, S., McCune, J.M., Reiter, M.K., Rouse, J., Rutenbar, P.: Device-enabled authorization in the Grey system. In: Proceedings of the 8th Information Security Conference, Springer Verlag LNCS 3650 (2005) 431–445