Cooperative Policy Control for Peer-to-Peer Data Distribution

Eric Anderson
University of Colorado
eric.anderson@cs.colorado.edu

Jun Li
University of Oregon
lijun@cs.uoregon.edu

ABSTRACT

Many network applications (such as swarming downloads, peer-to-peer video streaming and file sharing) are made possible by using large groups of peers to distribute and process data. Securing data in such a system requires not just data originators, but also those "distributors," to enforce access control, verify integrity, or make other *content-specific* security decisions for the replicated or adapted data.

In this paper, we introduce the concepts of *cooperative policy enforcement* and *request type checking*, and propose an implementation framework Q which uses these approaches to secure data in peer-to-peer systems.

The Q framework associates every data object with relocatable *policy descriptors* which distributors can use to determine whether a request for that object should be granted and whether a data transfer meets a request. With minimal changes to the application or the framework, Q can define and enforce arbitrarily sophisticated policies across a wide range of applications. Policies can be written to work across applications, or to include application-specific criteria and behavior.

We will also discuss integrating Q with several peer-to-peer applications, including Gnutella, distributed hash tables such as CAN and Chord, peer-to-peer video streaming, HTTP swarming and application-level routing.

Authors' Note

This paper was written for the *Symposium on Access Control Models and Technologies*, 2005. Neither of the authors have had the time to update or re-submit it, so we are releasing the manuscript as is.

1. INTRODUCTION

Peer-to-peer systems for data distribution and adaptation offer significant performance, scalability and reliability ad-

vantages over point-to-point transfers from a single server to all clients. However, in order to extend these benefits to the distribution of sensitive or commercially valuable data, it is necessary to provide support for access control, integrity verification, and other security assurances. Peer-to-peer distribution systems impose different requirements on the design of a security mechanism than centralized systems. These systems often rely on nodes, which we call "distributors" or "server peers," to replicate, sometimes adapt, and forward data. Any object may be handled by many distributors, and any distributor may handle objects from many sources. Access control is made more difficult because distributors may have no prior knowledge about the objects they hold, and thus have little basis for deciding which recipients are eligible. Because data can be adapted between the originator and recipient, it becomes hard for a recipient to verify that the content received is what the originator intended. End-to-end integrity checks do not allow for legitimate data adaptation by nodes in the middle, and per-hop checks do not detect illegitimate changes by those nodes.

In a peer-to-peer network, every data object may have different security requirements. Nodes serving or adapting a data object may have no knowledge of those requirements, unless they are explicitly represented and distributed with the data. Recipients may not trust the nodes from which they request objects, and may not know how to verify data authenticity when both benign and malicious adaptation are possible.

The Q security framework allows creators to assign objects access control and integrity policies that distributors can use to determine whether a request should be permitted and recipients can use to determine whether it was properly carried out. Access control policies can also require data confidentiality by specifying that only transfers using suitable encryption are permissible. Note that there are aspects of peer-to-peer network security that Q does not address (such as authenticating nodes, managing peer reputations or preventing protocol-level attacks), and Q is compatible with proposed solutions to the other problems. Though optimized for peer-to-peer networks, Q could also be applied to any scenario in which entities are responsible for managing large sets of discrete objects with differing security requirements.

This paper makes two primary contributions: a novel model of cooperative policy enforcement through relocatable per-

object permission descriptors, and a new highly-expressive approach to proof-carrying authorization, *request type-checking*. We present these approaches and a framework for using them to making access control and integrity decisions, and discuss ways of securing existing applications using this framework.

2. COOPERATIVE DISTRIBUTED POLICY ENFORCEMENT

2.1 Overview

The goal of our system is to provide for the consistent enforcement of policies, chosen by an object's originator, for all copies of the object which occur in a peer-to-peer network.

This may be regarded as an adaptation of an Originator Control (ORCON) meta-policy[17, 21] for use in an open, potentially untrusted environment.

We address two specific aspects of data security policy: Access control and integrity verification. Our approach to access control is as follows: Every object has an associated access control policy, which is encoded in a machine-readable metalanguage, discussed in section 3. Whenever a peer which has an object receives a request for the object, it consults the associated policy to determine whether the request should be granted. Whenever an object is transferred within the system, its policy must be transfered with it, to enable the receiving peer to make access control decisions. Similarly, every object has an integrity policy specifies acceptance criteria for the recipient to use.

After a server peer answers a request, the requester checks what is has received against the integrity policy. The requirement that the requester know the integrity policy in advance is not as stringent as it might at first seem: The policy may be obtained from the content originator or a trusted third party, or from an untrusted party (such as the server peer) as long as the client can check its authenticity. Verifying the integrity of the integrity policy is not a chicken-and-egg problem: Unlike data objects, which may potentially be legitimately altered in transit, policies are immutable, which makes a fixed verification strategy such as hash value or digital signature checking appropriate. The client must still obtain the policy's hash value or the creators public key from a trusted source.

2.2 System Model / Assumptions

1. An object's policies are determined by its creator prior to its insertion into the system, and do not change for the life of the object.

2. Any peer which is *authorized* by an object's creator to receive an object is also *trusted* to apply that object's access control policy correctly.

3. Any peer which holds an object may grant requests for that object, if the request is in accordance with the object's access control policy.

4. For any given object, there is an *access control policy* which specifies the criteria for determining which requests for the object should be granted.

5. For any given object, there is an *integrity policy* which specifies the criteria for determining whether a request has been properly carried out.

Assumption 2 weakens the traditional model of originator control[21, 17] (ORCON), in which authorized recipients of an object must be prevented from disclosing it to unauthorized parties. The relaxation is appropriate for three reasons:

1. Without a general usage control[24] (UCON) system, even the best mandatory access control *within* the distribution system cannot prevent information leakage through other channels. The design of such a system is orthogonal to our work.

2. At present, the best approaches to enforcing ORCON in a distributed system require all parties to use mutually-trusted tamper-resistant hardware or software[24, 23, 3], which a general purpose system for use on heterogeneous hosts cannot assume.

3. Any individual security policy may limit distribution to only those hosts using such a system. Thus, this requirement may be omitted from the design of our model without loss of generality.

3. REQUEST TYPE-CHECKING

Request type-checking is a mechanism for checking a request against a given policy. Requests are represented as statements in an application-defined formal language, and policies are represented as type systems. A request is accepted if it is a *correct* statement and it is possible to derive a desired type for the statement under the type system given in the policy. Note that while we present this in the context of access control, the process for integrity checking works equivalently, by trying to verify the distributor's claim to have done something correctly.

We briefly introduce logic- and language-based access control in general, discuss its limitations in this context, and then present request type-checking in greater detail. In section 4 we outline a design for a security framework based on this approach.

3.1 Background

Logics and logic programming languages have a number of properties which make them good candidates for use in policy descriptors. By compactly presenting decision-making rules, they can encode more complex policies than can be represented by access control lists [1] or fixed permission structures, and can scale to systems in which the set of principles is very large or unknowable. They also provide a clear separation of policy and mechanism, which simplifies reasoning about both.

There are several properties of peer-to-peer networks which make existing policy logics and access control systems based on them unsuitable. Most notably, such networks can actively transform data as well as replicating it, so an access control logic must be able to prove or disprove statements of the form "Process W may perform operation X on data Y

for requester Z," where *operation X may be an arbitrarily complex request* such as "encrypt a high-resolution, desaturated version of image Y with public key K and send it" rather than one of a finite set of operations such as {read, write, delete}.

Existing logics and languages such as D1LP[18] and Binder[12] can support multiple operations with distinct requirements, but every operation must be specifically enumerated, so there can only be a finite set. The set of operations may be encoded in the choice of predicates or of literal terms.

```
may-read(User, Object) :-
     foo.
may-write(User, Object) :-
     bar.
```

Figure 1: One predicate per operation

```
may(User,Object,read) :-
    foo.
may(User,Object,write) :-
    bar.
```

Figure 2: One literal per operation

In either case, every supported operation must be explicitly included. These are give in pseudo-Prolog syntax, so a single predicate with a disjunction is represented as two statements in figure 2. A request can be indirectly described by a *set* of assertions which are automatically added to the compliance-checker's environment. Such assertions can be either logic statements or simple key-value bindings, as in KeyNote's action environment. [27, 18]. For n such constructs appearing in the policy, at most 2^n distinct combinations are possible. There is no natural means for specifying a policy when the set of operations is potentially infinite.

Also, the range of checks and operations a policy writer might like an application to make available is large and growing. A person concerned about media files being leaked to the public might want a policy which says "Transfers to requesters who are only moderately trusted must include the addition of a watermark with the requester's identity" so that the lineage of the leaked copy could be traced. Existing security logics define a specific mapping between certain observable facts and statements in the logic (e.g. a signed certificate could map to `Bob says IsStudent(Alice)` in Delegation Logic), but there is no way to add mappings for new types of facts or operations.

3.2 Access Control through Type-checking

We propose to frame the problem of verifying an access request as deriving a type of *permitted* for the statement of the request, rather than proving the proposition that the request is permissible.

If the request is atomic, the two formulations are clearly equivalent. That is, the proof that $permitted(R)$ using a given logic l from the set of proposition P which represent a policy $P \vdash_l permitted(R)$ is isomorphic to the derivation

of $R : permitted$ using a type system t from the set of typing judgments p which represent the same policy $P \vdash_t R : permitted$. We propose the following refinements:

First, policies can be represented as *type systems* in a meta-language [25] rather than judgments in a fixed system. A policy can be written to contain the rules of an existing logic [28, 8, 7, 18] with properties the author desires, and propositions in that logic can be rewritten as axioms of the type system. That is, with a slight abuse of notation $P \vdash_l permitted(R)$ can be represented as $P \cup l \vdash R : permitted$. This effectively gives the policy writer a choice of existing or future policy logics, within the limits of what is representable in the meta-language.

Second, requests need not be atomic, nor have a fixed structure. Earlier, we raised the problem of *complex requests*. Consider an application which supports requests such as "encrypt a high-resolution, desaturated version of image Y with public key K and send it" instead of a fixed set of operations. If requests are given atomic representation in the access control process, all possible requests might need to be enumerated in an object's policies, which could be inefficient or impossible. We propose that requests be specified as statements in a formal language, with application-specific interpretation. This permits a type system to be written which uses syntax-directed reasoning to evaluate a request, and allows arbitrarily many distinguishable requests with even a small number of constructs appearing in a policy. A minimal example request, shown as an abstract syntax tree, is given in figure 3.
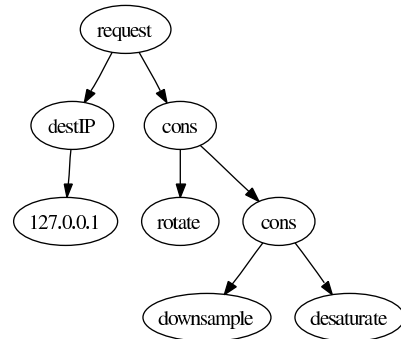

Figure 3: Request AST

An example policy fragment is given in figure 4. The notation is described in appendix A. This specifies that an *any_quality* action may be granted for trusted addresses, that a *low_quality* action may be granted for any address, and that a list of operations containing a `downsample` operation is *low_quality*. Thus, it will be possible to derive a type of *permitted* for the request whose abstract syntax tree is shown in figure 3 if 127.0.0.1 is typed as an *ipaddr* or *trusted_ipaddr*.

3.3 Credentials

Thus far, we have not considered the interpretation of the request, only its typing. All that is strictly required is that the

LOW VIDEO IMAGE ALLOWED TO ANY IP
$$\frac{\Gamma \vdash recipient : ipaddr \qquad \Gamma \vdash action : low\_quality}{\Gamma \vdash (\texttt{request } recipient\ action) : permitted}$$

ANY QUALITY IMAGE ALLOWED TO TRUSTED IP
$$\frac{\Gamma \vdash recipient : trusted\_ipaddr \qquad \Gamma \vdash action : any\_quality}{\Gamma \vdash (\texttt{request } recipient\ action) : permitted}$$

JOINING TWO ANY_QUALITY OPERATIONS GIVES ANOTHER
$$\frac{\Gamma \vdash head : any\_quality \qquad \Gamma \vdash tail : any\_quality}{\Gamma \vdash (\texttt{cons head tail}) : any\_quality}$$

JOINING A LOW_Q. AND ANY_Q. GIVES LOW_Q
$$\frac{\Gamma \vdash head : low\_quality \qquad \Gamma \vdash tail : any\_quality}{\Gamma \vdash (\texttt{cons head tail}) : low\_quality}$$

LIKEWISE THE OPPOSITE ORDER
$$\frac{\Gamma \vdash head : any\_quality \qquad \Gamma \vdash tail : low\_quality}{\Gamma \vdash (\texttt{cons head tail}) : low\_quality}$$

$$\frac{}{\Gamma \vdash (\texttt{rotate}) : any\_quality}$$

$$\frac{}{\Gamma \vdash (\texttt{compress}) : any\_quality}$$

$$\frac{}{\Gamma \vdash (\texttt{downsample}) : any\_quality}$$

$$\frac{}{\Gamma \vdash (\texttt{downsample}) : low\_quality}$$

Figure 4: Example partial policy

request means something to the application, and that policies are written with this same interpretation. That said, there are two main kinds of statements which an application needs to support to enable interesting security policies: Descriptions of the service requested, and assertions of the requester's credentials. As an example of the latter, consider a cryptographically signed certificate vouching for some fact about the requester. Such a credential might be represented as in figure 5. In that request, the "downsample" operation has been replaced with a compression, and a certificate is provided to establish the trustworthiness of the destination.

For a credential to be meaningful, it must be relevant to assigning a type to the request, and must also be true. The former is purely a formal property of the syntax tree and the policy's typing rules, and is thus not specific to credentials. However, credentials tend to be claims of something, which motivates the use of parametric types to describe them. The LF framework [26] allows a signature (policy) to specify not only specific types, but also rules for rules for new types:
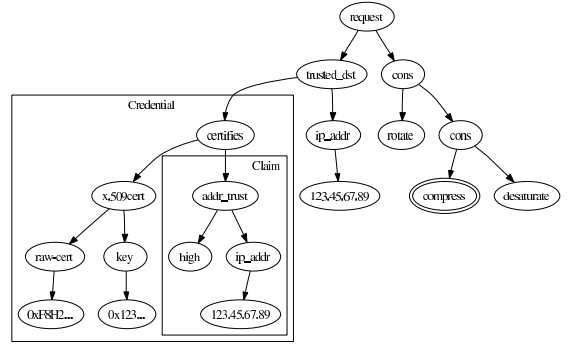
∀ INTRODUCTION
$$\frac{X \notin \Delta \quad \Delta \cup \{X\} \vdash t : type}{\Delta \vdash \forall X(t) : type}$$

Figure 5: Credential AST

∀ ELIMINATION
$$\frac{\Delta \vdash \forall X(t) : type \quad \Delta \vdash t' : type}{\Delta \vdash \{t'/X\}t : type}$$

(PARAMETRIC) CLAIM TYPES
$$\frac{\Delta \vdash t : type}{\Delta \vdash t\ valid\_claim : type}$$

Using these polymorphic types, one can define generalized typing rules for the role of certificates: Essentially "a certificate signed by a trusted key promotes any type 'foo' to 'valid claim of foo'" and "the 'certifies' construct applies a certificate to a specific type."

$$\frac{\Gamma \vdash_\Delta raw : byte\_array \quad \Gamma \vdash_\Delta key : trusted\_key}{\Gamma \vdash_\Delta (\texttt{x.509cert } raw\ key) : \forall X(X \to X\ valid\_claim)}$$

$$\frac{\Gamma \vdash_\Delta cert : t \to t\ valid\_claim \quad \Gamma \vdash_\Delta claim : t}{\Gamma \vdash_\Delta (\texttt{certifies } cert\ claim) : t\ valid\_claim}$$

Such a claim type can be used in rules such as "given a trusted certificate to the effect that a specific address is trusted, a transfer to that address is a transfer to a trusted destination."

$$\frac{\Gamma \vdash_\Delta cert : X\ good\_addr\ valid\_claim \quad \Gamma \vdash_\Delta ip\_addr : X}{\Gamma \vdash_\Delta (\texttt{trusted\_dst } cert\ ip\_addr) : trusted\_ipaddr}$$

Determining the truth of a credential requires assigning an interpretation to the request and checking this interpretation against actual fact. That is, in figure 5, the byte array under raw_key must actually be an x.509 certificate signed by the given public key, and the data signed must correspond to the claim given.

3.4 Request Generation and Verification

As a request defines not just the requester's desired conclusion ("this operation is permitted") but also the justification, it cannot be generated without knowledge of the policy which is to be satisfied. The generation of a satisfying request is essentially a standard proof-search process, but there are a several significant restrictions:

1. The resulting AST must be semantically meaningful.

2. The meaning must correspond to the requester's desired operation.

3. The meaning must be factually correct.

Restrictions 1 and 2 can be additional formal constraints for the theorem prover, but restriction 3 requires application-specific extension. The application needs to know what credentials it can supply, which can include context-specific facts such as "my IP address is X." The process of generating such a request can be arbitrarily time consuming, and even undecidable, depending on the choice of logic, but empirical studies suggest that it can be relatively efficient.[7]

The process of generating a request and type derivation can be separated from the process of verification, so that the requesting client performs the computationally intensive tasks. If the client sends the server peer its request and the associated type derivation, the server peer need only verify the derivation and check the factual statements in the request.

3.5 Integrity Checking

Integrity statements (the analog of requests) can be generated and verified in the same way. The semantic content of the credentials involved is more likely to pertain to the data itself than the identity of the server peer, but an integrity policy based on the trustworthiness of the distributor is plausible as well.

4. THE Q SECURITY FRAMEWORK

The goal of the Q security framework is to (1) enable an object creator to express her or his security requirements for that object as a set of policies and (2) enable all nodes that move, copy or adapt the object within a network to enforce that policy. An object creator can be as selective as she may desire in which nodes are allowed to have an object, but it is beyond the scope of this work to compel nodes to faithfully enforce policies.

Q defines a standardized but extensible language to describe requested or performed actions on a data object, and policies are expressed in the form of type systems for that language. Any authorization or integrity checking can be modeled as an attempt to derive the intended type for policy statements. This allows for reasoning about an infinite set of potential requests using a finite set of judgments and induction on the structure of the language.

In this section, we first describe how a general protocol can be defined to enforce policies, and then discuss the request language and processing mechanism. We address performance and scalability issues at the end of this section.

4.1 Protocol

A protocol that should be compatible with a wide range of applications is shown in Figure 6. Before a data object starts to travel from node to node, its creator will attach a "policy descriptor" with the object, which specifies the policies to be enforced on this object. Every node, upon the receipt of a data object, will receive the object's policy descriptor in addition to the data itself.

In the following we focus on the interchange between the peer requesting an object (referred to as a client peer) and the one providing it (referred to as a server peer). The security framework is protocol-neutral with regard to how peers are selected, the nature of the request and data, and over what mechanism requests are transmitted.
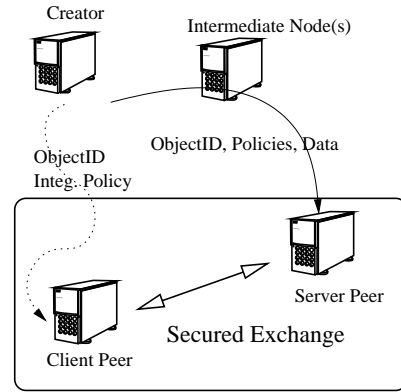

Figure 6: General Context

Before the interchange begins, it is assumed that the server peer has the object and knows the object's access policy and integrity policy, and that the client peer knows the object ID and integrity policy. Both peers must have stored or be able to generate any credentials or facts required to create and validate claims.

Client peers are responsible for generating a proof that their request complies with an object's access control policy. The proof generation process is computationally intensive and the execution time is unbounded and policy-dependent. Because client peers are likely to be more numerous than server peers, have more discretion over opening connections, and have greater incentives and trust in the object's originator, it is advantageous to assign the bulk of proof-generation to those systems.
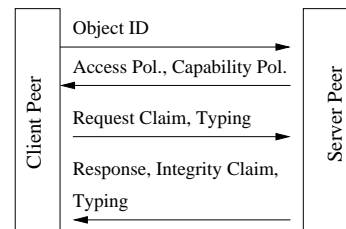

Figure 7: Request Protocol

As shown in figure 7, the client peer first requests the access policy from the server peer, and then searches for a claim expression which uses only known true facts, describes the desired transfer and satisfies both peers' capability policies and the access policy. This claim, along with its typing derivations, is sent to the server peer. The server verifies that the claim is factually correct, and then checks the typing derivations. If both are OK, the transfer is authorized. Following a symmetrical process, the server generates a claim that it properly fulfilled the request, and sends that claim along with a typing derivation to the client for verification. As the acceptance policy for any given object is fixed, and a small number of objects tend to account for the bulk of requests in many systems [9] it is likely that the server peer can cache proofs for common requests.

4.2 Claim Language

4.2.1 Core Language

The language of access requests and integrity assertions has been described as application-specific. Within any given application, requests and facts must have well-defined shared meanings to be of any value, so the claim language has fixed semantics. The elements of the claim language constrain what information a policy can reason about, but not what sort of reasoning process is used.

Further, there are a number of constructs which seem likely to be useful in a wide range of applications, such as those identifying hosts, communication channels, times, data transformations and cryptographic operations. Defining a core language which provides these makes it possible to write cross-application policies and increases the potential for code reuse.

4.2.2 Extensibility

Any application may freely extend the given core language with its own domain-specific primitives without having to modify the security framework or introducing incompatibility. Claim generation is a purely formal process: it can use any application-supplied construct allowed by the relevant policies without needing to know anything about its interpretation.

To extend the claim language with a given construct, an application must add types for it to the language definition, and must be able to assign truth or falsity to it when fact-checking claims. For the extension to be useful, that truth assignment should have some consistent meaning across those applications which implement the extension and any policies which refer to it. The image manipulation operations mentioned earlier are examples of application-specific constructs, as is video layer selection in section 5.3.

It is worth noting that security-related functions which are not explicitly a part of Q, such as data encryption, content watermarking, or steganography can be supported by extensions (if needed), and their use can be a required by Q policies.

A construct can only appear in a (valid) claim if both the generating and receiving peer include it in their language definitions, so it will not be used unless all parties support it.

4.2.3 Contraction

In addition to extending the language, an application or individual peer can contract it. If there is any function which a peer is unable or unwilling to support, the peer can prevent its use by removing the associated construct from its language definition. This will cause the request generation process not to consider requests using that construct.

4.3 Policy Processing

Policies in Q are systems that assign *types* to claims. This offers two key advantages over representing them as logic programs as previous systems have done: First, the policy represents the logic itself in addition to a set of propositions in the logic. This gives the policy writer the flexibility to define a system that is equivalent to any of the existing security logics, or any future logic, so long as the language to encode policies is sufficiently expressive. Second, this allows syntax-directed reasoning about structured requests, rather than regarding requests as atomic.

Q represents policies as signatures in the dependently-typed lambda calculus λ^Π, which is the type system underlying the LF logical framework[16]. LF is one of the most general systems with well-understood characteristics, and is sufficiently expressive that all of the security logics discussed can tractably be represented in it. LF, specifically the Twelf [26] implementation, is used for representing logics in Bauer's proof-carrying authorization [7] and Appel's and Necula's proof-carrying code [5, 20]. Several other security languages were either designed, as in the case of Delegation Logic [18] or subsequently discovered, as in the case of KeyNote and SPKI/SDSI[19] to be reducible to simple logics which are representable. A Q policy can thus be written to reason in at least the manners given in those systems.

There are extensions to LF's type system such as Linear LF, which [10] allows for use of linear contexts, and $\lambda P_{\le}$ which provides for fuller subtyping. [10, 6] An especially nice property of linear contexts is that they provide a flexible way of implementing credential thresholds. Linear LF would be a more flexible meta-language for policies than LF, but there has been less work on automated reasoning systems for it.

4.4 Performance and Scalability

This security framework does not impact the asymptotic complexity of any peer-to-peer system to which it is added. A constant amount of state is required per object per node to hold the applicable policies. Communication is only required when an object is requested, and only between the hosts involved in that request. The protocol given in section 4.1 requires two round-trip times, plus some computation time. At least one round-trip can always be combined with the actual data transfer, so only one is added. The message sizes are increased by the sizes of the policies, claims and typing derivations. Those sizes depend primarily upon the policy, which is a user-controlled variable. Higher-order logics are potentially undecidable, so the computation time, claim size and type derivation size are theoretically unbounded, but an application can try to avoid excessive time or message size.

5. SECURING PEER-TO-PEER NETWORKS WITH Q

5.1 Unstructured Overlays

In networks such as Gnutella [15] any object could be, but need not be, located at any node. This is the simplest scenario for applying Q: Any node which has an object just responds to searches for it. Once a transfer is requested, authorization and integrity are verified as given in section 4.1.

As an optional enhancement, the nodes having an object could attempt to evaluate whether the searcher could be authorized to receive the file, and only respond if so. A server peer can use automated theorem proving techniques to determine whether the known data (such as the requester's IP address) preclude a compliant request. If not, there is some possible request from the host which would satisfy the policy. Having server peers do so increases their computational workload but reduces network traffic by suppressing useless query replies. This also reduces the likelihood that clients will issue requests which will be refused. As a further enhancement, query responses could include the policy for the requested object. This would increase the query response message sizes, but entirely prevent the waste of clients issuing requests which will be refused.

5.2 Distributed Hash Tables

DHTs rely on fixing the location(s) within the overlay where given objects are stored. If those nodes are not permitted to hold the objects in question, then the scheme is broken. This is not a limitation of Q specifically, but rather an inherent negative interaction between access control and structured overlays.

This problem can be largely alleviated if the DHT is used to store pointers to objects rather than the objects themselves, and if the pointers are presumed to have no access control restrictions.

5.3 Peer-to-Peer video streaming

It is fairly easy to imagine that a layered video might have different security requirements for different layers, so as to provide for different levels of service to paying and non-paying viewers Especially if there are many levels of distinction, building a single distribution mesh with a single (differentiated) policy is more efficient than building separate meshes for every possible level.

Similarly, it is possible that a provider might want to allow any receiver to request excerpts of up to a certain length, or allow on-the-fly conversions between some formats but not others (to retain copy or use restriction features, for example).

Neither video layering nor format conversion are likely to appear in a generic claim language; to allow for security policies concerning these, a domain-specific language extension is required.

5.4 HTTP Swarming

Swarming [30] has been proposed as a mechanism for allowing hosts with modest bandwidth to serve large audiences. New peers first contact the centralized host and then discover other peers with the desired content. Since clients are already contacting the host, a simple policy might be to require tokens from the server before authorizing a transfer.

5.5 Application-level routing

Overlay networks can be used to route data through intermediate hosts rather than directly from the source to the destination at the IP layer. This is used to enhance reliability and performance in systems such as RON[4], for security as in proxy firewalls, or to mask the identity of either participant.

For such applications, a security policy might wish to restrict the set of hosts through which data may be routed, require end-to-end encryption for transfers through semi-trusted hosts, or otherwise place limits not only on the endpoints of a transfer but also on the path.

Such policies are made possible by defining the claim language to allow a single request to include multiple sub-requests such as a sequence of transfers.

6. RELATED WORK

A number of policy languages and access control mechanisms have been defined for client-server data transfer [8, 27, 28, 2]. These systems represent policies as sets of statements in some specified proof system, and access is permitted if the conclusion that a request should be granted can be proven using those statements as premises. Early systems such as KeyNote [27] and SPKI/SDSI [28] were not explicitly framed as logics, but are equivalent[19]. Many of these logics incorporate the ideas of trust management or distributed authentication. Such systems assume a *centralized authorizer* for any given request, but that authorizer incorporates claims by other parties (represented by cryptographically signed assertions) into its own set of premises, subject to its level of trust in those parties. This approach has been formalized and extended by Delegation Logic [18] and AF Logic [7], both of which provide generalized logics for reasoning about security relationships.

Proof-Carrying Authorization (PCA) optimizes this notion by placing the onus of proving that a request is permissible on the client, and give the server the computationally easier and less error-prone task of proof verification.

Distributed authentication [7, 13] avoids the need for either a client or server to be in possession of all the relevant information and credentials when a request is made, by providing for automatic fact-gathering. In Bauer's system, there are "fact servers" which recognized as authoritative sources for specific propositions, and are available to interact with the client. This approach adds requirement that fact servers be knowable and reachable, which may not be the case in a peer-to-peer context.

Usage Control (UCON), of which "Digital Rights Management" [14] (DRM) is a subset, is related to our work in that both involve the representation and application of per-object policies, but the focus of research is different: We are trying to answer the questions *how should policies be represented?* and *how should the acceptability of a proposed action be de-*

termined? while UCON and DRM are more concerned with *how should unacceptable actions be prevented?*[22].

Sadhu and Park classify UCON architectures in terms of where the reference monitor, which provides a point of control, is located[29]. None of the architectures given are sufficiently general to wholly support a system such as we describe: Server-only control (SRM) precludes releasing cleartext to clients at any time, which limits its utility to applications in which a central server is constantly available. Client-side control (CRM) requires tamper-resistant hardware for meaningful security, which limits the range of users and devices for which it will be acceptable.

7. CONCLUSIONS

Existing authorization systems, including distributed authentication, have relied on common administrative control to support objects being distributed from a single host or a fixed set of mirrors, but do not support open set of independent systems. Existing distribution mechanisms, such as swarming , Gnutella and Freenet allow a dynamic association between hosts and objects, but provide no per-object policy control. [15, 30, 11]

In this paper, we have presented two complementary security paradigms, cooperative policy enforcement and request type-checking, and outlined a design for a system which implements them. Cooperative policy enforcement provides for per-object access control and content-appropriate integrity checking, even when the group of hosts distributing an object is large and dynamic. A host can acquire objects dynamically and begin distributing them under the correct policies without administrative intervention.

Request type-checking allows security policies to reason about arbitrarily complex requests and allows applications to define arbitrary policy criteria. Not only the identity and credentials of the requester, but also the nature of the operation requested can be a factor in whether a request is permitted.

Our approach also enables application-specific criteria for access control and integrity, rather than limiting policies to the set of criteria considered by a logic's designer. By allowing applications to determine the factual correctness of a request, and policies to define their own logics, anything performable by an application can be considered by a security policy, ranging from network topology measurement to watermark checking, acoustical signature verification, and natural language processing for format-independent document integrity checking.

8. ACKNOWLEDGEMENTS

9. REFERENCES

[1] M. Abadi. Logic in access control. In *Proceedings IEEE Symposium on Logic in Computer Science*, pages 228–233, June 2003.

[2] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, September 1993.

[3] M. D. Abrams and P. B. Schneck. Controlling primary and secondary access to digital information. In *Proceedings of the 23rd National Information Systems Security Conference NISSC*, 2000.

[4] D. G. Andersen, H. Balakrishnan, M. F. aashoek, and R. Morris. The case for resilient overlay networks. In *HotOS-VIII*, May 2001.

[5] A. W. Appel. Foundational proof-carrying code. In *Logic in Computer Science*, 2001.

[6] D. Aspinall and A. Compagnoni. Subtyping dependent types. *Theoretical Computer Science*, 266(1-2):273–309, Septeber 2001.

[7] L. Bauer, M. A. Schneider, and E. W. Felten. A general and flexible access-control system for the web. In *Proceedings USENIX Security Symposiumm*, pages 93–108, August 2002.

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

[9] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: evidence and implications. In *Proceedings of INFOCOM*, 1999.

[10] I. Cervesato and F. Pfenning. A Linear Logical Framework. *Information & Computation*, 179(1):19–75, November 2002.

[11] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009:46+, 2001.

[12] J. DeTreville. Binder, a logic-based security language. Technical Report MS-TR-2002-21, Microsoft, 2002.

[13] J.-E. Elien. Certificate discovery using SPKI/SDSI 2.0 certificates. Master's thesis, Massachusetts Institute of Technology, 1998.

[14] J. S. Erickson. Fair use, drm, and trusted computing. *Communications of the ACM*, 46(4):34–39, 2003.

[15] Gnutella. `http://gnutella.wego.com/`.

[16] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *Proceedings 2nd Annual IEEE Symp. on Logic in Computer Science, LICS'87, Ithaca, NY, USA, 22–25 June 1987*, pages 194–204. IEEE Computer Society Press, New York, 1987.

[17] C. E. Landwehr. Formal models for computer security. *ACM Computing Surveys*, 13(3):247–278, September 1981.

[18] N. Li, B. N. Grosof, and J. Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *ACM Transactions on Information and Systems Security*, 6(1):128–171, February 2003.

[19] N. Li and J. C. Mitchell. Understanding SPKI/SDSI using first-order logic. In *Proceedings of IEEE Computer Security Foundations Workshop*, June 2003.

[20] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, Jan. 1997.

[21] D. of Central Intelligence. Control of dissemination of intelligence information. Directive No. 1/7, May 1981.

[22] J. Park and R. Sadhu. Towards usage control models: Beyond traditional access control. In *Proceedings of the 7th ACM Symposium on Access Control Models and Technologies SACMAT*, June 2002.

[23] J. Park and R. Sandhu. Security architectures for controlled digital information dissemination. In *Proceedings of ACSAC 2000*, 2000.

[24] J. Park and R. Sandhu. Originator control in usage control. In *Proceedings of POLICY 2002*, 2002.

[25] F. Pfenning. The practice of logical frameworks. In H. Kirchner, editor, *Proceedings of the Colloquium on Trees in Algebra and Programming*, pages 119–134, Linköping, Sweden, 1996. Springer-Verlag LNCS 1059.

[26] F. Pfenning and C. Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, 1999. Springer-Verlag LNAI 1632.

[27] The keynote trust management system version 2. IETF RFC 2704, September 1999. http://www.ietf.org/rfc/rfc2704.txt.

[28] R. L. Rivest and B. Lampson. SDSI – A simple distributed security infrastructure. Presented at CRYPTO'96 Rumpsession, 1996.

[29] R. Sadhu and J. Park. Usage control: A vision for next generation access control. In V. Gorodetsky, L. Popyack, and V. Skormin, editors, *Computer Network Security: Second International Workshop on Mathematical Methods, Models, and Architectures for Computer Network Security, MMM-ACNS*, pages 17–31, St. Petersburg, Russia, September 2003.

[30] D. Stutzbach, D. Zappala, and R. Rejaie. Swarming: Scalable content delivery for the masses. Technical Report UO-CIS-TR-2004-1, University of Oregon, January 2004.

APPENDIX
A. NOTATION

We use the following notation for logic and types in this paper. Inferences rules are represented as:

$$\frac{Premise \quad Premise}{Conclusion}$$

Conclusions derivable from no premises (axioms) are presented as:

$$\frac{}{Conclusion}$$

The symbol Γ refers to a mapping from variable names to types, and statements of the form "$\Gamma \vdash x : t$" mean that the given mapping establishes that expression x has type t. Similarly, Δ refers to a set of type variables, and "$\Delta \vdash t : type$" means that Δ establishes that t is a valid type. "$\Gamma \vdash_{\Delta} x : t$" is a statement that under Γ and Δ x is an expression of type t. Type substitution is denoted as follows: $\{t'/X\}t$ means "t with all occurrences of (free variable) X replaced with t'.

Language literals are given in `monospace` font, and meta-syntactic variables are *italicized*.

Thus,

$$\frac{\Gamma \vdash_{\Delta} cert : t \rightarrow t \ valid\_claim \quad \Gamma \vdash_{\Delta} claim : t}{\Gamma \vdash_{\Delta} (\texttt{certifies} \ cert \ claim) : t \ valid\_claim}$$

should be read as "A `certifies` node with two children has type $t \ valid_c laim$ in a given context if in the same context the first child has type $t \rightarrow t \ valid\_claim$ and the second has type t." Implicitly, t must be valid type in that context.