Lightweight Consistency Enforcement Schemes for Distributed Proofs with Hidden Subtrees (Extended Version)[*]

Adam J. Lee adamlee@cs.uiuc.edu
Kazuhiro Minami minami@cs.uiuc.edu
Marianne Winslett winslett@cs.uiuc.edu

Department of Computer Science
University of Illinois at Urbana-Champaign
201 N. Goodwin Avenue
Urbana, IL 61801

Abstract

In distributed proof construction systems, information release policies can make it unlikely that any single node in the system is aware of the complete structure of any particular proof tree. This property makes it difficult for queriers to determine whether the proofs constructed using these protocols sampled a consistent snapshot of the system state; this has previously been shown to have dire consequences in decentralized authorization systems. Unfortunately, the consistency enforcement solutions presented in previous work were designed for systems in which only information encoded in certificates issued by certificate authorities is used during the decision-making process. Further, they assume that each piece of certified evidence used during proof construction is available to the decision-making node at runtime.

In this paper, we generalize these previous results and present lightweight mechanisms through which consistency constraints can be enforced in proof systems in which the full details of a proof may be unavailable to the querier and the existence of certificate authorities for certifying evidence is unlikely; these types of distributed proof systems are likely candidates for use in pervasive computing and sensor network environments. We present modifications to one such distributed proof system that enable two types of consistency constraints to be enforced while still respecting the same confidentiality and integrity policies as the original proof system. Further, we present the details of a performance analysis conducted to illustrate the modest overheads (less than 30%) of consistency enforcement on distributed proof construction.

[*]A shorter version of this paper will appear in the Proceedings of the 12th ACM Symposium on Access Control Models and Technologies [13].

1 Introduction

The process of making informed authorization decisions in dynamic environments where trust relationships cannot be determined a priori is widely accepted as a difficult task. This is particularly true in context-rich environments such as pervasive computing spaces, as the set of permissible actions may depend on the physical context of the space. This context can be sampled through the use of sensors deployed throughout the environment. To address this complexity, several rule-based systems have been designed for specifying and checking authorization policies in pervasive computing environments (e.g., [2, 4, 11, 20]). Recently, frameworks for constructing and validating *distributed* proofs have been proposed to address the limitations of using centralized knowledge bases for making authorization decisions [6, 17, 24].

In authorization systems based on distributed proving, resource access requests are permitted if a resource owner can construct a well-formed proof tree whose root is a logical statement granting the requester access to the resource. The topology of a proof tree shows the logical dependencies among the facts in the tree; that is, the leaves of this tree represent base facts, while intermediate nodes represent inferences made using these facts. Such a proof tree need not be formed solely from facts in the resource owner's local knowledge base; subtrees of a proof may be produced by other entities in the network provided that the resource owner trusts the integrity of information provided by these entities (e.g., as in [6, 17, 24]). In some systems, information release policies may prevent portions of a subproof from being revealed to certain nodes in the proof tree [17]. An important observation is that the logical leaves of a distributed proof tree form one possible *view* of the state of the environment in which the proof was constructed. Resource access is granted because, in that view of the system, it was possible to construct a proof tree justifying the access request. If the facts making up a proof tree represent stable assertions (i.e., facts whose validity will not change), then this view is actually a snapshot of the system and the semantics of policy satisfaction remain the same as in centralized proof systems. However, if *any* facts in the proof tree are not constant, then in some circumstances, it is possible to form a proof tree justifying access to a particular resource that would have been denied in *any* centralized system. That is, an inconsistent view can lead a prover to think that certain logical facts were true simultaneously when, in fact, they were not. Clearly, this can lead to the permission of undesirable accesses to system resources.

For example, consider a hospital wired with sensors such as occupancy detectors, location tracking devices, and door lock sensors. Now, a clinician, Alice, decides to use the projector located in her office to review the medical records of several patients that she is working with. In order for the system to permit the use of the projector to view medical records, it must be the case that the occupancy of Alice's office is one, Alice is located in her office, and the door to her office is locked. When Alice requests this access, the system might first check that the occupancy of her office is one and then proceed to check that Alice is currently located in her office. As this check is being made, Bob enters the room and closes the door behind him, which automatically locks. The system determines that Alice is located in her office and then checks that the door is locked; since the door is locked the medical records are displayed on the projector. This is a clear violation of the policy protecting patient records that might have legal ramifications, as Bob may not be authorized to view the records being projected. In addition to this type of accidental violation of system view consistency, intentional attacks on the system are also possible.

The adverse effects of inconsistent views on authorization systems has been examined previously in the literature [14]. In this work, the authors focused on studying the properties of systems in which all attestations used during proof construction were encoded in certificates issued by one or more trusted Certificate Authorities (CAs). The solutions for enforcing the use of consistent states presented in [14] rely on the timing and sequencing of checks for certificate revocation that can be made using protocols such as OCSP [19] or COCA [26]. Unfortunately, these solutions cannot be used in proof construction frameworks that rely on simple digital signatures or keyed MACs to authenticate proof facts, including many of those designed to be used in pervasive computing or sensor network environments.

In this paper, we build upon the results presented in [14] and show how to ensure that distributed proofs constructed using these more general forms of trusted information can be formed by sampling consistent system states without impeding on the autonomy of nodes in the system (e.g., by requiring participation in a wide-scale transaction-management protocol). Further, we present solutions to the consistency problem that work even if some details of a proof tree are hidden from the query issuer by information flow policies; for comparison, the solutions presented in [14] assumed that the policy evaluator had complete knowledge of the proof tree formed during the protocol. Although we focus our presentation on authorization systems based on distributed proving, the techniques described in this paper are applicable to any system in which autonomous entities wish to leverage decentralized information to make decisions in a potentially adversarial environment.

The rest of this paper is organized as follows. In Section 2, we overview background material regarding the distributed proof construction protocol that we will modify to enforce view consistency constraints. Section 3 formally defines our system model and the levels of view consistency that we wish to enforce in this paper. In Section 4, we present modifications to an existing distributed proof construction protocol to enable the use of two types of consistent views when making authorization decisions. Further, we present proofs that the security and privacy properties of the underlying proof system have not been altered by our modifications. We quantitatively evaluate the performance impact of our consistency enforcement schemes in Section 5 and review related work in Section 6. We then present our conclusions and directions for future work in Section 7.

2 Background

In this section, we discuss the Minami-Kotz distributed proof construction protocol presented in [17], as later sections of this paper focus on modifying this protocol to ensure that authorization decisions are made using consistent states. Unfortunately, space limitations prevent us from presenting this proof system in its entirety, so we instead present several examples that illustrate the key features of this system; interested readers can refer to [17] for a more in-depth treatment of this proof construction system. We chose to explore the consistency problem within the context of this protocol as it allows portions of a proof tree to be hidden from certain entities participating in the construction of the proof tree, including the node issuing

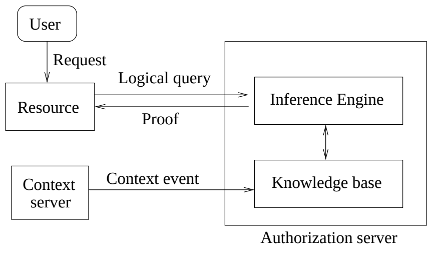

Figure 1: Structure of an authorization server.

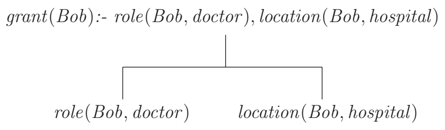
$grant(Bob)$:- $role(Bob, doctor), location(Bob, hospital)$
$role(Bob, doctor)$
$location(Bob, hospital)$

Figure 2: Sample proof tree.

the query, whereas most other distributed proof frameworks assume that the querying node gathers all supporting evidence locally prior to making a decision. The techniques developed in this paper for use in the Minami-Kotz proof construction system can also be applied to other distributed proof systems with less restrictive properties.

2.1 Structure of the Authorization Server

Figure 1 shows the structure of an authorization server consisting of a knowledge base and an inference engine. The knowledge base stores both authorization policies and facts including context information. The context server publishes context events and updates facts in the knowledge base dynamically. The inference engine receives authorization queries from remote servers, such as resource servers processing users' access requests. The inference engine then attempts to derive logical proofs justifying these queries using the facts in its local knowledge base and possibly even interactions with remote parties. If the inference engine cannot construct a proof, it returns a proof that contains a false value. In the open environment of pervasive computing, each server could belong to a different administrative domain.

Rules and facts in a knowledge base are represented as a set of Horn clauses in Prolog. For example, a medical database may define an authorization policy that requires a requester P to hold a role membership "doctor" and to be physically located at the "hospital" as follows.

$$grant(P) \text{ :- } role(P, doctor), location(P, hospital)$$

The atoms $role(P, doctor)$ and $location(P, hospital)$ on the right side of the clause are the conditions that must be satisfied to derive the granting decision $grant(P)$ on the left. If a user *Bob* issues a request to read a medical database, the proof tree in Figure 2 could be constructed based on the above rule. The root node in the tree represents the rule and the two leaf nodes represent the facts. Notice that the variable P in the rule is replaced with a constant *Bob*. A user's location, which is expressed with the $location$ predicate, is a dynamic fact; i.e., the second variable of the predicate $location$ should be updated dynamically as Bob changes his location.

2.2 Proof Decomposition

Multiple authorization servers in different administrative domains can cooperate to handle authorization queries in a peer-to-peer manner. These peer-to-peer interactions are guided by each entity's *integrity policies*, which specify sets of entities trusted to handle particular types of queries. For example, if Alice specifies the integrity policy $trust(location(P, L)) = \{Bob\}$, then she trusts Bob to accurately answer queries regarding the location of other entities. In the most basic case, the principal who issues a query trusts the principal who handles this query in terms of the integrity of the query result. As such, the handler principal need not disclose the entire proof tree that she generates, she needs only to return a proof that states whether the fact in the query was true. In general, however, the querier may not *completely* trust the query handler and thus her integrity policies might place constraints on the rules used by the handler to generate the proof tree. In this case, a more complete proof tree, whose intermediate nodes are digitally signed, would need to be returned by the handler. This way, the querier can verify that her integrity policies were respected.

Figure 3 describes one possible collaboration between a querier and handler. Suppose that host A run by principal Alice, who owns a projector, receives an authorization query $?grant(Dave, projector)$ that asks whether Dave is granted access to that projector. Since Alice's authorization policy in her knowledge base refers to a requester's location (i.e., $location(P, room112)$), Alice issues a query $?location(Dave, room112)$ to host B run by Bob. Alice chooses Bob, because Bob satisfies Alice's integrity policies for queries of the type $location(P, L)$. Bob processes the query from Alice, because Alice satisfies Bob's confidentiality policies for queries of the type $location(P, L)$ as defined in Bob's policy $acl(location(P, L)) = \{Alice\}$. Bob derives the fact that Dave is in *room112* from the location of his device using the facts $location(pda15, room112)$ and

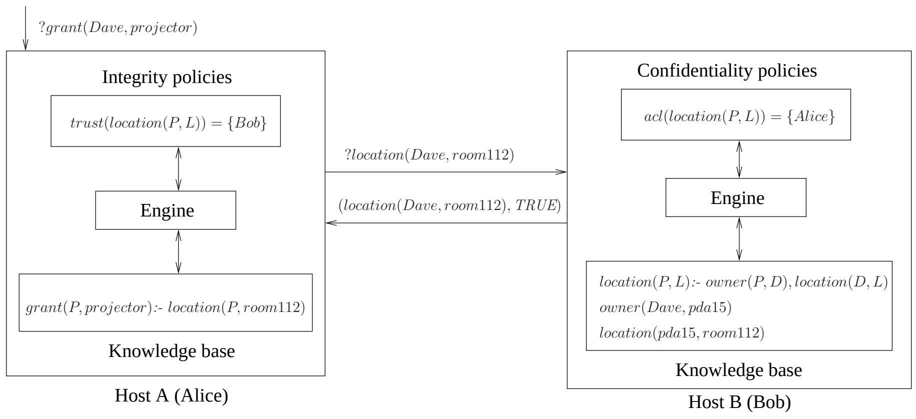

Figure 3: Remote query between two principals. Alice is a principal who maintains a projector, and Bob is a principal who runs a location server.

$owner(Bob, pda15)$. However, he only needs to return a proof that contains a single root node that states that $location(Dave, room112)$ is true, because Alice believes Bob's statement about people's location (i.e., $location(P, L)$) according to her integrity policies. The proof of the query is thus decomposed into two subproofs maintained by Alice and Bob.

2.3 Enforcement of Confidentiality Policies

Each fact provider maintains a set of *confidentiality policies* that determine which entities are authorized to receive the facts that she provides. These policies are enforced by encrypting a query result (along with a querier-provided nonce to ensure freshness) using the public key of an authorized receiver. Each query is accompanied by a list of upstream principals who could possibly receive the answer of the query; this enables the handler to choose an authorized recipient from the list of upstream principals that satisfies her confidentiality policies. It is therefore possible to obtain an answer for some initial query even when some number of intermediate principals in the distributed proof do not satisfy the confidentiality policies of a fact provider. Figure 4 shows an example collaboration among principals p_0, p_1, p_2, and p_3. When principal p_0 issues an authorization query q_0 to principal p_1, p_1 issues a subsequent query q_1, which causes principal p_2's queries q_2 and q_3. Since a receiver principal of a proof might not be a principal who issues a query, a reply for a query is a tuple $(p_i, (pf)_{K_i})$ where p_i is an identity of a receiver principal and $(pf)_{K_i}$ is an encrypted proof with the receiver's public key. We associate a receiver principal identity with an encrypted proof so that a principal who receives an encrypted fact can decide whether to attempt to decrypt that encrypted fact. We assume that, in this example, each principal who issues a query trusts the integrity of the principal who receives that query in terms of the correctness of whether the fact in the query is true or not. For example, p_0's integrity policies contain a policy $trust(q_0) = \{p_1\}$.

Suppose that query q_1's result (i.e., true or false) depends on the results of queries q_2 and q_3, which are handled by principals p_3 and p_4, respectively, and that p_3 and p_4 choose principals p_0 and p_1, respectively, as receivers since p_2 does not satisfy their confidentiality policies. Because principal p_2 cannot decrypt the results from principals p_3 and p_4, p_2 encrypts those results with the public key of principal p_1, which p_2 chose as a receiver. Principal p_2 forwards the encrypted results from p_3 and p_4 because the query result of q_1 is the conjunction of those results. Principal p_1 decrypts the encrypted result from p_2 and obtains the encrypted results originally sent from principals p_3 and p_4. Since p_1 is a receiver of the proof from p_4, p_1 decrypts the proof that contains a true value. Since a query result for q_0 depends on the encrypted proof from p_3, principal p_1 forwards it in the same way. The principal p_0 finally decrypts it and obtains an answer for query q_0. The key observation here is that principal p_0 is not aware of the fact that the query result is originally produced by principal p_3.

This proof system applies public-key operations only to a randomly generated symmetric key and uses that symmetric key to encrypt and decrypt a proof; that is, a proof consists of a new symmetric key encrypted with a receiver's public key and a proof encrypted with that symmetric key. In addition to the public-key encryption, the querier and handler principals use another shared symmetric key to protect other data fields (e.g., a re-

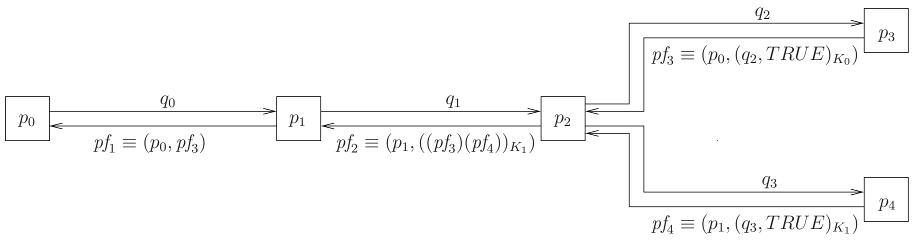
q_0
q_1
q_2
q_3
p_0
p_1
p_2
p_3
p_4
$pf_1 \equiv (p_0, pf_3)$
$pf_2 \equiv (p_1, ((pf_3)(pf_4))_{K_1})$
$pf_3 \equiv (p_0, (q_2, TRUE)_{K_0})$
$pf_4 \equiv (p_1, (q_3, TRUE)_{K_1})$

Figure 4: Enforcement of confidentiality policies. The first item in a proof tuple is a receiver principal, and the second item is a proof tree encrypted with the receiver's public key.

ceiver identity) in a query and a proof from eavesdroppers. We assume that the two principals share the symmetric key via a protocol using public-key operations when the querier and handler principal authenticate with each other for the first time.

3 Definitions

We begin this section by describing the system model within which the Minami-Kotz distributed proof construction protocol discussed in Section 2 was designed to be used. We then show that existing solutions to the view consistency problem are not applicable due to fundamental differences between system models. Lastly, we formally define the view consistency problem within the context of our system model and present the definitions of three important view consistency levels.

3.1 System Model

Distributed proof construction protocols were designed to be used in open-system environments consisting of a possibly infinite set of autonomous entities, $\mathcal{E}$. Each entity $e \in \mathcal{E}$ possesses one or more public key certificates that can be used to authenticate messages signed by e or to encrypt messages that are to be sent to e. These certificates are made publicly available by one or more key servers or through the use of decentralized peer-to-peer protocols. Without loss of generality, we will assume that each node uses only one public key certificate during the construction of any single distributed proof. We place no limitations on the temporal duration of executions of the proof construction protocol, nor do we assume any level of clock synchronization exists between entities in $\mathcal{E}$.

All evidence used during the construction of a distributed proof takes the form of assertions signed with the providing entity's private key. As was described in Section 2, the proof construction process is assumed to be policy-directed. Each entity maintains a collection of *integrity policies* that indicate which other entities are trusted to answer different types of queries; adherence to these integrity policies can be checked by verifying the signatures on responses to any issued subqueries. Each entity e also maintains a collection of *confidentiality policies* that control the release of subproofs generated by e. This interplay between integrity policies and confidentiality policies implies that the complete details of a proof tree may not be available to entities in the system. In particular, the querying entity will not learn any details of the proof tree beyond those specified by his or her integrity policies. Further, intermediate nodes in a proof tree may not learn whether the proof construction protocol was successful, as the results of subqueries issued by these nodes may be hidden from them by the query target's confidentiality policies (see Section 2.3 for an example of this behavior). These assumptions imply that the view consistency solutions developed in [14] cannot be used in this environment.

3.2 Problem Definition

As was observed in Section 1, the use of inconsistent views of a system during policy evaluation can lead to situations in which a policy evaluator believes that certain facts held true simultaneously when, in fact, they did not. We now more precisely define this problem.

Definition 1 (Validity). *An entity e can determine that some proof fact f is* valid *at time t if either (i) f is in e's local knowledge base at time t, or (ii) f is considered valid at time t by a remote entity who is trusted to provide information regarding f.*

Note that asserting the validity of some fact f at a particular time t by invoking case (ii) of the above definition is not a straight-forward task. Consider the case where some entity e issues a query for fact f to another entity e' at time t_{iss}. Due to delays in the network and processing delays at e and e', it is likely that e' will not receive the query until some time $t' > t_{iss}$. Similarly, e is unlikely to receive e''s response to his query until some time $t_{rcv} > t'$. Therefore, e cannot conclude that f was valid at either t_{iss} or t_{rcv}; he can only infer that f was valid at *some* time t where $t_{iss} \leq t \leq t_{rcv}$; we will discuss methods for fine-tuning these types of inferences later in the paper. As facts are collected and validated, an entity builds a *view* of the system that will be used to construct a proof of authorization.

Definition 2 (Fuzzy Validity Interval). *The interval $[t_s, t_e]$ is a* fuzzy validity interval *for some fact f if f can be shown to be valid at some (possibly unknown) time t such that $t_s \leq t \leq t_e$.*

Definition 3 (Concrete Validity Interval). *The interval $[t_s, t_e]$ is a* concrete validity interval *for some fact f if f can be shown to be valid at all times t such that $t_s \leq t \leq t_e$.*

Definition 4 (Fact State). *Let the set T contain all possible time stamps, let $\bot$ denote the null value, and let ℓ be a predefined length parameter. The* fact state *for a fact f as observed by some entity is then denoted by the five-tuple $s = \langle id, e, t_\alpha, t_\omega, \mathit{fuzzy}\rangle \in \{0,1\}^\ell \times \mathcal{E} \times T \cup \{\bot\} \times T \cup \{\bot\} \times \mathbb{B}$. The value id is an ℓ-bit identifier assigned to the fact f (which may simply be an encoding of that fact), e identifies the entity from which f was obtained, t_α and t_ω are local timestamps, and fuzzy is a Boolean value indicating whether $[t_\alpha, t_\omega]$ specifies a fuzzy or concrete validity interval. The set of all possible fact state tuples is denoted by $\mathcal{S}$.*

Entities in the system create fact state tuples as the validity of certain facts is revealed during the execution of the distributed proof protocol. In the remainder of this paper, we will use dot notation to access the fields of fact state tuples. For instance $s.id$ represents the identifier of the fact whose state is stored in s. Note that if given a fact state tuple s for a fact f has either of its $s.t_\alpha$ or $s.t_\omega$ fields set to $\bot$, then no conclusions can be drawn about the validity status of f.

Definition 5 (View). *A* view *is any collection of fact state tuples that has no more than one tuple for any $\langle id, e\rangle$ pair.*

We have now defined an entity e's view of the system as some collection of local observations that e has made regarding the validity of certain facts. Given that any such view contains only *local* observations, it is unlikely to capture a precise snapshot of the system state. As such, the consistency level of these views is of the utmost importance. Although such a view may contain data associated with any number of facts, unless noted otherwise, we assume without loss of generality that an entity e will only wish to enforce consistency constraints on views comprised of facts associated with a single distributed proof.

Definition 6 (View Consistency). *A view V is said to be ϕ-consistent if and only if V satisfies some predicate ϕ that places temporal constraints on the observed validity intervals of the facts whose state data are stored in V.*

3.3 Levels of Consistency

We now describe three increasingly-stringent levels of view consistency relevant to distributed proof construction protocols for use in the system model described in Section 3.1.

3.3.1 Incremental Consistency

The most basic definition of view consistency that one can imagine is what we will refer to as incremental consistency. Intuitively, an incrementally consistent view is a view in which each fact was valid at some point during the construction of the related proof tree. To formally define the notion of incremental consistency, we first define the predicates $\mathit{checked} : \mathcal{S} \rightarrow \mathbb{B}$, $\mathit{fuzzy} : \mathcal{S} \rightarrow \mathbb{B}$, and $\mathit{concrete} : \mathcal{S} \rightarrow \mathbb{B}$.

$$\mathit{checked}(s) \equiv (s.t_\alpha \neq \bot) \wedge (s.t_\omega \neq \bot) \wedge (s.t_\alpha \leq s.t_\omega) \quad (1)$$

$$\mathit{fuzzy}(s) \equiv \mathit{checked}(s) \wedge s.\mathit{fuzzy} \quad (2)$$

$$\mathit{concrete}(s) \equiv \mathit{checked}(s) \wedge \neg s.\mathit{fuzzy} \quad (3)$$

The predicate $\mathit{checked}(s)$ ensures that the fact state tuple s contains a fully-defined validity interval. The $\mathit{fuzzy}(s)$ predicate is true if and only if s encodes a fully-defined fuzzy validity interval; $\mathit{concrete}(s)$ is true if and only if s encodes a fully-defined concrete validity interval. Given these predicates, we can now formally define the notion of incremental consistency for distributed proof systems via the predicate $\phi_{inc} : 2^{\mathcal{S}} \times T \times T \rightarrow \mathbb{B}$ as follows:

$$\begin{aligned}\phi_{inc}(V, t_s, t_e) \equiv\ & \forall s \in V : checked(s) \\ & \wedge (fuzzy(s) \rightarrow \\ & \quad ((t_s \leq s.t_\alpha) \wedge (s.t_\omega \leq t_e))) \\ & \wedge (concrete(s) \rightarrow \\ & \quad ((s.t_\alpha \leq t_s \leq s.t_\omega) \vee \\ & \quad (s.t_\alpha \leq t_e \leq t_\omega))\end{aligned} \tag{4}$$

The predicate ϕ_{inc}—which is a reformulation of the predicate ϕ_{inc} presented in [14]—is satisfied by a view V during some interval $[t_s, t_e]$ if and only if each fact state tuple in the view contains a fully-specified validity interval, each fuzzy validity interval is a subinterval of $[t_s, t_e]$, and each concrete validity interval overlaps $[t_s, t_e]$ at some point. This gives us the following definition for an incrementally consistent proof construction and an associated theorem.

Definition 7 (Incremental Consistency). *A view V generated between the time that a given query was issued, t_{iss}, and the time that the completed proof tree was received by the issuer, t_{rcv}, is* incrementally consistent *if and only if $\phi_{inc}(V, t_{iss}, t_{rcv})$ is true.*

Theorem 1. *The Minami-Kotz distributed proof construction protocol always uses incrementally consistent views when evaluating authorization policies.*

Proof. Assume that the distributed proof construction algorithm succeeds in constructing a proof tree using a view that is not incrementally consistent. This implies that there exists some fact f that was not true at any point during execution of the proof construction protocol. This means that the validity status for f (which must be true for the proof to succeed) was contributed to the proof tree *before* the proof construction process was started or, equivalently, was a replayed validity status from an earlier execution of the protocol. However, each validity status returned by a fact provider is causally-linked to the query executed by a querier-provided nonce (see [17]), which prevents both the incorporation of old validity information and replay attacks. This implies that f was valid during the protocol execution, which is a contradiction. □

The fact that existing distributed proof construction protocols use incrementally consistent views when making authorization decisions is exactly what leads to the types of safety violations discussed in Section 1. This is because incremental consistency provides no guarantees regarding the overlap of the observed validity periods for facts whose state is stored in V in the event that *any* fact used during the proof construction is not a stable assertion. The other consistency levels defined in this section will address this problem.

3.3.2 Query Consistency

The next more stringent level of consistency that we define is query consistency. Informally, this consistency level guarantees that all facts used to construct a distributed proof were valid simultaneously at the time that the query triggering that proof construction was issued. We formally define query consistency in terms of the predicate $\phi_{query} : 2^{\mathcal{S}} \times T \rightarrow \mathbb{B}$, as follows:

$$\begin{aligned}\phi_{query}(V, t_{iss}) \equiv\ & \forall s \in V : concrete(s) \\ & \wedge (s.t_\alpha \leq t_{iss} \leq s.t_\omega)\end{aligned} \tag{5}$$

Definition 8 (Query Consistency). *A view V is* query consistent *with respect to a query issued at time t_{iss} if and only if $\phi_{query}(V, t_{iss})$ is true.*

If an authorization policy is satisfied using a query consistent view, the semantics of policy satisfaction in the distributed proof construction setting remain the same as if the proof had been constructed using a centralized proof framework supporting transactional evaluation (e.g., a Prolog theorem prover). In the event that any facts necessary to construct a given proof of authorization are unstable (i.e., their value can change once set), a view consistency level that is at least as strong as query consistency should be enforced to ensure that the satisfaction of a given authorization policy carries the same meaning as policy writers and analysts would expect it to have.

3.3.3 Interval Consistency

The most stringent consistency level that we consider in this paper is interval consistency. We say that some view V is interval consistent during some interval $[t_s, t_e]$ if each fact state tuple in V encodes a concrete validity interval that includes at least $[t_s, t_e]$. More formally, we define interval consistency using the predicate $\phi_{interval} : 2^{\mathcal{S}} \times T \times T \rightarrow \mathbb{B}$, as follows:

$$\begin{aligned}\phi_{interval}(V, t_s, t_e) \equiv\ & \forall s \in V : concrete(s) \\ & \wedge (s.t_\alpha \leq t_s \leq t_e \leq s.t_\omega)\end{aligned} \tag{6}$$

Definition 9 (Interval Consistency). *A view V is* interval consistent *for a time interval $[t_s, t_e]$ if and only if $\phi_{interval}(V, t_s, t_e)$ is true.*

The above definition of interval consistency is a reformulation of the definition of interval consistency presented in [14], altered to fit within the formalization of the consistency problem presented in Section 3.2. In distributed proving, the notion of interval consistency is useful for two primary reasons. First and foremost, interval consistency is important in the event that a resource provider wishes to monitor the conditions that lead to the permission of a given resource access. For instance, the hospital smart room discussed in Section 1 may wish to first check that Alice is the only person located in her locked office before allowing her to project patient records onto the wall and then continue to monitor these conditions. If her door subsequently became unlocked, for instance, access to the projector could be revoked.

At the implementation level, interval consistency can also be useful in the event that a proof tree is constructed that permits access to a given resource, but that view cannot be shown to be query consistent. The fact that a proof could be formed at all implies that it is possible that the facts that make up the proof were valid simultaneously, even though this could not be guaranteed from the view used to construct the proof. If it is faster to recheck a proof than it would be to generate the proof tree again, then this recheck could lead to an interval consistent view during the interval $[t_{rcv}, t_{recheck}]$, where t_{rcv} is the time that the original proof was returned to the resource provider and $t_{recheck}$ is the time at which the resource provider begins revalidation of the proof tree. We will explore this case further in Section 4.3.

4 Algorithm Details

In this section, we discuss modifications to the Minami-Kotz distributed proof construction algorithm that ensure the use of consistent system views during policy evaluation. As this algorithm trivially ensures that an incrementally consistent view is used (by Theorem 1), we will focus our discussion on creating query and interval consistent views.

4.1 Preliminaries

In this section, we will be concerned with both the correctness and security properties of our proposed proof construction algorithm modifications. In addition to proving the soundness of our consistency enforcement algorithms, we will also address their proximity to *ideal completeness*. A ϕ-consistency enforcement algorithm is said to be ideally complete if and only if it is capable of constructing ϕ-consistent views for all protocol executions in which an ideal algorithm run by an omniscient entity could construct a ϕ-consistent view [14]. Further, we will ensure that each proposed modification is a *policy-safe modification* to the proof construction protocol. That is, we will show that our modifications do not violate the integrity or confidentiality policies specified by each entity.

Algorithm 1 A query consistency enforcement algorithm

```
// Receive a fact response tuple relevant to a query issued at time $t_{iss}$
// from some entity $e$. Only invoked on true facts.
**Function** RcvFact($f \in \mathcal{F}, d \in T, e \in \mathcal{E}, t_{iss} \in T, V \in 2^{\mathcal{S}}$)
$t_{rcv} \leftarrow NOW$
**if** $t_{rcv} - t_{iss} \leq d(1 - \delta)$ **then**
    $V.insert(\text{Encode}(f), e, t_{iss}, t_{iss},$ false$)$
**else**
    $V.insert(\text{Encode}(f), e, t_{iss}, NOW,$ true$)$

// Check the query consistency condition on a view $V$ relative
// to a query issued at time $t_{iss}$.
**Function** CheckQuery($V \in 2^{\mathcal{S}}, t_{iss} \in T$)
**for all** $s \in V$ **do**
    **if** $s.fuzzy \vee (t_{iss} < s.t_\alpha) \vee (s.t_\omega < t_{iss})$ **then**
        **return** false
**return** true
```

4.2 Query Consistency

We now show that with relatively minor changes, the Minami-Kotz distributed proof construction protocol can be modified to use query consistent views when making authorization decisions. As presented in [17], this proof construction algorithm assumes that each knowledge base KB is defined as a subset of all possible facts, $\mathcal{F}$. Rather, we will define a knowledge base KB as a subset of $\mathcal{F} \times T$ in which each fact is associated with the local time at which it was inserted into KB. This allows each node to track the duration of a given fact's validity locally.

To leverage this new knowledge base format, the format of query responses must also be altered. Rather than an entity e responding to some query $?f$ with a Boolean response $b \in \mathbb{B}$ indicating whether f is considered valid by e (as in Section 2), they will instead respond with a *fact response tuple* of the form $\langle b, d \rangle \in \mathbb{B} \times T$. The b component of this tuple indicates whether e considers f to be valid, as before, and the d component of this tuple represents the length of time that e acknowledges that f has been true, or some duration less than this if the exact duration of validity is considered sensitive. In the event that f is a base atom, d is (at most) the difference between the current time and the time associated with f in e's knowledge base; if f is the head of a Horn clause f :- $f_1, \ldots, f_n$, then d is set to be (at most) the minimum such duration associated with any of $f_1, \ldots, f_n$. In the case that f is false, d is set to 0. Note that neither f nor any $f_1, \ldots, f_n$ need to be locally-stored facts.

Given the above modifications to the formats of entities' knowledge bases and query responses, we now present the details of Algorithm 1, which facilitates the creation of query consistent views. In Algorithm 1 and all other algorithms presented in this paper, we make the following assumptions regarding the local data structures accessible by entities in the system:

- The current local time is available via the local variable NOW.
- The absolute value of the maximum clock drift rate between any two entities in the system is no more than some constant δ. This does not imply that clocks are in any way synchronized, only that for each n time units that pass one entity, no less than $n(1-\delta)$ time units and no more than $n(1+\delta)$ time units pass at any other entity.
- Fact state tuples can be inserted into a view data structure via the function $insert : \{0,1\}^{\ell} \times \mathcal{E} \times T \times T \times \mathbb{B} \rightarrow \perp$. Note, for instance, that $V.insert(id, e, t, t', b)$ will replace any existing fact state tuples in V that have the identifier id and were received from entity e (see Definition 5).
- The function ENCODE : $\mathcal{F} \rightarrow \{0,1\}^{\ell}$ returns an encoding of some fact $f \in \mathcal{F}$ suitable for insertion into a view (see Definition 4).

Algorithm 1 consists of two functions that are to be used by the querying entity. Whenever the querier issues a new query, she records the query issue time t_{iss} and chooses a view V to which the results of her query are considered relevant; V need not be a new empty view. In the event that the response to her query is true, Alice invokes the RCVFACT function. If the fact provider attests that the fact whose status was queried was valid for some duration d that is longer than the time between when the query was issued and when its response was received, this function inserts a fact state tuple into V asserting that the corresponding fact was valid at time t_{iss}. Otherwise, a fact state tuple encoding a fuzzy interval is inserted into V. The function CHECKQUERY checks to see that ϕ_{query} is true, as defined by Equation 5.

Theorem 2. *If the function* CHECKQUERY(V, t_{iss}) *returns* true*, then V is query consistent relative to the query issue time t_{iss}, provided that V was constructed using only the* RCVFACT *function.*

Proof. The CHECKQUERY function clearly enforces the constraint that all fact state records encode concrete validity intervals that include the time t_{iss}. Thus CHECKQUERY$(V, t_{iss}) \leftrightarrow \phi_{query}(V, t_{iss})$ and V is query consistent with respect to the time t_{iss} by Definition 8, provided that all concrete validity intervals established by RCVFACT are correct. Lines 5 through 8 of Algorithm 1 ensure that RCVFACT inserts a concrete validity interval into V if and only if the validity duration, d, reported by the fact provider is longer than the query round trip time, even when adjusted to assume the largest possible clock drift between entities. Since the query and its associated response can be causally linked by nonces used in the underlying proof construction protocol (see [17]), we can infer that the fact provider sent its response to the query at some time $t \geq t_{iss}$. This implies that the fact associated with the state tuple being inserted into V was valid at t_{iss} because $t - d(1-\delta) \leq t_{iss} \leq t$ for all possible values of t such that $t_{iss} \leq t \leq t_{rcv}$ since $t_{rcv} - t_{iss} \leq d(1-\delta)$. □

Theorem 3. *Algorithm 1 is a policy-safe modification to the Minami-Kotz distributed proof construction protocol.*

Proof. In [17], the Minami-Kotz distributed proof construction algorithm was proven to construct a proof of authorization only if the integrity policies of every participating entity are satisfied. Since Algorithm 1 does not alter the mechanism through with the proof construction algorithm constructs proof trees, it does not affect the enforcement of any entity's integrity policies. We now show that Algorithm 1 does not affect the enforcement of any entity's confidentiality policies.

In [17], the distributed proof construction algorithm was also shown to construct proof trees only if the confidentiality policies of each participating entity are satisfied. Recall that these confidentiality policies are enforced by optionally encrypting query responses of form $b \in \mathbb{B}$ using a key bound to some entity e higher up the proof tree than the direct querying entity (see Section 2.3). Since the modified query responses of form $\langle b, d \rangle \in \mathbb{B} \times T$ returned by a fact provider using Algorithm 1 can be encrypted in this same manner, each entity's confidentiality policies are still enforced. Therefore, Algorithm 1 makes only query-safe modifications to the underlying distributed proof construction protocol. □

Although Algorithm 1 is sound (by Theorem 2), it is not ideally complete. It could be the case that a certain fact was valid at the time that a query was issued, even if the validity duration reported by the fact provider is less than the query round-trip time; this is an inevitable consequence of the use of casual orderings rather than synchronized clocks. Although not a violation of ideal completeness, this algorithm can also fail in the event that the validity of some fact is not monitored until the first query regarding this fact is issued. Both of these cases make it desirable to have an efficient means of revalidating a given view, as it is likely to be found consistent if rechecked. This leads directly to the stronger notion of interval consistency.

4.3 Interval Consistency

Establishing an interval consistent view typically involves observing that the validity statuses of the facts comprising the view do not change or fluctuate during the course of several observations of portions of the system [14]. In the distributing proving setting, one cannot simply construct a given proof twice to establish an interval of validity, as there would be no guarantee that the values of facts did not fluctuate between proofs or even that the same proof tree was generated for each query.[1] The requery method can succeed, however, if we leverage caches at intermediate nodes to ensure that the same proof tree is constructed at each invocation and that fluctuations can be detected (via cache misses caused by proactive revocations). The intermediate node caches proposed in [18] could be modified to suit this purpose.

Although this modified requery strategy for ensuring interval consistent views is appealing due to its simplicity, it is in fact a worst-case strategy for a number of reasons. First, this strategy requires excessive storage of data at intermediate nodes which is undesirable if nodes wish to remain autonomous. Second, the requery strategy requires that the entire proof tree be traversed twice, even though only the values managed by the leaves of the proof tree are of any significance to whether the proof succeeds; this results in high communication overheads. Lastly, due to reliance on intermediate node caches, a failure of *any* node contributing to the proof tree can cause the revalidation process to fail.

A perhaps more optimal strategy would be to alter the proof construction protocol in such a way as that the querying entity would learn not only whether a proof succeeded, but also a set of *association tuples*, each of which binds the identity of some leaf entity e in the proof tree to a fact identifier that can be used to recheck the status of the fact provided by e. This strategy eliminates the need for intermediate node caches, incurs the lowest possible overall communication overheads during a proof recheck, and fails only if a data-providing entity fails. Even though this leaf exposure strategy is optimal in many respects, it can potentially violate the confidentiality policies of the leaf entities.

[1]Recall that the portions of the proof tree outside of the querier's integrity policies are unknown to the querier; these portions of the proof tree may differ between invocations and go undetected.

In an attempt to balance the efficiency of the leaf exposure strategy with the privacy preservation of the requery strategy, we propose the *leaf indirection strategy* for constructing interval consistent views. As was the case with the query enforcement strategy presented in Section 4.2, we require slight modifications to formats of the entities' knowledge bases and query responses.[2] We will define a knowledge base KB as a subset of $\mathcal{F} \times \{0,1\}^\ell$ in which facts are associated with some locally-unique identifier. It is important that each time a fact is inserted into a knowledge base, it is associated with a previously-unused local identifier in $\{0,1\}^\ell$. Further, an entity e will respond to a query of the form $?f$ with a response tuple of the form $\langle b, (\langle e', id\rangle)_{K_q}\rangle \in \mathbb{B} \times \{0,1\}^n$. The b component of this tuple indicates whether e considers f to be valid, as in the unmodified proof system. The e' and id components of this tuple form an association tuple from the set $\mathcal{E} \times \{0,1\}^\ell$, as described above, though this association tuple is encrypted with the public key of the original querying entity, q. As in the leaf exposure strategy, e may choose to bind herself to the proof tree, in which case $e' = e$ and id is set to the identifier currently associated with f in e's knowledge base. However, e can instead choose a trusted *indirect entity*, ie, at random, obtain a nonce n from ie, and bind ie to the proof tree by setting $e' = ie$ and $id = n$. Each indirect entity maintains a small *remote cache* that associates locally-chosen nonces with ⟨entity, fact identifier⟩ pairs to facilitate the proof recheck process.

The leaf-indirection strategy for constructing interval consistent views is implemented by Algorithm 2. Prior to explaining this algorithm in detail, we first assume that entities have access to the following local data structures and methods, in addition to those required in Section 4.2:

- Each entity maintains a set of locally-trusted indirect entities, $Indirect$.
- The symbol $\leftarrow_r$ denotes random assignment from some set. For example, $e \leftarrow_r Indirect$ chooses a random member from the set of trusted indirect entities.
- An entity's node identifier can be accessed via the local variable ME.
- The function GETFRESHNONCE: $\perp \rightarrow \{0,1\}^\ell$ chooses a previously unused identifier to be associated with some fact or fact provider.

[2]Although the modifications required for the leaf indirection strategy are presented independently of the modifications required for query consistency, this need not be the case. In practice, both sets of modifications can be used together to allow for the creation of either query or interval consistent views.

Algorithm 2 An interval consistency enforcement algorithm

```
// Generate an association tuple for the fact associated with identifier $id$
// to be sent to the initial querying entity $q$
Function GenerateAssociation($id \in \{0,1\}^\ell, q \in \mathcal{E}$)
if $q$ not authorized to learn fact associated with $id$ then
    $e \leftarrow_r Indirect$
    $id' \leftarrow$ InsertRemote($e, id$)
    return $\langle e, id' \rangle$
else
    return $\langle ME, id \rangle$

// Accepts an entry to the $RemoteCache$ table after
// entity $e$ calls InsertRemote
Function InsertAssociation($e \in \mathcal{E}, id \in \{0,1\}^\ell$)
$id' \leftarrow$ GetFreshNonce()
$RemoteCache.insert(id', e, id)$
return $id'$

// Insert a fact state tuple associated with the association record $\langle e, id \rangle$
// bound to a query issued at time $t_{iss}$ into the view $V$.
Function RcvAssoc($\langle e, id \rangle \in \mathcal{E} \times \{0,1\}^\ell, t_{iss} \in T, V \in 2^{\mathcal{S}}$)
$V.insert(id, e, t_{iss}, NOW,$ true$)$

// Recheck the fact tuples making up a view $V$
Function RecheckView($V \in 2^{\mathcal{S}}$)
for all $s \in V$ do
    $t \leftarrow NOW$
    if AskRemote($s.id, s.e$) then
        if $s.fuzzy$ then
            $V.insert(s.id, s.e, s.t_\omega, t,$ false$)$
        else
            $V.insert(s.id, s.e, s.t_\alpha, t,$ false$)$

// Revalidate the fact identified by $id$
Function RecheckFact($id \in \{0,1\}^\ell$)
if $KB.contains(id)$ then
    return true
if $RemoteCache.contains(id)$ then
    $\langle e, id' \rangle \leftarrow RemoteCache.lookup(id)$
    $b \leftarrow$ AskRemote($e, id'$)
    if $\neg b$ then
        $RemoteCache.delete(id)$
    return $b$
else
    return false

// Check the interval consistency condition on a view $V$ relative
// to the time interval $[t_s, t_e]$
Function CheckInterval($V \in 2^{\mathcal{S}}, t_s \in T, t_e \in T$)
for all $s \in V$ do
    if $s.fuzzy \vee (t_s < s.t_\alpha) \vee (s.t_\omega < t_e)$ then
        return false
return true
```

- The local knowledge base is accessible via the data structure KB. The function $KB.contains : \{0,1\}^\ell \to \mathbb{B}$ checks whether the fact associated with a given identifier is currently in the local knowledge base.
- An entity's remote cache is accessible via the $RemoteCache$ data structure. This data structure has member functions $insert : \{0,1\}^\ell \times \mathcal{E} \times \{0,1\}^\ell \to \perp$, $contains : \{0,1\}^\ell \to \mathbb{B}$, $lookup : \{0,1\}^\ell \to \mathcal{E} \times \{0,1\}^\ell$, and $delete : \{0,1\}^\ell \to \perp$.

Algorithm 2 works as follows. An entity contributing a base fact to some proof tree invokes the GenerateAssociation function to generate the association tuple that will be propagated back up the proof tree to the initial querier. If the entity q for whom this association tuple is being prepared is authorized by the local entity's confidentiality policies to learn the value of the fact associated with id, this function binds the local entity to the provided fact identifier. If q is not authorized to learn of the local entity's involvement in the proof process, a randomly-chosen trusted indirect entity is bound to the proof tree via a call to the InsertRemote(e, id) function. This function triggers the execution of the InsertAssociation function at the entity e; InsertAssociation chooses a fresh nonce and binds this to the pair $\langle e', id \rangle$, where e' is the entity who called InsertRemote(e, id). The nonce is then returned to the entity e'. The initial querier thus receives both the proof tree constructed by the algorithm discussed in Section 2 and a list of association tuples binding either leaf entities or indirect leaf entities to the proof tree. Each of these association tuples is used to construct fact state tuples in some view V by using the RcvAssoc function.

The initial querier can then attempt to establish an interval of consistency through one or more calls to the RecheckView(V) function. For each fact state tuple $s \in V$, this function uses the AskRemote function to query the remote entity $s.e$ to see if the fact associated with $s.id$ is still valid. If the fact associated with s is still valid, then the validity interval in s is updated. Fuzzy validity intervals are turned into concrete validity intervals ranging from the end of the fuzzy interval until the time that the recheck was invoked; concrete validity intervals are just extended. The AskRemote(id, e) function works by invoking the RecheckFact(id) function at the entity e. This function returns true if the fact associated with id is still in e's local knowledge base or in the knowledge base of the entity associated with the nonce id in e's remote cache, and returns false otherwise. The function CheckInterval(V) checks to see that $\phi_{interval}(V)$ holds, as defined by Equation 6.

Theorem 4. *If the function* CheckInterval(V, t_s, t_e) *returns true, then V is interval consistent on the interval $[t_s, t_e]$ provided that V was constructed using only calls to the* RcvAssoc *and* RecheckView *functions.*

Proof. The CheckInterval(V, t_s, t_e) function enforces the constraint that all fact state tuples in V encode concrete validity intervals that include at least the interval $[t_s, t_e]$. Therefore, CheckInterval$(V, t_s, t_e) \leftrightarrow \phi_{interval}(V, t_s, t_e)$ which implies that V is interval consistent on the interval $[t_s, t_e]$ by Definition 9, provided that all concrete validity intervals established by RcvAssoc and RecheckView are correct.

RCVASSOC($\langle e, id \rangle, t_{iss}$) inserts a fuzzy validity interval bounded by the query issue time, t_{iss}, and the association tuple receipt time for the fact f described by identifier id. This is a legitimate action, as nonces used by the underlying proof construction protocol (see [17]) allow us to causally link the query and its response and therefore establish that the fact provider asserted f's validity at some time $t \geq t_{iss}$. We must now show that RECHECKVIEW updates these fuzzy intervals correctly.

Assuming that the ASKREMOTE function correctly determines whether a given fact is still true at the providing entity, RECHECKVIEW extends the validity interval for each fact state tuple whose corresponding fact is still valid. Assume that the recheck process for a fact state tuple s corresponding to some fact f starts when $NOW = t_r$ and succeeds. If s encodes the fuzzy validity interval $[s.t_\alpha, s.t_\omega]$, s is updated to encode the concrete validity interval $[s.t_\omega, t_r]$, since f was true at some time $t \leq s.t_\omega$ and was not yet revoked at some later time $t' \geq t_r$. If s encodes a concrete validity interval $[s.t_\alpha, s.t_\omega]$, it is extended to encode the concrete validity interval $[s.t_\alpha, t_r]$. We now show that RECHECKFACT(id)—which is invoked at entity e by the call ASKREMOTE(id, e)—correctly assesses the validity of the fact associated with the identifier id.

We must consider both the case in which the fact f associated with the identifier id was originally stored in e's local knowledge base and the case in which id was an entry in e's remote cache. In the case where f was stored in e's local knowledge base, line 35 of Algorithm 2 returns true if e's knowledge base still contains the fact f associated with id; we know that f has not yet been revoked because the GETFRESHNONCE function ensures that fact identifiers are not reused. If f has since been removed from e's knowledge base, then the call to $KB.contains(id)$ will fail. The check on line 37 will then fail because the state tuple associated with id was originally stored locally and thus would not be associated by GETFRESHNONCE with an entry in e's remote cache. This failure would then cause RECHECKFACT to return false, which implies that RECHECKFACT performs correctly in the case in which the fact f associated with identifier id was originally stored in e's local knowledge base.

We now consider the case in which id was originally associated with an entry in e's remote cache. Line 38 first determines the tuple $\langle e', id' \rangle$ associated with the identifier id in e's remote cache. If this lookup fails, we know that the fact that was indirectly associated with the identifier id has been revoked, as entries in e's remote cache are only removed after failed lookups (by line 41). By reasoning similar to that used above, the call to ASKREMOTE on line 39 will cause RECHECKFACT to return true in the event that the fact f' associated with id' in e''s knowledge base has not yet been revoked. Again, we know that this is not a false positive, as GETFRESHNONCE ensures that fact identifiers are used at most once. If f' has been removed from e''s knowledge base, but not from e's remote cache, this call to ASKREMOTE will return false. This will cause the entry associated with id to be removed from e's remote cache; RECHECKFACT will then return false, as expected.

The fact that RECHECKFACT behaves as expected implies that ASKREMOTE correctly assesses the continuing validity of remotely store facts. This, in turn, implies that RECHECKVIEW correctly updates the validity intervals encoded in the fact state tuples of a given view initially constructed by one or more calls to the RCVASSOC function. Since views can be correctly constructed by calls to the RCVASSOC and RECHECKVIEW functions and we have shown that CHECKINTERVAL(V, t_s, t_e) $\leftrightarrow \phi_{interval}(V, t_s, t_e)$, we can conclude that CHECKINTERVAL(V, t_s, t_e) returns true if and only if V is interval consistent on the interval $[t_s, t_e]$. □

Note that although Algorithm 2 is shown to be sound by Theorem 4, it is not ideally complete. That is, an omniscient entity may have been able to observe an interval consistent view even if Algorithm 2 fails. This can occur because the set of rechecks initiated after the time t_e takes a non-zero amount of time to complete. As with the completeness limitations discussed in Section 4.2, this is an artifact of relying on causal orderings to establish validity intervals, rather than perfectly synchronized clocks. We now show that Algorithm 2 is a policy-safe modification to the underlying distributed proof construction protocol.

Theorem 5. *Algorithm 2 is a policy-safe modification to the Minami-Kotz distributed proof construction protocol.*

Proof. As was the case with Algorithm 1, Algorithm 2 does not affect the construction of distributed proof trees. Therefore, the proof given in [17] stating that proof trees are constructed only if the integrity policies specified by each participating entity are respected still holds. We must now show that the confidentiality policies specified by each entity are still respected. To this end, we must show that no unauthorized entity along the path from the querier q to some fact provider e can learn both the fact f provided by e and f's validity as reported by e. To address the most general case, we will assume that learning e's identity is sufficient for an unauthorized entity to infer f. We then show that (i) each entity participating in the construction of the proof tree that is not entitled to know f cannot learn e's identity, and (ii) entities that do know f but should not learn f's validity cannot infer it.

We first treat case (i) and show that each unauthorized entity u in the proof tree that should not learn f does not learn e's identity. There are two sub-cases: $u = q$ and $u \neq q$. Consider the case where u is an *intermediate entity* in the proof tree; that is, $u \neq q$. In this case, u cannot learn e's identity, as the association tuple that might possibly bind e to the proof tree is encrypted with q's public key, K_q. In the case that $u = q$, we must show that u cannot learn e's identity. In this case, q receives an association tuple binding an indirect entity ie to the fact provided by e. Since ie is trusted by e not to reveal e's identity, q cannot learn the identity of e and thus cannot infer the hidden fact f provided by e.

Note that case (ii) is handled by the encryption of sensitive query responses as described in Section 2.3. Since the incorporation of association tuples into query responses does not affect this encryption process, the proof construction algorithm will correctly enforce the confidentiality of responses as proven in [17]. As we have shown that each entity that should not learn the fact f provided by e cannot learn f and that entities that should not learn f's validity cannot learn it, we can conclude that e's confidentiality policies are correctly enforced. Since both e's confidentiality policies and integrity policies are correctly enforced, we can conclude that Algorithm 2 is a policy-safe modification to the underlying proof system. □

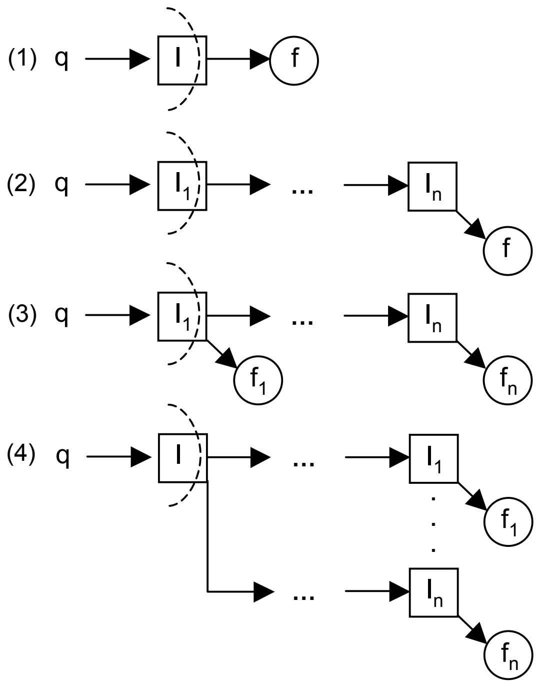

Figure 5: A diagram illustrating several possible proof tree structures. Inference nodes are represented by squares and fact leaves are represented by circles. The dashed lines indicate the expected leaves of the proof tree as perceived by the querier.

Although Algorithm 2 is a policy-safe modification to the Minami-Kotz distributed proof construction framework, it does nonetheless reveal additional information to the initial querying entity q. Specifically, in addition to knowing the portion of the initial proof tree specified by its integrity policies, q learns a list of association tuples declaring certain entities to be (possibly indirect) contributors of atomic facts to the proof tree. We now prove that this list of association tuples gives the q only minimal information regarding the structure of the generated proof tree beyond what is implied by its integrity policies.

Theorem 6. *Given the set of association tuples $\mathcal{A} = \{\langle e_1, n_1\rangle, \ldots, \langle e_i, n_i\rangle\}$ associated with a given proof tree, the initial querier q learns only the number of entities contributing facts to the proof tree, and in some cases, whether the proof tree extends beyond the proof tree implied by their integrity policies.*

Proof. Assume without loss of generality that each entity involved in the construction of a distributed proof makes at most one inference step. Let the set E contain the leaf entities implied by q's integrity policies, henceforth called the set of *expected leaves* of the proof tree. To prove this claim, we must explore two cases: $|E| = |\mathcal{A}|$, and $|E| < |\mathcal{A}|$. If $|E| = |\mathcal{A}|$, we know that each expected leaf node $e \in E$ is either a fact provider or the initiator of a chain of inference forming a linear subproof whose leaf provides a fact to the proof generated by q's query. If e is mentioned explicitly in an association tuple $a \in \mathcal{A}$, then q cannot conclude whether e contributed directly to the proof tree or was chosen as an indirect entity for the actual leaf, e', of the linear subproof initiated by e. These two indistinguishable sub-cases are shown in Figure 5 parts (1) and (2). Note that in Figure 5, each inference node I_k is associated with a unique entity e_k by our prior assumption. If e is *not* explicitly mentioned in any association tuple in $\mathcal{A}$, then q cannot differentiate between the case in which e contributed a fact to the proof tree via an indirect entity and the case in which e initiated a chain of inference resulting in a linear subproof.

If $|E| < |\mathcal{A}|$, then q knows that the proof tree certainly extends beyond the proof tree implied by her integrity policies and that at least one $e \in E$ initiated a subproof contributing multiple facts to the proof. In the event that $|E| > 1$, q cannot infer which entity or entities in E initiated subproofs contributing multiple facts to the proof, as the set of association tuples $\mathcal{A}$ encodes no structural information. If $|E| = 1$, then q clearly knows that the only $e \in E$ initiated a subproof contributing multiple facts to the proof tree. However, q cannot differentiate between linear and branching chains of inference, again, as no structural information is encoded in the set $\mathcal{A}$. This case is illustrated in Figure 5 parts (3) and (4). This shows that q learns no information beyond the number of facts used in the proof tree and whether, in some cases, the proof tree extends beyond the expected leaves of the proof tree. □

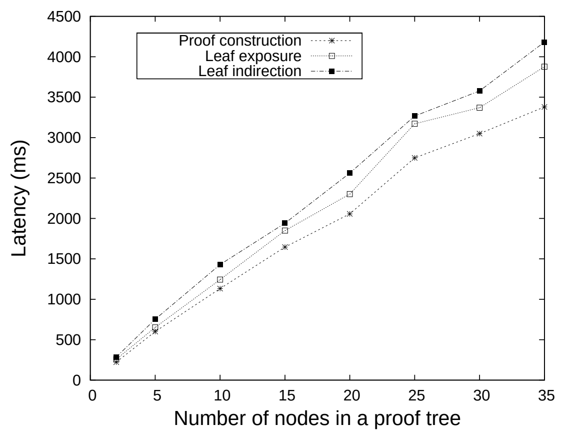

Figure 6: Latency for handling queries.

5 Evaluation

In this section, we measure the performance impact of our consistency enforcement algorithms. The environment in which we ran our tests consisted of a 25 node cluster connected with 100Mbit Ethernet. Each node has a 3.2GHz Intel Pentium D 940 dual-core processor and 2GB RAM, and runs RedHat Linux AS 4 and Sun Microsystem's Java runtime (v1.4.2). Our system has approximately 12,500 lines of Java code, of which about 600 lines represent extensions to the core implementation of the proof construction system described in [18]. We used the Java Cryptographic Extension (JCE) framework to implement RSA and Triple-DES (TDES) cryptographic operations. A 1024-bit public key whose public exponent is fixed to 65537 was used in all of our experiments and the RSA signing operation used MD5 [21] to compute the hash value for each message to be signed. We used Outer-CBC TDES in EDE mode [12] to perform symmetric key operations. The length of our DES keys was 192 bits, and the padding operation in TDES operations conforms to RFC 1423 [5].

During our experiments, we measured the latency of constructing distributed proof trees using two different strategies to ensure interval consistency. These experiments utilized 25 servers, each of which was run by a different principal. Each query issued during our experiments was of the form $?grant(P, R)$ where P is a principal and R is a resource. The body of each rule in any knowledge base is of the form $a_0(c_0), \ldots, a_{n-1}(c_{n-1})$ where each a_i is a predicate symbol and each c_i is a constant. Our experiments attempted to create proof trees containing up to 35 nodes. We believe that proof trees of this size are significantly larger than would be required in most applications, thus, our results should provide guidelines about the worst-case latency for a wide array of practical applications. Authorization, confidentiality, and integrity policies were generated for each of these principals automatically and in such a fashion as to ensure that valid proof trees of the appropriate size could be constructed. For each size of proof tree analyzed during these experiments, measurements were taken during the construction of ten different proof trees of varying internal structure.

Figure 6 compares the query-handling latencies of three different proof construction algorithms; each data point is an average of 50 runs (5 runs for each of the 10 different proof trees generated per proof size). The *proof construction* curve illustrates the cost of generating the proof tree corresponding to some initial query. Note that unless Algorithm 1 is used, no guarantees about the consistency level of the view used to construct these proof trees can be made. The *leaf exposure* curve illustrates the cost of using the leaf exposure strategy to guarantee that proofs are generated using interval consistent views. In this case, the identities of all leaf nodes in the system are forwarded to the initial querier, who can then recheck the validity of each base fact directly. Recall from Section 4.3 that, in many ways, this scenario represents a time-optimal strategy for constructing interval consistent views. The *leaf indirection* curve represents the cost of ensuring that proofs are generated using interval consistent views through the use of the leaf indirection strategy described in Algorithm 2.

Figure 6 shows that these two strategies for enforcing the use of interval consistent views cost little more than generating the initial distributed proof. Specifically, the leaf exposure strategy takes only about 10–15% more time than generating the initial proof tree, while the leaf indirection strategy takes only 25–30% more time than generating the initial proof tree. These results confirm our earlier conjecture that the leaf indirection strategy is a close approximation of the time-optimal leaf exposure strategy. The leaf indirection strategy is also vastly more efficient than the naive requery strategy which, by definition, would require 100% more time than generating the initial proof tree. Although these results depend on our specific implementations of the distributed proof construction and consistency enforcement algorithms, it is still interesting to note that it is possible to recheck proofs *much* faster than they can be constructed; this may lead to the design of more efficient distributed proof engines in the future. These efficiency results combined with Theorems 5 and 6 firmly establish the leaf indirection strategy (as implemented by Algorithm 2) as a low-cost, privacy-preserving method for ensuring the use of interval consistent views during the construction and evaluation of distributed proofs.

6 Related Work

The problem of sampling consistent system views during decentralized authorization protocols was first studied in [14]. This work focused on certificate-based authorization protocols in which the entity enforcing a given policy was assumed to have access to all certificates used during the policy satisfaction process (e.g., [6, 7, 8, 15, 23, 24, 25]). The solutions to the view consistency problem presented in [14] leveraged the semantics of certificate issuance and revocation to build consistent views based on particular orderings of online certificate validity checks; these checks are facilitated through protocols such as OCSP [19] and COCA [26]. We extend these previous results by demonstrating that lightweight view consistency enforcement schemes can also be designed for more general decentralized authorization frameworks in which (i) portions of a proof tree may be hidden from the policy evaluator and (ii) simple assertions authenticated with digital signatures or keyed HMACs are used as proof atoms, rather than CA-issued certificates. These more general authorization frameworks are likely candidates for use in pervasive computing systems and sensor networks.

The Antigone Context Framework (ACF) provides a general-purpose framework for incorporating contextual data into authorization policy enforcement systems [16]. ACF allows policy writers to incorporate contextual assertions into policies without requiring that the policy language include support for obtaining this data from the external world. Users of the ACF can write plug-ins for the framework that obtain this contextual information, which can then be accessed as policies are evaluated. These plug-ins could be used to enforce consistency constrains such as those discussed in this paper, although it is unclear how one would enforce consistency constraints that depend on *all* contextual facts used in a given policy. By contrast, we show several ways in which the underlying proof system can be used to enforce these types of constraints without requiring any involvement by policy writers.

Concurrency control and consistency enforcement in distributed systems [22], distributed databases [9], and distributed shared memory [1] is another area of closely related work. In general, solutions to the consistency problem in these domains assume that multiple entities will be updating values stored at multiple locations within the system and as such, maintaining data consistency is of concern to everyone. Therefore, the solutions presented typically involve the cooperation of multiple entities, as every entity has incentive to cooperate. However, in distributed authorization protocols, there is very little incentive for each autonomous entity participating in the proof construction to take part in complicated consistency preservation protocols, as the consistency of a particular view is only of concern to the policy evaluator. Therefore, the solutions developed in the distributed systems, distributed databases, and distributed shared memory literature are not suitable for our problem domain; the solutions that we develop in this paper require only the cooperation of a minimal number of participants in the protocol.

A final area of related work is the collection of system state snapshots in distributed systems. Collecting consistent snapshots that can be used to evaluate stable predicates over the system state is a well-known problem, to which a solution is presented in [10]. Unfortunately, the unstable nature of fact statuses prevents the use of this algorithm for solving the view consistency problem. There exist algorithms for collecting distributed state snapshots that can be used to evaluate unstable predicates (for a survey, see [3]), though these algorithms have very high overheads and make unreasonable assumptions about process cooperation for our problem domain.

7 Conclusions and Future Work

In this paper, we explored the problem of enforcing consistency constraints on the system views used during policy evaluation in an authorization system based on distributed proof construction. In particular, we focused on enabling the use of consistent system views when evaluating policies within the proof construction framework presented in [17]. This framework complicates the view consistency problem, as the confidentiality and integrity policies declared by entities in the system may render the full details of a given proof tree unavailable to the initial querier. Further, simple signed assertions are used as facts in the system, rather than CA-issued certificates.

Within this framework, we formally defined the view consistency problem and several important levels of view consistency. We then presented efficient algorithms for enforcing two interesting levels of view consistency, proved the soundness of each algorithm, commented on the proximity of these algorithms to ideal completeness, and proved that both algorithms represent policy-safe modifications to the underlying proof system. That is, neither algorithm has any effect on the proper enforcement of confidentiality or integrity policies defined by entities in the system. We then quantitatively evaluated the impact of these algorithms on an implementation the proof system presented in [17]; this impact was found to be minimal. Our solutions generalize previous work on the view consistency problem, which assumed that all assertions used during the proof construction process were encoded in CA-issued certificates and that each assertion used during the protocol was available to the policy evaluator for inspection [14].

One interesting area of future work involves the design of domain-specific view consistency levels. For instance, in pervasive computing environments with rapid contextual changes, the notion of interval consistency defined in this paper may be too strong, as fact validity may fluctuate often around some acceptable baseline; this could lead to situations in which views were repeatedly determined to be inconsistent and cause numerous service interruptions (e.g., consider a policy that is in some way predicated on the number of occupants in a busy hallway). Rather than falling back on the notion of query consistency, which provides no continuing validity checks, entities may wish to enforce a level of view consistency that provides guarantees somewhere between what is afforded by the query and interval consistency levels. For instance, a service provider might wish to enforce the constraint that at each time t, all facts used to justify resource access must have been simultaneously valid at some time t' such that $t - \Delta \leq t' \leq t$, where Δ is the length of a sliding window defined by the service provider. In other cases, an entity may wish to enforce different consistency constraints on different portions of a proof tree. Identifying these types of consistency levels and designing efficient algorithms to enforce their constraints while still respecting each node's autonomy could prove to be an interesting challenge. A related area of future work entails investigating methods for rapidly reevaluating policies or attempting alternate means of policy satisfaction in the event that certain view consistency constraints are not met, as timeliness is an important consideration for most authorization systems.

Acknowledgments

Lee and Winslett were supported by the NSF under grants IIS-0331707, CNS-0325951, and CNS-0524695 and by Sandia National Laboratories under grant number DOE SNL 541065. Minami was supported by the Office of Science and Technology at the Department of Homeland Security. Points of view in this document are those of the author(s) and do not necessarily represent the official position of the U.S. Department of Homeland Security or the Office of Science and Technology.

References

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, pages 66–76, Dec. 1996.

[2] J. Al-Muhtadi, A. Ranganathan, R. Campbell, and D. Mickunas. Cerberus: a context-aware security scheme for smart spaces. In *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*, pages 489–496. IEEE Computer Society, March 2003.

[3] O. Babaoğlu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In S. J. Mullender, editor, *Distributed Systems*, pages 55–96. Addison-Wesley, 1993.

[4] J. Bacon, K. Moody, and W. Yao. A model of OASIS role-based access control and its support for active security. *ACM Transactions on Information and System Security*, 5(4):492–540, 2002.

[5] D. Balenson. Privacy enhancement for internet electronic mail: Part III: Algorithms, modes, and identifiers. IETF RFC 1423, Feb. 1993.

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

[7] M. Y. Becker and P. Sewell. Cassandra: distributed access control policies with tunable expressiveness.

In *Proceedings of the 5th IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 159–168, June 2004.

[8] E. Bertino, E. Ferrari, and A. C. Squicciarini. Trust-$\mathcal{X}$: A peer-to-peer framework for trust establishment. *IEEE Transactions on Knowledge and Data Engineering*, 16(7):827–842, Jul. 2004.

[9] W. Cellary, E. Gelenbe, and T. Morzy. *Concurrency Control in Distributed Database Systems*. Elsevier Science Publishing Company, Inc., 1988.

[10] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, Feb. 1985.

[11] M. J. Covington, W. Long, S. Srinivasan, A. K. Dey, M. Ahamad, and G. D. Abowd. Securing context-aware applications using environment roles. In *Proceedings of the Sixth ACM Symposium on Access Control Models and Technologies*, pages 10–20, 2001.

[12] Data Encryption Standard (DES). Federal Information Processing Standard FIPS PUB 46-3, October 1999.

[13] A. J. Lee, K. Minami, and M. Winslett. Lightweight consistency enforcement schemes for distributed proofs with hidden subtrees. In *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies (SACMAT 2007)*, June 2007.

[14] A. J. Lee and M. Winslett. Safety and consistency in policy-based authorization systems. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, pages 124–133, Nov. 2006.

[15] J. Li, N. Li, and W. H. Winsborough. Automated trust negotiation using cryptographic credentials. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 46–57, 2005.

[16] P. McDaniel. On context in authorization policy. In *Proceedings of the Eighth ACM Symposium on Access Control Models and Technologies (SACMAT 2003)*, pages 80–89, June 2003.

[17] K. Minami and D. Kotz. Secure context-sensitive authorization. *Journal of Pervasive and Mobile Computing*, 1(1):123–156, Mar. 2005.

[18] K. Minami and D. Kotz. Scalability in a secure distributed proof system. In *Proceedings of the Fourth International Conference on Pervasive Computing (Pervasive)*, May 2006.

[19] M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. X.509 internet public key infrastructure online certificate status protocol—OCSP. IETF RFC 2560, June 1999.

[20] G. Myles, A. Friday, and N. Davies. Preserving privacy in environments with location-based applications. *IEEE Pervasive Computing*, 2(1):56–64, January-March 2003.

[21] R. L. Rivest. The MD5 message-digest algorithm. IETF RFC 1321, Apr. 1992.

[22] A. S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2002.

[23] W. Winsborough and N. Li. Towards practical automated trust negotiation. In *Proceedings of the 3rd International Workshop on Policies for Distributed Systems and Networks (POLICY'02)*, page 92, 2002.

[24] M. Winslett, C. C. Zhang, and P. A. Bonatti. Peer-Access: a logic for distributed authorization. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 168–179, 2005.

[25] T. Yu, M. Winslett, and K. E. Seamons. Supporting structured credentials and sensitive policies through interoperable strategies for automated trust negotiation. *ACM Transactions on Information and System Security*, 6(1), Feb. 2003.

[26] L. Zhou, F. B. Schneider, and R. van Renesse. COCA: A secure distributed online certification authority. *ACM Transactions on Computer Systems*, 20(4):329–368, Nov. 2002.