Confidentiality-Preserving Distributed Proofs of Conjunctive Queries (Extended Version)

Adam J. Lee[†] adamlee@cs.pitt.edu
Kazuhiro Minami[‡] minami@cs.uiuc.edu
Nikita Borisov[‡] nikita@uiuc.edu

[†]University of Pittsburgh
[‡]University of Illinois at Urbana-Champaign

Abstract

Distributed proof construction protocols have been shown to be valuable for reasoning about authorization decisions in open distributed environments such as pervasive computing spaces. Unfortunately, existing distributed proof protocols offer only limited support for protecting the confidentiality of sensitive facts, which limits their utility in many practical scenarios. In this paper, we propose a distributed proof construction protocol in which the release of a fact's truth value can be made contingent upon facts managed by other principals in the system. We formally prove that our protocol can safely prove conjunctions of facts without leaking the truth values of individual facts, even in the face of colluding adversaries and fact release policies with cyclical dependencies. This facilitates the definition of context-sensitive release policies that enable the conditional use of sensitive facts in distributed proofs.

1 Introduction

Distributed proof systems allow autonomous agents to reason about their surrounding environment. In these systems, an agent can make inferences using facts stored in its own knowledge base, as well as facts stored in the knowledge bases of others. This ability to derive new local knowledge based on remote facts allows external information defining the context of a given system to be formally included in the decision making process. As a result, recent years have seen the use of distributed proof systems become popular for making authorization decisions in open distributed environments, such as pervasive computing spaces [1, 2, 8, 22, 31]. Unfortunately, these existing protocols offer only limited support for protecting the confidentiality of facts whose release may be sensitive, as is often the case in pervasive environments.

Many distributed proof construction protocols assume that *every* principal is willing to disclose the truth value of *any* fact in its knowledge base to other principals in the system. Some systems improve upon this by allowing the use of identity-based access control lists (ACLs) to limit the disclosure of sensitive facts. However, these ACLs cannot incorporate system context to further limit information flow and are thus insufficient for many application domains. For example, consider a media controller application that controls a projector in a pervasive computing space. The controller may wish to allow access to some projector only to users who are presently located in the same room as the projector. Using identity-based ACLs, a location tracker would need to authorize the media controller to check users' locations, which means a compromise of the media controller would result in violation of users' location privacy. A better alternative would be to give the media controller access to users' locations only when they are requesting access to the projector in the first place.

Motivated by practical scenarios such as the above, we develop a distributed proof construction protocol that allows such context-sensitive release policies, where the disclosure of a fact may be conditional on other facts managed in remote knowledge bases. For example, the disclosure of some fact f in a principal p's local knowledge base may depend upon facts $f'_1, \ldots, f'_n$ that exist in up to n remote knowledge bases. Our protocol allows access to the truth value of f only when the other facts $f'_1, \ldots, f'_n$ are also true *simultaneously* with f. If the conjunction $f \wedge f'_1 \wedge \cdots \wedge f'_n$ is false, then the status of each individual fact remains hidden. We note that in our protocol, p need not be able to access the facts $f'_1, \ldots, f'_n$ itself in order to include them in the release policy for a given fact. In the context of the above example, this implies that the location tracker need not learn that a user is accessing the projector in order to place this

contingency upon the release of the user's location.

In this paper, we make the following contributions:

- We formalize the notion of confidentiality-preserving distributed proof, in which the disclosure of a fact's status can be made contingent upon the simultaneous truth of facts maintained in remote knowledge bases.
- We develop a "best case" trusted third party (TTP) model that embodies the ideal functionality of a confidentiality-preserving distributed proof system.
- We design a distributed algorithm that can be used to construct confidentiality-preserving distributed proofs. We prove that this algorithm reveals no more information than can be learned by interacting with the TTP, even in the face of malicious and colluding protocol participants.
- We show that our protocol functions correctly even if the release policies defined by multiple principals involved in a single proof have circular dependencies.
- We use our TTP model to reason about the types of inferences that can be made regarding confidential facts if concurrent executions or multiple runs of the distributed proof process are allowed. We then show that principals in the system can make local decisions to limit the types of inference that can be made.

The remainder of this paper is organized as follows. In Section 2, we formally define both our system model and the problem of confidentiality-preserving distributed proofs. Section 3 presents cryptographic primitives and a distributed algorithm that can be used to construct confidentiality- preserving distributed proofs. In Section 4, we prove the soundness of this protocol, even in the face of malicious participants. We then explore the types of inferences made possible by concurrent executions or multiple runs of our proof protocol. We show that these risks are minimal and can be greatly reduced through either intelligent release policy design or query rate limiting. Lastly, we compare our approach to other related work in Section 5 and present our conclusions in Section 6.

2 Environment and Definitions

In this section, we first describe the system model that will be used to formally reason about the distributed proof construction process. Within the context of this model, we then present a formal definition of the problem that this paper sets out to solve.

2.1 Basic System Model

We first describe the confidentiality-preserving distributed proof problem by means of an example that we will return to throughout this paper. Figure 1 shows several principals that exist within the same pervasive computing space. In this example, a user Bob wishes to access a digital projector located in a conference room. Other principals in the system include a media controller, *mc*, which mediates all access attempts to shared computing resources in the space; an inventory server, *is*, that keeps track of which resources are owned by which principals; a location service, *ls*, which tracks the locations of principals and devices in the space; and a role server, *rs*, that manages the various roles that can be taken on by principals in the space.

Each principal in the space manages a Datalog knowledge base that contains facts and derivation rules that allow the principal to reason about its surroundings. A Datalog fact is a predicate symbol followed by zero or more terms, where each term is either a lower-case or numerical constant, or a variable (denoted by an upper-case letter). For example, the location server's knowledge base contains the base fact $location(bob, 2124)$, which indicates that Bob is currently located in room 2124. Derivation rules are used to make inferences based on both facts in a principal's own knowledge base, as well as facts that exist in other principals' knowledge bases. For example, the location server's derivation rule $colocated(P_1, P_2) \leftarrow location(P_1, L) \wedge location(P_2, L)$ relies on the use of local facts to determine whether two principals are colocated. By contrast, the media controller's derivation rule $grant(U, P) \leftarrow$ "*ls* says $colocated(U, P)$" $\wedge$ "*rs* says $role(U, presenter)$" uses *quoted facts* to determine whether some user can access a given projector based upon facts in other knowledge bases. Note that P_1, P_2, U and P are free variables that can be bound at runtime to allow these rules to apply to arbitrary principals.

Lastly, we wish to allow principals to control the disclosure of their own local facts to remote parties. This is accomplished by means of access control lists (ACLs) that can be defined by each principal. In general, the ACL for a given fact identifies the conditions under which that fact can be disclosed to certain principals. For example, the role server's access control list indicates that the media controller is allowed to see all facts of the form $role(U, presenter)$, where U is a free variable that can be bound at runtime. In short, the media controller is always allowed to know which principals can take on the role "pre-

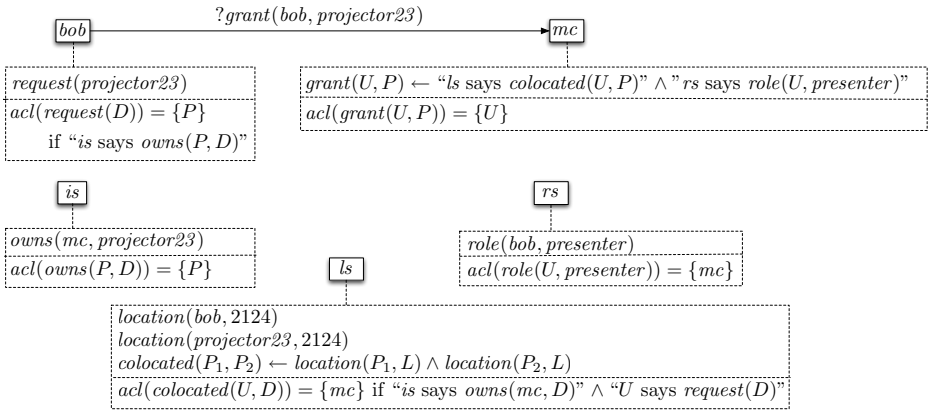

Figure 1: Knowledge bases and ACLs in an example pervasive computing system. Each principal's knowledge bases and ACL are enclosed in a dashed rectangle.

senter." The location server's ACL for facts of the form $colocated(U, D)$ is more restrictive, however, as it will only release these types of facts to the media controller if the inventory server says that the media controller owns the device that will be bound to the free variable D and the user U indicates that they are indeed trying to access the device D. Note that this strictly limits the ability of the media controller to track the locations of users in the system. This is in contrast to systems using only identity-based ACLs (as was discussed in Section 1).

2.2 Formal System Model

We will now more formally define the concepts introduced in Section 2.1. We will let $\mathcal{P}$ represent the set of all principals in the system. We will assume that any two principals $p_i, p_j \in \mathcal{P}$ can establish a private and authenticated communication channel between them by using a PKI or some other method of key distribution.

We use the symbol $\mathcal{F}$ to denote the set of all facts and the symbol $\mathcal{Q}$ to denote the set of all quoted facts of the form "p_i says f_i"; i.e., $\mathcal{Q} = \mathcal{P} \times \mathcal{F}$. Each principal p_i maintains a knowledge base $KB_i \subset \mathcal{F} \cup \mathcal{R}$, containing facts and derivation rules. Since any fact f_i stored in the knowledge base of a principal p_i can be written as the quoted fact "p_i says f_i", we model the set $\mathcal{R}$ of derivation rules as Horn clauses whose head is a fact and whose body is a conjunction of quoted facts; i.e., $\mathcal{R} = \mathcal{F} \times 2^{\mathcal{Q}}$. The ACL protecting facts managed by principal p_i is represented as a function $acl_{p_i} : \mathcal{F} \rightarrow 2^{2^{\mathcal{P}} \times 2^{\mathcal{Q}}}$ that maps a given fact in $f \in KB_i$ to a set of $(plist, qlist)$ pairs. Each $plist$ identifies a set of principals who can access f if and only if they can also verify that each q_i in the corresponding $qlist$ is also true. For example, the ACL defined by Bob in Figure 1 allows any principal P to access facts of the form $request(D)$, provided that the inventory server asserts that principal P is the owner of the device D.

In the remainder of this work, we assume that the construction of distributed proofs will take place in an asynchronous distributed system and thus we place no limitations on the temporal duration of the distributed proof construction process. Furthermore, we assume that each principal p_i acts autonomously and can add or remove facts from its knowledge base KB_i at any time *without* notifying other principals in the system. Lastly, we do not assume that any level of clock synchronization exists between the principals in $\mathcal{P}$.

2.3 Confidentiality-Preserving Distributed Proof

The high-level goal of confidentiality-preserving distributed proof is maximize the utility of a distributed proof environment while minimizing the amount of sensitive information disclosed by principals in the system. Since access control policies are often considered sensitive (e.g., see [20, 33]), we assume that ACLs are not publicly available. However, we do assume that for any two principals p_i and p_j, p_j can access the portions of p_i's ACL that are relevant to p_j. We call this the *restriction* of p_i's ACL relative to principal p_j.

Definition 1 (Restricted ACL)**.** *For any two principals p_i and p_j, the restriction of p_i's ACL relative to p_j is represented as the function $\widehat{acl_{p_i,p_j}} : \mathcal{F} \rightarrow 2^{2^{\mathcal{Q}}}$, which is defined as:*

$$\widehat{acl_{p_i,p_j}}(f) \equiv \{qlist \mid (plist, qlist) \in acl_{p_i}(f) \wedge p_j \in plist\}$$

Intuitively, the restricted ACL $\widehat{acl_{p_i,p_j}}(f_i)$ specifies a set of dependencies used to control the disclosure of the quoted fact "p_i says f_i" to principal p_j. However, each of these dependencies may also have other dependencies. To address this, p_j can use repeated application of other principals' restricted ACLs to compute all possible sets of quoted facts upon which the disclosure of "p_i says f_i" depends. For example, in the scenario depicted in Figure 1, the media controller can use $\widehat{acl_{ls,mc}}$ to determine that the disclosure of the fact "*ls* says $colocated(bob, projector23)$" depends upon the truth of facts "*is* says $owns(mc, projector23)$" and "*bob* says $request(projector23)$". The media controller can then check $\widehat{acl_{is,mc}}$, and $\widehat{acl_{bob,mc}}$ to verify that no other facts need to be incorporated in the conjunction to be proved.

As a result of the above observation, it is natural to formally define the ideal functionality of a confidentiality- preserving distributed proof system in terms of a trusted third party (TTP) that can be used as an oracle that proves conjunctions of facts. Prior to specifying this functionality, we first define the predicates $checkAcls : \mathcal{P} \times 2^{\mathcal{Q}} \rightarrow \mathbb{B}$ and $oracle : 2^{\mathcal{Q}} \rightarrow \mathbb{B}$ as follows.

$$\begin{aligned} checkAcls(p_0, conj) \quad &\equiv \quad \forall \text{“}p_i \text{ says } f_i\text{”} \in conj, \quad (1) \\ &\quad\ \exists (plist, qlist) \in acl_{p_i}(f_i) : \\ &\quad\ p_0 \in plist \wedge qlist \subseteq conj \\ oracle(conj) \quad &\equiv \quad \forall \text{“}p_i \text{ says } f_i\text{”} \in conj : \quad (2) \\ &\quad\ f_i \in KB_i \end{aligned}$$

For a given principal p_0 trying to prove the conjunction of quoted facts *conj*, the predicate $checkAcls(p_0, conj)$ is true if and only if allowing p_0 access to any "p_i says f_i" $\in$ *conj* does not require access to any quoted fact "p_k says f_k" $\notin$ *conj*. The predicate $oracle(conj)$ is satisfied if and only if each "p_i says f_i" $\in$ *conj* exists in the appropriate knowledge base. We now define the ideal functionality of a confidentiality-preserving distributed proof system.

Definition 2 (Ideal Functionaliy)**.** *A confidentiality-preserving distributed proof system can be modeled as a trusted third party that executes conjunctive queries on behalf of the principals in the system. When a principal p_0 wishes to prove the conjunction $C =$ "p_1 says f_1" $\wedge \cdots \wedge$ "p_n says f_n", the TTP takes the following actions:*

- *For each "p_i says f_i" $\in C$, notify p_i that p_0 has an interest in p_i's fact f_i.*
- *If $checkAcls(p_0, C)$ is* false*, send a query failure message to p_0. This failure message is distinct from the value* false*.*
- *If $checkAcls(p_0, C)$ is* true*, reveal the value of $oracle(conj)$ to p_0.*

Note that the above definition of a confidentiality-preserving distributed proof system entails several desirable properties. First, it allows each principal's ACL to remain private, as only access to restricted ACLs is needed to determine the minimal conjunction associated with a given fact of interest. Second, the truth value of a conjunction is revealed if and only if the ACLs of each fact within the conjunction are satisfied. Third, if the conjunction is false, the querier does not learn any information about the facts comprising the conjunction. Fourth, fact providers learn which individuals are interested in the facts in their knowledge bases, which allows them to audit usage patterns. Lastly, fact providers learn *neither* what conjunctions are being proved *nor* the status of other facts in the system, and thus cannot infer portions of other principals' ACLs. In the remainder of this paper, we describe how this functionality can be achieved in proof systems designed for use in asynchronous distributed environments.

3 Proof Construction Protocol

In this section, we describe our confidentiality-preserving distributed proof protocol. We begin with an overview of our protocol, which motivates discussion of the cryptographic primitives that will be used by participants to achieve the properties described in the previous section. We then present the algorithmic details of our protocol.

3.1 Protocol Intuition

Our confidentiality-preserving distributed proof protocol operates in two distinct phases. Assume that some querier p_0 wants to prove the conjunction of facts $conj =$ "p_1 says f_1" $\wedge \cdots \wedge$ "p_n says f_n". In the first phase of the protocol, the querier contacts each p_i to indicate an interest in their fact f_i. Each p_i generates a random number $s_{i,j}$ (henceforth called a *share*) for each "p_j says f_j" in $\widehat{acl_{p_i,p_0}}(f_i)$, encrypts each random share to its corresponding p_j, and returns the set of encrypted shares to the querier. Principal p_i then generates a final share, s_i, which is

stored locally, such that the product of all shares is 1.

During the second phase of the protocol, p_0 homomorphically combines all of the shares encrypted for a particular principal into a single share; this hides the number of principals whose release policies depend on a given p_i's fact f_i from non-colluding principals participating in the protocol. Additionally, p_0 adds a blinding factor to each share in order to ensure that colluding principals cannot determine the number of principals whose release policies depend on the fact "p_i says f_i". Each resulting encrypted share is then forwarded to the corresponding principal p_i. Principal p_i decrypts this ciphertext and combines it with the local secret share s_i that it generated during phase one of the protocol. If fact f_i was true from the time that phase one ended until the time that phase two began, p_i will return this newly-generated result. Otherwise, a random value will be returned. If the product of the results returned by each p_i equals the product of the blinding factors generated by p_0, then p_0 can determine that each "p_i says f_i" $\in$ *conj* was simultaneously true at some time t during the protocol. If *any* p_i returns a random result, p_0 will determine that *conj* is false, but cannot recover any other information regarding the truth of subclauses of *conj*. We now present a cryptographic construction that allows us to build such a protocol, and then discuss the full details of our confidentiality-preserving distributed proof construction protocol.

3.2 Cryptographic Foundations

At a first glance, the encryption scheme can be any public key scheme that supports homomorphic operations, such as ElGamal [11] or Pailler [24]. But notice that when p_i encrypts a share s_j for principal p_j, this occurs because p_i has an ACL which includes the clause "if p_j says f_j". However, p_i has no guarantee that p_0 will not forward the share to p_j while asking about some other fact, f'_j, thus violating the ACL constraints. Therefore, p_i needs to bind its encrypted share to the particular fact f_j. This could be done by encrypting $E_{K_j}(s_j||f_j)$ and using a non-malleable encryption scheme [9]. Non-malleability ensures that p_0 could not change an encryption of $s_j||f_j$ to an encryption of $s_j||f'_j$. However, a non-malleable encryption by definition cannot be homomorphic and thus cannot satisfy our needs.

Instead, we use a homomorphic identity-based encryption (IBE) scheme. An IBE scheme has a master public key (MPK) that can be used to encrypt to a given user ID, without knowing the user's specific public key: $\mathsf{Enc}(M, MPK, ID)$. The master secret key can be used to derive (extract) a private key for a particular user, which can then be used to decrypt all messages destined for that ID. In our case, rather than having a single master key, with identities referring to individual principals, we have each principal create a master key for an IBE scheme, with the *facts* in its knowledge base acting as identities. In other words, to encrypt a share for p_j, p_i computes $\mathsf{Enc}(s_{i,j}, K_j, f_j)$, where K_j is the master public key of principal j. This binds the encrypted share to the fact f_j.

The scheme we use is a homomorphic variant of the Boneh–Franklin IBE scheme [6], originally proposed by Ivan and Dodis [13]. The primitive is based on a bilinear mapping $e : G_1 \times G_1 \rightarrow G_2$ that satisfies the Bilinear Diffie–Hellman assumption [6]. We let g be a generator of G_1, and use a hash function $h : \{0,1\}^* \rightarrow G_1$ to define identities.The scheme is defined as follows:

Enc-Gen: Generate a secret key $s \in_R G_1$ and a master public key g^s.

Extract(ID)**:** Extract the secret key $h(ID)^s$.

Enc(M, g^s, ID)**:** $(g^r, M \cdot e(h(ID)^r, g^s))$, for a random r.

Dec(U, V, ID)**:** $V/e(h(ID)^s, U) = M$.

$\otimes$**:** $(U, V) \otimes (U', V') = (U \cdot U', V \cdot V')$

This homomorphism over the operator $\otimes$ works because the following two equations hold:

$$U \cdot U' = g^r \cdot g^{r'} = g^{r+r'} \quad (3)$$

$$\begin{aligned} V \cdot V' &= M \cdot e(h(ID)^r, g^s) \cdot \\ &\quad M' \cdot e(h(ID)^{r'}, g^s) \\ &= M \cdot M' \cdot e(h(ID), g^s)^r \cdot e(h(ID), g^s)^{r'} \\ &= M \cdot M' \cdot e(h(ID), g^s)^{r+r'} \\ &= M \cdot M' \cdot e(h(ID)^{r+r'}, g^s) \end{aligned} \quad (4)$$

Theorem 1. *The above Homomorphic IBE cryptosystem is IND-ID-CPA secure.*

The theorem can be proven by following the same argument as the original Boneh–Franklin IBE scheme [6]. This means that an adversary who has a ciphertext encrypted for a certain identity *ID* can perform key extraction queries for any number of identities $ID' \neq ID$, and yet is unable to infer anything about the plaintext. Note that the scheme does not (and cannot) offer chosen ciphertext security (IND-ID-CCA) because it allows homomorphic

operations. To address this issue, we add a session identifier, *sid*, to the fact name as the identity for encryption. In other words, each p_i produces $\mathsf{Enc}(s_j, K_j, p_0||f_j||sid)$. This ensures that while the underlying cryptography is not secure against chosen-ciphertext attacks, our protocol can guarantee that a principal will only decrypt a single message for a given identity, thereby rendering chosen ciphertext attacks impossible.

3.3 Protocol Details

We now describe the details of our confidentiality-preserving distributed proof construction protocol. In the remainder of this paper, we assume that each fact f_i in a principal p's knowledge base KB is associated with an identifier from the set $\{0,1\}^\ell$. Any time that a fact's status changes, its corresponding identifier will be changed at random. As we will see later in this section, this notion of fact identifiers will be used to allow queriers to ensure that all facts comprising the conjunction being proved were valid simultaneously during the proof construction protocol. Prior to explaining the full details of our protocol—which is presented in Algorithms 1 and 2—we first note the following:

- Each principal p_i acts as a key generator for an instance of the IBE scheme described in Section 3.2. The master public key of this instance of the IBE system are then made available to other principals in the system using the same mechanism used to distribute p_i's public key K_i.
- The sets M and C represent the message and ciphertext spaces of the IBE cryptosystem, respectively. (G_2 and $G_1 \times G_2$.)
- The symbol $\leftarrow_R$ denotes random assignment from some set. For example, $sid \leftarrow_R \mathbb{Z}_{2^{128}}$ chooses a random session identifier from the set of all 128-bit integers.
- The identifier *ME* is used as a placeholder for a given principal's unique identifier.
- The notation $E^{p_i}_{p_j||f_i||sid}(x)$ is used to denote the encryption of item x to principal p_i relative to the IBE key derived from p_j's identity, the fact f_i, and the session ID *sid*. Similarly, $D^{p_i}_{p_j||f_i||sid}(c)$ represents the decryption of the ciphertext c by principal p_i using the IBE key derived from p_j's identity, the fact f_i, and the session ID *sid*.

Algorithm 1 Functions used by the querier to build a proof.

```
1: // Phase 1: Indicate interest in variables in conj, collect encrypted shares.
2: Function StartQuery(conj ∈ 2^Q)
3: sid ←_R Z_{2^128}; expected ← 1; encShares ← ∅; contacted ← ∅
4: for all "p_i says f_i" ∈ conj do
5:     r ←_R M
6:     expected ← expected × r
7:     encShares(p_i, f_i) ← E^{p_i}_{ME||f_i||sid}(r)
8:
9: for all "p_i says f_i" ∈ conj do
10:    Choose depends ⊆ conj such that depends ∈ acl^_{p_i,ME}(f_i)
11:    resp ← Ask(p_i, f_i, depends, sid)
12:    if resp = ERROR then
13:        for all (p_k, f_k) ∈ contacted do
14:            RecoverShare(p_k, f_k, sid, encShares(p_k, f_k))
15:        return FAIL
16:    contacted.add(p_i, f_i)
17:
18:    // Update encrypted secret shares
19:    for all (p_j, f_j, E^{p_i}_{ME||f_j||sid}(s_j)) ∈ resp do
20:        encShares(p_j, f_j) ← encShares(p_j, f_j) ⊗ E^{p_j}_{ME||f_j||sid}(s_j)
21: return (sid, expected, encShares)
22:
23: // Phase 2: Send out decryption requests for encrypted shares
24: Function EndQuery(sid ∈ Z_{2^128}, expected ∈ M,
    encShares ∈ 2^{P×F×C})
25: share ← 1; failed ← false
26: for all (p_i, f_i, c_i) ∈ encShares do
27:     r ← RecoverShare(p_i, f_i, sid, c_i)
28:     if r = ERROR then
29:         failed ← true
30:     else
31:         share ← share × r
32:
33: // Return value
34: if failed then
35:     return FAIL
36: else
37:     return (share = expected)
```

Phase One: Share gathering

To initiate the confidentiality-preserving distributed proof process, the querier p_0 runs the code listed in Algorithm 1. The function StartQuery(*conj*) is invoked whenever p_0 wants to learn the status of some conjunction of quoted facts $conj \in 2^{\mathcal{Q}}$ managed in one or more remote knowledge bases. The querier first generates a random session identifier[1] and creates a random blinding factor for each quoted fact "p_i says f_i" $\in$ *conj*. Each random blinding factor is homomorphically encrypted using the session identifier *sid*. These encrypted blinding factors are then stored in the *encShares* table, which is used to keep track of the secret shares collected during phase one of the protocol. For each quoted fact "p_i says f_i" $\in$ *conj*, the querier then uses $\widehat{acl}_{p_i,p_0}$ to determine the subset of *conj* representing the release policy for "p_i says f_i"; in Algorithm 1, this subset

[1] $\mathbb{Z}_{2^{128}}$ is chosen to make the odds that two sessions will use the same identifier negligible.

is referred to as *depends*. The remote procedure call (RPC) stub ASK is then used to inquire about the status of f_i at p_i. All RPC communications are carried out over private and authenticated channels.

Principal p_0's call to $\text{ASK}(p_i, f_i, depends, sid)$ then invokes the function $\text{ASKRESPONSE}(p_0, f_i, depends, sid)$ at the principal p_i (see Algorithm 2). If p_0 has already asked about the status of fact f during session *sid* or if $depends \notin \widehat{acl_{p_i,p_0}}(f_i)$, this function raises an error. If the request is legitimate, p_i generates a random secret share for each "p_j says f_j" $\in depends$. Each share is homomorphically encrypted to its corresponding p_j using the key whose identifier is generated by concatenating the querier's identifier p_0, the fact f_j, and the session identifier *sid*. A final secret share is then generated such that the product of all secret shares is 1, and is saved in the *shares* table.

If f_i is a fact in p_i's extensional knowledge base (i.e., it is a base fact, not the head of some rule $r \in \mathcal{R}$), the current truth value of f_i and its corresponding fact identifier are retrieved from p_i's knowledge base and stored in the *ids* table; in the second stage of the protocol, p_i will use this information to determine whether the truth value of f_i changed during the protocol. If, on the other hand, f_i is part of p_i's intensional knowledge base (i.e., f_i is the head of a derivation rule), p_i executes the first stage of the distributed proof process recursively using the STARTQUERY function. The intermediate results returned by STARTQUERY are stored in the *proofState* table. At this point, p_i returns the list of quoted facts and secret shares to p_0.

If p_0's call to $\text{ASK}(p_i, f_i, depends, sid)$ returns the code ERROR, the STARTQUERY function will fail and p_0 will ask all previously contacted principals to decrypt the facts that they had previously disclosed. This ensures that the failed protocol execution appears indistinguishable from a successful protocol execution to all fact providers previously contacted. Otherwise, p_0 makes a note in the *contacted* table stating that principal p_i was successfully contacted regarding the fact f_i. The principal p_0 then uses the homomorphic property of the IBE cryptosystem to combine the secret shares returned by p_i with any corresponding secret shares or blinding factors already stored in the *encShares* table. At this point, each principal contributing a fact to *conj* has been contacted by p_0 and has set up the local state necessary to execute phase two of the protocol.

Algorithm 2 Functions used by fact providers to (conditionally) contribute information regarding locally-maintained facts to a proof being constructed by another principal.

```
1: // This function is used by principals to respond to queries
2: // issued using the ASK function.
3: Function AskResponse(p_0 ∈ P, f ∈ F, depends ∈ 2^Q,
   sid ∈ Z_{2^128})
4: // Make sure that this is a fresh request and p_0 is
5: // (conditionally) authorized to see f
6: if (shares(p_0, sid, f) ≠ ⊥) ∨ (depends ∉ acl^~_{ME,p_0}(f)) then
7:     return ERROR
8:
9: resp ← ∅; product ← 1
10: for all "p_j says f_j" ∈ depends do
11:     s_j ←_R M
12:     product ← product × s_j
13:     resp.add(p_j, f_j, E^{p_j}_{p_0||f_j||sid}(s_j))
14: Choose s such that s × product = 1
15: shares(p_0, sid, f) ← s
16:
17: if KB.managesFact(f) then
18:     (status, id) ← KB.lookup(f)
19:     ids(p_0, sid, f) ← id
20: else
21:     Use derivation rules and restricted ACLs to find a
        conjunction, conj, of all facts and dependencies needed to
        derive f
22:     proofState(p_0, f, sid) ← StartQuery(conj)
23: return resp
24:
25:
26: // This function is used by principals to respond to queries
27: // issued using the RecoverShare function.
28: Function RecoveryResponse(p_0 ∈ P, f ∈ F, sid ∈ Z_{2^128},
    c ∈ C)
29: // Make sure that this is a fresh request
30: if (shares(p_0, sid, f) = ⊥) ∨ (decrypted.contains(p_0, sid, f))
    then
31:     return ERROR
32:
33: d ← D^{ME}_{p_0||f||sid}(c)
34: s ← (d × shares(p_0, sid, f)); decrypted.add(p_0, sid, f)
35: if KB.managesFact(f) then
36:     (status, id) ← KB.lookup(f)
37:     if ¬status ∨ (id ≠ ids(p_0, sid, f)) then
38:         s ←_R M such that s ≠ d × shares(p_0, sid, f)
39: else
40:     if (proofState(p_0, f, sid) = FAIL) ∨
        ¬EndQuery(proofState(p_0, f, sid)) then
41:         s ←_R M such that s ≠ d × shares(p_0, sid, f)
42: return s
```

Phase Two: Consistency check and secret recovery

To start phase two of the protocol, p_0 invokes the ENDQUERY function using the intermediate state returned by the STARTQUERY function. This function uses the RECOVERSHARE RPC stub to ask each fact provider to decrypt the aggregate secret shares that p_0 has accrued. This invokes the RECOVERYRESPONSE function at each fact provider. If the fact provider has previously decrypted any other results related to this (p_0, sid, f_i) triple, an error is raised. If it is safe to proceed, the fact provider p_i decrypts the ciphertext provided by the querier, combines the result with the locally-stored secret share previously

generated, and records the fact that it has now decrypted a secret share for the session described by the triple (p_0, sid, f_i). If the fact f_i is part of p_i's extensional knowledge base and either f_i is false or the identifier associated with f_i has changed since the AskResponse function was invoked, then the previously-computed secret share is replaced with a random value. Similarly, if f_i is part of p_i's intensional knowledge base and either the first stage of the distributed proof process fails or the second stage of the distributed proof process determines that f_i is false, then the previously-computed secret share is replaced with a random value. The secret share (or random value) is then returned to the querier.

If the product of all the responses gathered from each fact provider is equal to the product of the blinding factors generated prior to stage one of the protocol, the querier can determine that each fact in the conjunction *conj* was simultaneously true at some point during the protocol execution. If *any* fact provider returns a random value during stage two of the protocol, the value computed by p_0 will be randomized and *conj* will be determined to be false.

Note that principals must remember all session identifiers they have previously observed. If loose clock synchronization is available, the amount of state to be maintained can be reduced by using the current time for the session identifier. If Δ is the largest expected difference between any two principals' clocks, a principal can remove from the *shares* table any sessions that are more than Δ old, and simultaneously forbid any queries that use a session identifier that is more than Δ time ago. Similarly, a principal that has lost its previous state can ensure security by refusing to answer queries for the first Δ period of time after it is back up.

3.4 An Example

We now revisit the example depicted in Figure 1 of Section 2.1 to explain in more detail an example execution of the protocol described above. In the interest of space and clarity of presentation, we consider only a subset of this scenario in which the media controller wishes to prove "*bob* says *request*(*projector23*)". After using its access to $\widehat{acl_{bob,mc}}$ and $\widehat{acl_{is,mc}}$, the media controller learns that it must actually prove the conjunction of quoted facts "*bob* says*request*(*projector23*)$''$ $\wedge$ "*is* says *owns*(*mc*, *projector23*)$''$.

Figure 2 depicts the execution of phase one of the protocol from Section 3.3, using f_{bob} and f_{is} as abbreviations for *request*(*projector23*) and *owns*(*mc*, *projector23*), respectively. During this stage of the protocol, the media controller first generates two blinding factors b_{bob} and b_{is}. The blinding factor b_{bob} (resp. b_{is}) is then stored in the *encShares* table after being encrypted to Bob (resp. the inventory server) using a key bound to the media controller's identifier, the fact *request*(*projector23*) (resp. *owns*(*mc*, *projector23*)), and the session identifier *sid*. The media controller then uses the Ask function to notify Bob and the inventory server that it is interested in the status of certain facts stored in their respective knowledge bases.

Since Bob's disclosure of the fact *request*(*projector23*) depends on the status of the fact *owns*(*mc*, *projector23*) in the knowledge base of the inventory server, Bob generates a secret share $s_{bob,is}$ that is then encrypted to the inventory server using a key derived from the media controller's identifier, the fact *owns*(*mc*, *projector23*), and the session identifier *sid*. Bob then generates a secret share s_{bob} such that $s_{bob} \times s_{bob,is} = 1$. The secret share s_{bob} is stored locally in the *shares* table, while the encrypted share for the inventory server is disclosed to the media controller. This encrypted secret share is then homomorphically combined with the inventory server's encrypted blinding factor that is stored in the *encShares* table. Since the inventory server's disclosure of the fact *owns*(*mc*, *projector23*) has no dependencies, it stores the value 1 in its *shares* table and discloses an empty set of dependencies to the media controller. At this point, the media controller updates its *contacted* list and phase one of the protocol is complete.

During stage two of the protocol, the media controller asks Bob to decrypt the value $E^{bob}_{mc||f_{bob}||sid}(b_{bob})$ and asks the inventory server to decrypt the value $E^{is}_{mc||f_{is}||sid}(b_{is}s_{bob,is})$. Both principals decrypt the requested values and combine them with the secret shares stored in their *shares* tables. The values $b_{bob}s_{bob}$ and $b_{is}s_{bob,is}$ are then returned to media controller by Bob and the inventory server, respectively. Since $b_{bob}s_{bob}b_{is}s_{bob,is} = (b_{bob}b_{is})(s_{bob,is}s_{bob}) = b_{bob}b_{is}$, which is the expected value generated by the media controller prior to the start of the protocol, the media controller can conclude that the conjunction "*bob* says *request*(*projector23*)" $\wedge$ "*is* says *owns*(*mc*, *projector23*)" is true.

4 Discussion

In this section, we formally discuss the properties of the confidentiality-preserving distributed proof protocol that was presented in Section 3. We begin

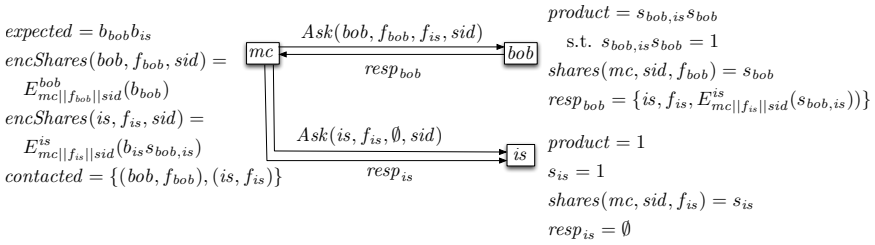

Figure 2: A sample execution of phase one of the protocol.

by discussing the soundness and completeness of this protocol, and conclude by examining limitations of confidentiality-preserving distributed proof in scenarios involving multiple concurrent runs.

4.1 Protocol Characteristics

We begin our treatment of the formal properties of our confidentiality-preserving distributed proof protocol by making the following claim regarding the correctness of this protocol. A full proof of this claim can be found in Appendix A.

Theorem 2 (Soundness)**.** *Given a conjunction of quoted facts conj* $\in 2^{\mathcal{Q}}$*, if* EndQuery(StartQuery(*conj*)) *returns* true*, then there exists some time t during the execution of the protocol at which every quoted fact "*p_j says f_j*" ∈ conj was simultaneously* true*.*

Although above theorem asserts the soundness of this proof construction protocol, it is not complete. That is, if EndQuery(StartQuery(*conj*)) returns false, there could still have existed some time t during the execution of the protocol at which each quoted fact in *conj* was simultaneously true. In the general case, designing a sound and complete distributed proof system requires synchronized clocks, which is contrary to the definition of an asynchronous system. Due to space limitations, we do not prove this claim, but instead refer the reader to [16] where a similar claim was proven for trust negotiation systems (e.g., [30]), which are a specific type distributed proof system. However, we can make a more limited claim regarding the soundness and completeness of our proof construction protocol, the proof of which can be found in Appendix B.

Theorem 3 (Stable Proof Behavior)**.** *Assume that a querier p_0 wants to prove the conjunction of facts conj, $\widehat{acl_{p_i,p_0}}(f_i) \subseteq$ conj for each "p_i says f_i" ∈ conj, and there exists a well-formed proof graph for the conjunction of facts conj whose leaves are stable between times t_1 and t_2. In this case,* EndQuery(StartQuery(*conj*)) ↔ *conj if* StartQuery *is invoked after time t_1 and* EndQuery *returns before time t_2.*

While completeness cannot be established in the general case, the above theorem shows that the protocol presented in this paper is both sound and complete under realistic assumptions. This is in contrast to a trivial protocol that always returns false. Such a protocol is sound, but cannot be shown to be complete under *any* realistic assumptions.

Finally, we note that our protocol preserves the necessary constraints from the ideal functionality. We do this through a series of theorems, the proofs of which can be found in Appendix C.

Theorem 4 (Query Privacy)**.** *During the construction of a distributed proof, a malicious subset of providers $p_1, \ldots, p_m$ will learn which local facts each p_i provides, but will learn nothing about the other facts comprising the conjunction being proved.*

Theorem 5 (Query Validity)**.** *During the construction of a distributed proof, given a subset of malicious fact providers, $p_1, \ldots, p_m$, if one of the honest fact providers, p_i, provides a* false *fact, the conjunction received by the querier p_0 is* false*.*

Note that a malicious fact provider can always *falsify* a conjunction by simply performing the protocol as if its fact is false. However, the same is true inside the TTP model.

Theorem 6 (Limited Disclosure)**.** *During the construction of a distributed proof, a malicious querier p_0, colluding with set of fact providers, $p_1, \ldots, p_m$, learns the same amount of information as it would by interacting with the TTP functionality, as long as the ACLs of honest fact providers form a strongly-connected component.*

Here, we consider ACLs to form a graph, where if p_i's ACL includes the condition p_j says f_j, then there is a directed arrow from p_i to p_j. Note that our protocol provides a form of limited disclosure when

the ACLs do not form a strongly-connected component, but we will reserve the discussion of the exact security property to Section 4.3.

The above theorems guarantee that, as long as the strongly-connected condition is satisfied, the protocol presented in Section 3 provides principals with the same guarantees afforded by the ideal functionality discussed in Section 2.3. Namely, each principal's ACLs remain private, the truth value of a conjunction is revealed if and only if each fact comprising the conjunction is *simultaneously* valid, and fact providers learn neither what conjunctions are being proved nor the status of other facts in the system.

4.2 Concurrency and Multiple Runs

It is important to note that the ideal functionality described by Definition 2 is concerned with the construction of a *single* proof. As a result, the semantics of a confidentiality-preserving distributed proof treat only the confidentiality of facts within a single run of any given proof protocol. This implies that although one failed proof does not leak any information regarding the status of the facts making up the conjunction being proved, it may be possible to use information collected during *multiple* proofs to infer the status of sensitive facts. For example, consider the case in Figure 3, in which the disclosure of the quoted fact "p_1 says f_1" depends upon the quoted fact "p_2 says f_2". If a proof in the TTP model fails, the querier will learn only that the conjunction "p_1 says f_1" $\wedge$ "p_2 says f_2" is false. However, if the same principal then attempts to prove "p_2 says f_2" and succeeds, he can conclude that "p_1 says f_1" was likely false during his previously attempted proof.

Notice that in this case, the ACLs are *weakly connected*, and thus Theorem 6 says nothing about the security of the query. However, as we will show in Section 4.3, even when ACLs are not strongly connected, a single run of our protocol reveals only as much information as several *non-overlapping* queries of the TTP (i.e., queries that do not share facts). Since combining multiple non-overlapping queries into one does not offer a querier any advantage, for the rest of this sub-section we will assume that the querier is only able to learn the truth value of a single conjunction during each run of the protocol.

Note that the above types of leakage scenario are similar to attacks in which individual records in an "anonymized" data source are identified by combining access to the anonymized data with external knowledge (e.g., see [14] and [23]). Recently, a number of syntactic [17, 21, 28, 32] and semantic [10] approaches have been proposed to limit these types of inferences. However, the inference problem in distributed proof systems differs from this existing work in two important ways: (i) fact status values change over time and (ii) *principals* in the system control the functional dependencies used to infer data (i.e., ACL constraints). These two important differences from the more traditional data anonymization domain can be leveraged to develop realistic inference mitigation approaches.

Strict Reference Monitors The only way to strictly prevent inferences made using the observations from multiple distributed proofs is to partition the set of allowable proofs based upon previous history. This can be accomplished by keeping track of previous queries. Each principal p_i can locally enforce the constraint that no other principal p_j is allowed query the status of fact $f_i \in KB_i$ more than once. This prevents inference, but it does so at the cost of making the proof system less useful over time. Note that if we want to guard against colluding principals, p_i must allow only a single query for f_i from *any* principal, further limiting the utility of the system.

Epoch-Based Reference Monitors To increase system utility over the strict reference monitor model, we can leverage the observation that in many proof systems, the status of facts is expected to fluctuate over time (e.g., consider facts representing user and device locations, room occupancy predicates, or other physical phenomena). As a result, reference monitors can actually enforce the above types of constraints on a sliding-window basis. The length of such a window allows the uncertainty introduced by the transient nature of fact status values to occlude inferences to whatever level is deemed necessary.

Policy Design In certain circumstances, principals in the system may be able to prevent inferences made during the construction of multiple proofs by intelligent policy engineering decisions. Specifically, if the dependency graph whose nodes are quoted facts in the conjunction to be proved and whose edges are entailed by the ACL entries protecting these facts is strongly-connected, then requesting any fact in the conjunction requires the disclosure of all other facts in the conjunction. For example, Figure 4 applies this principle to prevent the inference made possible in Figure 3. Obviously, this method prevents the inference problem, but can only be applied in the event that facts in a given conjunction are never used outside of the context of that conjunction.

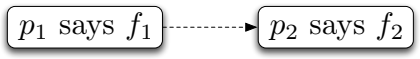

Figure 3: An example ACL dependency graph that permits inference.

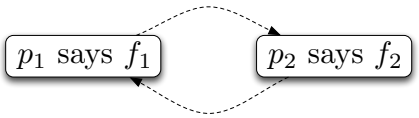

Figure 4: An example ACL dependency graph that prevents inference.

The choice of inference control mechanism to apply in any given situation is highly dependent on the environment within which a given proof system is to be deployed. Furthermore, the three approaches to inference control described above afford a range of trade-offs that can be taken into consideration by system architects at deployment time. As a result, we argue that the notion of confidentiality-preserving distributed proof described in this paper is in fact quite reasonable. It provides much more protection for confidential facts than existing proof systems, and can be further tuned to adjust the privacy/utility tradeoff as needed for any particular deployment environment.

4.3 Multiple-Run Restrictions

Theorem 6 makes no statement about the security of the conjunction in the case where the ACLs are not strongly connected. Ideally, we would like to be able to prove the same statement in this case; however, the statement turns out to be not true.

First, it is easy to see that if the ACLs of facts in the conjunction are not weakly connected, the querier learns the truth of each weakly connected component separately. This is because there are no encrypted values that are sent from principals in one component to principals in another. Hence, the execution of the protocol is equivalent to multiple parallel executions of the protocol on each weakly connected component. In fact, the querier could use different session IDs for each component without being detected. Second, even within a weakly connected component, the querier has the ability to learn conjunctions other than the one that is ostensibly being queried. For example, if the conjunction in question is "p_1 says f_1" $\wedge$ "p_2 says f_2", and $acl_2(f_2) = \{p_0\}$, then p_0 can execute the protocol in such a way that it learns "p_2 says f_2". Both of these problems are a direct consequence of hiding the full conjunction that is being proven from the fact providers: by design, p_2 should not be able to tell whether p_0 is trying to prove "p_1 says f_1" $\wedge$ "p_2 says f_2" or just "p_2 says f_2".

Fortunately, we can still state a useful security property that is provided by our protocol: the protocol allows a querier to learn as much as could be learned by multiple *non-overlapping* interactions with the TTP.

Theorem 7. *During a single construction of a distributed proof, given a conjunction C, a malicious querier p_0 can learn the truth of a set of conjunctions $C_1, \ldots, C_n \subset C$, under the condition that the ACLs of all the facts comprising each conjunction C_i are weakly connected, $C_i \cap C_j = \emptyset$ for $i \neq j$, and $checkAcls(p_0, C_i) =$ true. No other conjunctions will be revealed by the construction of a distributed proof.*

This means that in the above example, p_0 can learn the truth of "p_1 says f_1" $\wedge$ "p_2 says f_2" or "p_2 says f_2", but not both. Therefore, the strict or epochal reference monitor policies suggested in Section 4.2 will still work, since any time a principal learns a conjunction C, whether it be the one it ostensibly queried or not, all the principals who have facts in C will be notified and will forbid any further conjunctions that overlap with C.

Note, however, that it is impossible to give a simulation argument for Theorem 7. This is because a simulator that is interacting with p_0 and simulating p_1 and p_2 cannot know whether p_0 is interested in the conjunction "p_1 says f_1" $\wedge$ "p_2 says f_2" or simply "p_2 says f_2". Therefore, it cannot issue a single query to the TTP to provide p_0 with the correct answer without having to guess. Thus, we provide a more ad hoc proof of Theorem 7 in Appendix C.

It is easy to see that in the above example, our protocol behaves like an oblivious transfer protocol, with p_0 acting as the chooser and receiving either of the two conjunction values from the senders p_1 and p_2. Previous work on oblivious transfer has been able to develop protocols where a simulation argument can be used to prove sender security (e.g., [3]), so we are optimistic that in our future work we can modify the protocol so that it admits a full simulation proof.

5 Related Work

Although earlier distributed proof construction systems [1, 2, 8, 22, 31] are clearly related to the notion of confidentiality-preserving distributed proof, the work presented in this paper differs in two important ways. First, most existing systems have no mechanism for protecting the release of sensitive facts. Other proof systems allow fact disclosures to be protected by identity-based ACLs [22] or more complex release policies that must be *centrally verified* by the fact provider [31]. Our proof system allows fact providers to be assured that complex conditions protecting the disclosure of their facts will be enforced *without* requiring central verification. This allows fact providers to write release policies that reference quoted facts in the system to which they themselves do not have access. Second, existing proof systems do not consider the issue of simultaneous truth of facts, and simply assume that the proof construction protocol samples a consistent system state. In [15], the authors show how a distributed proof system [22] can be extended to ensure the simultaneous truth of facts sampled in a given proof. The proof system presented in this paper considers this goal from the outset.

In trust negotiation approaches to authorization (e.g., see [4, 5, 20, 29, 30]), principals can protect access to sensitive credentials using complex release policies similar to those described in this paper. However, once again, evidence attesting to the fact that these policies are satisfied must be collected and centrally verified by the entity disclosing the credential. This has two implications: evidence of partial policy satisfaction can be learned by principals in the system and principals can only write release policies that they are authorized to verify. The proof system presented in this paper addresses both of these problems by using a two-phase approach to *distributed* policy enforcement, which has the benefit of not leaking any partial state.

Brands's digital credential scheme [7] allows a credential owner to selectively reveal information encoded in the attributes of that credential. The credential owner can demonstrate that confidential attributes in the credential satisfy a linear relation or some expression in propositional logic without disclosing additional information about the individual attributes. This is similar to the guarantees afforded by our system, but it does not address the case where confidential attributes are maintained by multiple parties in a decentralized environment. The oblivious commitment-based envelope (OCBE) construction proposed by Li and Li [19, 18] provides similar guarantees, again, at a per-credential granularity.

Our problem can be viewed as a special case of the secure multi-party computation (SMC) problem. In this class of problems, some collection of principals wishes to calculate the result of a public function of n private inputs. General solutions exist for this class of problems [12], but can be quite inefficient for large problem sizes. These solutions also assume that all principals involved in a computation are aware of one another, which is something that our protocol seeks to avoid. Prabhakaran and Rosulek propose a specific SMC algorithm for computing aggregated OR that is similar to our construction [25], but in their construction the fact providers communicate directly to each other.

6 Conclusions

In this paper, we develop the notion of a confidentiality-preserving distributed proof system, in which the disclosure of a sensitive fact f can be made contingent upon facts $f'_1, \ldots, f'_n$ managed by other principals in the system. In this type of proof system, the querier learns only the truth value of the conjunction $f \wedge f'_1 \wedge \cdots \wedge f'_n$, which limits the knowledge leaked during the proof construction process. This enables *controlled* usage of sensitive facts, thereby increasing the utility of the proof system while minimizing the effects of this process on user privacy. We first formalized this notion of confidentiality-preserving distributed proof using a trusted third party (TTP) model of computation, and then developed a distributed algorithm for constructing confidentiality-preserving distributed proofs in asynchronous distributed systems.

We formally proved that our confidentiality-preserving distributed proof algorithm is sound, and that it reveals no more information than can be learned by interacting with the ideal functionality embodied by the TTP model. We then examined potential avenues of inference made possible in the ideal model—and thus in our proof construction algorithm—through multiple or concurrent proof constructions. Although these information leaks are worrisome, we showed that they can be limited based on local decisions made by fact providers.

Acknowledgements

We would like to thank Manoj Prabhakaran and Mike Rosulek for their discussions about the cryptographic primitives used in our protocol and the security proofs, and the anonymous reviewers for their helpful suggestions. This research was supported in

part by the National Science Foundation under award CNS–0716421.

References

[1] A. W. Appel and E. W. Felten. Proof-carrying authentication. In *Proceedings of the Sixth ACM Conference on Computer and Communications Security*, Nov. 1999.

[2] 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.

[3] M. Bellare and S. Micali. Non-interactive oblivious transfer and applications. In G. Brassard, editor, *Advances in Cryptology—CRYPTO'89*, volume 435 of *Lecture Notes in Computer Science*, pages 547–557. Springer, Aug. 1989.

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

[5] P. Bonatti and P. Samarati. Regulating service access and information release on the web. In *Proceedings of the Seventh ACM Conference on Computer and Communications Security*, pages 134–143, 2000.

[6] D. Boneh and M. Franklin. Identity based encryption from the Weil pairing. *SIAM Journal of Computing*, 32(3):586–615, 2003.

[7] S. A. Brands. *Rethinking Public Key Infrastructure and Digital Certificates*. MIT Press, Cambridge, MA, USA, 2000.

[8] J. DeTreville. Binder, a logic-based security language. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 105, 2002.

[9] D. Dolev, C. Dwork, and M. Naor. Nonmalleable cryptography. In *STOC '91: Proceedings of the twenty-third annual ACM symposium on Theory of computing*, pages 542–552, New York, NY, USA, 1991. ACM.

[10] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Proceedings of the Third Theory of Cryptography Conference*, pages 265–284, Mar. 2006.

[11] T. Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[12] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *Proceedings of the 19th annual ACM Conference on Theory of Computing*, pages 218–229, New York, NY, USA, 1987. ACM Press.

[13] A. Ivan and Y. Dodis. Proxy cryptography revisited. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium (NDSS 2003)*, Feb. 2003.

[14] T. Kohno, A. Broido, and K. Claffy. Remote physical device fingerprinting. *IEEE Transactions on Dependable and Secure Computing*, 2(2):93–108, 2005.

[15] 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*, pages 101–110, 2007.

[16] A. J. Lee and M. Winslett. Enforcing safety and consistency constraints in policy-based authorization systems. *ACM Transactions on Information and System Security*, to appear.

[17] K. LeFevre, D. J. DeWitt, and R. Ramakrishnan. Mondrian multidimensional k-anonymity. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*, Apr. 2006.

[18] J. Li and N. Li. A construction for general and efficient oblivious commitment based envelope protocols. In *Proceedings of 8th International Conference on Information and Communications Security (ICICS)*, pages 122–138, Dec. 2006.

[19] J. Li and N. Li. OACerts: oblivious attribute certificates. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 3(4):340–352, Oct. 2006.

[20] J. Li, N. Li, and W. H. Winsborough. Automated trust negotiation using cryptographic credentials. *ACM Transactions on Information and System Security*, to appear.

[21] A. Machanavajjhala, J. Gehrke, D. Kifer, and M. Venkitasubramaniam. ℓ-diversity: Privacy beyond k-anonymity. In *Proceedings of the 22nd*

International Conference on Data Engineering (ICDE), Apr. 2006.

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

[23] A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets (how to break anonymity of the Netflix prize dataset). In *Proceedings of 29th IEEE Symposium on Security and Privacy*, May 2008.

[24] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proceedings of EUROCRYPT—Advances in Cryptology*, 1999.

[25] M. Prabhakaran and M. Rosulek. Cryptographic complexity of multi-party computation problems: Classifications and separations. *Electronic Colloquium on Computational Complexity (ECCC)*, 15(50), 2008.

[26] M. Prabhakaran and M. Rosulek. Homomorphic encryption with CCA security. In L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, editors, *ICALP (2)*, volume 5126 of *Lecture Notes in Computer Science*, pages 667–678. Springer, 2008.

[27] M. Prabhakaran and M. Rosulek. Towards robust computation on encrypted data. In J. Pieprzyk, editor, *ASIACRYPT*, Lecture Notes in Computer Science. Springer, 2008. To appear.

[28] L. Sweeney. k-anonymity: A model for protecting privacy. *International Journal on Uncertainty, Fuzziness and Kowledge-based Systems*, 10(5):557–570, 2002.

[29] W. H. Winsborough and N. Li. Towards practical automated trust negotiation. In *Proceedings of the Third IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 92–103, June 2002.

[30] W. H. Winsborough, K. E. Seamons, and V. E. Jones. Automated trust negotiation. In *Proceedings of the DARPA Information Survivability Conference and Exposition*, pages 88–102, Jan. 2000.

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

[32] X. Xiao and Y. Tao. m-invariance: Towards privacy preserving re-publication of dynamic datasets. In *Proceedings of the ACM Conference on Management of Data (SIGMOD)*, pages 689–700, June 2007.

[33] 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):1–42, Feb. 2003.

A Proof of Theorem 2

To prove Theorem 2, we must address each of the three cases identified in Figure 5. In this figure, square nodes represent invocations of the proof protocol, while rounded nodes are used to identify quoted facts. We call these *invocation nodes* and *fact nodes*, respectively. Solid arrows are called *proof edges* and are used to identify relationships between facts and invocations of the proof protocol. More specifically, proof edges either identify the quoted facts making up a given conjunction (e.g., as in Figure 5(a)), or link a recursive instance of the proof protocol to the quoted fact requiring that a sub-proof be generated (e.g., as in Figure 5(b)). Dashed arrows are called *dependency edges* and are used to identify ACL dependencies between nodes in the proof graph. Dependency edges are labeled with the name of the principal to which the disclosure dependency applies.

Given this notation, we will now prove Theorem 2 by showing that if EndQuery(StartQuery(*conj*)) returns true, then each "p_i says $f_i \in conj$ was simultaneously true from the time that the call to StartQuery returned to the time at which EndQuery was invoked.

Case 1

The class of proof graph identified by Figure 5(a) encompasses all proofs in which recursive invocations of the proof protocol are not needed to prove some conjunction of facts. That is, for each "p_i says f_i" in the conjunction to be proved, f_i is part of p_i's extensional knowledge base (i.e., f_i is *not* derived using derivation rules). These proofs, by definition, have proof edges connecting a single invocation node to each of the n fact nodes comprising the conjunction, as well as dependency edges linking any number of fact nodes. Note that cycles may occur in the proof graph, but no cycle includes any proof edges.

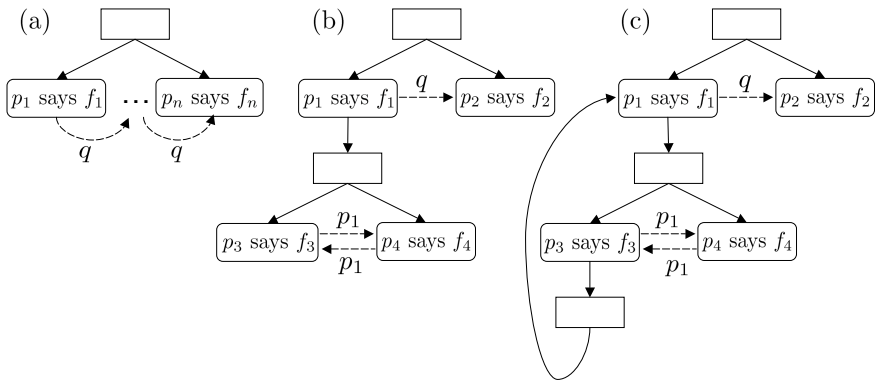

Figure 5: Three classes of confidentiality-preserving distributed proof graphs.

Assume that the proof graph entailed by a querier p_0 trying to prove the conjunction of facts *conj* is of the form described above. Further, assume that EndQuery(StartQuery(*conj*)) returns true, but that some fact "p_i says $f_i \in$ *conj* was not true for the entire interval between the first and second phases of the proof construction protocol. Here we have four cases to consider: (i) f_i was false for the entire interval; (ii) f_i was false at the end of phase one, but true at the start of phase two; (iii) f_i was true at the end of phase one, but false at the start of phase two; or (iv) f_i was true at the end of phase one and the start of phase two, but became false one or more times in the intervening period. In cases (i) and (iii), the check at line 37 of Algorithm 2 will fail because f_i is false during stage two of the protocol. In cases (ii) and (iv), this check will fail because the random identifier associated with f_i that was saved on line 19 of Algorithm 2 will be different from that checked on line 37 of Algorithm 2, as it would have changed when the status of f_i changed. In all cases, this causes p_i to return a random group element to p_0, rather than the decrypted secret share needed to recover the expected value generated by p_0 at the start of phase one. This implies that EndQuery(StartQuery(*conj*)) will return false, which is a contradiction.

Case 2

The class of proof graph identified by Figure 5(b) encompasses all *recursive invocation-acyclic* proof graphs. That is, proof graphs in which recursive invocations of the proof protocol are permitted, but there do not exist any cycles involving proof edges. To prove Theorem 2 in this case, we proceed by induction on the number of invocation nodes on the longest path starting from the root invocation node. In the base case, there is only one invocation node (i.e., the root of the proof graph). This is simply Case 1, above, and thus Theorem 2 holds in the base case. We now assume that Theorem 2 holds if up to $n-1$ invocation nodes exist on any single path.

We now show that Theorem 2 holds for any recursive invocation-acyclic proof graph containing n invocation nodes on its longest path. Note that the proof graph rooted at the nth invocation node is, again, an instance of Case 1. Thus, Theorem 2 holds for this nth protocol invocation. Since the first phase of this nth protocol invocation ends before the first phase of the $n-1$st protocol invocation ends, and the second phase of the nth protocol invocation starts after the second phase of the $n-1$st protocol invocation starts, all facts used in the nth protocol invocation are true between the end of phase one and the start of phase two in the $n-1$st protocol invocation. As a result, the inductive hypothesis tells us that Theorem 2 holds for the remaining $n-1$ protocol invocations. Thus, Theorem 2 holds for all recursive invocation acyclic proof graphs.

Case 3

The last class of proof graph is identified by Figure 5(c), and includes all proof graphs in which cycles containing proof edges exist. The proof construction protocol presented in Algorithms 1 and 2 fails to terminate if invoked to prove a conjunction of facts that entails a proof graph of this structure. For example, note that in Figure 5(c), each time that p_1 is asked about the status of f_1, a recursive invocation of the protocol occurs in which p_1 is again asked about the status of f_1. Detecting these types of cycles is an orthogonal problem to that which we set out to solve, although we note that standard techniques can be used to detect these types of cycles and terminate the protocol. To keep the code in Algorithms 1 and 2 simple, however, we omitted such code. In either case (i.e., non-terminating execution or aborted exe-

cution) the soundness of our protocol is unaffected, and thus Theorem 2 holds.

Since Cases 1–3, above, address all possible classes of proof graph upon which our protocol can be invoked, we have thus proven Theorem 2. □

B Proof of Theorem 3

The proof of Theorem 2 shows that EndQuery(StartQuery(*conj*)) → *conj*. We must now show that *conj* → EndQuery(StartQuery(*conj*)). That is, if each "p_i says f_i" ∈ *conj* is true throughout the duration of the protocol, the protocol will determine that *conj* is true. To prove this claim, we must examine proof graphs whose structures are described by Figures 5(a) and (b), as proof graphs with cycles containing proof edges are not considered well-formed.

Case 1

We first address proof graphs that do not require recursive invocations of the proof protocol (see Figure 5(a)). Assume without loss of generality that *conj* consists of n quoted facts. Prior to issuing any queries, the querier p_0 generates a random blinding factor b_i for each "p_i says f_i" ∈ *conj*. Each b_i is then encrypted to its corresponding p_i using an IBE key derived using the querier's identifier, the fact f_i, and the session id *sid*. The product of all such b_i's is then stored locally by p_0 in the variable *expected*. The above process takes place on lines 3–7 of Algorithm 1.

We now assume that *conj* = true, but EndQuery(StartQuery(*conj*)) = false. This implies that either (i) some p_i returns a random response during phase two of the protocol, or (ii) each p_i properly decrypts the encrypted shares provided to them during phase two of the protocol, but the product of all such shares computed by p_0 does not equal *expected*. Case (i) will not occur, as the fact identifier recorded for each "p_i says f_i" on line 19 of Algorithm 2 will match the fact identifier retrieved on line 36 of Algorithm 2, as each fact remains true throughout the execution of the protocol (by assumption). Furthermore, Case (ii) cannot occur, as the product of all secret shares generated by any single p_i on lines 9–15 of Algorithm 2 equals 1 (since, by assumption, each "p_i says f_i" remains true throughout the duration of the protocol). Therefore, the product of the secret shares generated by *all* n fact providers equals 1, which implies that p_0 will recover the value *expected* $= \prod_{i=1}^{n} b_i$ at the end of stage two of the protocol. Since neither Case (i) nor Case (ii) can occur, we have a contradiction.

Case 2

The second case that we must address is illustrated by Figure 5(b) and encompasses all recursive invocation-acyclic proof graphs. As in the proof of Case 2 of Theorem 2, we proceed by induction on the the number of invocation nodes on the longest path starting from the root invocation node. In the base case, there is only one invocation node (i.e., the root of the proof graph). This reduces to Case 1, above, and thus *conj* → EndQuery(StartQuery(*conj*)) in the base case. We now assume that *conj* → EndQuery(StartQuery(*conj*)) if up to $n-1$ invocation nodes exist on any single path.

We now show that *conj* → EndQuery(StartQuery(*conj*)) in any recursive invocation-acyclic proof graph containing n invocation nodes on its longest path. Consider the proof tree rooted at the nth invocation node. Since *conj* is true throughout the duration of the protocol (by assumption), all facts comprising the nth proof tree are true throughout the duration of the nth protocol invocation. Thus, the nth protocol invocation is an instance of Case 1, above, and therefore will return true. As a result, the remaining $n-1$ protocol invocations will return true by the inductive hypothesis, and the claim holds.

Since Cases 1 and 2 address all possible proof graphs to which Theorem 3 applies, we can conclude that *conj* → EndQuery(StartQuery(*conj*)) in all relevant cases. Furthermore, since Theorem 2 gives us that EndQuery(StartQuery(*conj*)) → *conj*, it follows that *conj* ↔ EndQuery(StartQuery(*conj*)) and Theorem 3 holds. □

C Proof of Theorems 4–7

We prove several security properties of our protocol and thus show that it models the ideal TTP functionality.

Theorem 4 (Query Privacy)**.** *During the construction of a distributed proof, a malicious subset of providers $p_1, \ldots, p_m$ will learn which local facts each p_i provides, but will learn nothing about the other facts comprising the conjunction being proved.*

Proof. During phase 1, the querier p_0 reveals only the fact in which p_0 is interested. During the sec-

ond phase, the blinding factor r ensures that p_i cannot distinguish the second-phase query from random. Hence no additional information about the query can be learned in the second phase. □

Theorem 5 (Query Validity)**.** *During the construction of a distributed proof, given a subset of malicious fact providers, $p_1, \ldots, p_m$, if one of the honest fact providers, p_i, provides a* false *fact, the conjunction received by the querier p_0 is* false*.*

Proof. Since p_i's fact is false, it will provide a random response during the RecoverShare function. Therefore, the querier will obtain a random result in EndQuery, and declare the conjunction to be false. □

Theorem 6 (Limited Disclosure)**.** *During the construction of a distributed proof, a malicious querier p_0, colluding with set of fact providers, $p_1, \ldots, p_m$, learns the same amount of information as it would by interacting with the TTP functionality, as long as the ACLs of honest fact providers form a strongly-connected component.*

Proof. We can present a simple simulation argument. Consider an adversary A that represents the querier p_0 and the malicious parties $p_1, \ldots, p_m$, and interacts with the honest parties $p_{m+1}, \ldots, p_n$. We will construct a simulator S that interacts with the real $p_{m+1}, \ldots, p_n$ through ideal functionality and simulates $p_{m+1}, \ldots, p_n$, to the adversary. We will show that A cannot distinguish whether it is interacting with S or the real honest parties.

After the first phase, S learns the list of facts that A is interested from each of the honest parties. It then expands the list of facts into a conjunction that will pass *checkAcls* by adding facts and principals from the ACLs of the honest parties; since these new principals will be simulated, S sets the ACL of the new principals to allow unrestricted access to their respective facts by p_0. Note that there might be multiple expanded conjunctions that might satisfy *checkAcls*; for our purposes, it is sufficient to find one. It then sends the query to the TTP, simulating the new principals by setting their facts to true.

If the TTP returns true, then it must be the case that all of the honest parties' facts are true. S then finishes the protocol by simulating the honest parties in phase 2 with true values for their facts. Since the simulation is identical to each p_i following the protocol, it is clear that A cannot distinguish the simulation from real interaction.

If the TTP returns false, S simulates phase 2 by simulating the honest parties with false values for their facts. We now show that A cannot distinguish this from the real execution of the protocol. Because the conjunction is false, in the real execution of the protocol, some honest party p_k must have a false fact. It is clear that p_k behaves identically in the real protocol and simulation.

Next consider an honest party p_i whose ACL depends on p_k. In this case, p_i will generate a number of shares, $s_{i,i}, s_{i,j}, s_{i,k}, \ldots$. The share $s_{i,k}$ will be encrypted for p_k in the first phase of the protocol, and discarded by p_k in the second phase. Since the encryption scheme is IND-ID-CPA secure, and since the session ID prevents repeated queries, A cannot learn the value of $s_{i,k}$. Leaving out $s_{i,k}$, the other shares are uniformly distributed and uncorrelated each other. If p_i's fact is true, its answer will be a multiple of $s_{i,i}$, and thus be uniformly randomly distributed and uncorrelated with anything in the adversary's view; likewise, if p_i's fact is false, the answer will be chosen uniformly at random. Thus in both cases, the real p_i's behavior cannot be distinguished by A from the simulated p_i, which sets its value of the fact to false.

We can next consider principal whose ACL depends on principals such as p_i, and continue by induction to show that every honest p_i behaves in a manner undistinguishable between simulation and the real protocol. Note that all honest principals will, after some number of steps, depend on p_k, since the ACL graph is strongly connected. □

Theorem 7. *During a single construction of a distributed proof, given a conjunction C, a malicious querier p_0 can learn the truth of a set of conjunctions $C_1, \ldots, C_n \subset C$, under the condition that the ACLs of all the facts comprising each conjunction C_i are weakly connected, $C_i \cap C_j = \emptyset$ for $i \neq j$, and $checkAcls(p_0, C_i) =$* true*. No other conjunctions will be revealed by the construction of a distributed proof.*

To prove this theorem, we first need an additional assumption about the Homomorphic IBE scheme that we are using:

Assumption 1. *An adversary who knows the master public key in the Homomorphic IBE scheme, but does not know the private key corresponding to identity ID, can obtain valid ciphertexts encrypted for ID in one of the following ways only:*

- *Encrypting a known value, M, to obtain $E_{ID}(M)$*
- *Using the homomorphism $\otimes$ on two ciphertexts $E_{ID}(M)$ and $E_{ID}(M')$ to obtain $E_{ID}(M \cdot M')$*

Since CPA does not impose any bounds on malleability, this assumption does not follow from the

IND-ID-CPA property of the HIBE scheme we are using. Our assumption requires that the scheme have *limited* malleability; i.e., ciphertexts can be manipulated only by the homomorphism. Prabhakaran and Rosulek formalize this notion for homomorphic (non-IBE) encryption as Homomorphic-CCA security [26], but at the same time show that their definition is impossible to achieve for a binary homomorphic operation that performs a group operation on two ciphertexts. In further work, they show that it is possible to implement such a construction with the caveat that the ciphertext grows with the number of binary operations [27], which would violate the query privacy property. In future work, we will investigate whether, by bounding the number of fact providers, we can use their construction to create a protocol with more well-founded proof.

Proof. Consider an adversary who wants to learn C_1 and C_2, where $C_1 \cap C_2 \neq \emptyset$. Since both C_1 and C_2 are weakly connected, there must be a dependency from "p_i says $f_i'' \in C_1 \setminus C_2$ to "p_j says $f_j'' \in C_2$ or vice versa, for some p_i and p_j. (We will assume without loss of generality that the dependency is from C_1 to C_2). Therefore, during the first phase of the protocol, p_i will return $E^{p_j}_{ME||f_j||sid}(s_{i,j})$. Given the assumption, p_0 now has two choices in the second phase of the protocol. It can use homomorphic encryption to provide p_j with *some* multiple of $E^{p_j}_{ME||f_j||sid}(s_{i,j})$, or it can give it an encrypted value unrelated to $E^{p_j}_{ME||f_j||sid}(s_{i,j})$. In the former case, the result returned by p_j will be randomized if "p_i says f_i'' is false. Therefore, p_0 will not learn the truth of C_2. In the latter case, p_0 will not be able to distinguish the cases where "p_i says f_i'' is true and "p_i says f_i'' is false, hence it cannot learn the value of C_1. □