Traffic Pattern-Based Content Leakage Detection for Trusted Content Delivery Networks

Shaymaa Taha Ahmed

Department of Computer Science

University of Diyala / College of Basic Education

Abstract—In this paper, Due to the increasing popularity of multimedia streaming applications and services in recent years, the issue of trusted video delivery to prevent undesirable content-leakage has, indeed, become critical. While preserving user privacy, conventional systems have addressed this issue by proposing methods based on the observation of streamed traffic throughout the network. These conventional systems maintain a high detection accuracy while coping with some of the traffic variation in the network (e.g., network delay and packet loss), however, their detection performance substantially degrades owing to the significant variation of video lengths. In this paper, we focus on overcoming this issue by proposing a novel content-leakage detection scheme that is robust to the variation of the video length. By comparing videos of different lengths, we determine a relation between the length of videos to be compared and the similarity between the compared videos.. We finally analyze the different approaches presented using both analytical evaluation of the overheads and simulations to guide the decision makers to which approach to use.

Index Terms—Streaming content, leakage detection, traffic pattern, degree of similarity

1 Introduction

Cloud computing has recently emerged as a computing paradigm in which storage and computation can be outsourced from organizations to next generation data centres hosted by companies such as Amazon, Google, Yahoo, and Microsoft. Such companies help free organizations from requiring expensive infrastructure and expertise in-house, and instead make use of the cloud providers to maintain, support, and broker access to high-end resources. From an economic perspective, cloud consumers can save huge IT capital investments and be charged on the basis of a pay-only-for-what you-use pricing model.

One of the most appealing aspects of cloud computing is its elasticity, which provides an illusion of infinite, on-demand resources making it an attractive environment for highly-scalable, multi-tiered applications. However, this can create additional challenges for back-end, transactional database systems, which were designed without elasticity in mind. Despite the e orts of key-value stores like Amazon's Sampled, Dynamo, and Google's Big table to provide scal-able access to huge amounts of data, transactional guarantees remain a bottleneck.

To provide scalability and elasticity, cloud services often make heavy use of replication to ensure consistent performance and availability. As a result, many cloud services rely on the notion of eventual consistency when propagating data throughout the system. This consistency model is a variant of weak consistency that allows data to be inconsistent among some replicas during the update process, but ensures that updates will eventually be propagated to all replicas. This

Makes it di cult to strictly maintain the ACID guarantees, as the ’C’ (consistency) part of ACID is sacrificed to provide reasonable availability.

In systems that host sensitive resources, accesses are protected via authorization policies that describe the conditions under which users should be permitted access to resources. These policies describe relationships between the system principals, as well as the certified credentials that users must provide to attest to their attributes. In a transactional database system that is deployed in a highly distributed and elastic sys-tem such as the cloud, policies would typically be replicated very much like data among multiple sites, often following the same weak or eventual consistency model. It therefore becomes possible for a policy-based authorization system to make unsafe decisions using stale policies.

Interesting consistency problems can arise as transactional database systems are deployed in cloud environments and use policy-based authorization systems to protect sensitive resources. In addition to handling consistency issues amongst database replicas, we must also handle two types of security inconsistency conditions. First, the system may suffer from *policy inconsistencies* during policy updates due to the relaxed consistency model underlying most cloud services. For example, it is possible for several versions of the policy to be observed at multiple sites within a single transaction, leading to inconsistent (and likely unsafe) access decisions during the transaction. Second, it is possible for external factors to cause *user credential inconsistencies* over the lifetime of a transaction. For instance, a user’s login credentials could be invalidated or revoked after collection by the authorization

Server, but before the completion of the transaction. *In this paper, we address this confluence of data, policy, and credential inconsistency problems that can emerge as transactional database systems are deployed to the cloud.* In doing so we make the following contributions:

We formalize the concept of *trusted transactions*. Trusted transactions are those transactions that do not violate credential or policy inconsistencies over the lifetime of the transaction. We then present a more general term, *safe transactions*, that identifies transactions that are both trusted and conform to the ACID properties of distributed database systems (Sec. 2).

We define several different levels of policy consistency constraints and corresponding enforcement approaches that guarantee the trustworthiness of transactions being executed on cloud servers (Sec. 3).

We propose a Two-Phase Validation Commit (2PVC) protocol that ensures that a transaction is *safe* by checking policy, credential, and data consistency during transaction execution (Sec. 4).

We carry out an experimental evaluation of our proposed approaches (Sec. 5), and present a trade-o discussion to guide decision makers as to which approach is most suitable in various situations (Sec 6).

Finally, Sec. 7 describes previous related work, while Sec. 8 presents our conclusions.

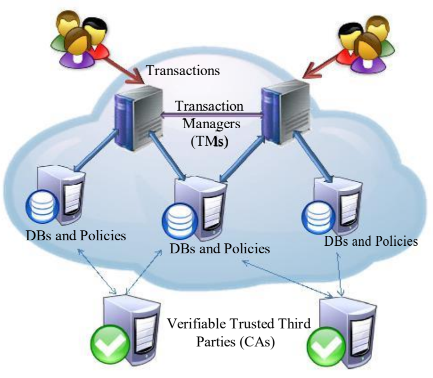
2 System Assumptions and Problem Definition

2.1 System Model

Fig. 1 illustrates the interaction among the components in our system. We assume a cloud infrastructure consisting of a set of **S** servers, where each server is responsible for hosting a subset D of all data items **D** belonging to a specific application domain (D **D**). Users interact with the system by submitting queries or update requests encapsulated in ACID transactions. A transaction is submitted to a *Transaction Manager* (TM) that coordinates its execution. Multiple TMs could be invoked as the system workload increases for load balancing, but each transaction is handled by only one TM.

We denote each transaction as T **=** q_1**;** q_2**; : : : ;** q_n, where q_i **2** Q is a single query**/**update belonging to the set of all queries Q. The start time of each transaction is denoted by (T), and the time at which the transaction finishes execution and is ready to commit is denoted by **!**(T). We assume that queries belonging to a transaction execute sequentially, and that a transaction does not fork sub-transactions. These assumptions simplify our presentation, but do not a ect the correctness or the validity of our consistency definitions.

Let **P** denote the set of all authorization policies, and let P_{s_i} (D) denote the policy that server s_i uses to protect data item D. We represent a policy P as a mapping P : **S** $2^{\mathbf{D}}$ **!** 2^R A N that associates a server and a set of data items with



Transactions
Transaction Managers (TMs)
DBs and Policies
DBs and Policies
DBs and Policies
Verifiable Trusted Third Parties (CAs)

Fig. 1. Interaction among the system components

a set of inference rules from the set R, a policy administrator from the set A, and a version number. We denote by **C** the set of all credentials, which are issued by the Certificate Authorities (CAs) within the system. We assume that each CA o ers an online method that allows any server to check the current status of credentials that it has issued [5]. Given a credential c_k **2 C**, (c_k) and **!**(c_k) denote issue and expiration times of c_k , respectively. Given a function m : Q **!** 2^D that identifies the data items accessed by a particular query, a *proof of authorization* for query q_i evaluated at server s_j at time t_k is a tuple **h**q_i**;** s_j**;** P_{s_j} (m(q_i))**;** t_k **;** C**i**, where C is the set of credentials presented by the querier to satisfy P_{s_j} (m(q_i)). In this paper, we use the function *eval* : **F** T S **!** B to denote whether a proof f **2 F** is valid at time t **2** T S .

To enhance the general applicability of the consistency models developed in this paper, the above formalism is in-emotionally opaque with respect to the policy and credential formats used to implement the system. For instance, this formalism could easily be used to model the use of XACML policies [6] as the set of inference rules R, and traditional (e.g., X.509 [7]) credentials for the set **C**. On the other hand, it can also model the use of more advanced trust management policies (e.g., [8], [9]) for the inference rules R, and the use of privacy-friendly credentials (e.g., [10], [11]) for the set **C**.

2.2 Problem Definition

Since transactions are executed over time, the state information of the credentials and the policies enforced by different servers are subject to changes at any instance of time, therefore it becomes important to introduce precise definitions for the different consistency levels that could be achieved within a transactions lifetime. These consistency models strengthen the trusted transaction definition by defining the environment in which policy versions are consistent relative to the rest of the system. Before we do that, we define a transaction's *view* in terms of the different proofs of authorization evaluated during the lifetime of a particular transaction.

Definition 1: (View) A transaction's *view* V^T is the set of proofs of authorization observed during the lifetime of a transaction [(T); **!**(T)] and defined as V^T **= f** f_{si} **j** f_{si} **=** **h**q_i; s_i; P_{si} $(m(q_i))$; t_i; C**i ^** q_i **2** T **g**. _

Following from Def. 1, a transaction's view is built incrementally as more proofs of authorization are being evaluated by servers during the transaction execution. We now present two increasingly more powerful definitions of consistencies within transactions.

Definition 2: (View Consistency) A view V^T **= fh**q_i; s_i; P_{si} $(m(q_i))$; t_i; C**i**; : : : ; **h**q_n; s_n; P_{sn} $(m(q_n))$; t_n; C**ig** is *view consistent*, or -consistent, if V^T satisfies a predicate -consistent that places constraints on the versioning of the policies such that -consistent (V^T) **$ 8**$_{i;j}$: $ver(P_{si})$ **=** $ver(P_{sj})$ for all policies belonging to the same administrator A, where function ver is defined as $ver : P$ **!** N. _

With a view consistency model, the policy versions should be internally consistent across all servers executing the transaction. The view consistency model is weak in that the policy version agreed upon by the subset of servers within the transaction may not be the latest policy version v. It may be the case that a server outside of the S servers has a policy that belongs to the same administrative domain and with a version $v^{\mathbf{0}}$ **>** v. A more strict consistency model is the global consistency and is defined as follows.

Definition 3: (Global Consistency) A view V^T **= fh**q_i; s_i; P_{si} $(m(q_i))$; t_i; C**i**; : : : ; **h**q_n; s_n; P_{sn} $(m(q_n))$; t_n; C**ig** is *global consistent*, or -consistent, if V^T satisfies a predicate -consistent that places constraints on the versioning of the policies such that -consistent (V^T) **$ 8**$_i$: $ver(P_{si})$ **=** ver(**P**) for all policies belonging to the same administrator A, and function ver follows the same aforementioned definition, while ver(**P**) refers to the latest policy version. _

With a global consistency model, policies used to evaluate the proofs of authorization during a transaction execution among S servers should match the latest policy version among the entire policy set **P**, for all policies enforced by the same administrator A.

Given the above definitions, we now have a precise vocabulary for defining the conditions necessary for a transaction to be asserted as "trusted".

Definition 4: (Trusted Transaction) Given a transaction T **= f**$q1$; $q2$; **: : :** ; q_n**g** and its corresponding view V^T, T is *trusted* i **8**f_{si} **2**V_T : $eval(f_{si}; t)$, at some time instance t : (T) t **!**(T) **^** (-consistent(V^T) _ -consistent(V^T)) _

Finally, we say that a transaction is *safe* if it is a trusted transaction that also satisfies all data integrity constraints imposed by the database management system. A safe transaction is allowed to commit, while an unsafe transaction is forced to rollback.

3 Trusted Transaction Enforcement

In this section, we present several increasingly stringent approaches for enforcing trusted transactions. We show that each approach guarantees during the course of a transaction. Fig. 2 is a graphical depiction of how these approaches could be applied to a transaction running across three server, and will be referenced throughout this section. In this figure, dots represent the arrival time of a query to some server, stars indicate the times at which a server validates a proof of authorization, and dashed lines represent view- or globally-consistency policy synchronization among servers.

3.1 Deferred Proofs of Authorization

Definition 5: (Deferred Proofs of Authorization) Given a transaction T and its corresponding view V^T, T is trusted under the Deferred proofs of authorization approach, i at commit time **!**(T), **8**f_{si} **2**V_T : $eval(f_{si}$; **!**$(T))$ **^** (-consistent(V^T) _ -consistent(V^T)) _

Deferred proofs present an optimistic approach with relatively weak authorization guarantees. The proofs of authorization are evaluated simultaneously only at commit time (using either view or global consistency from Defs. 2 and 3) to decide whether the transaction is trusted.

3.2 Punctual Proofs of Authorization

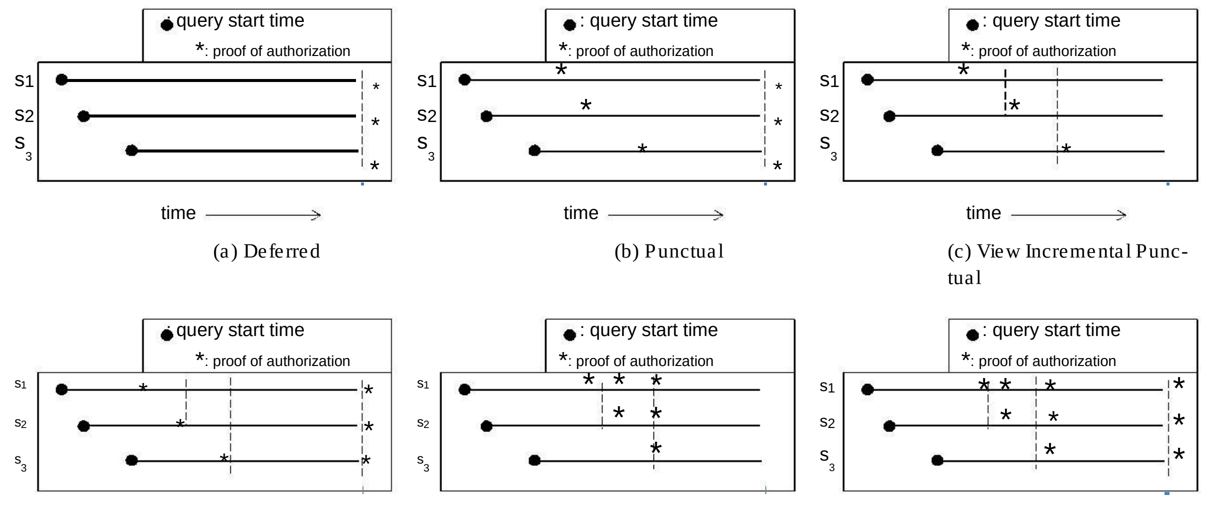
Definition 6: (Punctual Proofs of Authorization) Given a transaction T and its corresponding view V^T, T is trusted under the Punctual proofs of authorization approach, i at any time instance t_i : (T) t_i **!**(T) **8**f_{si} **2**V_T : $eval(f_{si}$; $t_i)$ **^** $eval(f_{si}$; **!**$(T))$ **^** (-consistent(V^T) _ -consistent(V^T)) _

Punctual proofs present a more proactive approach in which the proofs of authorization are evaluated instantaneously whenever a query is being handled by a server. This facilitates early detections of unsafe transactions which can save the system from going into expensive undo operations. All the proofs of authorization are then re-evaluated at commit time to ensure that policies were not updated during the transaction in a way that would invalidate a previous proof, and that credentials were not invalidated.

Punctual proofs do not impose any restrictions on the freshness of the policies used by the servers during the transaction execution. Consequently, servers might falsely deny or allow access to data. Thus, we propose two more restrictive approaches that enforce some degree of consistency among the participating servers each time a proof is evaluated.

3.3 Incremental Punctual Proofs of Authorization

Before we define the Incremental Punctual proofs of authorization approach, we define a *view instance*, which is a view snapshot at a specific instance of time.



(a) Deferred

(b) Punctual

(c) View Incremental Punctual

(d) Global Incremental Punctual

(e) View Continuous

(f) Global Continuous

Fig. 2. Different variants of proofs of authorization

For view consistency, no consistency check at commit time is required, since all participating servers will be view consistent by commit time. On the other hand, the global consistency condition necessitates another consistency check at commit time to confirm that the policies used have not become stale during the window of execution between the last proofs of authorization and commit time.

3.4 Continuous Proofs of Authorization

We now present the least permissive approach which we call Continuous proofs of authorization.

For the global consistency case (Def. 3), the TM retrieves the latest policy version from a master policies server (Step 2) and uses it to compare against the version numbers of each participant (Step 3). This master version may be retrieved only once or each time Step 3 is invoked. For the former case, collection may only be executed twice as in the case of view consistency. Once the TM receives the replies from all the participants, it moves on to the validation phase. If all polices are consistent, then the protocol honors the truth value where any FALSE causes an ABORT decision and all TRUE cause a CONTINUE decision. In the latter case, if the TM retrieves the latest version every round, global consistency may execute the collection many times. This is the case if the policy is updated during the round. While the number of rounds is theoretically infinite, in a practical setting, this should occur infrequently.

In Continuous proofs, whenever a proof is evaluated, all previous proofs have to be re-evaluated if a newer version of the policy is found at any of the participating servers. At commit time, Continuous proofs behave similarly to Incremental Punctual proofs. In contrast with the Incremental Punctual proofs, if later executing servers are using newer policy versions $_{(T)}$ all previous $_{(T)}$ servers must (i) update $_{(T)}$ their policies to be consistent with the newest one, and (ii) re-evaluate their proofs of authorization using the newer policies. In the case of global consistency, all servers will be forced to use the latest policy version at all times. Therefore, we consider this variant of our approaches to be the strictest approach of all giving the best privacy and consistency guarantees.

The decision of which approach to adopt is likely to be a strategic choice made independently by each application. As with$_{\omega(T\ )}$ any trade$_{\alpha(T)}$-o , the stronger the security$_{\omega(T)}$ and accuracy given by an approach, the more the system has to pay in terms of implementation and messages exchange overheads. Further discussion of these trade-o s will be presented in Sec. 6.

4 Implementing Safe Transactions

A safe transaction is a transaction that is both trusted (i.e., satisfies the correctness properties of proofs of authorization) and database correct (i.e., satisfies the data integrity constraints). We first describe an algorithm that enforces trusted transactions, and then expand this algorithm to enforce safe transactions. Finally, we show how these algorithms can be used to implement the approaches discussed in Sec. 3.

4.1 Two-Phase Validation Algorithm

A common characteristic of most of our proposed approaches to achieve trusted transactions is the need for policy consistency validation at the end of a transaction. That is, in order for a trusted transaction to commit, its TM has to enforce either view or global consistency among the servers participating in the transaction. Toward this, we propose a new algorithm called *Two-Phase Validation* (2PV).

As the name implies, 2PV operates in two phases: *collection* and *validation*. During collection, the TM first sends a Prepare-to-Validate message to each participant server. In response to this message, each participant (1) evaluates the proofs for each query of the transaction using the latest policies it has available and (2) sends a reply back to the TM containing the truth value (TRUE/FALSE) of those proofs along with the version number and policy identifier for each policy used. Further, each participant keeps track of its reply (i.e., the state of each query) which includes the id of the TM (TM_{id}), the id of the transaction (T_{id}) to which the query belongs, and a set of policy versions used in the query's authorization ($v_i; p_i$).

Once the TM receives the replies from all the participants, it moves on to the validation phase. If all polices are consistent, then the protocol honors the truth value where any FALSE causes an ABORT decision and all TRUE cause a CONTINUE decision. In the case of inconsistent policies, the TM identifies the latest policy and sends an Update message to each out-of-date participant with a policy identifier and returns to the collection phase. In this case, the participants (1) update their policies, (2) re-evaluate the proofs and (3) send a new reply to the TM. Algorithm 1 shows the process for the TM.

In the case of view consistency (Def. 2), there will be at most two rounds of the collection phase. A participant may only be asked to re-evaluate a query using a newer policy by an Update message from the TM after one collection phase.

For the global consistency case (Def. 3), the TM retrieves the latest policy version from a master policies server (Step 2) and uses it to compare against the version numbers of each participant (Step 3). This master version may be retrieved only once or each time Step 3 is invoked. For the former case, collection may only be executed twice as in the case of view consistency. In the latter case, if the TM retrieves the latest version every round, global consistency may execute the collection many times. This is the case if the policy is updated during the round. While the number of rounds are theoretically infinite, in a practical setting, this should occur infrequently.

4.2 Two-Phase Validate Commit Algorithm

The 2PV protocol enforces trusted transactions, but does not enforce not safe transactions because it does not validate any integrity constraints. Since the *Two-Phase Commit* atomic protocol (2PC) commonly used to enforce integrity constraints has similar structure as 2PV, we propose integrating these protocols into a *Two-Phase Validation Commit* (2PVC) protocol.

Algorithm 1: Two-Phase Validation - 2PV(TM)

```
1  Send "Prepare-to-Validate" to all participants
2  Wait for all replies (a True/False, and a set of policy
     versions for each unique policy)
3  Identify the largest version for all unique policies
4  If all participants utilize the largest version for each
     unique policy
5      If any responded False
6          ABORT
7      Otherwise
8          CONTINUE
9  Otherwise, for all participants with old versions of policies
10     Send "Update" with the largest version number of each
         policy
11     Goto 2
```

2PVC can be used to ensure the data and policy consistency requirements of safe transactions.

Specifically, 2PVC will evaluate the policies and authorizations within the first, voting phase. That is, when the TM sends out a Prepare-to-Commit message for a transaction, the participant server has three values to report: (1) the YES or NO reply for the satisfaction of integrity constraints as in 2PC, (2) the TRUE or FALSE reply for the satisfaction of the proofs of authorization as in 2PV, and (3) the version number of the policies used to build the proofs ($v_i; p_i$) as in 2PV.

The process given in Algorithm 2 is for the TM under view consistency. It is similar to that of 2PV with the exception of handling the YES or NO reply for integrity constraint validation and having a decision of COMMIT rather than CONTINUE. The TM enforces the same behavior as 2PV in identifying policies inconsistencies and sending the Update messages. The same changes to 2PV can be made here to provide global consistency by consulting the master policies server for the latest policy version (Step 5).

The resilience of 2PVC to system and communication failures can be achieved in the same manner as 2PC by recording the progress of the protocol in the logs of the TM and participants. In the case of 2PVC, a participant must forcibly log the set of ($v_i; p_i$) tuples along with its vote and truth value. Similarly to 2PC, the cost of 2PVC can be measured in terms of log complexity (i.e., the number of times the protocol forcibly logs for recovery) and message complexity (i.e., the number of messages sent). The log complexity of 2PVC is no di erent than basic 2PC and can be improved by using any of log-based optimizations of 2PC such as *Presumed-Abort* (PrA) and *Presumed-Commit* (PrC) [12]. The message complexity of 2PVC was analyzed.

4.3 Using 2PV & 2PVC in Safe Transactions

2PV and 2PVC can be used to enforce each of the consistency levels defined in Sec. 3. Deferred and Punctual (Defs. 5 and 6) proofs are roughly the same. The only difference is that

Algorithm 2: Two-Phase Validation Commit - 2PVC (TM)

```
1  Send "Prepare-to-Commit" to all participants
2  Wait for all replies (Yes/No, True/False, and a set of
     policy versions for each unique policy)
3  If any participant replied No for integrity check
4      ABORT
5  Identify the largest version for all unique policies
6  If all participants utilize the largest version for each
     unique policy
7      If any responded False
8          ABORT
9      Otherwise
10         COMMIT
11 Otherwise, for participants with old policies
12     Send "Update" with the largest version
         number of each policy
13     Wait for all replies
14     Goto 5
```

Punctual will return proof evaluations upon executing each query. Yet this is done on a single server, and therefore does not need 2PVC or 2PV to distribute the decision. To provide for trusted transactions, both require a commit-time evaluation at all participants using 2PVC.

Incremental Punctual (Def. 8) proofs are slightly different. As queries are executed, the TM must also check for consistency within the participating servers. Hence, a variant of the basic 2PV protocol is used during the transaction execution. For view consistency, the TM needs to check the version number it receives from each server with that of the very first participating server. If they are di erent, the transaction aborts due to a consistency violation. At commit time, all the proofs will have been generated with consistent policies and only 2PC is invoked. In the global consistency case, the TM needs to validate the policy versions used against the latest policy version known by the master policies server to decide whether to abort or not. At commit time, 2PVC is invoked by the TM to check the data integrity constraints and verify that the master policies server has not received any newer policy versions.

Finally, Continuous proofs (Def. 9) are the most involved. Unlike the case of Incremental Punctual in a view consistency, Continuous proofs invoke 2PV at the execution of each query which will update the older policies with the new policy and re-evaluate. When a query is requested, its TM will (1) execute 2PV to validate authorizations of all queries up to this point, and (2) upon CONTINUE being the decision of 2PV, submit the next query to be executed at the appropriate server, otherwise the transaction aborts. The same actions occur under global consistency with the exception that the latest policy version is used as identified by the master policy server.

5 Evaluations

5.1 Environment and Setup

We used Java to implement each proof approach described in Sec. 3 with support for both view and global consistency. Although the approaches were implemented in their entirety, the underlying database and policy enforcement systems were simulated with parameters chosen according to Table 1. To understand the performance implications of the di erent approaches, we varied the (i) protocol used, (ii) level of consistency desired, (iii) frequency of master policy updates, (iv) transaction length, and (v) number of servers available.

Our experimentation framework consists of three main com-ponents: a randomized transaction generator, a master policy server that controls the propagation of policy updates, and an array of transaction processing servers.

Our experiments were run within a research lab consisting of 38 Apple Mac Mini computers. These machines were running OS X 10.6.8 and had 1.83 GHz Intel Core Duo processors coupled with 2GB of RAM. All machines were connected to a gigabit ethernet LAN with average round trip times of 0.35 ms. All WAN experiments were also conducted within this testbed by artificially delaying packet transmission by an additional 75 ms.

For each simulation and each possible combination of parameters, 1000 transactions were run to gather average statistics on transaction processing delays induced by the particular protocol and system parameter choices. The randomized transactions were randomly composed of database reads and writes with equal probability. To simulate policy updates at di erent servers, the master policy server picks a random participating server to receive the updates.

Given that our interest in this article lies in exploring the average performance of each of the di erent approaches, we made few assumptions to simplify the experimentations and help limit the influence of other factors on transaction execution time. Specifically, we assumed the existence of a single master policy server that has to be consulted for the latest policy version belonging to a specific policy administrator. This simplifies the 2PV protocol and reduces the number of exchanged messages to realize the latest version among

TABLE 1
Simulation Parameters

Parameter	Value(s)
Times of policies update	once during operations, once per participant join, or once at com-mit time
Disk read latency	1-3 ms
Disk write latency	12-20 ms
Authorization check delay	1-3 ms
Data integrity constraint verification	1-3 ms
Transaction size	Short: 8-15 operations, Medium: 16-30 operations, or Long: 31-50 operations

We now investigate two cloud-based applications that are representative of larger classes of interesting applications to show how requirements can impact the choice of consistency enforcement scheme. In particular, we consider three orthog-onal axes of requirements: *code complexity* (which is directly related to trusted computing base size), *transaction mix* (i.e., write-only, read/write with internal reads, and read/write with materialized reads), and *policy/credential update frequency*.

Application: Event Scheduling. Consider an Event Mon-itoring Service (EMS) used by a multi-campus university to track events within the university and to allow sta , faculty members, and student organizations to make online event registrations. The university is using a cloud infrastructure to host the various EMS databases and execute the di erent transactions. Users have varying access privileges that are governed by authorization policies and credentials issued by a university-wide credentialing system. In this system, read requests must be externalized to users during the transaction execution so that intermediate decisions can be made. Further-more, the university system in general has infrequent policy and credentials updates, and requires lower code complexity to minimize code verification overheads.

Recommendation: In this case, the use of Punctual proofs makes the most sense. Note that this approach has low code complexity, performs fast, and is suitable for systems with infrequent updates. Read externalization is also permissible, as policies are checked prior to each operation in the transaction.

Application: Sales Database This example is derived from the travelling salesperson example in. According to company requirements, a customer's data should only be read by the representatives in the operations region of that customer, while any other data should not be materialized until commit time. The company also experiences very frequent policy and credential updates, as representatives are frequently assigned to different operational regions. The company considers security to be very important as to avoid incorrect authorization decisions that might leak customer information. Finally, the company has enough resources to manage complex code, but still requires reasonable execution latency.

Recommendation: This Company should use the Continuous global approach for the highest accuracy to avoid any information leakage at runtime, or Continuous view for slightly lower accuracy. This provides a good balance between accuracy and performance, at the cost of higher code complexity.

7 Related Work

Relaxed Consistency Models for the Cloud. Many database solutions have been written for use within the cloud environment. For instance, Amazon's Dynamo database; Google's Big Table storage system; Face book's Cassandra; and Yahoo!'s PNUTS. The common thread between each of these custom data models is BASE with a relaxed notion of consistency provided in order to support massively parallel environments.

Such a relaxed consistency model adds a new dimension to the complexity of the design of large scale applications and introduces a new set of consistency problems. In, the authors presented a model that allows queriers to express consistency and concurrency constraints on their queries that can be enforced by the DBMS at runtime. On the other hand, introduces a dynamic consistency rationing mechanism which automatically adapts the level of consistency at run-time. Both of these works focus on *data* consistency, while our work focuses on attaining both *data* and *policy* consistency.

Reliable Outsourcing. Security is considered one of the major obstacles to a wider adoption of cloud computing. Particular attention has been given to client security as it relates to the proper handling of outsourced data. For example, proofs of data possession have been proposed as a means for clients to ensure that service providers actually maintain copies of the data that they are contracted to host. In other works, data replication has been combined with proofs of irretrievability to provide users with integrity and consistency guarantees when using cloud storage.

To protect user access patterns from a cloud data store, introduces a mechanism by which cloud storage users can issue encrypted reads, writes and inserts. Further, pro-poses a mechanism that enables entrusted service providers to support transaction serialization, backup, and recovery with full data confidentiality and correctness. This work is orthogonal to the problem that we focus on in this article, namely consistency problems in policy-based database transactions.

Distributed Transactions. Cloud TPS provides full ACID properties with a scalable transaction manager designed for a NoSQL environment. However, Cloud TPS is primarily concerned with providing consistency and isolation upon data without regard to considerations of authorization policies.

There has also been recent work that focuses on providing some level of guarantee about the relationship between data and policies. This work proactively ensures that data stored at a particular site conforms to the policy stored at that site. If the policy is updated, the server will scan the data items and throw out any that would be denied based on the revised policy. It is obvious that this will lead to an eventually consistent state where data and policy conform, but this work only concerns itself with local consistency of a single node, not with transactions that span multiple nodes.

Distributed Authorization. The consistency of distributed proofs of authorization has previously been studied, though not in a dynamic cloud environment (e.g.,). This work highlights the inconsistency issues that can arise in the case where authorization policies are static, but the credentials used to satisfy these policies may be revoked or altered. The authors develop protocols that enable various consistency guarantees to be enforced during the proof construction process to minimize these types of security issues. These consistency guarantees are similar to our notions of safe transactions. However, our work also addresses the case in which policies in addition to credentials—may be altered or modified during a transaction.

8 Conclusions

Despite the popularity of cloud services and their wide adoption by enterprises and governments, cloud providers still lack services that guarantee both data and access control policy consistency across multiple data centers. In this article, we identified several consistency problems that can arise during cloud-hosted transaction processing using weak consistency models, particularly if policy-based authorization systems are used to enforce access controls. To this end, we developed a variety of light-weight proof enforcement and consistency models—i.e., Deferred, Punctual, Incremental, and Continuous proofs, with view or global consistency—that can enforce increasingly strong protections with minimal runtime overheads.

We used simulated workloads to experimentally evaluate implementations of our proposed consistency models relative to three core metrics: transaction processing performance, accuracy (i.e., global vs. view consistency and regency of policies used), and precision (level of agreement among trans-action participants). We found that high performance comes at a cost: Deferred and Punctual proofs had minimal overheads, but failed to detect certain types of consistency problems. On the other hand, high accuracy models (i.e., Incremental and Continuous) required higher code complexity to implement correctly, and had only moderate performance when compared to the lower accuracy schemes. To better explore the differences between these approaches, we also carried out a trade analysis of our schemes to illustrate how application-centric requirements influence the applicability of the eight protocol variants explored in this article.

Acknowledgments. This work was supported in part by the National Science Foundation under awards CCF–0916015,
CNS–0964295, CNS–1017229 and IIS–1050301.

References

[1] M. Armbrust *et al.*, “Above the clouds: A berkeley view of cloud computing,” University of California, Berkeley, Tech. Rep., Feb. 2009.

[2] S. Das, D. Agrawal, and A. El Abbadi, “Elastras: an elastic transactional data store in the cloud,” in *USENIX HotCloud*, 2009.

[3] D. J. Abadi, “Data management in the cloud: Limitations and opportu-nities,” IEEE Data Engineering Bulletin, Mar. 2009.

[4] A. J. Lee and M. Winslett, “Safety and consistency in policy-based authorization systems,” in *ACM CCS*, 2006.

[5] M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams, “X.509 internet public key infrastructure online certificate status protocol - ocsp,” RFC 2560, Jun. 1999, http: //tools.ietf.org/html/rfc5280.

[6] E. Rissanen, “extensible access control markup language (xacml) version 3.0,” Jan. 2013, http: //docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html.

[7] D. Cooper *et al.*, “Internet x.509 public key infrastructure certificate and certificate revocation list (crl) profile,” RFC 5280, May 2008, http://tools.ietf.org/html/rfc5280.

[8] J. Li, N. Li, and W. H. Winsborough, “Automated trust negotiation using cryptographic credentials,” in *ACM CCS*, Nov. 2005.

[9] L. Bauer *et al.*, “Distributed proving in access-control systems,” in *Proc. of the IEEE Symposium on Security and Privacy*, May 2005.

[10] J. Li and N. Li, “OACerts: Oblivious attribute based certificates,” *IEEE TDSC*, Oct. 2006.

[11] J. Camenisch and A. Lysyanskaya, “An e cient system for non-transferable anonymous credentials with optional anonymity revocation,” in *EUROCRYPT*, 2001.