CPOL: High-Performance Policy Evaluation

Kevin Borders, Xin Zhao, Atul Prakash
University of Michigan
Department of Electrical Engineering and Computer Science
Ann Arbor, MI 48109-2122
{kborders, zhaoxin, aprakash}@umich.edu

ABSTRACT

Policy enforcement is an integral part of many applications. Policies are often used to control access to sensitive information. Current policy specification languages give users fine-grained control over when and how information can be accessed, and are flexible enough to be used in a variety of applications. Evaluation of these policies, however, is not optimized for performance. Emerging applications, such as real-time enforcement of privacy policies in a sensor network or location-aware computing environment, require high throughput. Our experiments indicate that current policy enforcement solutions are unable to deliver the level of performance needed for such systems, and limit their overall scalability. To deal with the need for high-throughput evaluation, we propose CPOL, a flexible C++ framework for policy evaluation. CPOL is designed to evaluate policies as efficiently as possible, and still maintain a level of expressiveness comparable to current policy languages. CPOL achieves its performance goals by efficiently evaluating policies and caching query results (while still preserving correctness). To evaluate CPOL, we ran a simulated workload of users making privacy queries in a location-sensing infrastructure. CPOL was able to handle policy evaluation requests two to six orders of magnitude faster than a MySql implementation and an existing policy evaluation system. We present the design and implementation of CPOL, a high-performance policy evaluation engine, along with our testing methodology and experimental results.

Categories and Subject Descriptors

K.4.1 [**Public Policy Issues**]: Privacy – *policy enforcement, privacy policy*; C.4 [**Performance of Systems**] – *design studies*.

General Terms

Measurement, Performance, Design, Security.

Keywords

Policy evaluation, performance, privacy policy.

1. INTRODUCTION

Design of policy evaluation systems has been a focus of the security community [1, 4, 2]. However, performance of such systems has generally not received much attention. Applications are emerging, such as privacy enforcement for real-time location-tracking infrastructure, where performance is an issue. Current policy evaluation systems are unable to deliver the needed throughput for these applications. In fact, our work was motivated by an NSF infrastructure project to deploy a location sensing network in our computer science building and the need for a privacy/access control infrastructure to allow only authorized access to users' location data.

Enforcing privacy in a location sensing network presents unique challenges. Unlike traditional access control, access to a person's location information may depend on the current environment and state of the user, such as the time of day and the user's location. For example, employees may not want their locations to be known after work hours or when they are outside of their workplace. In addition, location sensing networks have the potential to receive a large number of queries depending on the number of users. In one of our simulations, with one thousand users occasionally monitoring the locations of friends, looking up information about others nearby, and browsing buildings to look for empty labs, conference rooms, or places to study, over 50,000 requests were generated in one second. Determining whether or not to grant access to a user's location alone can be complicated due to the potential complexity of access conditions. This problem is exacerbated even further by having a large volume of requests.

Current policy enforcement systems are able to express complex access conditions based on a user's location and time of day, but are unable to handle a large number of requests that are likely to be seen in a real location sensing network. Keynote is a popular trust management system that has been used in the past to enforce privacy constraints in a location publish/subscribe framework [11]. The current implementation of Keynote, however, was not adequate to handle large workloads. In our experiments with the system (described in Section 4.1), it was only able to handle a few requests per second with 500 principals in the system. With only 100 principals, KeyNote was able to do a lot better, but this number falls far short of what would be required for a medium or large-sized location sensing network.

Another potential solution for enforcing privacy policies is to use a database management system (DBMS). A DBMS already has the power to manage large datasets, and is flexible enough to allow for expressive access conditions. Restrictions based on the time of day and a user's current location can be placed in policy table entries and evaluated at query time using Boolean 'WHERE' clauses. Despite being fairly straightforward to set up and use, the maximum throughput of a dedicated and optimized MySql database [10] is only a few thousand queries per second, which is not enough to handle real-time queries for a moderately-sized location infrastructure.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CCS'05, November 7–10, 2005, Alexandria, Virginia, USA.
Copyright 2005 ACM 1-59593-226-7/05/0011...$5.00.

To address the need for a high-performance policy evaluation system, we propose CPOL, a flexible C++ framework for policy management. In this paper, we focus on using CPOL to evaluate privacy policies for location-aware services. However, CPOL's architecture and programming interfaces are general-purpose, allowing it to be applied to other domains. Examples of how to do this are given throughout the paper.

The main goal of CPOL is to deliver good throughput with a strong emphasis on maintaining expressiveness. Our objective is for CPOL to be able to handle requests from clients in a medium to large-sized location sensing infrastructure (10,000 or more users). It is equally important, however, that it can express a wide range of policies so that users are able to have fine-grained control over their private information.

CPOL is designed to be comparable in expressiveness to the KeyNote trust management system [1]. In some cases, CPOL ends up being even more expressive than KeyNote. CPOL has support for groups/roles and provides more control over delegation. A more detailed discussion of the differences between KeyNote and CPOL can be found in section 3.1.

One key design feature of CPOL that enhances its performance is a policy evaluation cache. CPOL can cache results more effectively than other systems, while still maintaining correctness, because it is better at determining when to invalidate entries. A typical DBMS cache, for example, only works if query parameters stay exactly the same between requests. If one of the query parameters is the current time, then caching in a real-time system becomes completely ineffective. Using CPOL, however, the application developer can restrict the domain of access conditions. This helps because a single policy can be forced to use a limited set of time intervals, which allows the caching subsystem to calculate an accurate time-to-live for each entry, significantly increasing the hit rate. A similar mechanism can be used with conditions on other variables to calculate how much they can change before invalidating the result.

To evaluate CPOL, we used it to process queries for simulated movement data in a university building. We collected information about privacy preferences, usage scenarios, and daily habits by interviewing thirty potential users. Using that information, we simulated users going to and from class, labs, offices, restrooms, and vending machines throughout the day. Queries on the system were generated by users looking up the locations of acquaintances, others nearby, or everyone in a building. Based on this workload, the CPOL evaluation engine was able to handle queries from approximately 300,000 users in real time with a sample rate of 30 seconds. A MySql database was only able to handle 5,000 users, while KeyNote could not even handle 1000.

The remainder of this paper is laid out as follows: section 2 discusses related work, section 3 presents the design and implementation of CPOL, section 4 describes the evaluation and results, and section 5 wraps up with a conclusion and future work.

2. RELATED WORK

The work most directly related to CPOL is in the area of policy specification and evaluation. Two popular policy systems that are similar to CPOL are Keynote and Ponder [1, 4]. Keynote is a trust management system that consists of an assertion language and a compliance checker. The assertion language is expressive enough to represent conditional access policies based on time and location. It does have a few limitations, however, which are discussed later in the comparison with CPOL (Section 3.1). The compliance checker works well for checking a small number of policies, but was found to have low throughput when evaluating a large number of policies, as can be the case with access control enforcement for sensor data in a large sensor network. To be fair to Keynote, it was intended for use in a distributed environment where each node only has a small number of policies. Nevertheless, CPOL is not restricted to use in a distributed architecture.

Ponder is another policy specification language that supports many different types of policies for wide variety of applications. The Ponder language is more expressive than KeyNote and CPOL in several ways. It supports obligation policies as well as negative authorizations and policy typing/inheritance. Although Ponder can express a wide range of policies, it does not address performance requirements. CPOL provides a solution for enforcing policies in applications where performance is an issue, while still retaining a high level of expressiveness.

There has also been significant research in the field of privacy for ubiquitous computing and location aware services. A general toolkit called Confab is available for developing privacy-aware applications [7]. Confab provides full support for conditional access to location data based on the current time and the content of the data. The key difference between Confab and CPOL is that Confab relies on individuals in the system to manage their own information. This includes handling location requests and enforcing privacy policies locally. In Confab, policy evaluation overhead is not much of an issue, and is not discussed in [7] because a single machine only has to handle requests for one user.

Confab is not the only system that pushes privacy management to the end user; Spreitzer and Theimer take the same approach in their system [13] This architecture does have its benefits, such as inherent performance scalability and not having to trust a central server to properly protect one's privacy. However, it presents a deployment scalability problem by requiring each individual user to find a trusted machine on which to run a privacy agent. This can be particularly problematic if many of the users do not have persistent internet connections at home, or turn their computers off when not in use. CPOL instead focuses on the problem of policy enforcement using a centralized server, even if some of the users in the system opt to deploy their own privacy agents. To address computational scalability concerns, CPOL is designed is designed to handle a large volume of requests, and is designed in such a way that it could be distributed over multiple machines in the future.

CPOL is also applicable to other domains requiring high throughput. Consider a situation where users of a mobile messaging service want to only receive messages from friends and family when at home or on vacation, and from co-workers or immediate family members when at work. Using CPOL, these permissions may be processed efficiently for a large number of users by the service provider at a central server. This can be useful if devices operate under power constraints or are frequently turned off.

The database community has also done significant work on performance issues related to location-aware applications [9, 6, 14]. Standard transactional databases, such as MySql [10], are not very good at handling continuous queries on real-time data. To deal with this problem, special-purpose databases have been developed that are optimized for continuously updated spatio-temporal datasets [9, 6, 14]. Although these systems are able to handle a very large number of queries, they do not address the

CPOL Access Rule Fields	
Owner:	The owner is the entity whose resources are controlled by this rule.
Licensee(s):	The licensee is the entity or group that will receive privileges. If multiple licensees are specified, then all licensees must request access together for the rule to apply.
Access token:	The access token contains information about the rights assigned by this rule.
Condition:	CPOL verifies that the condition is true before granting the access token to the target.

Figure 1. A CPOL access rule has four fields: a rule owner, rule target, access token, and condition. Rules govern access to all entities in the system.

potential privacy implications associated with distributing location data. CPOL provides a solution to this problem by being able to handle large number of privacy requests in real time.

3. DESIGN

CPOL is designed with the following goals in mind:

1. It provides a high level of expressiveness. In particular, our goal was that its policies be comparable in expressiveness to KeyNote policies.
2. The policy engine can process a large volume of evaluation requests on a single machine.

To provide a specific context for discussion, we will examine how CPOL can be used to enforce privacy policies in the domain of location-aware services. In this scenario, each user has a location that is protected by access rules, and can query other users' locations. Users may also join roles such as "Students" or "Professors," and can give access to an entire role using a single policy. To provide easier management, people also have the ability to delegate administrative rights to others. Along with the ability to delegate, people are also able to revoke rules created by a particular user. This can be useful if someone who has been given delegation rights is later found to be malicious or has a password compromised.

The rest of the section presents the detailed design of CPOL. It is laid out as follows: section 3.1 compares CPOL to KeyNote, section 3.2 presents CPOL's top-level architecture and interfaces, section 3.3 discusses access tokens, section 3.4 talks about access conditions, and section 3.5 provides an in-depth discussion of caching in CPOL.

3.1 Comparison with KeyNote

This section highlights CPOL's expressiveness, as well as its limitations, by comparing it to KeyNote [1], another policy enforcement system that is widely used. CPOL can express a wider range of policies than KeyNote in some cases. KeyNote is able to have recursively-nested condition clauses and complex licensee fields, but it lacks control over delegation privileges, roles, and multiple access levels. Having support for roles and greater control over delegation makes CPOL much better suited for location privacy enforcement.

CPOL's current prototype has a C++ interface for specifying policies, rather than a custom language. In general, the interface can be connected to a policy reader, a GUI front-end, or a web server for policy management. In this section we will show CPOL's rules in a KeyNote-like syntax for the purpose of comparison.

The Keynote trust management system is structured around chains of signed assertions that authorize principals to execute actions. At the root of each chain is a policy authorized by the system that gives a principal control over a certain domain of actions. For example, in the location privacy system, a delegation chain may consist of:

1. A policy authorized by the system giving complete control to an administrator
2. A policy authorized by an administrator giving a user Alice complete control over her location data.
3. A policy authorized by Alice giving Bob access to her location 9 AM to 5 PM Monday – Friday.

Both CPOL and KeyNote are based on "closed-world" assumption – a principal does not have a right unless explicitly authorized by the system and there are no negative rights.

In KeyNote, the result of evaluating a policy for a requested action is often true or false; but in general, KeyNote returns a *compliance value* that can represent a variable level of trust. CPOL generalizes that notion by returning an *access token* (a multi-variable object). This access token encapsulates a set of allowed rights, instead of a single right. Consider a situation where CPOL is used to determine a user's access rights to a file in a filesystem. A user would have to make three separate calls to KeyNote to determine read, write, and execute status (because each right may be delegated separately and rules would have to be evaluated for each right). CPOL returns an access token that lists all privileges in one request. Figure 1 shows the fields of a CPOL rule in a Keynote-style syntax.

```
Owner: Alice
Licensee: Bob
AccessToken {
        LocationResolution = RoomLevel
        IdentityResolution = Name
        DelegationPrivileges = None
}
Condition  {
        AfterTime = 9 AM
        BeforeTime = 5 PM
        InBuilding = {Library, CS}
        NotInRoom = {ConferenceRoom 1010 CS}
}
```

Figure 2. CPOL rule giving Bob access to Alice's location with room-level precision (with no delegation privileges) 9 AM – 5 PM when Alice is in the library or the CS building, except in conference room 1010.

One limitation of KeyNote is the inability to control delegation. Once Alice has given Bob access to her location, Bob implicitly has the ability to give that same level of access to anyone else. Alice may not want Bob to grant his rights to others. KeyNote has no obvious way of controlling this. Like KeyNote, CPOL also supports delegation of rights. In CPOL, however, delegation privileges can be explicitly separated from other access rights, giving authorizers more control over their information. An example of a CPOL policy that grants Bob access to Alice's information, but does not allow Bob to delegate rights can be seen

Evaluation Interface	
`AccessToken GetAccess(IDList Requester, EntityID Owner, State Inputs)`	
Management Interface	
Entity Functions	
`EntityID`	`CreateEntity(String Description)`
`Boolean`	`RemoveEntity(EntityID Remove)`
Group Functions	
`GroupID`	`CreateGroup(EntityID Owner, String Description)`
`Boolean`	`RemoveGroup(EntityID Requester, GroupID Remove)`
`Boolean`	`AddMember(EntityID Requester, GroupID Group, EntityID Member)`
`Boolean`	`RemoveMember(EntityID Requester, GroupID Group, EntityID OldMember)`
`Boolean`	`SetGroupRights(EntityID Requester, GroupID Group, EntityID User, GroupToken Rights)`
Rule Functions	
`RuleID`	`AddRule(EntityID Requester, EntityID Owner, IDList Licensee, AccessToken A, Condition C)`
`Boolean`	`RemoveRule(EntityID Requester, EntityID Owner, RuleID OldRule)`

Figure 3. CPOL's external functions. CPOL also has functions to enumerate entities, groups, and rules that are omitted here for the sake of brevity.

in Figure 2. Of course, Bob could leak or misuse Alice's location data contrary to Alice's wishes, but that is beyond the scope of our threat model.

CPOL, unlike KeyNote, supports roles to help simplify policy administration as well as to improve performance. In CPOL, roles can be created with a unique identifier and have members added to them. Then, a person can grant rights to all the members of a role by specifying that role's identifier as the recipient of access rights. This functionality is very useful for location privacy management. A user may wish to make some location information available to a large group of people such as "Students" or "Faculty." Using KeyNote, there is no easy way of granting access to large groups of people. It could be done by individually assigning rights to every member of a group, or by assigning rights to a group key. Some groups may have hundreds or thousands of members in a large system, making both of these solutions very impractical. The first method would require individual users to maintain hundreds of policies. Using group keys creates a separate key management problem, and also requires policies to be updated whenever the key changes due to a membership update. CPOL's solution is much more straight-forward and does not add this additional overhead.

KeyNote does provide several features that are not supported in CPOL. In the "Licensees:" field of a KeyNote policy, one can specify Boolean expressions instead of a single key. CPOL allows for a list of principals that must all be present for the access to succeed, but does not directly support K-of or a logical OR of licensees. However, much of the functionality of a logical OR can be achieved through the use of roles. For the purposes of enforcing privacy, we found it useful to require multiple requesters under some circumstances, such as allowing emergency personnel to access a users' location data only when it is also requested by an authorized manager. Being unable to support more complex licensee sets was not a significant limitation.

In the "Condition:" field of a KeyNote policy, it is possible to nest clauses and grant different levels of trust. CPOL does not support nested conditions or multiple access levels in a single policy. Instead, these complex conditions must be reduced into multiple single-condition policies. An example of a nested KeyNote condition would be "A=True -> (B=True -> Allow, C=True -> Log)", which is the same as two policies with conditions: "A=True && B=True -> Allow" and "A=True && B=True -> Log". Requiring users to split up policies this way again was not a significant limitation.

KeyNote also allows principals to digitally sign policies so they can be distributed over an un-trusted medium. This allows creation of policies in a decentralized fashion; if Alice wants to grant a right to Bob, she can simply sign a policy rule granting that right and send it to him. CPOL does not currently support this mechanism of policy distribution. In CPOL, Alice would have to authenticate to a policy administration service that is authorized to update the global rule set in order to make changes. The policy service is then responsible for tagging rules with Alice as the author. It can use any authentication mechanism, such as Kerberos or public keys, independent of the CPOL system.

Finally, CPOL provides an incremental way to update the global rule set. The rule can only be updated by adding or remove individual rules. CPOL takes advantage of this when managing cache entries. Instead of invalidating the whole cache when a change is made to the rule set, CPOL only has to invalidate a small subset of the entries (those that with the same authorizer) to guarantee correct results.

3.2 Top-Level Architecture and Interfaces

The CPOL policy engine is designed to be deployed along with a data management system and a web or application server. It has two top-level interfaces, one for communicating with each of these two external processes. The functions associated with each interface can be seen in Figure 3. CPOL's evaluation back-end has only one function, *GetAccess*, which determines an entity's access rights given an owner and a set of inputs. The evaluation interface is designed to be connected to a data management process that is responsible for enforcing access policies. In the case where CPOL is used to manage location privacy, the data management process is likely to be a publish/subscribe type system [11] or a modified spatio-temporal database [9]. The data manager should be able to understand rights that are returned in the access token by *GetAccess*, and is solely responsible for

passing correct parameters and ensuring that access levels are enforced properly.

CPOL's management front-end contains functions to add and remove entities, roles, and access rules from the system. The management interface can be connected to a web server that uses authentication and encryption, giving users easy and secure access to their privacy policies. The management interface's functions are described here to give a better idea of how users can interact with CPOL. The first two functions available in the management interface, *AddEntity* and *RemoveEntity*, are used to create and delete people and objects in CPOL. These functions do not check any credentials; they assume that the call is being made by an administrative user. It is the font-end's job to authenticate administrators and not allow normal users to make calls to *AddEntity* and *RemoveEntity*.

The next set of management functions is used to create and modify groups. These functions do take the identity of the requesting entity as an argument, and should be exposed to all users so that they can create and modify their own groups. Privileges to update groups, list members, and use groups in access rules can be assigned by calling *SetGroupRights*. The group creator owns the group by default, and must explicitly grant permissions to other users for them to be able to access the group. Universal groups such as 'Everyone' and 'Students' should be maintained by a system administrator to ensure their integrity.

The last two management functions, *AddRule* and *RemoveRule*, are used to modify access policies. Each function takes the identity of the requester as an argument. If the requester and the owner are the same, then the function call is automatically allowed. If the requester is not the owner, then the function checks to see if the owner has given the requester privileges to perform the operation. If this is the case, then the function call is allowed, otherwise it will return with permission denied. If the *AddRule* function call is allowed, then a new access rule will be inserted into the owner's policy list. The rule grants the licensee a given level of access A when condition C is true. The rule licensee can be an entity ID, group ID, or list of IDs. If it is a list, then all entities in the list must collectively request access in order for the rule to apply. This mechanism can be used to require multiple requesters for access to sensitive information.

Formally, CPOL can be represented as a five-tuple {*E*, *G*, *M*, *A*, *R*} of entity IDs, group IDs, membership pairs, access tokens, and rules. These elements have the following properties:

- ***E* – Entity IDs** uniquely identify people and objects in the system. Each entity is able to request access to other entities' resources and can have policies governing access to its own resources.
- ***G* – Group IDs** uniquely identify groups of entities. Groups in CPOL are equivalent to roles in traditional role-based access control [5].
- ***M* – Membership Pairs** associate groups with entities. Each pair contains one group ID *g* and one entity ID *e* and can be represented as <*g*,*e*>. A mapping $G \rightarrow E$ or $E \rightarrow G$ can be derived from the set of membership pairs.
- ***A* – Access Tokens** represent privileges that may be assigned in the system and can be any arbitrary set of values. In the simplest case *A* could be {*true*, *false*}. For a file access control system, *A* could be something more complex like the power set of {*Read*, *Write*, *Execute*}. The next section discusses access token definition in greater detail.
- ***R* – Rules** consist of four fields as seen in Figure 1. These fields have the following properties:
  - ***e* – Owner** must be a member of *E*, $e \in E$.
  - ***l* – Licensees(s)** must be a subset of $E \cup G$. This licensee set almost always contains one element, unless a rule requires multiple requesters for additional security.
  - ***a* – Access Token** must be a member of *A*, $a \in A$.
  - ***c* – Condition** is an expression that must evaluate to *true* or *false* given a set of inputs. Other than this restriction, information stored in the condition, the set of inputs, and the function used to evaluate conditions can be anything. Section 3.4 provides an in-depth discussion of access conditions.

3.3 Access Tokens

Access tokens represent rights that are given to an entity in the system. When a request is made by the database back-end for an entity's resources, CPOL returns an access token indicating the resulting set of privileges. This token is defined by a C++ class, which can contain arbitrary values. These values describe the extent to which access is granted. The application developer can define the *AccessToken* and is also responsible for making sure the data management process can understand its contents. The access token also has four member functions that must be defined. These functions can be seen in Figure 4. They are used to combine, compare, add, and remove access tokens. Other than the parameters and the return type, the functions can be defined in any way. How they are defined affects the delegation of rights and what it means for an entity to hold multiple access tokens in the
system.

AccessToken Member Functions	
`Boolean`	`Operator≥(AccessToken`
`AccessToken`	`Add(AccessToken OtherToken)`
`Boolean`	`AddAccess(AccessToken RequesterToken)`
`Boolean`	`RemoveAccess(AccessToken RequesterToken)`

Figure 4. The AccessToken object's four member functions. Operator≥ determines if one token contains another, Add combines two access tokens, AddAccess checks if a requester can add a new rule, and RemoveAccess see if a requester can remove a rule.

To better illustrate the specification of access token values and member functions, we will examine how they are defined in the location-aware privacy enforcement implementation of CPOL. The access tokens have varying permission levels along three dimensions: location resolution (*L*), identity resolution (*I*), and delegation (*D*). The resulting set of access tokens *A* is $L \times I \times D$. The location resolution element restricts the level of detail that can be seen with respect to an entity's position. The five members of *L* are {*LocNone, LocBuilding, LocFloor, LocRoom, LocExact*}, which allow the licensee to see the following information about the authorizer: nothing, current building, floor, room, or exact location. The identity resolution element *I* has five members:

{*IdentNone, IdentPerson, IdentJob, IdentAffiliation, IdentName*}, which respectively disclose: nothing, type (person or object), job (student, professor, etc. in a university), affiliation (computer science dept., etc.), or the authorizer's exact name.

3.3.1 Delegation

Delegation is used in CPOL to allow the holder of an access token to create and delete access rules on behalf of an authorizer. The delegation access token element, *D* has three possible values {*Normal*, *Admin*, *Delegate*}. *Normal* indicates that the grantee cannot create access rules for the authorizer. *Admin* allows the grantee to add and remove rules with access tokens that have up to the same location and identity resolution as the current token, but only have *Normal* delegate access. *Delegate* allows the grantee to create and delete rules with access tokens that have up to the same location and identity resolution as the current token, but may have *Admin* access as well. (A possible modification here would be to allow *Delegate* holders to create *Delegate* tokens as well, instead of only *Admin* tokens. This could lead to arbitrarily long delegation chains, but may be desirable in some situations.)

Access tokens for the privacy enforcement system also keep track of the delegation chain used during their creation. The delegation chain field, *DChain*, does not affect the level of access granted by the token, and is treated as a comment by the data management system. *DChain* can be used by the management interface to organize access rules. They can be arranged into a tree structure in order to make it easier for the original authorizer to revoke entire branches. The contents of the delegation chain also persist through user updates. So, if Alice grants rights on the behalf of Bob and is later removed from the system, then Bob can still see which rights she added. This feature is useful in the situation where someone with *Delegate* privileges is later found to be malicious and the authorizer wants to delete all the rules created by that person. Also, *DChain* can be used to prevent people from deleting rules that they did not create. For example, assume both Alice and Bob have delegate privileges. If Alice grants access to a third user, Charlie, she may not want Bob to be able to remove Charlie's privileges. This can be enforced by requiring a user to be in a token's delegation chain in order to remove it.

3.3.2 Access Token Functions

The first function in Figure 4, *Operator* ≥, is used to determine if one token contains all the rights granted by another token. CPOL uses this function to cut down on evaluation time. Once an entity has obtained a token, CPOL does not need to continue evaluating rules that are contained by that token. For example, in the simplest case where possible access tokens are {*yes*, *no*}, *yes* ≥ *no* would be true, and *no* ≥ *yes* would be false. In the privacy implementation of CPOL, *Operator* ≥ compares elements for each of the three access dimensions one by one, and is only true if all of the elements in one token are greater than or equal to all of the elements in another token. Here are a few examples:

- *T* ≥ {*LocNone*, *IdentNone*, *Normal*} will always be **True**.
- {*LocFloor*, *IdentJob*, *Normal*} ≥ {*LocBuilding*, *IdentJob*, *Normal*} is **True**.
- {*LocExact*, *IdentName*, *Normal*} ≥ {*LocBuilding*, *IdentPerson*, *Admin*} is **False**.

The next access token function, *Add*, combines two tokens into one. *Add* defines what it means to simultaneously hold two access tokens. Before *Add* is called on two tokens *a* and *b*, CPOL checks that *a* ≥ *b* and *b* ≥ *a* are both false. If one token contains the other, then CPOL just discards the lesser token. If the two tokens *a* and *b* do not contain each other, there are a number of ways to combine them. First, *Add* could simply throw away one of the two tokens using a heuristic to determine which is more preferable. This method does not work very well in situations where access tokens grant independent privileges, such as for {*read*} and {*execute*} file access tokens. Instead of selecting one of the two access tokens, *Add* could combine them by taking the union of their access rights. This would be a good solution for file access privileges, but it does not work well for location privacy. Ignoring delegation, an entity could possess two access tokens: {*LocBuilding*, *IdentName*} and {*LocExact, IdentPerson*}. Having both of these tokens at the same time means that someone should be able to access the authorizer's name and building, or see the authorizer's precise location, but know nothing about the authorizer's identity. Combining these tokens into {*LocExact, IdentName*} would give the authorizer's exact location and name, which could be a severe violation of privacy. To avoid this problem, the location-aware privacy enforcement implementation of CPOL combines two access tokens in *Add* by linking them together to form a list. This way, when an entity holds two access tokens, it can choose which one to use depending to the needs of the application.

The last two access token functions, *AddAccess* and *RemoveAccess*, allow for the delegation of rights. When the owner of a policy list makes a request to add or remove a rule, that request is automatically granted. When someone else attempts to create or delete a rule, however, CPOL must check if that person has been given permission to do so by owner. In this case, *GetAccess* is called to obtain the requester's access token, which is passed to *AddAccess* or *RemoveAccess*. After determining whether the requester has sufficient privileges to create or delete the rule, *AddAccess* and *RemoveAccess* return *true* to allow the operation, or *false* to prevent it. Furthermore, *AddAccess* has a chance to update the new token's delegation chain.

In the privacy enforcement example, *AddAccess* and *RemoveAccess* first check that the requester's access token is greater than or equal to the target rule's access token. Then, the function will verify that the delegation level of the rule's access token is less than the delegation level of the requester. If it is not, then *AddAccess* and *RemoveAccess* return *false*. If the delegation level is higher, then *AddAccess* will copy the delegation chain from the requester's access token to the new access token, appending the requester's identity to the end. *RemoveAccess* will just delete the rule. As an example of how this works, when a user with {*LocBuilding*, *IdentName, Admin*} access tries to add a rule with token {*LocBuilding*, *IdentJob, Normal*}, the operation is allowed. If that same user tried to add a rule with {*LocFloor*, *IdentJob, Normal*} or {*LocBuilding*, *IdentJob, Admin*}, however, then the request would be denied.

3.4 Access Conditions

In CPOL, access conditions give users more control over when others can access their resources. An access condition is a Boolean expression that can take inputs from the surrounding environment. These inputs are encapsulated in a *State* object and can include things such as the current time or the authorizer's

Condition Member Functions	
`Boolean`	`Test(State Inputs)`
`Condition`	`Add(Condition OtherCondition)`
`Condition`	`Subtract(Condition OtherCondition)`

Figure 5. The Condition object's three member functions. Test evaluates the condition, Add combines two conditions, and Subtract combines a condition that is currently true with one that is false.

location. Each rule in CPOL can have a condition associated with it that must evaluate to *true* in order for the rule to apply.

The capability to specify conditional access is especially important when enforcing privacy for location-aware services. Without conditions, anyone who had access to a person's location would be able to see where that person is 24 hours a day 7 days a week. This could cause a variety of problems. As an example, supervisors may want high-resolution access to their employees' location during work hours, while employees may not want their bosses to know where they are on weekends. Location privacy enforcement needs access conditions to be able to handle these types of situations.

To define the range of possible conditions and their meaning within CPOL, the application developer must specify three things:

1. **Condition Inputs**. These inputs are specified within a *State* object. The *State* object is implemented as a C++ class, which can have arbitrary data members. A *State* object is passed into CPOL on every call to *GetAccess*.
2. **Condition Content**. Conditions, much like access tokens, are implemented as a C++ *condition* class. This class can have any arbitrary data members, which are used to store the contents of the condition.
3. **Condition Functions**. The object has three member functions, which can be seen in Figure 5. The *Test* function evaluates a condition. *Add* and *Subtract* are used to combine two conditions into one for cache invalidation.

The manner in which the *State* and *Condition* objects are defined has a major effect on performance and expressiveness. Conditions are evaluated for every single access rule and have the potential to become a bottleneck if the *Test* function is slow. On the other hand, if the *Test* function is too simple, then it can be hard to express useful conditions. To give a better understanding of how the *State* and *Condition* objects should be defined, the remainder of the section describes how they are specified in the location privacy enforcement system.

For the privacy implementation, we decided to use two condition inputs to make up the *State* object: the current time and the authorizer's location. When combined, these two variables can be used determine the authorizer's current situation, the content of the information being disclosed, and that content's sensitivity. According to a study by Lederer et al. [8], these are three of the five most important factors when determining privacy preferences. Two other important factors are the recipient's identity and the intended usage. CPOL already uses the recipient's identity during rule evaluation. The intended usage is not included in the set of condition inputs because intent is very hard to enforce from a security perspective. Once someone has obtained location information, it is easy for that person to transform or store the information in violation intended usage, especially if the data resides on a personal computer. For this reason, *State* objects were limited to the current time and the authorizer's location.

Condition objects in the location privacy system contain two elements that are used to restrict access: location modifiers and time modifiers. A single location modifier specifies a particular area in which access is allowed or denied. The area could be a whole building, a floor, or a single room. An example of a location modifier would be "Allow access when I am in the computer science building." One condition is able to have up to four location modifiers, allowing for a condition such as "Allow access when I am in the computer science building, except when I am on the third floor or in the restroom." A condition is also allowed to have one time modifier, which contains a single time interval and a weekday mask. Some examples of time modifiers are "Only allow access Monday-Friday 9 AM to 5 PM" or "Only allow access Thursday 7 PM to 9 PM." Note here that it would be easy to modify conditions so that they can contain an arbitrarily large number of location and time modifiers. It would be possible to allow negative time modifiers as well (i.e. "Not Monday-Friday 12 PM to 1 PM"). These modifications that a policy administrator could make have performance and security implications. Allowing a larger number of modifiers could degrade performance and may make it easier to run a denial of service attack on CPOL. For the purposes of the location-aware privacy implementation, we found that one time and four location modifiers were sufficiently to express almost all realistic access conditions.

The last part of the *Condition* object is the definition of three member functions *Test, Add*, and *Subtract*, which can be seen in Figure 5. *Test* is called by *GetAccess* for every rule that applies to the requester. *GetAccess* sends *Test* a *State* object with the current set of inputs and *Test* returns a Boolean truth value. For location privacy enforcement, *Test* first checks to see if the current time is within the interval specified by the time modifier. It then goes through the location modifiers one by one making sure that the authorizer is within all of the allowed areas, and not in any of the forbidden ones. If the input state satisfies all time and location modifiers, then *Test* returns *true*, otherwise it returns *false*.

The two other *Condition* functions, *Add* and *Subtract*, are used when more than one rule applies to a requester. These functions do not affect the resulting access token and are used for cache invalidation purposes only. *Add* defines how two conditions are merged when their access tokens are combined using the access token *Add* function. Note that the resulting access condition is only used for cache invalidation, so it does not have to be a precise intersection of two original conditions. For the privacy example, the condition's *Add* function will take the time modifier that expires first and a location modifier with the finest granularity. This way, a cache entry can be given an accurate time-to-live, and will also expire if the authorizer changes rooms, floors, or buildings depending on the level of resolution in the location modifiers. *Subtract* is very similar to the *Add* function. It is called when a rule applies to a requester, but the condition is currently false. In the location-privacy system, the *Subtract* function works just like *Add*, except that it switches the start and end time in the subtracted condition's time modifier. This way, the condition will expire when it becomes valid and could affect the requester's access level. Together, *Add* and *Subtract* ensure that the merged condition will expire if one of the two original

<table>
<tr><th>Bits</th><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td></tr>
<tr><th>Field</th><td colspan="2">Move</td><td colspan="2">Hours</td><td colspan="6">Minutes</td><td colspan="6">Seconds</td></tr>
<tr><th colspan="3">Move Possible Values:</th><td colspan="14">00 – Entry invalidated on Room change
01 – Entry invalidated on Floor change
10 – Entry invalidated on Building change
11 – Entry never invalidated by location change</td></tr>
<tr><th colspan="3">Hours:</th><td colspan="14">0-3 remaining, 3 if no timeout.</td></tr>
<tr><th colspan="3">Minutes:</th><td colspan="14">0-60 remaining or 63 for no timeout.</td></tr>
<tr><th colspan="3">Seconds:</th><td colspan="14">0-60 remaining or 63 for no timeout.</td></tr>
</table>

Figure 7. CacheCondition has a 16-bit field that contains the cache entry's time-to-live and a movement tolerance.

conditions expires, allowing CPOL to correctly cache access tokens.

3.5 Caching

The most critical performance improvement in CPOL is caching. In the location-aware privacy implementation, CPOL was able to process 99.8% of all requests from the cache, reducing the average handling time from 6 µs to 0.6 µs. CPOL is able to achieve such a high hit rate because it gives the application developer control over invalidation, which can be complicated in the case of conditional access. The remainder of this section describes how cache invalidation is handled in CPOL, and gives an example of how it is done accurately and efficiently for the location privacy enforcement system.

The CPOL cache is structured as an in-memory map keyed by <*Requester*, *Owner*> pairs. Whenever *GetAccess* is called by the data management back-end, CPOL uses the *Requester* and *Owner* parameters to look for a cache entry. If an entry exists, then it will contain an *AccessToken*, a *CacheCondition*, and the last input state used to obtain access. The *CacheConditon* is an object defined by the application developer. It may have arbitrary data members, just like a *Condition*, but it should be as compact and easy to evaluate as possible. The member functions of *CacheCondition* can be seen in Figure 6. *GetAccess* tests the *CacheCondition* by calling *StillGood* with the current and previous *State* objects to see if the entry is still valid. If it is, then *GetAccess* updates the previous state in the cache entry and returns the *AccessToken*. If the entry not valid or no entry was found, then *GetAccess* evaluates access normally. When it is done, *GetAccess* stores the resulting *AccessToken*, the current *State*, and a *CacheCondition* in a new cache entry.

CacheCondition Member Functions
`Boolean   StillGood(State LastState, State CurrentState)`
`Void      Set(Condition FullCondition, State FirstState)`

Figure 6. The CacheCondition object's two member functions. StillGood tests the CacheCondition to see if it is still valid, and Set initializes the CacheCondition from a full access condition and a current state.

In addition to invalidation using *CacheCondition* objects, entries can also be removed by eviction, modification of the owner's rule list, or change in the requester's group membership. Replacement occurs when the cache has exceeded a pre-defined maximum size. An entry is selected for eviction using a standard clock replacement algorithm. Valid entries are not evicted very often because the cache is stored in main memory can be very large.

To ensure that policy updates go into effect immediately, a mapping can be kept from owners to cache entries at the option of the policy administrator. This way, all the cache entries for a particular owner can be removed when that owner's rule set is modified. This additional mapping will take up extra memory, but it provides better security guarantees. Without an owner to entry mapping, policy updates may take hours or days to go into effect if stale entries exist with long timeout values. Another mapping from requesters to cache entries can be kept so that group membership updates to go into effect immediately. Entries with the same requester can be thrown out when the requester's group membership list changes. This mapping also takes extra memory and provides better security guarantees.

In the location privacy system, *CacheCondition* objects contain a timeout and a movement tolerance. These two values are stored in a16-bit field, which can be seen in Figure 7. When the *Set* function is called with a *Condition* object, it first calculates the amount of time left until the time modifier will expire. If there is no time modifier, then it sets all of the time remaining bits to 1 to indicate no timeout. Next, *Set* looks at all of the location modifiers, and sets the movement tolerance to one of {*Room*, *Floor*, *Building*, *None*} based on the most fine-grained modifier.

In the location privacy system, when *GetAccess* calls *StillGood* to determine if a condition is still true, *StillGood* looks at the difference between the times and locations of the new and old states. First, it checks to see if the time since the last request is greater than the timeout value. If it is, then *StillGood* returns *false*, otherwise it subtracts the time difference from the time remaining in the *CacheCondition*. Next, *StillGood* verifies that the owner has not changed rooms, floors, or buildings since the last request, depending on the movement tolerance. If the two states pass both of these tests, then the cache entry is valid and *StillGood* returns *true*, otherwise it returns *false*.

Using these simple cache conditions, CPOL is able to achieve a very good hit rate and a fast processing time. Because the *CacheCondition* is not as precise as the original condition, some entries with fine-grained location modifiers may be invalidated when the original condition is still true. This has a minimal effect on hit rate, however, since people tend to stay in the same room for hours at a time. On the other hand, only looking at the difference between times and locations allows cache hits to be processed very quickly taking an average of .33 µs, which is much faster than processing full conditions.

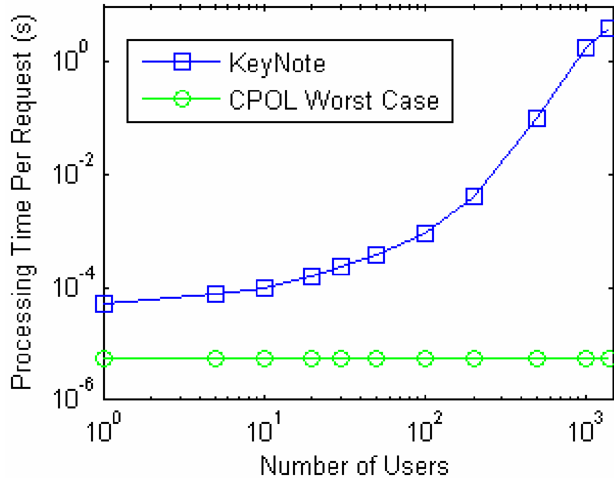

Figure 8. KeyNote vs. CPOL request time comparison.

4. EVALUATION

In this section, we test CPOL's performance by measuring query processing time, memory usage, and ability to handle requests from a simulated location infrastructure. For these experiments we configured CPOL, as well as KeyNote and a MySql database, to handle privacy requests from a location sensing network. All of the tests were run on the same computer, which had an AMD Athlon™ XP 2200 processor and 512 MB of RAM. The first set of benchmarks measure processing time and give a good representation of CPOL's general performance. These results do not rely on a specific cache hit rate and can be extrapolated using the expected hit rate and number of conditional access rules to come up with expected evaluation times for a particular application. The memory usage statistics are also applicable to implementations other than privacy enforcement, and can be adjusted based on the expected size of *State*, *AccessToken,* and *Condition* objects, as well as the size of the cache.

The final part of the evaluation presents results from a simulated privacy query workload. This workload was generated using a schedule-based movement simulator and is specific to privacy enforcement for location-aware service. These results do not necessarily apply to other scenarios due to varying request load and cache hit rate, but they do provide a good example of CPOL's capabilities. The simulations also give insight into how KeyNote and MySql's longer request processing times translate into an overall performance penalty.

4.1 Individual Requests

For the first part of the evaluation, we compared individual request times from CPOL, KeyNote, and MySql. Each system was initialized with n users where n varied from 1 to 1000. The requests were made by one user requesting access to someone else's location. No roles were used since they are not supported by KeyNote. In all three systems, each entity's rule set was populated with ten rules, with each rule granting access to one other randomly selected user These rules also had the following conditions: (1) Time had to be "Monday through Friday 9 AM to 5 PM" (2) The authorizer's had to be on one of two floors in a particular building, but not in one of two specific rooms. In KeyNote, the access conditions were stored in the "Condition:" clause. In CPOL, they were set directly inside of the *Condition* object. In MySql, each condition was stored in the rule table and evaluated at query time in the "WHERE:" clause. Note here that the total number of rules for this simulation is $10 * n$.

Table 1. Individual Request Processing Times for KeyNote, MySql, and CPOL.

Request Type (500 Users, 5000 Rules)			Processing Time (µs)
KeyNote			101,000.00
MySql Database			459.00
CPOL	Cache Hit		0.33
	Cache Miss	Access	5.50
		No Rule	3.50
	No Cache	Access	4.50
		No Rule	2.00

After making requests with the number of users n varying from 1 to 1000, we observed that the processing times stay constant for CPOL and MySql, while they increase dramatically for KeyNote. Figure 8 shows the difference in time between KeyNote requests and worst case CPOL requests (Cache miss on a successful access). KeyNote's request processing time starts out at 53 µs for n=1. At this point, a worst-case CPOL request takes 5.5 µs, approximately ten times faster than a KeyNote request. As the number of users increases, however, KeyNote's processing time starts growing very quickly. With n=50 users, a KeyNote request takes 374 µs, increasing to 936 µs at 100 users, 101 ms at 500 users, and so on up to 1.77 seconds for a *single* request with n=1000 users. In the worst-case CPOL still only takes 5.5 µs to process a request for the same 1000 users. Although KeyNote has satisfactory performance for a low number of users, it becomes extremely slow after more than a few hundred users with ten policies each.

Table 1 shows request processing times under different circumstances for CPOL compared to MySql and KeyNote. For a cache hit, CPOL is able to process a request in 0.33 µs, which is three orders of magnitude better than a MySql database and five to six orders of magnitude better than KeyNote. If a request misses in the cache, or if a cache is not used, then CPOL requests take approximately ten times longer, anywhere from 2 µs to 5.5 µs. The exact time depends on whether CPOL has to evaluate a condition for one of the ten rules. For implementations with many complex access conditions, requests may take a bit longer than 5.5 µs. Note here that with more complicated conditions, the database will take longer than 459 µs as well. Half of MySql's processing time is already spent evaluating conditions. In a system with mostly unconditional access rules and a cache, CPOL misses are likely to take closer to 3.5 µs. With a hit rate of 90%, most implementations should be able to handle a sustained throughput of approximately 1,000,000 requests every second.

4.2 Memory Usage

The next part of the evaluation looks at CPOL's memory consumption in relation to the number of users, roles, policies, memberships, and cache entries in the system. Memory usage is especially important because CPOL is an in-memory system. The tests used in this section were run on the privacy enforcement implementation of CPOL. Although the *State*, *Condition*, *AccessToken*, and *CacheCondition* objects all have application-specific definitions, their sizes only affect rules and cache entries,

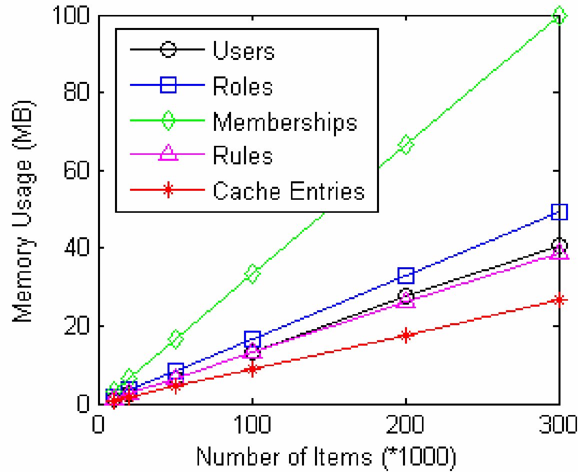

Figure 9. Memory usage per item for Users, Roles, Memberships, Access Rules, and Cache Entries.

which are likely to be similar between implementations. The sizes also include indexing overhead, so discrepancies between object definitions for different applications will not have a major impact on total memory usage.

Figure 9 shows the amount of memory consumed per item for users/entities, roles, role memberships, access rules, and cache entries. These five items are all pretty close in size with membership entries being the largest and cache entries the smallest. Memberships take up more space because a two-way mapping is kept for easy role management and quick lookup at request time. Cache entries are smaller than other objects because they are stored in a more compact format. Using the sizes in Figure 9, a location privacy system that has 500 MB available, where each user has ten access rules and is a member of ten roles (10,000 total roles), can support approximately 500,000 entities with a 2,000,000 entry cache. This number could be increased slightly by optimizing CPOL's data structures for space. However, memory was not a bottleneck when evaluating privacy requests for large numbers of users.

4.3 Privacy Request Workload

For the final step of the evaluation, we used CPOL to process a simulated workload of privacy queries in a location-aware environment. The problem for us was coming up with a suitable workload, given that the technology is not yet widely deployed. GSTD is one general-purpose movement generator [15]. It uses randomly moving points to represent objects in the system, creating chaotic behavior. This can useful in some applications, but is not a very realistic representation of typical human movement. There are special-purpose generation algorithms that produce more realistic movement patterns for particular applications such as vehicle traffic [3] and fishing boat movement [12] simulation. When testing CPOL, we found that GSTD and special-purpose generators do not do a very good job of generating realistic movements for people inside of a building. Instead of aimlessly walking around throughout the day, people have fixed schedules that include classes, lab time, meetings, etc. and tend to stay in one place for duration of a scheduled event. To evaluate CPOL, we needed a movement generation algorithm that accounted for these factors. Accurately modeling human movement is particularly important for evaluating the effectiveness of caching.

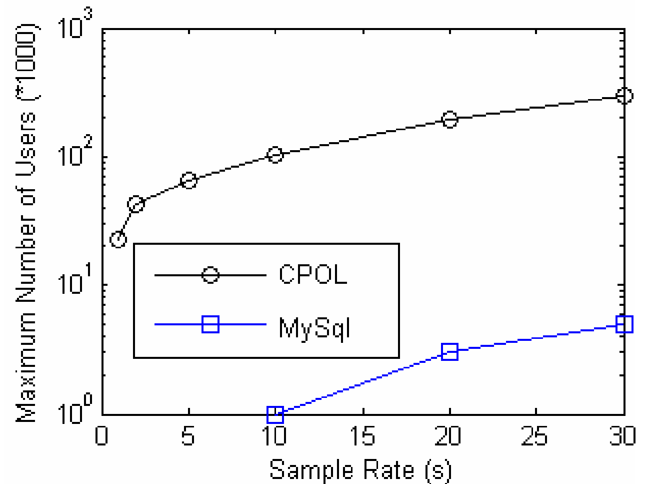

Figure 10. Maximum number of supported users for different location sample rates.

For above reasons, we created a new movement generation algorithm that uses semi-random scheduling. The simulator is designed for a university environment, where users go between classrooms, labs, offices, bathrooms, and vending machines. During initialization, the simulator fills classroom schedules with different "courses." Once courses have been created, each user "enrolls" using a target number of credits. Throughout the week, users enter and leave the building through random exits, attending class and spending time in offices, labs, and other locations. The amount of time spent in the building outside of class is determined by a target number of hours.

To get a better idea of how many hours students typically spend in class, offices, labs, or other locations, we interviewed thirty potential users. During these interviews, we also asked people about their personal privacy preferences and what services they would be interested in using. The three main uses of location information were looking up friends or associates ("Show person A"), querying information of nearby users ("Show people in this room"), and general building-wide requests ("Show everyone in this building") to look for empty computer labs or quiet places to study. The last type of query was optimized by only allowing users to request access as a role member such as "student," instead of as an individual. This way, the database back-end only has to query every user and prepare the query result once per time step for all requesters. The service usage information was applied to the simulator, which generated requests based on frequency of use and the percentage of people who said they would run each query. Finally, each person's privacy preferences were reflected in CPOL by creating access lists and friend roles based on answers from the interviews.

After CPOL was initialized with individual privacy preferences, it read requests from a simulator output file. This output file contained requests from users in one building of average size (it had four lecture halls and four classrooms in which 1000 students attended class throughout the week). To test CPOL's maximum capacity, the requests and users from the single building were replicated multiple times, modeling a campus-wide workload. The location update rate also varied from one to thirty seconds. For each update rate, the number of buildings was continually increased until CPOL was unable to process all of the queries for a simulation step before it was over.

We also ran the same experiment on a MySql server and on KeyNote. The results from CPOL and MySql can be seen in Figure 10. Keynote is not shown here because it was unable to process queries for users in a single building in 30 seconds. MySql was also unable to handle requests in real-time for 1000 users with sample times of less than 10 seconds. With a thirty-second sample rate, MySql was able to handle requests for approximately 5,000 users, while CPOL was able to handle approximately 300,000, sixty times more than MySql.

5. CONCLUSION AND FUTURE WORK

Traditionally, performance has not been major focus in the design of policy systems. Applications are emerging, however, that require policies to be evaluated with a low latency and high throughput. CPOL provides a good solution for evaluating policies in such applications. It is more expressive than KeyNote in many cases, and it delivers much better throughput. During evaluation, we found that CPOL was able to process a single request two to three orders of magnitude faster than a similar database solution, and four to five orders of magnitude faster than KeyNote, depending on the expected cache hit rate. To test a more realistic workload, CPOL was set up to enforce privacy constraints for location-aware services. Using a generator that modeled people moving in a university building, CPOL was able to handle requests from approximately 300,000 users in real-time with a thirty-second sample rate, while a MySql database was only able to support 5,000 and KeyNote was too slow to do real-time processing for even 1,000 users.

In the future, we would like to expand CPOL to support distribution across multiple machines. The request load could be split up amongst a group of policy evaluation engines, allowing for even greater scalability. In addition, we hope to use CPOL for policy management in other application domains, and explore the integration of different GUI front-ends for easy policy management.

ACKNOWLEDGEMENTS

Kevin Borders and Atul Prakash were funded in part by a grant from the Intel Corporation. The equipment for the research was funded by the National Science Foundation under grants 0303587 and 0325332. We thank our colleague, Professor Jignesh Patel, for discussions on database research in the area of privacy and on continuous queries on spatial data.

REFERENCES

[1] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The KeyNote Trust Management System Version 2. *Internet RFC 2704*, September 1999.

[2] M. Blaze, J. Feigenbaum, and M. Strauss. Compliance Checking in the PolicyMaker Trust Management System. In *Proceedings of the Financial Cryptography Conference, Lecture Notes in Computer Science*, vol. 1465, pages 254-274. Springer, 1998.

[3] T. Brinkhoff. A Framework for Generating Network-Based Moving Objects. *GeoInformatica*, Vol. 6, No. 2, 153-180, 2002.

[4] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. In Morris Sloman, editor, *Proceedings of Policy Worshop*, 2001, Bristol UK, January 2001.

[5] D. Ferraiolo and R. Kuhn. Role-based access control. In *Proceedings of 15th NIST-NCSC National Computer Security Conference*. Baltimore, MD. pp. 554–563, October 1992.

[6] B. Gedik and L. Liu. Mobieyes: Distributed processing of continuously moving queries on moving objects in a mobile system. In *Proceedings of the 9th Conference on Extended Database Technology (EDBT 2004)*, Heraklion-Crete, Greece, March 2004.

[7] J. Hong and J. Landay. An Architecture for Privacy-Sensitive Ubiquitous Computing. In *Proceedings of the Second International Conference on Mobile Systems, Applications, and Services (Mobisys 2004)*. Boston, MA. pp. 177-189, 2004.

[8] S. Lederer, C. Beckmann, A. Dey, and J. Mankoff. Managing Personal Information Disclosure in Ubiquitous Computing Environments. University of California, Berkeley, Computer Science Division, *Technical Report UCB-CSD-03-1257*, July 2003.

[9] M. Mokbel, X. Xiong, and W. Aref. SINA: Scalable Incremental Processing of Continuous Queries in Spatio-temporal Databases. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, Paris, France, pp. 623-634, 2004.

[10] MySQL, Inc. The mysql database manager. http://www.mysql.org, 2004.

[11] L. Opyrchal, A. Prakash, A. Agrawal, "Designing a Publish-Subscribe Substrate for Privacy/Security in Pervasive Environments." In *First Workshop on Pervasive Security, Privacy and Trust (PSPT)*, Boston, MA, August 2004.

[12] J. Saglio and J. Moreira. Oporto: A Realistic Scenario Generator for Moving Objects. *GeoInformatica*, Vol. 5, No. 1, 71-93, 2001.

[13] M. Spreitzer and M. Theimer. Providing location information in a ubiquitous computing environment. In *Proceedings of Fourteenth ACM Symposium on Operating System Principles*. Asheville, NC: ACM Press, December 1993.

[14] Y. Tao and D. Papadias. Spatial Queries in Dynamic Environments. *ACM Transactions on Databases Systems* (*TODS*), 28(2): 101-139, 2003.

[15] Y. Theodoridis, J. Silva, and M. Nascimento. On the generation of spatio-temporal datasets. In *Proceedings SSD*, 1999.