CLUSTERING AND RECOMMENDATION TECHNIQUES FOR ACCESS CONTROL POLICY MANAGEMENT

by

Said M. Marouf

A dissertation submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in
Computing and Information Systems

Charlotte

2012

Approved by:

Dr. Mohamed Shehab

Dr. Bei-Tseng Chu

Dr. Cem Saydam

Dr. Taghi Mostafavi

ABSTRACT

SAID M. MAROUF. Clustering and recommendation techniques for access control policy management. (Under the direction of DR. MOHAMED SHEHAB)

Managing access control policies can be a daunting process, given the frequent policy decisions that need to be made, and the potentially large number of policy rules involved. Policy management includes, but is not limited to: policy optimization, configuration, and analysis. Such tasks require a deep understanding of the policy and its building components, especially in scenarios where it frequently changes and needs to adapt to different environments. Assisting both administrators and users in performing these tasks is important in avoiding policy misconfigurations and ill-informed policy decisions. We investigate a number of clustering and recommendation techniques, and implement a set of tools that assist administrators and users in managing their policies. First, we propose and implement an optimization technique, based on policy clustering and adaptable rule ranking, to achieve optimal request evaluation performance. Second, we implement a policy analysis framework that simplifies and visualizes analysis results, based on a hierarchical clustering algorithm. The framework utilizes a similarity-based model that provides a basis of risk analysis on newly introduced policy rules. In addition to administrators, we focus on regular individuals whom nowadays manage their own access control polices on a regular basis. Users are making frequent policy decisions, especially with the increasing popularity of social network sites, such as Facebook and Twitter. For example, users are required to allow/deny access to their private data on social sites each time they install a 3rd party application. To make matters worse, 3rd party access requests are mostly uncustomizable

by the user. We propose a framework that allows users to customize their policy decisions on social sites, and provides a set of recommendations that assist users in making well-informed decisions. Finally, as the browser has become the main medium for the users online presence, we investigate the access control models for 3rd party browser extensions. Even though, extensions enrich the browsing experience of users, they could potentially represent a threat to their privacy. We propose and implement a framework that 1) monitors 3rd party extension accesses, 2) provides fine-grained permission controls, and 3) Provides detailed permission information to users in effort to increase their privacy awareness. To evaluate the framework we conducted a within-subjects user study and found the framework to effectively increase user awareness of requested permissions.

ACKNOWLEDGEMENTS

While at UNC-Charlotte, I was greatly fortunate to work with a group of outstanding colleagues and friends. Therefore, it is my pleasure to dedicate this dissertation to those who have contributed directly or indirectly to this dissertation.

First and foremost, I would like to express my many thanks and sincere gratitude to my dear advisor Prof. Mohamed Shehab. His continuous guidance and support have made this work possible and have made for a great research experience throughout my time at UNC-Charlotte.

My many thanks go out to those who have collaborated with me on various research papers and topics throughout the past few years, including Dr. Anna Squicciarini, Doan Minh Phuong, Smitha Sundareswaran, Christopher Hudel, Dr. Moo Nam Ko, Hakim Touati, Adharsh Desikan, and Gorrell Cheek. They were a great source of inspiration and positive criticism that was valuable to me and my research.

I would also like to express my great appreciation to my Ph.D. committee members, Prof. Mohamed Shehab, Prof. Bei-Tseng Chu, Prof. Cem Saydam, and Prof. Taghi Mostafavi. Their assistance, feedback, and guidance have been invaluable in preparing this work.

Last and not least, my deepest gratitude goes to my father and mother for everything they have done for me throughout my journey to completing my Ph.D. Their infinite support, encouragement, and love have been invaluable to me. My love goes to my lovely wife Walaa Marouf for her continuous encouragement, patience and unconditional love that have enabled me to successfully complete my Ph.D. I also send my love to my lovely princess Fatma, my prince Mousa, and my younger prince Mohamed for being a continuous source

of motivation. Finally, my thanks and love go to my 12 sisters and 2 brothers in Palestine for their continuous good wishes and love.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Managing access control policies is a complex process, given the complex nature of policy languages and the large number of attributes and rules involved. Policies can involve thousands of rules, leading to thousands of relations among policy attributes. Interpreting and understanding such a large number of relations is difficult, and the possibilities of introducing policy misconfigurations is high. Another issue with large policies, is the difficulty in optimizing them for optimal performance, that is, if policy rules are configured properly, potential performance bottlenecks can be removed. Adding to this complexity, is the fact that not all administrators are well versed and proficient in all access control policy languages. For example, a Linux-based server could incorporate both an operating system (OS) level policy using SELinux [60], and a web service level policy using XACML [49]. Both policies can easily involve tens of thousands of rules and attributes, which makes it difficult for average administrators to manage without appropriate policy management tools.

Access control policy management is no longer a task strictly assigned to administrators. Nowadays, with the increase in privacy awareness [1, 10], and the wide adoption of third party applications on social sites (e.g. 3rd party Facebook apps) and internet browsers (e.g. 3rd party Chrome extensions), regular individuals have become themselves admins on their own privacy polices. Managing access control policies is an essential task in the daily lives

of individuals, who have to protect their private data (e.g., email address, location info, birthday, etc.) and content (e.g., photos, videos, browser bookmarks, etc.), from unwanted accesses by other online users and by third party applications. Security aware users will manage their policies to the degree they can, that is, current privacy preserving mechanisms do not provide users the capability to completely control their online privacy. For example, at installation time, third party Facebook applications can request a set of permissions to access a user's Facebook profile data. At this point, users are given two options: 1) Grant the application all requested permissions, or 2) Opt-out of installing the application (all-or-nothing). It is clear that there is space to improve, and that security aware users should be given fine-grained controls over their access control policies. In regards to security unaware users, new tools should be introduced to guide and assist them in understanding privacy issues and in making well-informed policy decisions.

The challenges facing both administrators and individuals call for better privacy preserving mechanisms and better tools for managing access control policies. Such tools should provide the following:

- Simplified policy management: When access control policies involve thousands of attributes and rules, policy management tools need to simplify the way administrators and users interact with a policy. This can be achieved by providing new presentation layers that allow for easier interpretation of the policy, and the ability to focus on relevant policy information without the need for a deep understanding of the policy language. Simplifying policies, also allows administrators to easily and properly analyze existing policies.

- Assisted policy management: Assisting users and administrators in understanding the consequences of their policy decisions is important, especially given the large number of attributes and rules within policies, and the complex nature of existing policy languages. Guiding users and administrators is achievable by utilizing existing decisions made by other potentially well-informed parties. That is, tools should make use of existing knowledge in providing guidance to those making new policy decisions. Guiding users could also be achieved by providing simplified descriptions of the various access permissions.
- Fine-grained policy controls: We believe, that users should be able to control their privacy policies to the extent they wish. That is, controlling individual privacy attributes should be possible without the need to limit the options to "grant all accesses" or "nothing". To achieve this, new tools are required which extend upon current authorization flows.
- Policy Optimization: Policies with large numbers of rules, can easily introduce performance bottlenecks, especially when faced with a huge number of policy evaluation requests. Configuring such large policies for optimal performance outcomes, requires knowledge of all policy rules, and the ability to understand the outcome of each possible configuration. There is a need for tools that can take on this task, and provide for configurations that lead to ideal performance.

XACML Policies: Many web services have adopted XACML (eXtensible Access Control Markup Language) as the standard for specifying their access control policies. XACML policies can introduce performance bottlenecks when a large number of policy rules are

involved. In our investigation, we found that existing XACML policy evaluation engines, such as Sun's Policy Decision Point Engine (PDP) [64], suffer from such performance bottlenecks. For a 100,000 random policy evaluation requests, we found that a policy with 4000 rules, requires Sun's PDP up to 1,152,460ms to evaluate. Even smaller policies, for example of 75 rules, took up to 32,223 ms to evaluate. Such evaluation times are not sufficient for running web services under high request loads. The bottleneck in existing engines results from the sequential nature of evaluating policy rules. We believe that with the proper policy structure optimization, i.e. the structure of its rules, we can achieve improved performance outcomes.

Third Party Application Authorization: In our research we mainly focus on two types of third party applications: 1) Social networking applications, and 2) Internet browser extensions. Third party social networking applications run on social sites such as Facebook and Twitter, and are widely adopted by users who wish to add new services on top of a site's core services. To do so, applications need to be authorized by users for a set of requested accesses/permissions. For example, an application can request permission to access a user's birthday information on Facebook.

Third party browser extensions are also widely adopted and enrich the user browsing experience. Extensions also request permissions that allow for performing privileged tasks such as accessing a user's browsing history, or executing custom scripts within certain webpages visited.

The primary disadvantage of existing authorization mechanisms, is the lack of fine-grained controls, that is, users have to authorize all requested accesses, or choose not to

install an application in the first place. Authorizing third party applications can be problematic, if they are malicious, and seek to use a user's private data inappropriately. For this reason, it is important that existing authorization methods be extended to provide fine-grained controls. We investigated third party applications on Facebook, and found that, among popularly requested accesses, individuals - when given the choice - will, in the majority of cases, deny the request.

SELinux Policies: The U.S. National Security Agency, introduced Security Enhanced Linux (SELinux) for the purpose of incorporating a system-wide Mandatory Access Control (MAC) architecture into the Linux operating system. SELinux provides fine-grained access control through its policy language, but in exchange, the language is very complex, leading to complex policies that are hard to interpret and difficult to manage. SELinux policies are mainly based on *types*, which represent labels on processes and files. That is, policy rules are written in regards to these types. SELinux also comes with a set of default policies that potentially satisfy the needs of most Linux systems. When investigating SELinux's default policies, we found that the *targeted* default policy contains over 1,780 types, and over 1,500,000 rules. Another *strict* version of the default policy, contained over 2,300 types and 1,700,000 rules. With such a large number of types and rules, it's clear why many administrators face difficulties in managing SELinux policies [70, 44, 34].

1.1 Research Problem

The difficulty in managing access control policies can be due to a number of factors:

- Complex Policy Language: Policy managers, whether normal individuals or administrators, are not well versed in existing complex policy languages. Because of this,

it is difficult to easily interpret policies which in many cases can lead to policy misconfigurations.

- Complex Policy Structure: Access control policies can involve large numbers of attributes and rules which lead to difficulties in understanding the relations among policy attributes and rules. This also leads to difficulties in optimizing policies for optimal performance.
- Limited Policy Management Tools: The lack of proper policy management tools, that are able to guide users and administrators in making better policy decisions. This becomes essential when decisions need to be made on new policy attributes and rules, that is, introducing unknown elements into an existing policy. Such new elements, can compromise the overall security of a system.

In the light of the existing challenges facing both administrators and individuals, we investigate a number of clustering and recommendation based techniques for managing access control policies. Clustering access control policies could potentially optimize its structure, leading to better performance outcomes. It can also simplify the policy presentation, hence resulting in a policy that is easier to interpret and understand. On the other hand, recommendation-based techniques could help in guiding users and administrators in making well-informed policy decisions. These techniques can be based on the collective collaboration of a community, and on existing knowledge regarding a policy.

We define our research problem as follows:

Problem Statement: *The average administrator and individual face many challenges when managing their access control policies. These challenges can lead to policy misconfigura-*

tions, performance bottlenecks, and ill-informed policy decisions.

In this research proposal, we plan on overcoming existing challenges in access control policy management. Our hypothesis is as follows:

Hypothesis Statement: *Applying effective clustering and recommendation techniques onto access control policies will allow for a simpler and more effective policy management process that also guides users and administrators when making important policy decisions.*

1.2 Overview of proposed solution

Managing access control policies is a complex and challenging process, which requires executing a number of various tasks. We focus on three primary tasks:

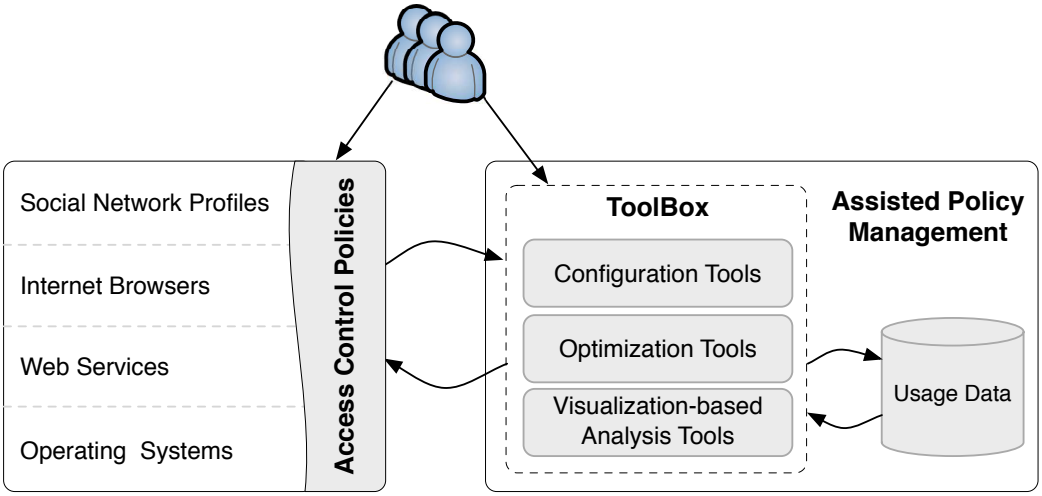


Figure 1: Assisted Policy Management Model

1. Policy Configuration: Configuring an access control policy involves making a set of decisions on what accesses should be allowed or denied. These decisions are what define the policy, hence defining the overall security of the system under this policy. If ill-informed decisions are made, they can easily compromise the security of a system. Ill-informed decisions can be a result of: 1) complex policies that involve large

numbers of rules and relations that are difficult to interpret, 2) The lack of a deep understanding of the policy language, and the understanding of possible consequences related to certain policy decisions. 3) The lack of fine-grained controls on the policy decisions that need to be made.

2. Policy Optimization: Optimizing access control policies is an essential task that can involve: 1) Removing out-dated policy components, 2) Prioritizing policy components according to demand, and 3) Assessing a policy's overall structure, and finding optimal structures, that lead to enhanced access control request evaluations. By performing these optimization tasks, the storage footprint decreases, and performance bottlenecks can be avoided. Such tasks become very difficult when dealing with large policies, or policies that need to rapidly adapt to different scenarios and environments.
3. Policy Analysis: Analyzing access control policies is a fundamental task, and is a basis for the previous two tasks mentioned above. That is, to properly configure and optimize a policy, proper analyses need to take place. By analyzing a policy, administrators are able to discover: potential misconfigurations, redundant policy rules, out-dated components, and more. They are also able to get a deeper understanding of existing relations within a policy, and how various policy components are able to interact with each other. With large policies, the analysis process gets tricky, due to the vast number of relations, and the difficulty in presenting large amounts of analysis data in an easily and interpretable fashion.

In the light of these challenging tasks, and the fact that more usable and suitable tools are needed to accommodate the needs of average administrators and individuals, we propose a set of assisted policy management tools (see Figure 1), based on recommendation and clustering techniques.

1.2.1 Recommendation-Based Policy Management Tools

The premise of these tools is to guide administrators and users in making better policy decisions. Guidance is provided in the form of recommendations on new policy decisions. Recommendations represent quantified indications of how common, or how risky, certain decisions are. Recommendations are based on data collected from various sources, such as the collaborative decisions of communities, application behavior, existing stable policies, and the nature of incoming access control policy requests. Based on provided recommendations, administrators and users can make more well-informed decisions when configuring their policies.

The proposed tools are also able to optimize policies by generating recommendation values that are the basis of ranking/prioritizing policy components according to their demand. That is, components that are most frequently used within a certain range of time, get priority over components less used. Components with higher priority are given more importance when evaluating incoming requests, which potentially removes performance bottlenecks.

Another advantage of recommendation-based tools is the ability to provide recommendations on a fine-grained level. Our proposed tools provide the necessary fine-grained policy controls, accompanied with fine-grained recommendations.

1.2.2 Clustering-Based Policy Management Tools

Clustering-based tools analyze the properties of existing policy components. From these properties, they are able to identify groups/clusters of tightly related components. Identified clusters can then be used to: 1) provide abstractions on the components of a particular cluster, that is, focus on what a cluster represents rather than what each single component does. 2) optimize a policy's structure based on its identified clusters, which potentially results in improved policy evaluation times, by redirecting incoming requests to their appropriate clusters, rather than the whole policy.

Our proposed clustering-based tools also provide effective policy analysis capabilities. This is achieved by utilizing identified clusters to discover new relations among policy components, and to visualize the results of policy analyses. Visualization occurs at the cluster level, which dramatically reduces the complexity of analysis results.

The remainder of this dissertation is organized as follows: Chapter 2 discusses some of the preliminaries regarding our research. Chapter 3 proposes an adaptive and clustering-based approach for evaluating XACML policies, and discusses the experimental results done. In Chapter 4 we propose a recommendation-based open authorization framework that provides user with recommendations on permissions requested by third party social networking applications. It also discusses our fine-grained control mechanism. Chapter 5 proposes a framework that provides fine-grained controls on third party browser extension permissions, in addition to increased user privacy awareness. The results of a user study on the effectiveness of the framework are also discussed. Chapter 6 proposes a visualized-based approach for analyzing SELinux policies, and discusses a risk-based model for newly

added policy rules. Finally, in Chapter 7 we conclude the dissertation and discuss potential future paths for extending upon this research.

CHAPTER 2: PRELIMINARIES

2.1 XACML Policies and Access Requests

In this section we provide the logic formalism adopted to denote XACML policies and access requests. XACML policies are composed of five basic components, namely, *PolicySet*, *Policy*, *Target*, *Rule*, and *Policy and Rule Combining algorithm* for conflict resolution. The root of the XACML policy is the PolicySet element, which is defined as follows:

Definition 1. *PolicySet is a tuple $PS = (id, t, P, PC)$, where: id is the PolicySet id, t is the PolicySet Target element, and takes values from the set $\{$Applicable, NotApplicable, Indeterminate$\}$, $P = \{p_1, \ldots, p_n\}$ is the set of policies, and PC is the policy combining algorithm.*

A *Policy* element is a set of rules and conditions that control access to protected resources which we refer to as objects. A policy contains a *target*, a set of *rules*, and a *rule combining* algorithm.

Definition 2. *A policy is a tuple $P = (id, t, R, RC)$, where: id is the policy id, t is the policy target element, and takes values from the set $\{$Applicable, NotApplicable, Indeterminate$\}$, $R = \{r_1, \ldots, r_n\}$ is the set of rules, and RC is the rule combining algorithm.*

The Target element t specifies a set of predicates on the request attributes, which must be met in a PolicySet, Policy or Rule to apply to a given request. The attributes in the target element are categorized into *Subject*, *Resource* and *Action*. The attribute values in a request

are compared with those included in the Target, if all the attributes match then the Target's PolicySet, Policy or Rule is said to be Applicable. If the request and the Target attributes do not match then the request is NotApplicable, and if the evaluation results in an error then the request is said to be Indeterminate. If a request satisfies the target of a policy, then the request is further checked against the rule set of the policy; otherwise, the policy is skipped without further examining its rules. The Target predicates can be quite complex, and can be constructed using functions and attributes. The rule combining algorithm RC respectively allows one to specify the approach to compute the decision result of a policy when the policy contains rules evaluating to conflicting effects. The policy combining algorithm PC follows the same logic but at the PolicySet level.

A *Rule* identifies a complete and atomic authorization constraint that can exist in isolation with respect to the policy in which it has been created. We define rules as follows.

Definition 3. *A Rule is a tuple* $r = (id, t, e, c)$*, where:* id *is the rule id,* t *is the rule target element, and takes values from the set* $\{$*Applicable, NotApplicable, Indeterminate*$\}$*,* e *is the rule effect, where* $e \in \{Permit, Deny\}$*, and* c *is a boolean condition against the request attributes.*

The rule target element is similar to the policy target instead it indicates the requests applicable to the rule. The condition c is a boolean function with respect to the request attributes. The rule's effect e, which can be Permit or Deny, is returned if the rule's condition c evaluates to true. The rule evaluation can also be Indeterminate in case of an error, or NotApplicable if the rule's target doesn't apply to the request's attributes. Access requests are typically matched against a policy set. A policy set is the root of an XACML policy, it holds policy elements and, possibly, other policy sets. We denote access requests accord-

```
<PolicySet PolicySetId="PSID"
  PolicyCombiningAlgId="permit-overrides">
  <Target/>
  <Policy PolicyId="PID"
    RuleCombiningAlgId="permit-overrides">
    <Target/>
    <Rule RuleId="RID1" Effect="Deny">
      <Target>
       <Subjects>
        <Subject>Bob</Subject>
        <Subject>John</Subject>
       </Subjects>
       <Resources>
        <Resource>file2</Resource>
       </Resources>
       <Actions>
        <Action>
         <ActionMatch MatchId="string-equal">
          <AttributeValue DataType ="string">
            read
          </AttributeValue>
          <ActionAttributeDesignator
           AttributeId ="AID1" DataType ="string"/>
         </ActionMatch>
        </Action>
       </Actions>
      </Target>
    </Rule>
    <Rule RuleId="RID2" Effect="Permit">
      <Target>
       <Subjects>
        <Subject>Bob</Subject>
       </Subjects>
       <Resources>
        <Resource>file1</Resource>
       </Resources>
       <Actions>
        <Action>
         <ActionMatch MatchId="string-equal">
          <AttributeValue DataType ="string">
            read
          </AttributeValue>
          <ActionAttributeDesignator
           AttributeId ="AID2" DataType ="string"/>
         </ActionMatch>
        </Action>
       </Actions>
      </Target>
    </Rule>
  </Policy>
</PolicySet>
```

Figure 2: XACML Policy Set example

ing to the following notation. Let S, O, A and X denote all subjects, objects, actions and context variables in an access control system respectively.

Definition 4. *(Access Request) An access request q is the tuple (s, o, a, x), where $s \in S$ is the subject making the request, $o \in O$ is the requested object, $a \in A$ is the requested action on object o, and $x \in X$ are the context attributes.*

Let us consider the PolicySet listed in Figure 2 which contains one policy with 2 rules.

The first rule specifies that “Both Bob and John are denied read access to file2” where each “Bob” and “John” is a *Subject*, “denied” is the rule *Effect*, “read” is the *Action*, and “file2” is the *Object or Resource*, whereas the second rule says “Bob has permission to read file1”, “Bob” being the *Subject*, “has permission” the *Effect*, “read” the *Action*, and “file1” the *Object*. Either rule could be accompanied with context parameters (Environment Attributes) as part of a rule’s condition such as time, system variables, history, or location. A target is a condition on subject $s \in S$, object $o \in O$ and the action $a \in A$. If the request satisfies the target conditions of a rule (policy) then we say that the rule (policy) is *applicable* to the request, otherwise it is *not applicable*. That is, if Bob makes a request to read file1, his request would be applicable to the second rule which would return a Permit.

2.2 Third Party Application Authorization and APIs

Most of the major online platforms such as Facebook, Google, and Twitter, provide an open API which allows third party applications to directly interact with their platform. APIs provide a mechanism to read, write, or modify user information on these platforms through other third party applications on behalf of users themselves. An API comes with a set of methods, each representing a certain user interaction executed through a third party application. For example, the FriendCameo [18] Facebook application is able to post content (e.g. messages, photos) to a user’s Facebook feed/wall using Facebook’s */profile_id/feed* API method, where *profile_id* is the targeted Facebook user ID. It is important to note that third party applications can potentially execute any API call on behalf of a user, relying on the type and scope of permissions granted to these apps. In the previous example, the FriendCameo application could only perform the */profile_id/feed* API call, given the user

has granted it the "publish_stream" permission. The full set of permissions available to third party apps are defined by the online platforms, and it is up to third party applications to request the proper subset of permissions required. We believe users should have the final decision on whether to grant requested permissions or not.

2.2.1 OAuth Standard

With an increasing trend towards offering online services that provide third party applications the ability to interact through open APIs and access user resources, OAuth was introduced as a secure and efficient mechanism for authorizing third party applications [52]. Traditional authentication models such as the client-server model require third party applications to authenticate with online services using the resource owner's private credentials, typically a username and password. This requires users to present their credentials to third party applications, hence granting them broad access to all their online resources with no restrictions. A user may revoke access from a third party application by changing her credentials, but doing so subsequently revokes access from *all* third party applications that continue to use her previous credentials. These issues are amplified given the high number of third party applications that potentially get access to a user's online resources. OAuth uses a mechanism where the roles of third party applications and resource owners are separated. It does not require users to share their private credentials with third party applications, instead it issues a new set of credentials for each application. These new set of credentials are per application, and reflect a unique set of permissions to a user's online resources. In OAuth, these new credentials are represented via an *Access Token*. An Access Token is a string which denotes a certain scope of permissions granted to an application,

it also denotes other attributes such as the duration the Access Token is considered valid. We are mainly interested in the scope attribute within an Access Token. Access Tokens are issued by an authorization server after the approval of the resource owner. In this research we extend upon this authorization stage of the OAuth 2.0 protocol.

When a third party application needs to access a user's protected resources, it presents its Access Token to the service provider hosting the resource (e.g. Facebook, Twitter) which in turn verifies the requested access against the scope of permissions denoted by the Token. For example, Alice (resource owner) on Facebook (service provider and resource server) can grant the FriendCameo application (client) access to her email address on her Facebook profile without ever sharing her username & password with FriendCameo. Instead, she authenticates the FriendCameo application with Facebook (authorization server) which in turn provides FriendCameo with a proper Access Token that denotes permission to access Alice's email address.

OAuth provides multiple authorization flows depending on the client (third party application) type (e.g. web server, native applications). We focus on the *Authorization Code* flow shown in figure 3 and detailed in the OAuth 2.0 specification [52]. The authorization code flow is used by third party applications that are able to interact with a user's web browser, and are able to receive incoming requests via redirection. The authorization flow process consists of three parties: 1)End-user (resource owner) at browser, 2)Client (third party application), and 3)Authorization server (e.g. Facebook). Our main focus is on steps "(A)" and "(B)" within the authorization code flow [52]. Step "(A)" is where third party applications initiate the flow by redirecting a user's browser to the authorization server and pass along the requested scope of permissions. In step "(B)", the authorization server authenti-

cates the end-user, and establishes her decision on whether to grant or deny the third party application's access request.

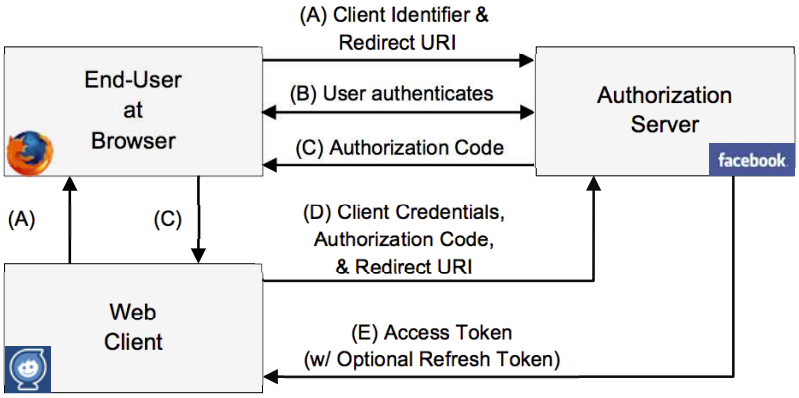


Figure 3: Authorization Code OAuth Flow

2.2.2 OAuth and User Privacy

One of the main reasons behind OAuth was to increase user privacy by separating the role of users from that of third party applications. OAuth uses the concept of Access Tokens, where a token denotes a set of credentials granted to third party applications by the resource owners [52]. This avoids the need for users to share their private credentials such as their username & password. It also allows users to revoke access to a specific third party application by revoking its Access Token.

OAuth 2.0 allows third party applications to request a set of permissions via the `scope` attribute, and for users to grant/deny such requests. If a user grants a third party application's request, then an Access Token (denoting the `scope`) is issued for that application, hence granting it the scope of permissions requested. The `scope` attribute represents the set of permissions requested by third party applications, and is our main focus in this work. In the authorization code OAuth flow seen in figure 3, the `scope` parameter is part of the request URI that is generated by third party applications (Step "(A)" in figure 3). The

`scope` is a list of space-delimited strings, each string mapped to a certain permission or access level. For example, the FriendCameo application requests permission to post to a user's Facebook feed/wall, to log in to Facebook chat, to access her email address, and to check her friend's online/offline presence. FriendCameo requests these permissions with a `scope` attribute value of "`publish_stream, xmpp_login, email, friends_online_presence`". The `scope` value becomes part of the OAuth request URI sent to the authorization server (Facebook's OAuth implementation uses commas rather than spaces to separate each requested permission). Step "(B)" of figure 3 is where users grant/deny the requested `scope` value.

2.3 Collaborative Filtering in Recommendation Systems

Recommendation systems are systems that try to assist users in evaluating and making decisions on items by providing them opinions and prediction values as a set of recommendations [55]. These set of recommendations are usually based on other people's opinions and the potential relevance of items to a target user. The first recommender system Tapestry [19], followed the approach of "Collaborative Filtering" in which users collaborate towards filtering documents via their individual reactions after reading certain documents. Since then, the "Collaborative Filtering" approach has been widely adopted and is accepted as a highly successful technique in recommender systems [38, 45, 39, 63].

In a context of access control and user privacy, items in a collaborative filtering model can be mapped to individual privacy attributes or permissions. Users have to make decisions on privacy attributes, i.e. grant them to third party applications or not. This is similar to other recommendation systems in which users make decisions on items, e.g. to rent or

not rent a certain movie. Users can benefit from recommendations on privacy attributes which are based on the collaborative decisions of all users. Similarly, users benefit from movie ratings in making their decision to rent a movie.

2.4 Third Party Browser Extensions

Third party browser extensions are widely used within major browsers such as Firefox, Chrome, and Safari [50, 66]. Users can enhance their browsing experience by adding new functionalities or modifying the core browser functionalities. To provide extended functionaly, extensions request a set of permissions which have to be authorized by their users. We focus on the permission model for Google Chrome extensions, where extensions request permissions at install time but also have the ability to request optional permissions after installation.

2.4.1 Chrome Extensions

Chrome extensions are built using a mix of required and optional components. Specifically, a required `manifest.json`, at least one `html` file (`background.html` or `popup.html`), and other additional resources such as JavaScript files, images, and other HTML files.

Manifest: The `manifest.json` file is a required component for each extension, and provides information on an extension's properties, requested permissions, and other attributes. In this paper, we focus on the `permissions`, `plugins`, and `content_scripts` properties within the manifest. These are properties related to the privacy of the user when using third party extensions.

Background Page: An optional HTML page that many extensions use for managing

background activities. This is used by extensions that need to stay active at all times or be able to perform continuous tasks. Our proposed framework targets background pages when adapting third party extensions to our model.

Content Scripts: These are scripts that run within the context of a webpage that extensions want to interact with. That is, the content script can read and modify a webpage and pass messages back to its parent extension. An example extension that uses content scripts is the Google Dictionary extension which shows a popup with the description of a selected word within a webpage. Extensions declare the hosts targeted by their content scripts within the `manifest.json`. Note that extensions are also able to programmatically inject custom scripts into webpages using the `chrome.tabs.executeScript` API.

NPAPI Plugins: For purposes of supporting legacy code, Chrome allows for embedding NPAPI plugins within newly developed extensions. NPAPI plugins allow for executing native code, i.e. calling native binary code from within an extension's JavaScript. This gives an extension user level access to the user's machine. Such extensions, if compromised, could highly risk the user's privacy. iMacros [29] is a popular extension that uses NPAPI plugins to store a user's recorded macros on to the the file system.

2.5 SELinux Policies

SELinux policies are considered quite difficult to manage due to the granular level of controls they provide [70, 44, 34]. Even though this is true, an SELinux policy at its core is no different than other access control policies in which a set of rules are introduced to enforce and achieve an overall security goal. A typical access control policy rule is built

around a *subject* which is granted certain *actions* on a certain *object*. For example, John (subject) is allowed to play (action) all mp3 files (object) on a system. The same model is applied in SELinux policy rules but with more elaborate and fine-grained levels of control. SELinux labels each resource, such as files and processes within an SELinux-enabled system with a *security context*. A security context is a label that usually incorporates three fields: 1) SELinux User, 2) Role, and 3) Type. Our focus is on the "Type", which represents the core of access control rules that determine what *subject-types* have what accesses on which *object-types*. Object-types are defined to group file objects, whereas subject-types are defined for processes. Objects that fall under the same object-type, are similar in which subjects access them. Subjects or processes that are under the same subject-type, are similar in which objects or files they access. An example of an object-type is the `user_home_t` type, which is used to group files owned by a user and reside in his/her home directory. Grouping here, is achieved by setting the type within each file's security context to `user_home_t`. An example subject-type is the `httpd_t` type, which belongs to the Apache HTTP server process.

We also focus on Access Vector (AV) *allow* rules within an SELinux policy. AV *allow* rules are responsible for allowing accesses between types. A typical AV allow rule specifies how a subject-type is allowed to interact with an object-type. The building blocks of any AV allow rule are the following:

- Subject-type: The subject of the access control rule which is granted certain accesses.
- Object-type: The object or resource to be accessible by the subject of this rule.
- Object-class: Each object within SELinux falls under a certain class (*object-class*).

Each object-class has a corresponding set of applicable actions (permissions). For example, *file* and *dir* are object-classes that respectively correspond to files and directories within a system. Having object-classes allows for easier management of permissions on objects. For example, a *read* permission has a different interpretation when applied to files vs. directories, hence having an associated permission set for each object-class allows for easier interpretation of the intended permission, i.e. *read* on object-class *file* is not the same as *read* on object-class *dir*.

- Permissions: For each object-class there is an associated set of permissions, i.e. a set of actions that the subject can take on the object. For example, the *file* class has the permissions *read, write, create, rename* an so forth.

Following is an example AV allow rule written in the SELinux AV rule syntax:

```
allow httpd_t httpd_log_files_t : file {read create}
```

this reads as: allow the subject-type `httpd_t` to *read* and *create* files of object-type `httpd_log_files_t`. Or in a more readable format this reads: Allow the Apache HTTP process to read and create its log files.

2.5.1 Custom SELinux Policy Modules

Administrators (admins) are frequently required to write custom policy modules for new services and applications that are installed onto a Linux system. Such modules contain a set of new policy rules that are incorporated into the existing SELinux policy to allow the new services to function properly. Given the nature of SELinux policies in respect to the large number of types and rules they contain, admins rely on policy tools [32, 27] for generating new policy rules that can be used to adapt new services.

audit2allow [32] and SEEdit's *audit2spdl* [74] are two of the most common tools for generating new policy rules based on audit logs. It is then the admin's responsibility to either add these new rules to the existing policy directly, or tweak them before hand.

CHAPTER 3: ADAPTIVE REORDERING & CLUSTER-BASED FRAMEWORK FOR EFFICIENT XACML POLICY EVALUATION

The adoption of XACML as the standard for specifying access control policies for various applications, especially web services is vastly increasing. This calls for high performance XACML policy evaluation engines. A policy evaluation engine can easily become a bottleneck when enforcing XACML policies with a large number of rules. We propose an adaptive approach for XACML policy optimization. We apply a clustering technique to policy sets based on the K-means algorithm. In addition to clustering we find that, since a policy set has a variable number of policies and a policy has a variable number of rules, their ordering is important for efficient execution. By clustering policy sets and reordering policies and rules in a policy set and policies respectively, we formulated and solved the optimal policy execution problem. The proposed clustering technique categorizes policies and rules within a policy set and policy respectively in respect to target subjects. When a request is received, it is redirected to applicable policies and rules that correspond to its subjects; hence, avoiding unnecessary evaluations from occurring. We also propose a usage based framework that computes access request statistics to dynamically optimize the ordering access control to policies within a policy set and rules within a policy. Reordering is applied to categorized policies and rules from our proposed clustering technique.

3.1 Related Work

Much research has been done on optimizing XACML policy evaluation. In [40], Liu et al. present one of the most interesting proposals on optimization of XACML policies so far. Liu et al, focus on improving performance by numericalizing and normalizing XACML Policies. The numericalization is used to convert the string policies into numbers as numerical comparison is more efficient. Further, normalized policies are converted into a flat policy structure. In doing this, the authors replace the different rule-combining algorithms with only one, viz. First-Applicable. They then proceed to convert the numericalized, normalized policies into tree data structures for efficient policy evaluation.

Miseldine [46] proposes to achieve policy optimization by minimizing the average cost of finding a match at the rule level the target level and the policy level. The work assumes no changes to the XACML specification, in that the Sun's XACML implementation is not altered. Miseldine approaches this problem by using *policy configurations*. A policy configuration is the relationship of policy and rule targets to members of the set of rules R, the set of subjects S and the set of actions A. Combinations of sets are sought such that policy targets are formed from S.R, R.A or S.A .

Kolovski [37] formalizes XACML policies using description logics (DL), and exploits existing DL verifiers to conduct policy verification. Their policy verification framework can detect redundant XACML rules. The idea of removing redundant policies is interesting and may be useful to improve evaluation times. However, it is yet to be validated whether the improvement will be worth the time needed to remove redundant policies, and how significant the overall improvement would be.

One related area where similar optimization techniques are often explored is Firewall Filtering [23, 22]. In this respect, our work on optimization of XACML policies shares some similarities to the optimization of firewall filtering approaches. Firewall optimization is different from that of XACML policy optimization in that a major portion of the traffic packets match a small subset of the firewall rules, and the same distribution of traffic is maintained over a significant period of time. This skewness is not experienced in the incoming requests for an XACML policy. Also note that firewall rules have an order of precedence defined, while rules in an XACML policy do not. These two properties of firewall rules allow the authors to prove in [23] and [22] that the optimal firewall rule ordering problem is NP-Complete. Despite these differences between firewall filtering optimization and optimization of XACML policies, we can still draw from the body of work on firewalls, specifically from [23].

3.2 Policy and Rule Reordering Framework

When a web server needs to enforce an XACML policy with a large number of rules, its XACML policy evaluation engine may easily become the performance bottleneck for the server. To enable an XACML policy evaluation engine to process simultaneous requests of large quantities in real time, especially in face of a burst volume of requests, an efficient XACML policy evaluation engine is necessary. In such environments the requests' distribution is dynamic in terms of volume, types and type of requesters. Motivated by such observation, we develop an adaptive framework that dynamically determines the best ordering according to the incoming requests and the recently received history of requests and executions.

3.2.1 Execution Vector and Policy Permutation

In what follows for the sake of presentation we focus on policy permutation where a similar approach is adopted for PolicySet permutation. We define a policy permutation as follows:

Definition 5. (Policy Permutation) *Given a policy P with a rule set $P.R = \{r_1, \ldots, r_n\}$, a policy permutation π is a policy P_π generated by the following procedure:*

(0) $P_\pi.R = \{\}$, $P_\pi.id = P.id$, $P_\pi.t = P.t$, *and* $P_\pi.RC = P.RC$.

(1) P' *is a copy of P.*

(2) Select a random rule r_i *from* P' *and append* r_i *to the end of* P_π.

(3) Repeat step ***2*** *until* P' *is empty.*

Policy permutation may alter the correctness of a policy, and result in different evaluations for a same set of requests. We are interested in policy permutations that do not alter the policy evaluation results for any request.

Definition 6. (Safe Policy Permutation) *A safe policy permutation π of a policy P is safe iff all requests permitted (denied) by the permuted policy P_π are also permitted (denied) by P.*

We assume all requests are well formed such that the policy evaluation returns PERMIT or DENY by the PDP. Using such an assumption, we provide the below theorem:

Theorem 0.1. Safe Permit (Deny) Overrides Permutation. *A policy P having a rule combining algorithm $P.RC$ set to Permit-Overrides or Deny-Overrides is safe with respect to all possible policy permutations.*

Proof. The semantics of the permit overrides is that if any rule evaluates to permit then the final authorization decision is permit. Assuming each rule returns either permit or deny then the policy evaluation of a policy P, with a permit overrides rule combing algorithm is the disjunction of all the rule results represented by: $E(P) = E(r_1) \vee \cdots \vee E(r_n)$. The disjunction operator is commutative where $a \vee b = b \vee a$, and associative where $(a \vee b) \vee c = a \vee (b \vee c)$, thus the evaluation of the policy P and any permutation P_π are equal $E(P) = E(P_\pi)$. The deny override follows similar semantics and follows a similar proof. □

Using Theorem 0.1, policies with permit override or deny override rule combining algorithms can be permuted without affecting the policy semantics. This does not hold for other rule combining algorithms such as First-Applicable. We focus our discussion on permit and deny override combining algorithms for reordering optimization. As discussed in the following sections, policy based categorization is independent of the rule combining algorithm used.

Given a policy permutation π and a given request q, a subset of rules is of relevance. We represent an ordering of such rules as the *execution vector*.

Definition 7. *(Execution vector)* $\Gamma = [r_1, \ldots, r_n]$ *is the execution vector representing the set of applicable rules, where rule* r_i *is executed before rule* r_{i+1}. $\pi(i)$ *refers to the position for rule* r_i *in execution vector.*

According to Theorem 1, any policy execution vector for a policy P having permit overrides rule combining algorithm will evaluate to the same effect as P, the challenge is to evaluate the execution vector that will provide the lowest latency. Hence, we need to define

the rule weights in order to present our optimal rule ordering approach.

3.2.2 Computation of Rule Weights

Our approach relies on statistics and metrics collected as PDP receives requests. Statistics are collected at two separate levels: *policy* and *rule* level. At the policy level, we are interested in understanding how often a policy applies, and by which class of users. At the rule level, it is important to identify the class of efficient execution vectors. In order to collect meaningful metrics, we assign to each rule (policy) weights that reflect the dominance of this rule in the requests. The weights are based on the PDP returned values, and constructed based on the 1) frequency and the 2) complexity of the rule (policy).

During a given time interval the number of times a policy P_i or a rule r_j gets evaluated is referred to as the hit frequency. We refer to the hit frequency by f and use the dot notation to refer to policy $(P_i.f)$ and rule $(r_j.f)$ hit frequency. Statistics with respect to the hit frequency are accumulated as follows:

- *Policy (Rule) Permit Ratio:* Records the ratio between the number of times a policy (rule) returns a permit with respect to the number of times a policy (rule) gets evaluated, where $P_i.p$ and $r_j.p$ represent the policy and rule permit ratios respectively.
- *Policy (Rule) Deny Ratio:* Records the ratio between the number of times a policy (rule) returns a deny with respect to the number of times a policy (rule) gets evaluated. Where $P_i.d$ and $r_j.d$ represent the policy and rule deny ratios respectively.
- *Policy (Rule) Hit Ratio:* Records the ratio between the number of times a policy (rule) is applicable with respect to the number of times a policy (rule) gets evaluated. Where $P_i.a$ and $r_j.a$ represent the policy and rule hit ratios respectively.

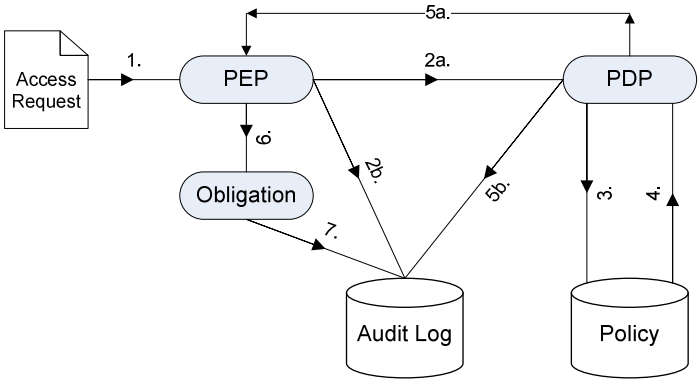


Figure 4: Log Based XACML Policy Evaluation Framework

Note that all the above statistics are easily derived from the XACML execution log (see Figure 4). In addition to the rule evaluation statistics we also consider the rule computational complexity. Rules vary from simple conditions to more complicated statements that require the parsing of an XML document or querying a database. The rule complexity metric is related to the number of operations required to execute the rule, we compute it as the number of boolean atomic conditions appearing in a rule, both at target and at the condition element. Let $n(t)$ denote the number of conditions in the Target element (denoted as t according to Def. 3), and let $n(c)$ be the number of conditions in the Condition element c. XACML supports over 100 standard functions that could be used in the boolean conditions, for example the $Belong\_to$. We assign a cost m_i to each standard function std_i appearing in the rule. m_i is computed by estimating the average execution time of the function. The simple atomic boolean conditions are assigned a constant cost k. For a rule r_j the complexity metric is given by:

$$E_j = k * (n(r_j.t) + n(r_j.c)) + \sum_{std_i \in r_j} m_i$$

where std_i represents a uniquely identified standard function appearing in r_j. Using both

the accumulated rule statistics and the complexity metric for a rule r_j we compute the rule cost as follows:

$$c_j = \beta * E_j + \alpha * F_j$$

Here, β and α are weights that allow system administrators to tune the computation cost, based on the local constraints, such as the available processing power and network bandwidth.

The rule cost is designed to represent the cost of computing a rule, the complexity metric E_j easily represents the rule cost, however the other component is based on the rule's accumulated statistics F_j. The value of F_j is based on the rule combining algorithm, for example if a rule combining algorithm is Permit-Overrides then the metric F_j is based on the decreasing function with respect to the rule permit ratio ($r_j.p$) or an increasing function with respect to the rule deny ratio ($r_j.d$). Intuitively, this implies that the rules need to be reordered such that for a policy with the permit overrides rule combining algorithm, the rule r_j with the lowest c_j is to be evaluated first.

3.2.3 Optimal Rule Reordering

Using the rule cost metrics we present our optimal rule reordering problem. Given a policy (P_i), the optimal request execution problem (REP) is to find an execution sequence that requires the minimum number of rule evaluations. We assume that rules within policies are evaluated sequentially. The policy P_i, composed of n rules $\{r_1, \ldots, r_n\}$, where $\pi(j)$ refers to the position (depth) for rule r_j in the policy execution vector. The cost associated with rule r_j as computed in Section 3.2.2 is referred to as c_j. The expected cost (i.e.,

average search length) for a given permutation π is given by:

$$\Phi_i = \sum_{j=1}^{n} c_j \pi(j)$$

The main challenge is to compute the optimal policy permutation π that will generate the minimum expected policy execution cost. Additionally, among the possibly optimal π, we need to ensure the policy permutation to be safe, as defined in Definition 6. By computing Φ_i we are able to generate a cost metric for each policy P_i.

A policy set PS is composed of a set of policies $\{P_1, \ldots, P_m\}$. We assume the policies are executed sequentially. Using the minimum policy expected cost Φ_i, and the collected policy evaluation statistics, we compute the policy set execution sequence. The position of policy P_i in the policy set execution sequence is referred to by $\xi(i)$. The expected cost (average search length) for a given policy set (PS_k) permutation ξ is given by:

$$\Psi_k = \sum_{i=1}^{m} \Phi_i \xi(i)$$

The costs Φ_i and Ψ_k are minimized when policies and rules are ordered in ascending order with respect to their costs [57]. Figure 5, shows the algorithm used at both the policyset and policy levels.

For example, consider a school database. During certain time periods, the access requests would be more uniform and from the same class of users (e.g. at the beginning of a semester most requests would be from students needing to register for courses, whereas faculty requests will be much less), while during other time periods, more heterogeneous set of requests may be submitted. In section 3.4 of this proposal, we show how our framework adapts to the different types of requests received and how we can benefit from policy/rule

```
Algorithm: optimize_policyset
Input: Policy Set PS = {P_1, ..., P_m},
Output: Optimal Policy Set Permutation PS*

 1:   if PS.PC = Permit-Overrides or Deny-Overrides
 2:      PS* ← {}
 3:    for each P_i ∈ PS
 4:       P_i* ← optimize_policy(P_i)
 5:       if PS.alg = Permit-Overrides
 6:          P_i*.c = α * P_i*.Φ + β * P_i.p^-1
 7:       elseif PS.alg = Deny-Overrides
 8:          P_i*.c = α * P_i*.Φ + β * P_i.d^-1
 9:       PS*.insert(P_i*) //Priority Queue on P_i*.c
 10:    return PS*
 11:  return PS


Algorithm: optimize_policy
Input: Policy P = {r_1, ..., r_n},
Output: Optimal Policy Permutation P*
 1:   if P.RC = Permit-Overrides or Deny-Overrides
 2:      P* ← {}
 3:    for each r_j ∈ P
 4:       E_j = k * (n(r_j.t + r_j.c)) + Σ_{std_i ∈ r_j} m_i
 5:       if P.RC = Permit-Overrides
 6:          F_j = r_j.p^-1
 7:       elseif P.RC = Deny-Overrides
 8:          F_j = r_j.d^-1
 9:       c_j = β * E_j + α * F_j
 10:      P*.insert(r_j) //Priority Queue on c_j
 11:    P*.Φ = Σ_{j=1}^{n} c_j π(j)
 12:    return P*
 13:  return P
```

Figure 5: Optimal PolicySet and Policy Reordering

reordering.

Weights can be updated according to two different strategies: 1) periodically, 2) based on the last ρ received requests. In the first case, we update the weight values using the latest statistics. New execution vectors are constructed using fresh rule weights in order to boost up the hit performance close to its optimum level. The update period should be based on the predictable incoming request (e.g., certain months of the year) flow changes. In the latter case, the optimal execution vectors are constructed based on the computed rule weights. The incoming access requests are then processed according to the ordering determined. Intuitively, the maximal reduction is obtained when the incoming requests perfectly match the requests' distribution. Notice that more than one execution vector could be optimal and safe. However, since not all rules have the same complexity, different

execution vectors may sensibly influence the overall evaluation time, even if a safe and efficient policy permutation is found.

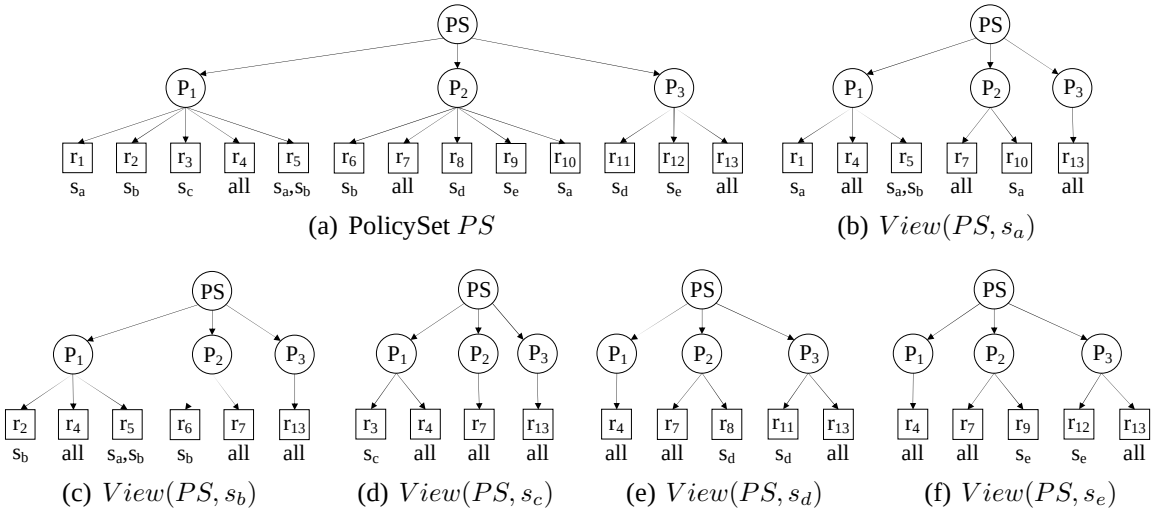


Figure 6: Policy Set and Views

3.3 Categorization Based Optimization

The optimization problem minimizes the average request evaluation time. This approach is ideal if the policy requests follow a uniform statistic. However, this approach is unlikely to be satisfactory in scenarios where the requests' distribution is dynamic in terms of volume and type of requesters. If we solely rely on reordering, assuming a role based access control (RBAC) system of two roles, say *student* and *faculty*, where there are on average 100 student requests for every faculty request, the computed statistics will be guided by the student requests. As such, the optimization problem presented above will favor the student role. Reordering rules and policies in these circumstances is not sufficient, as the computational cost will not be given by the evaluation of the rules themselves, rather it will heavily depend on the time spent on finding the applicable policies to the given request.

Hence, in order to further improve the efficiency of the rule reordering, we resort to clus-

tering the policies. Building on execution vectors, an intuitive mechanism is to categorize the policies based on the subjects. Starting from a set of $L[S]$ clusters, where $L[S]$ is the number of subjects in S, the goal is first to reduce the number of categories in order to allow the reordering to have a considerable effect on the execution time. Second, to reduce the memory footprint needed for caching the categories. When the categorization is done on a per-subject basis, to record an improvement in the execution time the policies must be adequately large. This happens because, when there is a category for each subject, there is essentially a unique execution vector for that subject. When large policies are evaluated, the categorization helps provide a good match for the execution vector and hence fewer rules are evaluated, thereby improving the evaluation time. In case of small policies, to make categorization effective, we need to decrease the number of categories to be searched in order to find the execution vector. In order to resolve this issue, we resort to further clustering the requests. Figure 6, shows a PolicySet and the different applicable views based on the involved subject, where each view could serve as a subject based category.

To achieve these results, we propose adopting an algorithm based on the K-Means clustering method [71]. Generally speaking, the K-Means algorithm is used to cluster *m* objects based on attributes into *k* partitions, $k < m$. Each cluster consists of a "center" around which individual elements of the data set being clustered are grouped together. This grouping is done based on some measure of similarity to the other elements in that cluster. In our domain, the number of clusters N_c and the centers of these clusters, i.e. N_c subjects are chosen at random from the set of subjects *S*. The set of centers (or clusters) is referred to as C_s. Each subject $S_i \in S$ is considered, and its similarity $D_{i,k}$ is calculated with respect to each subject $S_k \in C_s$ in the different clusters. S_i will be added to that cluster where the

similarity $D_{i,k}$ is maximum. The strength of this simple algorithm lies in the way the similarity metric $D_{i,k}$ is calculated. The similarity metric aims to cluster together the subjects that share a large number of policies which are applicable to all of them. Let $\mathbb{P}_i$ represents the set of policies applicable to a given subject S_i and let $L[\mathbb{P}_i]$ be the number of policies applicable to that subject. The number of policies shared between two subjects, S_i and S_k is given by $L[\mathbb{P}_i \cap \mathbb{P}_k]$. The fraction of the number of policies shared between the two subjects that are a part of $L[\mathbb{P}_i]$ is given by $\Theta_{i,k}$, where:

$$\Theta_{i,k} = \frac{L[\mathbb{P}_i \cap \mathbb{P}_k]}{L[\mathbb{P}_i]}$$

The similarity metric $D_{i,k}$ between subject S_i and S_k is calculated as follows $D_{i,k} = \Theta_{i,k} + \Theta_{k,i}$. The subject S_i is grouped with the cluster centering on S_k where $D_{i,k}$ is maximum. This ensures that only those subjects which have a large number of policies in common are grouped together. In general, the clustering is more effective when the number of shared policies is large, i.e. when $L[\mathbb{P}_i \cap \mathbb{P}_k]$ is large. The number of clusters N_c should be chosen carefully. The larger the value N_c, the lesser visible will the effect of reordering be. This is more evident when we consider the fact that as N_c approaches $L[S]$, we essentially experience the initial effect of having $L[S]$ unique categories for each of the subjects. On the other hand, should N_c be too small, the improvement obtained by categorization is completely lost, because as N_c approaches '1', all the subjects belong to the same cluster. In other words, there are no clusters at all.

This algorithm allows us to tune our optimization approach such that we can either maximize the improvement due to clustering or due to reordering, or both, based on the specific context.

3.4 Experimental Results

Our experiments were conducted on both synthetic policies and real world-based policies. The synthetic polices were divided into two sets of test suites. The first test suite deals with XACML policy sets where subjects have a small number of applicable rules. The second suite investigates policy sets where subjects have a large number of applicable rules, and will show the significant effect of applying our reordering technique to large policy sets. The real world-based policy sets are policies built using existing data sets, and properly modified to fit our framework without changing the semantics -or the structure- of the policies. Precisely, we tested the policies by Fisler et al. [17], which they used for their Margrave tool. Our experiments ensure that all policies are loaded into memory before executing any request evaluations. This ensures that evaluation times are not skewed by any policy loading time. All tests were conducted using 100,000 randomly generated XACML requests. All requests have a single value for the subject, resource, and action.

Our experimental process includes two main stages; First, the setup stage and, Second, the request evaluations. The setup stage includes three sub-stages:

S1. Categorization of the experimental policy sets. Categorization is performed as explained in Section 3.3. The number of categories used for each policy set ranges from N to $N/10$, where N is the number of unique subjects within a policy set,

S2. Training stage that collects the results of request evaluations (permit, deny, not-applicable, indeterminate) subsequently used for the reordering stage,

S3. Reordering policies within the policy set and all rules within each policy according to the statistics we gathered during the training stage.

The setup stage needs to be executed only once, however the sub-stages (S2) and (S3) could be executed repeatedly to retrain and reorder the policies and rules to achieve better performance. For our tests, we chose not to repeat the sub-stages, and thus measure the performance in the worst case scenario. The results of categorization and reordering are cached in memory. During the second stage the access requests are actually evaluated, using the ordering and categories set up in the previous stage. The processing time is the time needed to evaluate a request against a policy subjected to our setup stage plus the time to make a decision on that request. The preprocessing time is the time needed to complete the setup stage.

3.4.1 Real World-Based policies

The experiments on real world-based policies used the policy sets by Fisler et al. [17], specifically CodeA, CodeB, CodeC, & CodeD. We also added another policy that we call CodeDMod, which is an enlarged version of the policy CodeD. This policy set contains 11 policies and 75 rules in total. We include this policy in order to evaluate the performance of our framework with larger real world policies. As highlighted by [36], it is difficult to access large real world policies that are publicly available, due to the confidential information these policies typically carry. Another issue highlighted by other authors [28] is the fact that XACML policies tend to get larger and more complicated with time, hence we introduced CodeDMod to represent such a large policy.

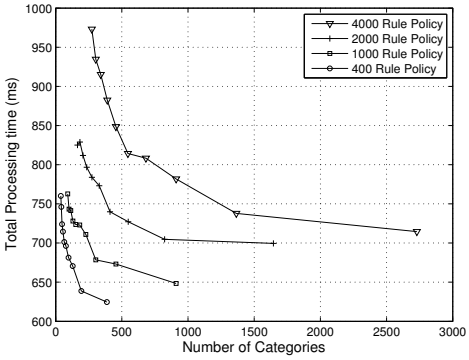
The results of the experiments done on the real world-based policies are summarized as follows: In all cases we obtain at least a 78% performance improvement over Sun's PDP. Despite the nature of our framework which best suits large policies, our optimization engine

still provides a significant performance boost in the case of smaller policies, e.g. CodeA is a policy set with only 2 rules. The policy CodeDMod which is a much larger policy, shows a performance boost of over 91% over Sun's PDP. We also notice the difference between using categorization only and the effect of adding reordering to the framework. Reordering boosts the evaluation performance up to 22% over using categorization only. This is noticeable in the case of CodeDMod where reordering has an effect on its 11 policies' and 75 rules' order. In the smaller policy sets CodeA, CodeB, CodeC, & CodeD, reordering does not provide a big performance boost over categorization only, but still gives up to 8.5% better performance in the case of CodeA.

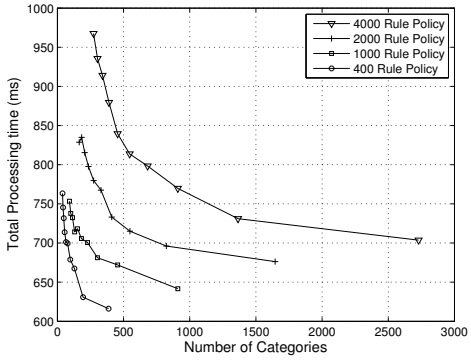
3.4.2 Synthetic Policies

We test our framework against large synthetic policies to show the scalability of the framework and the high performance that it provides in the case of very large policies. We divide the synthetic policies into two test suites, each of which has policy sets of sizes ranging from 400 to 4000 rules. The following sections explain the test suites' results in detail.

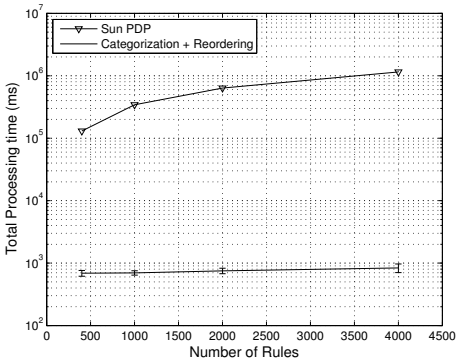
Test Suite I Results: This test suite deals with policy sets where, each subject has a few number of applicable rules. This test case is used to emphasize the effect of our categorization technique, whereas our reordering technique may have a minor effect. This test suite uses policy sets of 4000, 2000, 1000, and 400 rules. For each policy set, rules are divided evenly among 100 policies. For the sake of testing the *Permit Overrides* combining algorithm is used for all the test policy sets and policies. Using this test suite our approach is 1638 times faster than the Sun PDP.



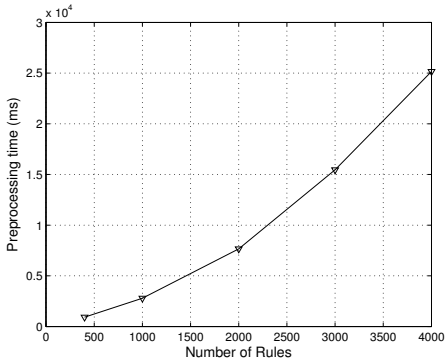
(a) Effect of categorization on evaluation time w.r.t # of categories used with no reordering.



(b) Effect of categorization and reordering on evaluation time w.r.t # categories used.



(c) Evaluation times comparison between our approach and Sun PDP.



(d) Preprocessing times including categorization and reordering.

Figure 7: Experimental Results for Test Suite 1.

Results with Categorization Only: We carried out a first set of tests only applying the categorization technique with no reordering. The number of categories used for each policy set was varied from N to $N/10$, where N is the number of unique subjects within a policy set. The preprocessing time for this approach is the time needed for categorizing a policy set (sub-stage S1.). When using N categories, results show that preprocessing a policy set of 100 policies and 4000 rules takes about 25138 ms and a policy set of 100 policies and 400 rules takes about 913 ms. When $N/10$ categories are used, preprocessing times are 23464 ms and 487 ms for the 4000-rule and 400-rule policy sets respectively.

The experimental results demonstrate that the total processing times for our approach is at least 172 times faster than Sun's PDP. For a policy set of 100 policies and 4000 rules while using $N/10$ categories, it takes 973.1 ms to evaluate 100,000 random requests, whereas Sun's PDP takes about 1152460 ms. A policy set with 400 rules takes 760.2 ms and Sun's PDP takes about 130421.3 ms. When N categories are used, total processing times are 714.6 ms and 624.6 ms for the 4000-rule and 400-rule policy sets respectively. Figure 7(a) shows the complete results when using categorization alone with respect to the number of categories used, which range from 0 to 3000.

Results with Categorization plus Reordering: For this set of tests, we applied the categorization technique, followed by our reordering technique. The number of categories used also range from N to $N/10$. We make use of all sub-stages within the setup stage. Preprocessing time in this case is the time for both categorization and reordering of rules. The results for this set of tests are reported in Figure 7(b). The experimental results shows that the total processing times for our approach is at least 171 times faster than Sun's PDP. For a policy set of 100 policies and 4000 rules while using $N/10$ categories, it takes 967.5 ms to evaluate 100,000 random requests, whereas Sun's PDP takes about 1152460 ms. A policy set with 400 rules takes 763 ms and Sun's PDP takes about 130421.3 ms. When N categories are used, total processing times are 703.7 ms and 616.2 ms for the 4000-rule and 400-rule policy sets respectively. Figure 7(b) shows our complete results when using categorization plus reordering with respect to the number of categories used. Figure 7(c) is a comparison between our approach with categorization plus reordering and Sun's PDP. The plot representing our approach is an average of the best and worst case we obtained

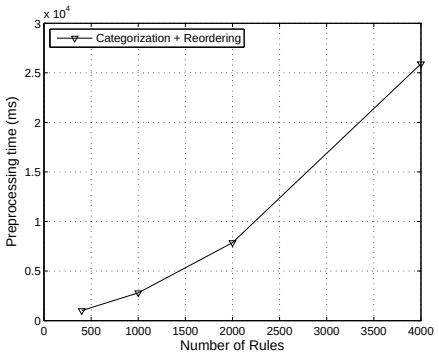
from using different numbers of categories. The results obtained by this set of tests report a very slight performance improvement due to the reordering.

Reordering rules is not a significant factor to performance because of the low number of rules applicable to each subject. Reordering's effect can be better appreciated for policy sets with many rules applicable to each subject.

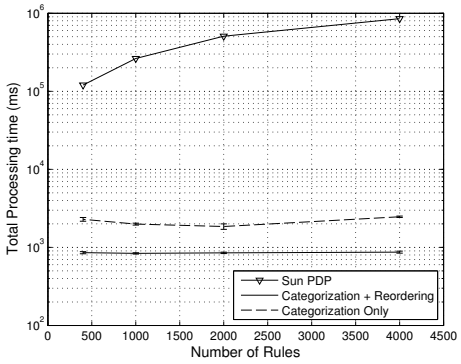
With regards to preprocessing, our results show that preprocessing time is proportional to the number of rules, as reported in Figure 7(d). Preprocessing a policy set of 100 policies and 4000 rules while using N categories takes about 25158 ms, and a policy set with 100 policies and 400 rules takes about 925 ms. When $N/10$ categories are used, preprocessing times are lower, 23472 ms and 491 ms for the 4000-rule and 400-rule policy sets respectively. Our tests also show that the preprocessing times are proportional to the number of categories used. More categories lead to higher preprocessing times due to the extra processing needed to match similar subjects to a common category. Next, we present a second test suite highlighting the advantages of the reordering effect.

Test Suite II Results: We generated a second test suite that could allow us to observe the impact of reordering on performance. This suite simulates a scenario where each subject within a policy set is guaranteed to have a significant number of applicable rules. This case might occur when a specific subject has high privileges and has access to a high number of resources. In this case the subject will have a high number of rules permitting him access to these resources.

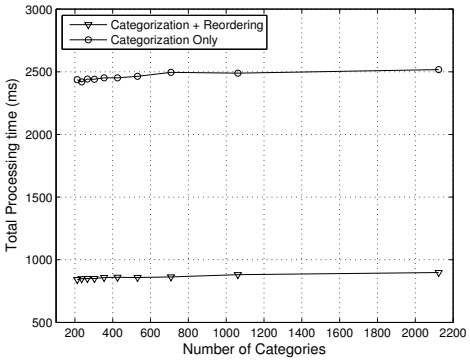
When reordering happens in such a scenario, there will be no need to go over all rules within a subject's category. As expected, this test suite showed a significant performance



(a) Preprocessing times (categorization + reordering).



(b) Sun PDP evaluation times compared to categorization only and categorization + reordering.



(c) Performance boost from reordering w.r.t the # of categories using a 4000-rule policy.

Figure 8: Experimental Results for Test Suite 2.

advantage for the categorization plus reordering approach over the categorization only approach. We used policy sets of 4000, 2000, 1000, and 400 rules (different from the ones used in first test suite). For each policy set, rules are divided evenly among 100 policies. Overall, our results for this test suite show that our approach is 949 times faster than Sun's PDP engine. Similar to the first test suite, we conducted experiments using categorization only and categorization with reordering.

Results with Categorization Only: The preprocessing times for this case are inline with the times for the analogous set of tests of the first test suite. Precisely, when using N

categories, preprocessing a policy set of 100 policies and 4000 rules takes about 25397 ms and a policy set of 100 policies and 400 rules takes about 978 ms. When $N/10$ categories are used, preprocessing times are 28633 ms and 1075 ms for the 4000-rule and 400-rule policy sets respectively.

As in the previous test case, the results for total processing times show a very significant improvement in performance over Sun's PDP. Our results indicate that our mechanism provides at least 48 times faster evaluation. For a policy set of 100 policies and 4000 rules while using $N/10$ categories, it takes 2437.2 ms to evaluate 100,000 random requests, whereas Sun's PDP takes about 851477 ms. A policy set with 400 rules takes 2272.2 ms and Sun's PDP takes about 120230.3 ms. For N categories, total processing times are 2517.6 ms and 2242.5 ms for the 4000-rule and 400-rule policy sets respectively.

Results with Categorization plus Reordering: Figure 8(a) reports the preprocessing times for this approach. Our results show that preprocessing a policy set of 100 policies and 4000 rules while using N categories takes about 25902 ms and a policy set with 100 policies and 400 rules takes about 1007 ms. When $N/10$ categories are used, preprocessing times are 31052 ms and 1061 ms for the 4000-rule and 400-rule policy sets respectively. Although the policies are different, we notice that the gathered times are very similar to the times recorded for preprocessing the set of policies used for the first test suite (reported in Figure 7(d)). This observation leads to the conclusion that the preprocessing time is not influenced by the type of policies used. The preprocessing times are almost negligible when compared to the highly significant performance improvement in total processing times over Sun's PDP, not to mention that preprocessing times correspond to the setup stage

of our framework which only occurs once within a policy set's lifetime or upon a client's request.

Figure 8(b) compares Sun's PDP total evaluation times with our results from the second test suite. The total processing time of our approach is at least 139 times faster than Sun's PDP. As shown, for a policy set of 100 policies and 4000 rules while using $N/10$ categories, it takes 842.3 ms to evaluate 100,000 random requests, whereas Sun's PDP takes about 851477 ms. A policy set with 400 rules takes 867.5 ms and Sun's PDP takes about 120230.3 ms. When N categories are used, total processing times are 897.6 ms and 830 ms for the 4000-rule and 400-rule policy sets respectively.

For the 4000-rule policy set used in this test suite, results indicate that categorization plus reordering has a 65.4% performance improvement over using categorization alone. Figure 8(c) shows the performance boost reordering provides with respect to the number of categorizations used. The figure shows that adding reordering to categorization provides over 1.6 seconds of an advantage over the use of categorization only.

We notice a slight improvement in performance when the number of categories is reduced. This result is explained by the fact that the policy set we used has many rules that are applicable to all subjects, which means the resulting categories are not much different from the original categories.

3.4.3 Adaptability of Reordering Approach

Figure 9, demonstrates how our reordering approach adapts to the incoming requests received by the PDP. As mentioned earlier in the reordering approach, we have a reordering process that reorders both policies within a PolicySet and rules within all policies. The

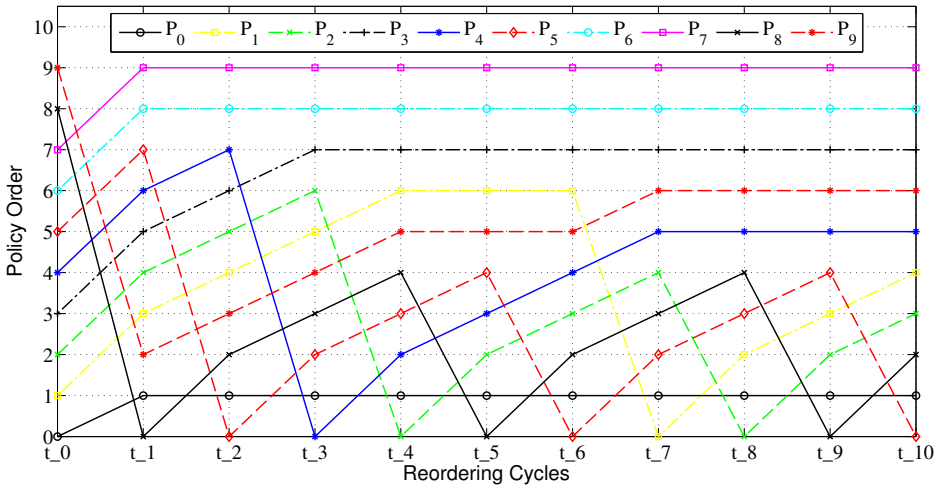


Figure 9: Policy Order and Reordering Cycles

reordering happens according to the number of Permits/Denies a policy or rule triggers. Figure 9 shows how the order of 10 policies within a PolicySet changes with respect to time. The orders of policies ranges from 0 to $L[P_i]$, where $L[P_i]$ is the number of applicable policies for subject S_i (The size of a subject's policy execution vector). Order 0 reflects the highest ranked policy (the policy most requested). Figure 9 shows the policies within a policy execution vector for a particular subject, in this case subject S_1. It is important to notice that each reordering cycle (a single reordering process) is dependent on all previous cycles. In Figure 9, t_0 represents the initial time before reordering, and t_n represents the time at which n reordering cycles have been executed (reordering of policies/rules based on the evaluation results at $t_{n-1}, t_{n-2}, \ldots, t_0$). As time passes and more reordering cycles occur, one can notice how the order of some policies starts to settle at a certain position. For instance, if one looks at policy P_7, it gets pushed to order 9 at t_1, this is due to the low number of Permits/Denies returned by this policy. Whereas if one looks at policy P_0, it gets to order 1 and stays there as it is requested very frequently. Policy P_4 settles after t_7. Other

polices settle for a while and then get reordered as the incoming requests might influence their order positions. The ordering of these policies depends on the incoming requests and how they trigger the accumulated number of Permits/Denies a policy evaluates to. Each subject within a policy set will reflect a similar adaptation process to the one in Figure 9, each of which prioritizes their applicable policies and rules according to the statistics from previous reordering cycles.

To clarify how the adaptation process would actually occur, let us look into a case scenario e.g. a school. At the beginning of a semester, most access requests would be driven by students wanting to register for their courses. The adaptation process would move policies/rules that are applicable to students and favor their incoming requests to the top of a policy set, which will result in faster evaluation times for such similar future requests. Within a semester, where most midterms are given, many faculty requests for inserting or updating student grades will be recorded. In this case, the adaptation process will favor faculty requests by moving policies/rules within a policy set to the top, and hence favoring these requests. Whenever there is a flow of similar requests from different subjects within the school, the policy set will adapt to the best configuration that will result in the best evaluation results.

Figure 10, demonstrates the average request evaluation times for subject S_1 with respect to time. As the time proceeds, a number of reordering cycles occur, hence influencing the order of subject S_1's policies within its policy execution vector and rules within its rule execution vector. The reordering process will push the most requested policies and rules that evaluate to Permit/Deny up to the front of the corresponding execution vectors. This will result in faster evaluation times as depicted by our test results in Figure 10. Note

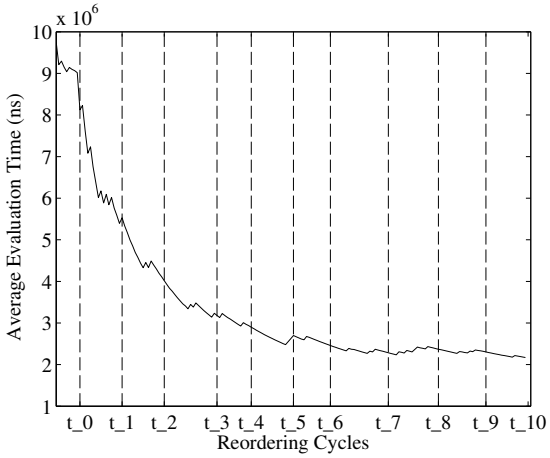


Figure 10: Average Request Evaluation Time and Reordering Cycles.

that the average request evaluation time gradually decreases as more reordering cycles are executed and thus adapt to the incoming different request trends.

CHAPTER 4: RECOMMENDATION MODELS FOR OPEN AUTHORIZATION

The Open Authorization protocol (OAuth) was introduced as a secure and efficient method for authorizing third party applications without releasing a user's access credentials. However, OAuth implementations don't provide the necessary fine-grained access control, nor any recommendations i.e. which access control decisions are most appropriate. We propose an extension to the OAuth 2.0 authorization that enables the provisioning of fine-grained authorization recommendations to users when granting permissions to third party applications. We propose a multi-criteria recommendation model that utilizes application-based, user-based, and category-based collaborative filtering mechanisms. Our collaborative filtering mechanisms are based on previous user decisions, and application permission requests to enhance the privacy of the overall site's user population. We implemented our proposed OAuth extension as a browser extension that allows users to easily configure their privacy settings at application installation time, provides recommendations on requested privacy permissions, and collects data regarding user decisions.

4.1 Related Work

Developing usable tools that provide fine-grained control over user private data is an emerging problem in online platforms especially within social networks [20, 2, 7, 24]. Studies such as the one by Acquisti and Gross [21, 2] indicate user concern over their privacy on social networks while most users did not apply strict privacy settings on their

online social profiles. This was mostly due to the lack or poor understanding of what privacy controls are available to them. Felt et al. [13] detail a novel solution for protecting privacy within social networking platforms through the use of an application programming interface to which independent application owners would agree to adhere to. The approach requires developers to adopt a privacy proxy instead of utilizing already existing technologies such as the popular OAuth 2.0 authorization flow. Recently Felt et al. reviewed the permissions requested by current applications [15]. While some of their findings apply to the context of Android applications, they confirm that up-front permission requirements for installation may help APIs achieve their full potential in a secure fashion, while still be useful for end-users. Fang and LeFevre's work asserts the value in providing highly accurate privacy settings with reduced user input [12]. Using real user input, they infer a set of privacy-preferences using a machine learning approach. While the authors' study is based on real users, they do not provide a technique that applies the inferred privacy settings onto a user's real online profile. Besmer et al. [6] demonstrated in their research the value of social navigation cues in prompting users to make informed privacy decisions; where that research was not concerned with the type of data and arbitrarily assigned a recommended positive or negative cue for each item, our research is very specifically tied to data types and our recommender model provides cues that are based on real user privacy decisions. While much has been researched about the privacy impacts of recommender systems themselves [56, 53, 8], little research appears to be available for the use of recommender systems in aiding privacy and security systems. One notable exception is in the research of Kelly et al. [33] where the authors demonstrated the benefit of combining collaboration among a user population in the suggestion of an individual user's privacy policy. They also propose

an incremental model for optimizing a user's policy over time. This approach is not optimal when dealing with third party applications, that once installed, can harvest a user's private social network data. Shehab et al. [61] proposed an access control framework that allows users to specify the data attributes to share with applications and the degree of specificity. The framework requires many changes to existing authorization models and requires developers to go through a cumbersome deployment process.

4.2 Proposed OAuth Flow

We propose an extension to the OAuth 2.0 authorization code flow, by introducing two new modules into the flow: 1) A Permission Guide that guides users through the requested permissions, and shows them a set of recommendations on each of the requested permissions, and 2) A Recommendation Service that retrieves a set of recommendations for the requested permissions following a collaborative filtering model as seen in Section 4.2.2.

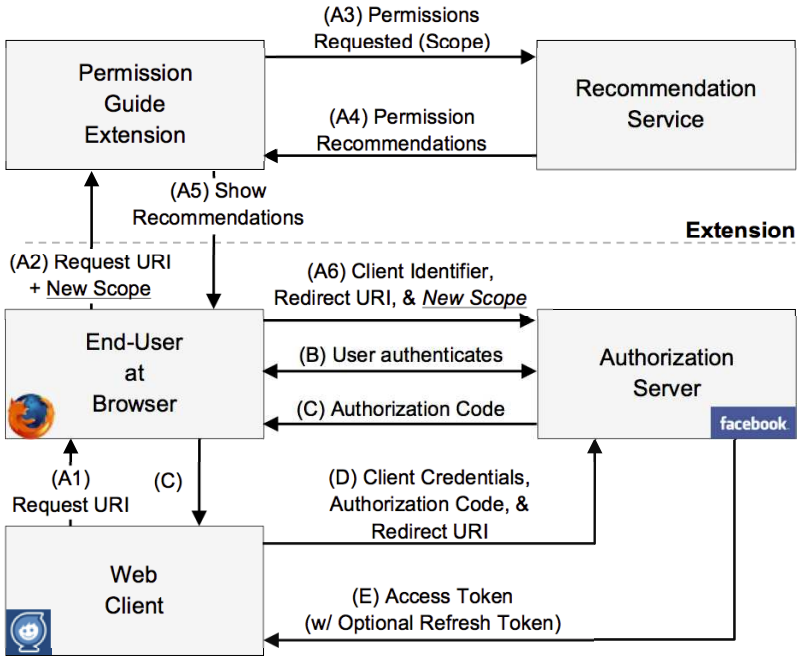


Figure 11: Proposed OAuth Flow

Our OAuth extension focuses on step “(A)” of the authorization code flow in OAuth 2.0 [52]. We revise step “(A)” to become a six stage process as shown in Figure 11 and explained in the following steps:

A1: The client redirects the browser to the end-user authorization endpoint by initiating a request URI that includes a `scope` parameter.

A2: The Permission Guide extension captures the `scope` value from the request URI and parses the requested permissions. At this step the extension allows users to choose a subset of the permissions requested.

A3: The Permission Guide extension requests a set of recommendations on the parsed permissions. This is achieved by passing the set of permissions to our Recommendation Service.

A4: The Recommendation Service returns a set of recommendations for the permissions requested by the client.

A5: Using the set of returned recommendations, the extension presents the permissions with their respective recommendations in a user friendly manner.

A6: The Permission Guide extension redirects the end-user’s browser to a new request URI with a new scope (`scope'`), assuming the user chooses to modify the requested permissions.

4.2.1 Permission Guide

The Permission Guide is represented by a browser extension that integrates into the authorization process by capturing the `scope` parameter value within the request URI generated by a third party application. Once the `scope` is captured, the extension parses the

requested permissions and presents them in a user friendly manner as shown in Figure 16. A readable label of each requested permission is shown to the end-user e.g. it shows “Facebook Chat” rather than `xmpp_login`.

The extension also shows users a set of recommendations for the requested permissions. For each permission there is a thumbs-up and thumbs-down recommendation value. These recommendations represent prediction values that we calculate following our model in section 4.2.2. These prediction values represent the likeliness of a user to grant or deny a certain permission based on her previous decisions and on the collaborative decisions of other users. Users who have not made any decisions yet, are shown recommendations based on other user decisions.

The extension also allows users to customize the requested permissions by checking or unchecking individual permissions, where a checked permission is one the user wishes to grant to the third party application and an unchecked permission is one she wishes to deny access to. Once a user decides on the permissions she wishes to grant and deny, she simply needs to click a *Set Permissions* button on the extension (blue button in Figure 16). This will trigger the extension to generate a new request URI with a new scope `scope'`, and forward the user’s browser to this new request URI. `scope'` will always be a subset of the original requested scope, i.e. `scope'` $\subseteq$ `scope`. An example `scope'` for the FriendCameo application could be as follows:

```
scope'=publish_stream
```

reflecting the user’s desire to allow FriendCameo to post to her feed/wall, but deny it access to her email, Facebook chat and friend’s online/offline presence. Note that using a

subset of the permissions requested could potentially hinder the functionality of a third party application once installed.

Our Permission Guide extension also collects the user's decisions on the requested permissions, hence allows us to generate a data set of decisions to be used in our recommendation model explained in section 4.2.2. That is, our Recommendation Service as seen in Figure 11 will utilize these decisions in making its recommendation predictions. These decisions are uploaded to our servers once a user sets her desired permissions within the extension, i.e. clicks the *Set Permissions* button. The data uploaded to our servers includes: `app_id, requested_perms, decisions, recommendations`, where the `app_id` is the application's unique id which is assigned by the service provider (e.g. Facebook), the `requested_perms` is the scope of permissions requested by the third party application, the `decisions` are the individual user decisions (grant or deny) on each of the requested permissions, and the `recommendations` are the recommendation values at the time the user made her decisions.

Our goal is to provide a simple user interface for interacting with permission requests, hence increasing user awareness and providing an easy mechanism for guiding users in making their decisions.

4.2.2 Recommendation Model

We propose a Recommendation Service component that extends upon our Permission Guide extension. Let $\mathcal{A}, \mathcal{U}$ and $\mathcal{P}$ represent the set of applications, users and permissions respectively. A user $u_i \in \mathcal{U}$ can make a decision $d_i \in \{grant, deny\}$ on a permission $p_j \in \mathcal{P}$ for an application $a_k \in \mathcal{A}$. An application a_k which requests permissions $p_1, \cdots, p_m$ is

mapped to a set of decisions $d_1, \cdots, d_m$ made by the user installing a_k.

4.2.3 Collaborative Filtering

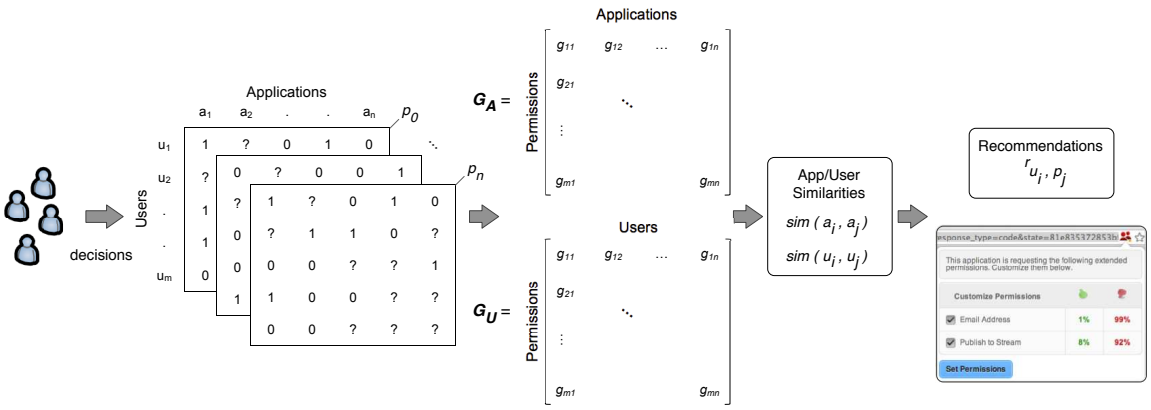


Figure 12: Collaborative-based model

Our model follows the multi-criteria recommendation model where user recommendations are calculated per criterion [38, 3]. The model utilizes the set of permissions $\mathcal{P}$ as a set of criteria, i.e. each permission $p_j \in \mathcal{P}$ represents an individual criterion within the model. The multi-criteria approach fits our model as decisions are made per permission (criteria) rather than an application as a whole. We model a user's utility for a given application with the user's decisions $d_1, \cdots, d_m$ on each individual permission $p_1, \cdots, p_m$ using Function 1.

$$D : Users \times Applications \rightarrow d_1 \times \cdots \times d_m \tag{1}$$

Function 1 represents a user's overall decision on a certain application via the set of decisions made on each individually requested permission. That is, a user u_i makes a decision d_i on an application a_k with respect to an individual permission. For each permission p_j, there exists a matrix $\mathcal{C}_{p_j}$ representing user decisions on p_j for each application $a_k \in \mathcal{A}$, see Figure 13. A matrix entry d_i with a value of 1 denotes a user has *granted* a_k the permission

p_j, whereas a 0 denotes a *deny*. Entries with "?" values denote the user is yet to make a decision on permission p_j for application a_k. Our model provides recommendations to users that guide them in making these future decisions. Applications that do not request a permission p_j have an empty entry in $\mathcal{C}_{p_j}$ and are handled properly in our implementation.

For example, let $p_1 = birthday$, $p_2 = email$, and $p_3 = location$, where each represents a single criterion within a three-criteria model. Let $u_1 = Alice$ who installed application a_1 that requests access to the permissions *birthday*, *email*, and *location*. As illustrated in Figure 13, Alice has granted a_1 the permissions *birthday* and *location* ($d_1 = grant, d_3 = grant$), whereas denied *email* ($d_3 = deny$). Alice has yet to make a decision on a_2 i.e. a single decision on each requested permission $\in \{birthday, email, location\}$. Our proposed model utilizes the set of decisions for each $\mathcal{C}_{p_j}$, hence providing a recommendation that fits each criterion.

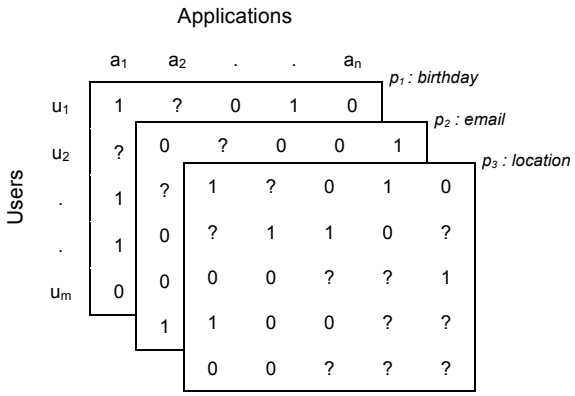


Figure 13: A three-permission (criteria) model

Figure 12 illustrates our overall collaborative model. The model relies on decisions made by the community users, and utilizes them in building the multi-criteria matrices $\mathcal{C}$ for each permission. By utilizing the $\mathcal{C}$ matrices, we generate two probability matrices, G_A and G_U, as seen in Figure 12. G_A is app-based, whereas G_U is user-based. G_A captures

the probability of a certain application being granted a certain permission, whereas G_U captures the probability of a certain user granting a certain permission.

Figure 14 shows an example G_A matrix, with a set of applications (a_1, a_2, a_3, a_4, a_5), permissions (`birthday`, `email`, `location`, `sms`, `photos`) and their corresponding $G_A(j,k)$ values. For example, $G_A(location, a_2) = 0.15$, denotes a low probability of the permission `location` being granted to application a_2 by users who installed a_2. Our proposed collaborative model adopts an item-based and user-based collaborative filtering process. In our model, items are applications, hence we refer to item-based filtering as application-based filtering. User-based filtering utilizes the user-based probability values of G_U, whereas application-based filtering utilizes the app-based probabilities of G_A as seen in Figure 12.

Permissions \ Applications	a_1	a_2	a_3	a_4	a_5
birthday	0.6	0.75	1	0.2	0.3
email	0	0.9	0.25	0.7	0.1
location	1	0.15	0	0.35	0
sms	0	0.4	0	1	0.5
photos	0.2	0	0.6	0.25	0

Figure 14: Example $G_A(j,k)$ values.

4.2.3.1 Application-based Filtering

Our application-based filtering process relies on the app-based probability values of G_A shown in Figure 12. Each entry $G_A(j,k)$ in G_A represents the overall probability of permission p_j being granted to application a_k.

To generate recommendations on the requested permissions, we first detect the nearest-

neighbors for the target application requesting the permissions. The nearest-neighbors in app-based filtering are the applications most similar to the target application. Collaborative filtering algorithms have mainly been based on one of two popular similarity measures namely the Pearson Correlation and Cosine-similarity [25, 45]. We measure similarities between applications using the G_A values, and by calculating the Pearson correlation values between them. Equation 2 represents our application-based similarity measure, which is the Pearson correlation value between applications a_i and a_j, where $\mathcal{P}$ is the set of all permissions in our system and $\overline{G_A}(a_i)$ is the average probability for application a_i being granted a permission in $\mathcal{P}$.

$$sim(i,j) = \frac{\sum_{\forall p \in \mathcal{P}} (G_A(p,i) - \overline{G_A}(a_i))(G_A(p,j) - \overline{G_A}(a_j))}{\sqrt{\sum_{\forall p \in \mathcal{P}} (G_A(p,i) - \overline{G_A}(a_i))^2 \sum_{\forall p \in \mathcal{P}} (G_A(p,j) - \overline{G_A}(a_j))^2}} \tag{2}$$

Applications that don't request a certain permission p_j have a $G_A(j,i)$ of zero. Applications which are similar and highly correlated, are those which request a similar set of permissions, and have similar $G_A(j,i)$ values for each of their requested permissions. For example, if both applications a_1 and a_2 requested the same set of permissions $\{p1, p_2\}$, and they have a $G_A(p_1,a_1) = G_A(p_1,a_2)$ and a $G_A(p_2,a_1) = G_A(p_2,a_2)$, then a_1 and a_2 are considered highly correlated and their application-similarity value $sim(i,j)$ will be close to 1. When predicting recommendation values for permissions of application a_i, we make sure they are based on a_i's *nearest neighbors*, that is, the set of applications where $sim(a_i,a_j)$ is highest. With application-based filtering, users collaborate towards increasing or decreasing the $G_A(j,k)$ values, hence filtering applications according to the willingness of users

to grant them certain permissions.

4.2.3.2 User-based Filtering

User-based filtering relies on the G_U values, where each entry $G_U(j,k)$ in G_U represents the overall probability of permission p_j being granted by a focus user u_k. Permission recommendations in this case are based on the focus user's nearest-neighbors, that is, the users most similar to the focus user. Similar to application-based filtering, we use the Pearson correlation to measure similarities between users. Equation 3 represents our user-based similarity measure, which in terms is the Pearson correlation value between users u_i and u_j, where $\overline{G_U}(u_i)$ is the average probability of user u_i granting a permission in $\mathcal{P}$.

$$sim(i,j) = \frac{\sum_{\forall p \in \mathcal{P}} (G_U(p,i) - \overline{G_U}(u_i))(G_U(p,j) - \overline{G_U}(u_j))}{\sqrt{\sum_{\forall p \in \mathcal{P}} (G_U(p,i) - \overline{G_U}(u_i))^2 \sum_{\forall p \in \mathcal{P}} (G_U(p,j) - \overline{G_U}(u_j))^2}} \tag{3}$$

With user-based filtering, a focus user u_i is given recommendations based on those users most similar to him/her. Users with more similar probabilities of granting a certain permission will be more similar, hence, potentially reflect a similar willingness to grant/deny a certain permission.

We use both application-based and user-based filtering to calculate a recommendation value on permissions requested by application a_i on behalf of user u_i.

4.2.4 Prediction Model

When a user u_i, say Alice, wants to install application a_k, we calculate a set $\mathcal{R}_k$, where $r_{i,j} \in \mathcal{R}_k$ is a prediction value for permission p_j requested by a_k. $r_{i,j} \in \mathcal{R}_k$ is a prediction of how likely Alice would be willing to grant p_j to a_k.

The recommendation value $r_{i,j}$ is based on either our app-based filtering or user-based filtering approaches. That is, the recommendations are either based on a_i's nearest-neighbors (most similar applications) or u_i's nearest-neighbors (most similar users). Equations 4 and 5 show the recommendation value for app-based and user-based filtering respectively. Note that we calculate $r_{i,j}$ for each p_j requested by an application a_k.

$$r_{i,j} = \overline{G_A}(p_j) + \frac{\sum_{a \in \mathcal{N}} sim(a_k, a) * d_{j,a}}{\sum_{a \in \mathcal{N}} |sim(a_k, a)|} \tag{4}$$

$$r_{i,j} = \overline{G_U}(p_j) + \frac{\sum_{u \in \mathcal{N}} sim(u_i, u) * d_{j,a_k}}{\sum_{u \in \mathcal{N}} |sim(u_i, u)|} \tag{5}$$

In Equation 4, $\overline{G_A}(p_j))$ reflects the average probability that permission p_j is granted over all applications in $\mathcal{A}$, and is easily calculated via it's corresponding row in the G_A matrix. Similarly, in Equation 5, $\overline{G_U}(p_j)$ represents the average probability that permission p_j is granted over all users in $\mathcal{U}$, and is calculated via it's corresponding row in the G_U matrix. Note that both $\overline{G_A}(p_j))$ and $\overline{G_U}(p_j)$ are driven by all users within our system. In both equations, $\mathcal{N}$ represents the target application's nearest-neighbors and the focus user's nearest-neighbors respectively. The size of $\mathcal{N}$ depends on the similarity measures used, and can be adjusted to follow a preset threshold within the implementation, e.g. only include neighbors with a similarity above 0.8.

Finally, $d_{j,a}$ in Equation 4 represents u_i's (focus user) previous decisions on permission p_j for each application $a \in \mathcal{N}$. In Equation 5, d_{j,a_k} is a neighboring user's decision on p_j for the focus application a_k. Note that the $sim(u_i, u)$ value will either increase or decrease the effect of a neighboring user's decision, based on how similar the neighboring user is

to the focus user. Both $d_{j,a}$ and d_{j,a_i} are captured via the $\mathcal{C}_{p_j}$ matrix explained earlier (see Figure 13).

Notice that the prediction values calculated are based on a user's previous decisions and on the decisions of other users, hence capturing the essence of collaborative filtering. In cases of insufficient data, prediction models could refrain from generating predictions, or utilize collaborative filtering systems based on probabilistic, hybrid, or clustering approaches for generating predictions. We decided not to provide predictions in such cases.

4.2.4.1 Category-based Predictions

To further enhance the results of our recommendation predictions, we propose a category-based model that takes into consideration an application's category. Example application categories include Games, Utilities, Entertainment, etc. Categories can increase the precision of our predictions especially for applications that request similar permissions for different purposes. For example, two applications might request access to a user's email address, where the first application is a game and the second is a task manager. In this example scenario, a user's email could be used for different purposes, i.e. a task manager could use it for sending reminder emails, whereas a game could use it to send promotions for other games. A user would probably be more willing to grant email permission to the task manager as it could be of more benefit to the user. Granting or denying a certain permission will be driven by the user's perception of the requested permission. We believe that similar permissions requested by apps within the same category will be perceived similarly by users. Hence, by providing recommendation predictions based on application categories, we can reflect more precise user perceptions within our recommendations.

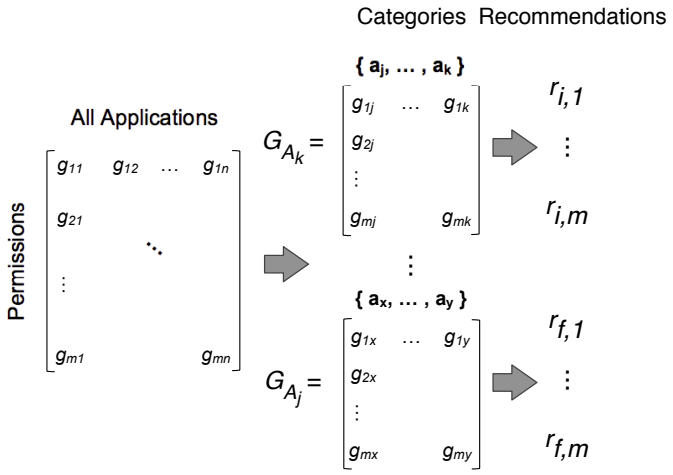


Figure 15: Application category probability matrices

When generating category-based predictions, we follow a modified version of our application based filtering model for calculating similarities. To calculate the set of nearest neighbors for a certain application a_i, we only consider other applications that fall into the same category as a_i. Figure 15 shows two probability matrices G_{A_k} and G_{A_j}, which are extracted from the overall G_A matrix explained previously. G_{A_k} and G_{A_j} represent the permission probabilities for applications within the categories k and j respectively. Let $\mathcal{A}_k \subseteq \mathcal{A}$ be the set of applications that belong to category k, and $\mathcal{N}_i$ be a_i's nearest neighbors where $\mathcal{N}_i \subseteq \mathcal{A}_k$. Note that a_i's nearest neighbors can be found by calculating the similarities between a_i and applications within $\mathcal{A}_k$ rather than all applications in $\mathcal{A}$. For example, in Figure 15, the nearest neighbors for a_y are found among the set of apps $\{a_x \dots a_y\}$, and the similarities are calculated using G_{A_j}. For application $a_i \in \mathcal{A}$ that belongs to category k, we calculate recommendation predictions following Equation 6.

$$r_{i,j} = \overline{G_{A_k}}(p_j) + \frac{\sum_{a \in \mathcal{N}_i} sim(a_i, a) * d_{j,a}}{\sum_{a \in \mathcal{N}_i} |sim(a_i, a)|} \tag{6}$$

Where $\overline{G_{A_k}}(p_j)$ reflects the average probability that permission p_j is granted over applica-

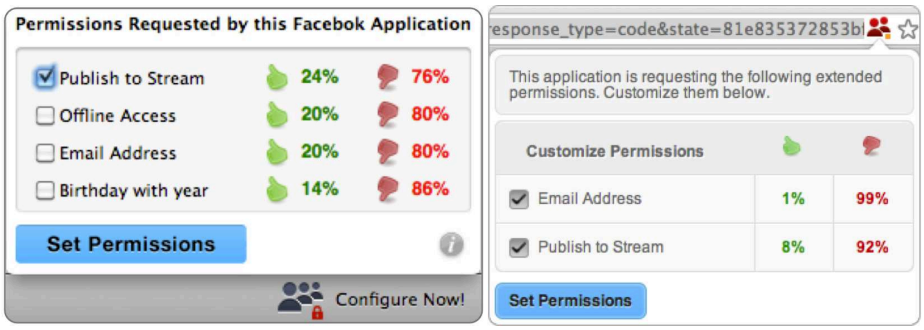
tions in $\mathcal{A}_k$, i.e. apps that fall within a_i's category. Category-based predictions are more efficient in that they do not rely on all applications within our system, but rather on a smaller subset of categorized applications. This allows for faster prediction calculations, in addition to the potentially more precise recommendations.

4.3 Experiments

We evaluate our proposed OAuth 2.0 extension using Facebook as our target platform. Facebook is an ideal target given its large user base of over 800 million users, and its extensive application directory of of over 7 million third party applications [11]. Facebook is also one of the major platforms to adopt the OAuth 2.0 protocol, which makes it a good fit for our evaluation process. The proposed extension is not limited to Facebook and can be extended to other OAuth 2.0 platforms. To evaluate our proposed OAuth 2.0 extension, we implemented two main components: a Permission Guide, and a Recommendation Service.

Permission Guide: Our proposed Permission Guide in section 4.2.1 was implemented as a browser extension for both Firefox and Chrome browsers, using a combination of Mozilla's XML User Interface Language, the Google Chrome browser APIs and Javascript. Figure 16 shows the extension user interface for both Firefox and Chrome. Javascript was used to interact with our back-end recommendation service API. The extension was tested on the latest Firefox and Chrome browsers on Mac OS X 10.6/10.7, Linux CentOS and Windows (Vista, 7) machines.

Once installed, the extension resides within the user's browser and begins monitoring, waiting for a Facebook application installation process to commence. The extension does not otherwise interfere with a user's browsing experience. Once a Facebook application



FireFox Extension UI Chrome Extension UI

Figure 16: Extension UI, Firefox and Chrome.

installation process is detected, the extension is activated and presented to the user.

An installation process is detected by parsing the URLs a user visits and searching for a *Permission Request*. A *Permission Request* for Facebook applications can be identified by locating the substrings *permissions.request* and either of *facebook.com/connect/uiserver* or *facebook.com/dialog/permissions.request*. If a request is detected, the extension looks for the type of request issued, i.e. Basic permission vs. Extended permission access. A basic permission access request is identified by a missing or empty `scope` attribute within the URL. Otherwise, if the *scope* attribute is located, the extension recognizes that an extended permission access request is in progress.

Recommendation Service: The service is a PHP based solution running on Apache 2.2.14 with MySQL 5.1.5 as the data store solution. We run the service on a desktop machine running Linux CentOS, with 2GB RAM and a 2.0 GHz Intel Xeon CPU. The recommendation service applies the recommendation based schema explained in section 4.2.2 by providing two private API methods which are used by our extension. The first API method is the `getRecommendations` method which accepts an `app_id` and a set of requested permissions. It then returns a set of recommendations in a JSON format which

maps a recommendation value to each permission. The second API method provided is the `postDecisions` method which is invoked by our extension when a user makes her decision on the requested permissions. This API method takes an `app_id`, a set of requested permissions, a set of user decisions on these permissions, and the set of recommendation values displayed at decision time. These values are stored onto our recommendation back-end server and used later in our recommendation based schema.

For our evaluation purposes, we are primarily focused on extended permission requests because those are the permissions which are customizable by users on the targeted platform (Facebook). For basic permission requests, our extension notifies users that basic access is requested, and no customization is possible. Whereas for extended permission requests our extension performs the following:

1. Extracts the permissions requested by parsing the `scope` value from within the request URI. For Facebook, the `scope` value is a list of comma-delimited strings, each string representing a certain requested permission.
2. Asynchronously retrieves recommendations for the set of requested permissions by calling our API method `getRecommendations`. Once the recommendations are retrieved, the extension UI is updated properly.
3. Dynamically generates the user interface to be shown to the user based on the requested permissions and their respective recommendation values. Figure 16 shows an example interface for

```
scope = publish_stream, offline_access, email, birthday
```

Once the user makes a decision on the permissions she would like to grant/deny by

clicking the ”Set Permissions” button, the extension will perform two actions: 1)Invoke our `postDecisions` API method passing along the user decisions. 2)Generate a new `scope` value using the permissions granted by the user. Using this new scope `scope'` the user is then redirected to a customized application request URI, resulting in a new Facebook application permission request page. At this point the user has defended herself against unnecessary application accesses. Note that our approach prevents an application from acquiring permissions before its actual installation. The current approach by Facebook allows the removal of permissions only after applications are installed, which is realistically not sufficient because applications have already acquired access to the data.

4.3.1 User Study

To evaluate our proposed framework, we perform a user study on our browser extension FBSecure. The study's main research questions were: *1) Do permission recommendations (positive/negative) affect the user's willingness to allow/deny permissions requested by third party applications?* and *2) Are users more willing to share their friends' privacy attributes in comparison to their own?* We use statistical measures to evaluate the success of our proposed framework as discussed in Section 5.6.1.2.

4.3.1.1 Methedology

Our proposed browser extension is hosted under the name of FBSecure on the Mozilla Add-Ons website (Firefox version) and, the Google Chrome web store website (Chrome version). In addition, it was posted on our lab website (`http://liisp.uncc.edu/fbs`). Twitter was also used as a means of recruiting participants for this study which was approved by UNC Charlotte IRB (Protocol# 11-05-24). FBSecure was installed by over

3528 Facebook users who installed over 1561 unique Facebook applications. The results summarized in this section are based on the population of users who installed our browser extension, use Facebook, and sought out privacy extensions. This user sample is mainly biased towards privacy aware users, but also includes regular users recruited via Twitter, whom did not specifically seek out privacy extensions.

4.3.1.2 Study Results

We gathered over 7200 user decisions on 56 different Facebook extended permissions. We evaluate our recommendation model based on the user decisions collected during the usage of the extension. For every application permission request, our extension enabled the collection of the details of the requested permission, the generated recommendation, and the user selected permission settings. Figure 19, shows the probability of applications requesting different permissions, for example we found that the most popular requested permission is the `publish_stream` permission, which enables apps to post messages on a user's wall, and is requested by 42% of the Facebook apps. Other popular permissions include `email, offline_access` and `user_birthday`.

Over all our user population, Figure 17 shows how likely users were willing to grant different permissions. Our results show that users have varying willingness towards different permissions, for example the likelihood of a user giving an application access to his email is only 31%, while users are more likely to share their status (65%) with apps. Note that some permissions requested give applications access to user's friends' information, for example `friends_location` permission. To investigate the permissions that users are more willing to grant on their friends' data compared to their own data we conducted a

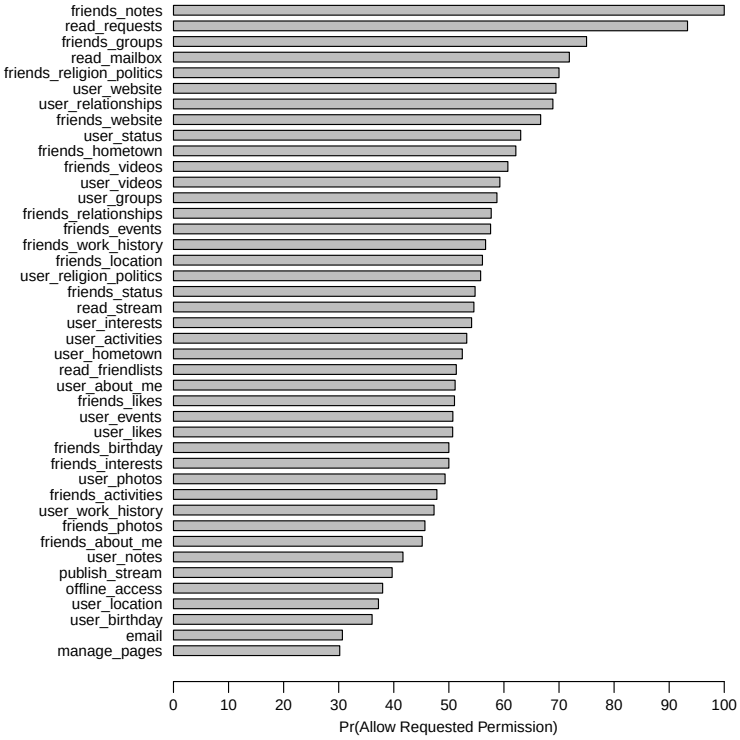


Figure 17: Probability of Allowing a Requested Permission

t-test on the likely of allow statistic collected when users are asked for permission to access both their data and their friends' data. With a significance level of 5%, Figure 18 shows the permissions for the hypothesis that users are more willing to share their friends data is accepted. For example, it is statistically significant that users are more willing to share with apps their friends' birthday compared to their birthday.

Figure 20(a), summarizes the distribution of the number of permissions requested by applications, with an average of 3.1 permissions requested per application. Figure 20(b), shows the average number of granted permissions for apps requesting permissions, and it can be noted that on average applications are granted around 44.7% of the permissions that are requested. Figure 20(c), shows the distribution of number of applications by users who installed the extension, on average the extension was used to install 5.2 applications.

Attribute	**User** (μ, σ)	**Friend** (μ, σ)	**p-value**
notes	(0.42, 0.50)	(0.98, 0.21)	0.0019
birthday	(0.38, 0.45)	(0.48, 0.46)	0.0123
location	(0.38, 0.44)	(0.57, 0.45)	0.0144
groups	(0.57, 0.47)	(0.75, 0.42)	0.0253
work_history	(0.45, 0.43)	(0.58, 0.44)	0.0313
religion_politics	(0.56, 0.48)	(0.71, 0.50)	0.0377
online_presence	(0.38, 0.42)	(0.56, 0.49)	0.0456
events	(0.51, 0.51)	(0.60, 0.58)	0.0475
videos	(0.58, 0.41)	(0.61, 0.44)	0.0491

Figure 18: T-test user and friend permissions

The extension provides users with recommendations for each of the application requested permissions. The recommendation is presented to the user as thumbs up and thumbs down with their associated recommendation values based on the the recommender models presented in previous sections. We are interested in evaluating whether the recommender system properly predicts the user's decision. Also we are interested in evaluating what is the lowest (highest) recommendation value that will influence users into granting (denying) a requested permission, we refer to this value as the threshold T. Where users said to be encouraged to grant the permission if the recommendation is higher than T and to deny otherwise. In this case we have four possible outcomes for the recommended and decided value, see Figure 21.

In literature there are several proposed metrics for evaluating recommender system performance, we focus on the most adopted metrics in literature which are based on three measures namely accuracy, precision and recall [26]. Accuracy of the recommender system is the degree of closeness of the recommender system to the actual decision taken by the user, which is calculated as $\frac{TP+TN}{TP+TN+FP+FN}$. The precision or the repeatability of the recommender system, is a measure of the degree to which repeated recommendations un-

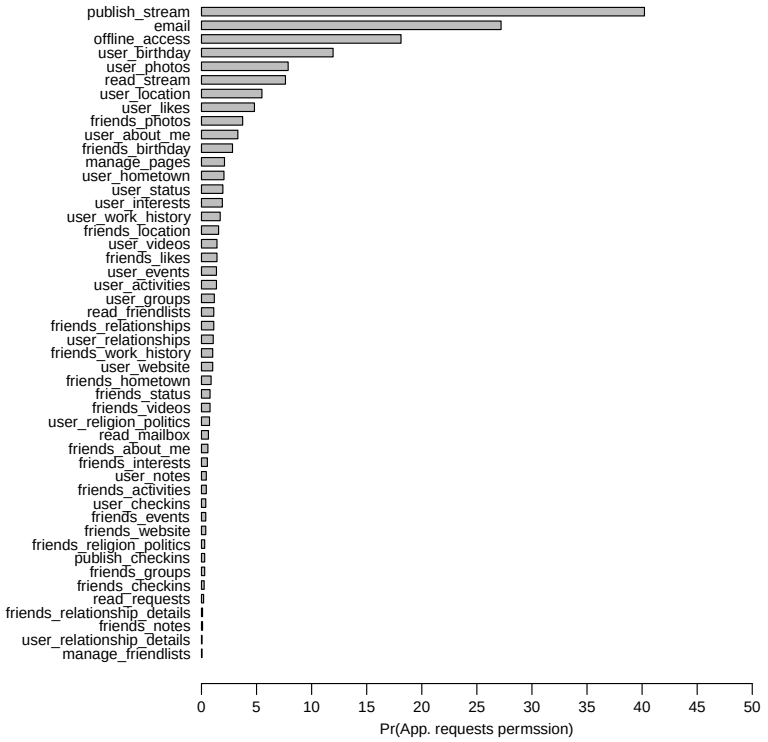
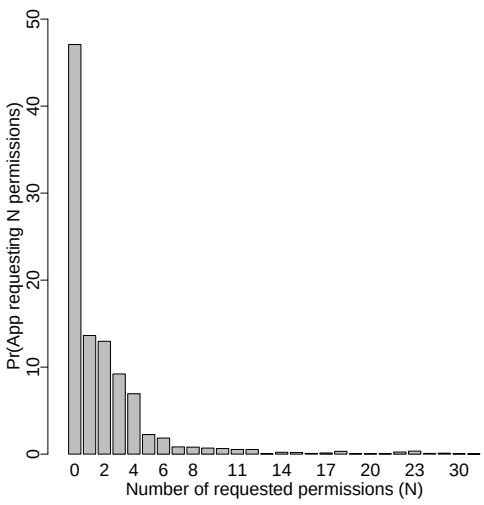
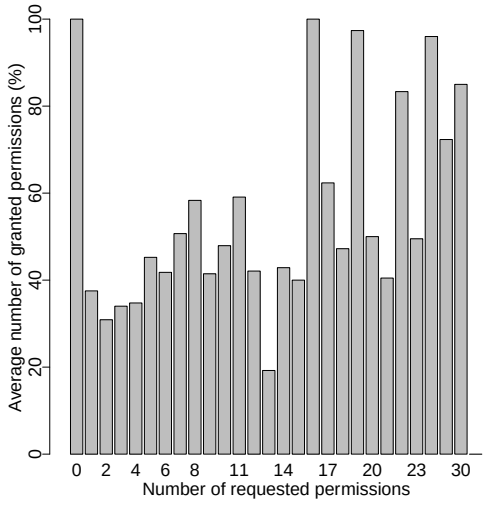


Figure 19: Probability of Requesting a Permission

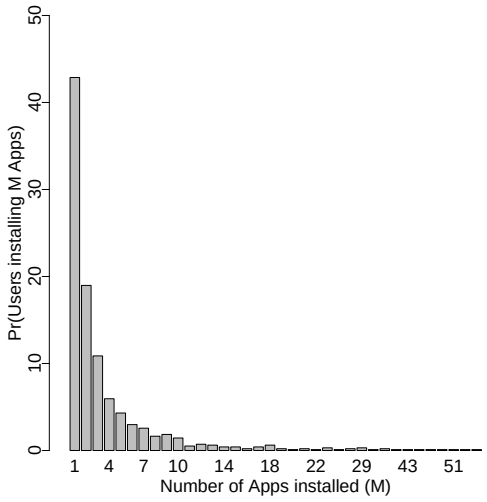
der the similar conditions generate the same results, which is computed as $\frac{TP}{TP+FP}$. The recall or sensitivity is a measure of the ability of the recommender system to select instances of either to recommend or not, which is computed as, $\frac{TP}{TP+FN}$. Figure 22, shows the accuracy, precision and recall calculated for different threshold values. The experiments were conducted to evaluate the proposed application based, user based and category based recommendation models. The application and category based approaches maintained an accuracy of over 90%. The category based approach provided the highest accuracy, this is due to the refined application similarity value as apps in a given category provide a better context for providing recommendations for apps from the same category. The precision and recall are inversely proportional with a break even region around the threshold value of 45%, which could explain that the recommendation value of 45% or higher is an indication



(a) App. Requested Permissions



(b) App. Granted Permissions



(c) User Extension Usage Stats

Figure 20: Permission Probability Profiles

that the system is recommending to grant the requested permission, and lower than 45% is recommending to deny the permission. Also note that the system achieves a precision and recall values of 92-85% and 75-85% around this threshold.

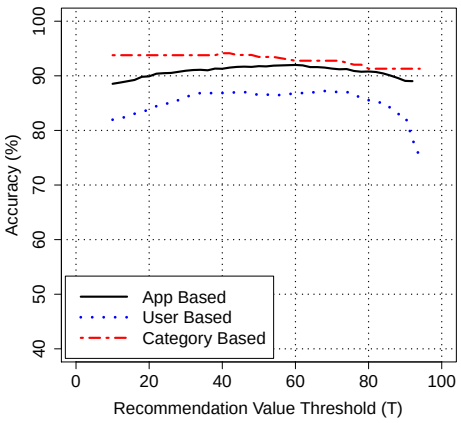
In addition to investigating the accuracy, precision and recall measures we further investigated the causality of our recommendation scheme. That is, are users less likely to grant permissions when using the recommendation based scheme. To investigate, our browser

	Recommended	**Not Recommended**
Used	True Positive (TP)	False Negative (FN)
Not Used	False Positive (FP)	True Negative (TN)

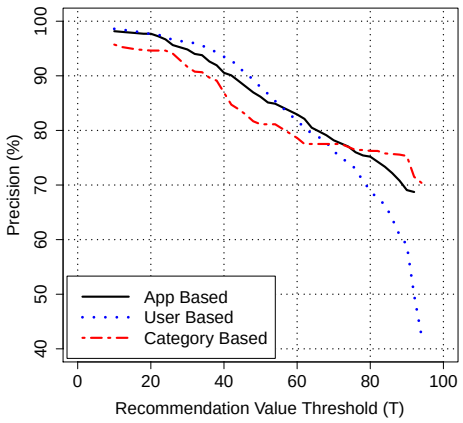
Figure 21: Classification based on user decisions

extension was designed to accommodate two groups of users. The first group (G1), are users who were not shown the recommendation values (see Figure 23). The second group (G2), are users who were shown the recommendation values generated by the recommendation system (see Figure 24). The extension randomly selected users who belonged in each of the groups. For each group we recorded the users' *openness*, which is the percentage of granted permissions for each application installed.

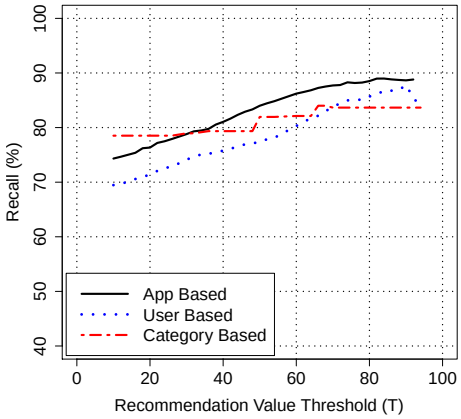
The average user openness of G1 and G2 were 66.5% and 30.7% respectively, which indicates that users who were not presented with the recommendation were more likely to grant permissions to applications. To compare the two groups we performed a T-test of the hypothesis to investigate the following question, "on average, are users in G2 less open than users in G1?". Using the collected data, with a significance level of 5% this hypothesis was accepted (P-Value of 0.0001). These results show that the users who were presented with the recommendation values were less open to granting permissions to applications. The results presented in this experiment are based on the average openness values calculated over all installed apps in both groups. Figure 25, shows the expected openness for the two groups for specific permissions for which the hypothesis was accepted.



(a) Accuracy



(b) Precision



(c) Recall

Figure 22: Recommendation system accuracy, precision and recall evaluation

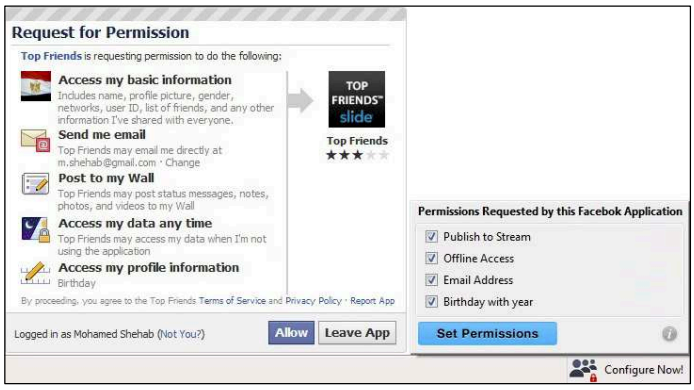


Figure 23: Group (G1) with no recommendations shown

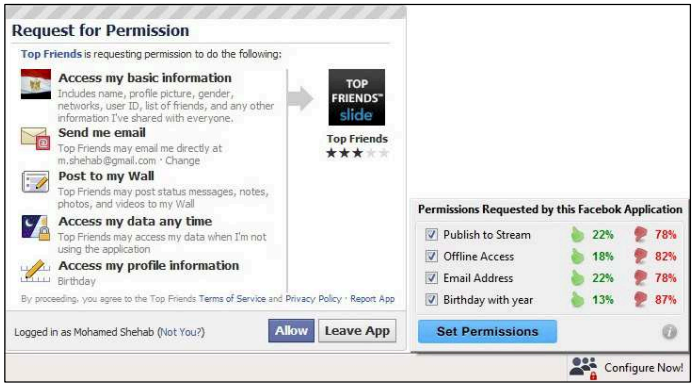


Figure 24: Group (G2) with recommendations shown

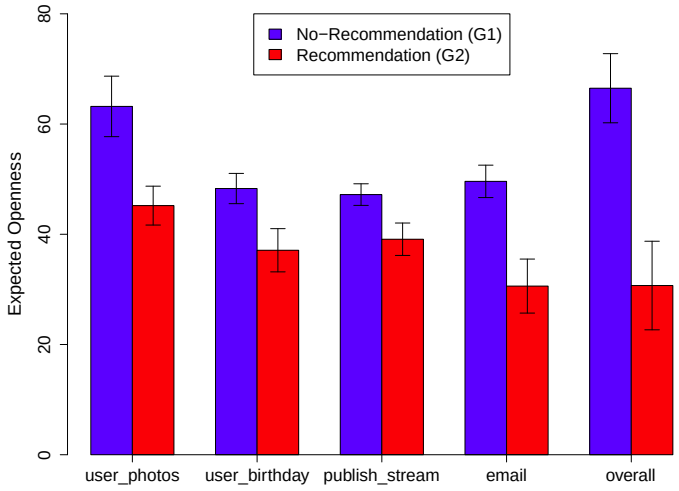


Figure 25: Groups (G1) and (G2) Expected Openness

CHAPTER 5: 3RD PARTY BROWSER EXTENSION POLICY MANAGEMENT

Today's online activities such as social networking, banking and other daily online activities have increased the users' online presence and made the browser a main portal for users. Users are increasingly enriching their browsing experience with third party applications that provide new functionalities and improve upon existing ones. Third party browser extensions are popularly used by millions of users [50, 66], especially with their wide availability on online portals such as Google's Chrome Web Store.

Regardless of the popularity and benefits of third party browser extensions, they could potentially threat the privacy of their users. This lead platforms such as Google Chrome to introducing permission models that control third party extension accesses, especially those regrading sensitive user data. These models allow developers to declare the permissions their extensions require. Extension users on the other hand are responsible for making their own access control decisions on requested permissions. Users are usually warned of requested permissions and provided with brief descriptions on what they mean.

Existing browser permission models suffer from limitations when it comes to protecting user privacy against 3rd party extensions. Limitations mainly involve insufficient access control techniques, and limited user awareness. Some browsers provide an Incognito mode that disables 3rd party extensions by default. For example, Google Chrome allows users to enable/disable extensions in this mode, but lacks fine-grain permission customization.

In this work we analyze the Google Chrome permission model for 3rd party extensions and discuss some of its limitations, in addition to some potential threats on user privacy under Chrome. We propose a runtime framework that improves upon the existing Chrome extension permission model. The framework contributes the following:

- Runtime API Monitoring: Chrome extension APIs are monitored in runtime, which increases the user's awareness by informing them of API accesses at the moment they occur.
- Fine-grained Runtime Access Control: The proposed framework gives users the capability to customize extension permissions. Users can deny/allow an individual permission and its associated APIs. Users are also able to prevent APIs of specific permissions from accessing certain webpages they visit. E.g., users can prevent extensions from reading the URL of their banking website, even though the extension was originally granted permission to do so.
- A Chrome extension called "REM" that implements the proposed framework. The extension provides users a simple user interface for monitoring extension accesses, customizing their extension permissions, and getting details on requested permissions as seen in Figure 28, 29, & 30.
- Finally, we conduct a user study that evaluates our Chrome extension "REM" and focuses on measuring REM's effect on user awareness towards extension permissions.

5.1 Related Work

In the last few years several extension vulnerabilities have been discovered, which include stealing cookies, key logging, expose confidential information, and hijack the local

operating system [14, 75, 5, 65]. In a white paper, Freeman et al. [41] investigated the possible security attacks on Firefox extensions.

Bandhakavi et al. [4], proposed applying static information-flow analysis to the JavaScript code used in the third party applications. They described a set of unsafe flow patterns that may lead to security vulnerabilities. This approach provides a mechanism to query the extension code for the defined unsafe flows and does not provide a mechanism to enable the user to monitor application behavior and control its access. Similarly static analysis [42] has been proposed to address security of web applications such as identifying SQL injection [72], and cross-site scripting [43, 69].

Dynamic analysis techniques have also been used to trace information flow properties of JavaScript as it is being executed by the browser [35, 75]. Dhawan et al. [9] proposed a memory tainting approach to trace propagation of tainted objects during JavaScript execution and to raise alerts if an object containing sensitive information is accessed in an unsafe way. These approaches are effective in tracing dynamic program flow, however usually require users to install a modified or recompiled browser or JavaScript engine.

5.2 Chrome Extension Permissions

Third party Chrome extension developers are able to declare permissions needed by their extensions to fulfill certain functionalities, and to access certain Chrome APIs. Such permissions can be declared as required using the `permissions` manifest property. For example, an extension might request access to browser cookies, or a user's browsing history in order to interact with their associated Chrome APIs. The set of such possible permissions are defined by Google within the Chrome extension API documentation. Developers

can also declare permissions as optional, which is ideal for permissions not required immediately by extensions. Additional permissions can also be requested by an extension when updated.

5.2.1 Permissions and Chrome APIs

Once an extension acquires its requested permissions, it can access the Chrome APIs associated with each permission, i.e, certain Chrome APIs require certain permissions to execute successfully. For example, the `chrome.cookies.get` API call requires the `cookies` permission. We look at each requested permission, and find all the reachable API calls an extension can perform, which allows us to precisely monitor all potential extension accesses, as explained in our proposed framework in Section 5.4. The full permission to API mappings were generated by scanning the Chrome extension documentation, specifically the manifest permissions and their associated `chrome` modules. By mapping each permission to a set of associated API calls, we can control and monitor an extension's specific accesses. The exception to this rule is any extension using an NPAPI plugin, which allows for native code execution outside of the context of the Chrome browser. That is, NPAPI accesses do not occur through the Chrome APIs.

5.2.2 User Awareness

Users are warned about some of the permissions that are requested at installation time, and have the option to either continue installing an extension with the requested permissions, or cancel the installation process. Warnings are also shown to users if a certain extension is updated and requests additional permissions, or if an optional permission is being requested. Note that not all permissions trigger a warning message. Such permis-

sions will be granted to an extension without the user's explicit approval. An example of such permissions is `cookies`. We think the rational behind this is that these permissions rely on other requested permissions that do trigger warnings. For example, an extension that requests the `cookies` permission can only access cookies for the hosts it has access to. The list of hosts that can be accessed by a certain extension are listed within its manifest file as part of the permissions attribute, and are shown to the user at install time. The caveat here is that not all users will presume giving access to a certain host could also lead to granting access to its cookies. For example, if a user grants access to `<all_urls>` (all urls), this could potentially mean access to all cookies in the user's browser. Another

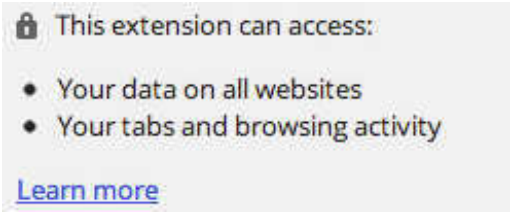


Figure 26: Permission details in the Standard method

issue involves warnings that do not reflect a precise description of what is being granted to an extension. For example, an extension that requests the `history` permission will trigger a warning that says "It can access: Your browsing history", which could potentially be misinterpreted as the list of all URLs a user has visited. But the matter of fact is that the `history` permission also provides an extension with information regarding a user's browsing behavior, e.g. how the user reached a certain website (by typing the url, clicking a link, via a bookmark, etc.), the time they visited a website, and the number of visits too. Such information can be valuable to third parties and could potentially be used for undesired purposes from a user's point of view. In our proposed framework, we provide

users with detailed information and feedback on the permissions and accesses granted to an extension as seen in Figure 28 and 30. Currently, the “Standard” method for discovering an extension’s permissions is to visit it’s page on the Chrome Web Store and looking at the details tab as seen in Figure 26. From there, users have the option to discover more about the permissions requested by visiting yet another webpage. In Section 5.6 we show that our proposed extension REM performs better in increasing user awareness and understanding of an extension’s permissions.

5.2.3 Permission Dependency

Extension permissions sometimes rely on other permissions, i.e. it is not sufficient for an extension to request one permission without the other. Hence, certain functionalities within an extension will require a chain of permissions to execute successfully. A

Chrome API	**Direct**	**Indirect**

`cookies.get`
`cookies.remove`
`cookies.set`
`cookies.getAll`
`cookies.getAllCookieStores`
`cookies.onChanged`
	`cookies`	`host`

`tabs.captureVisibleTab` `tabs.executeScript`	`tabs`	`host`

Table 1: API Direct and Indirect Permissions

popular permission requested by extensions is the `host` permission, which is declared within the manifest as a match pattern. The pattern dictates the hosts that are accessible by extensions. Example patterns include: `http://*/*` (all hosts using the http scheme), `http://example.com/foo.html` which matches that specific url, and `<all_urls>` which matches all urls. The importance of the `host` permission emerges when extensions

use other permissions such as the `cookies` or `tabs`. For example, an extension may request `cookies` permission and assume it can read all cookies using the `cookies.getAll` API. This isn't true, unless the extension requested a `host` permission that covers all URLs associated with the desired cookies. Figure 1 shows an example set of APIs and the various permissions required to use them. Two types of permissions are shown, direct and indirect. Direct permissions are immediately associated with the API method, whereas the indirect ones are additional required permissions. By understanding these dependencies our proposed framework can better monitor and control the specific accesses made by extensions.

5.3 User Privacy and Threats

Users have widely adopted browser extensions and have become acclimated to using them on a regular basis. With this wide spread of extensions, especially ones developed by third parties, the threats to user privacy have increased [14, 75, 5, 65]. The permission model adopted by Google Chrome does provide some means for controlling the permissions given to extensions, but there are still areas that can be improved to provide for better privacy and protection against potential threats.

5.3.1 Threats

Extensions with excessive permissions represent a higher threat to user privacy, especially those that are poorly written and include security vulnerabilities. Excessive permissions are those that are deemed inappropriate or unnecessary in certain privacy related scenarios. For example, granting a `host` permission of `<all_urls>` to a Twitter client extension could be deemed excessive, as it most likely would only require access to `http://*.twitter.com/*`. In the following, we discuss some potential threats when

extensions gain excessive Chrome permissions.

Host Permissions: The `host` permission is a popular permission requested by third party extensions and is declared as a match pattern within the extension's manifest. The match pattern represents the webpages extensions would like to access, which could range from a specific webpage (by specifying a specific URL) to all webpages with a schema of `http`, `https`, `file`, or `ftp` (Using the `<all_urls>`). Figure 2 shows the requested host permission patterns requested by the top hundred rated extensions on the Chrome Web Store. The most popular patterns requested where the `http://*/*` and `https://*/*` patterns. Note that the occurrences of match patterns do not sum up to 100, that is because extensions can declare multiple patterns. Extensions with excessive host permissions could

Host Pattern	**Occurrences (100)**
`<all_urls>`	5
`*://*/*`	4
`https://*/*`	38
`http://*/*`	46
Wild Card Subdomain	18
Specific Host	12

Table 2: Host permission patterns requested by the top 100 rated extensions

potentially succeed in performing attacks on user privacy, especially when combined with other permissions such as the `tabs` or `cookies` permission. With the `tabs` permission, extensions are able to programmatically execute their own custom JavaScript using the `chrome.tabs.executeScript` API. Such scripts are allowed to run on webpages that satisfy the extension's `host` permission. Hence, with an excessive `host` permission, custom scripts are executed on a wider range of webpages. The threats on user privacy arise when custom scripts are vulnerable to attacks such as Cross Site Scripting, that is, a

script could potentially execute malicious code embedded within webpages visited by the user. Such a scenario would allow the malicious code to perform with the privileges of the compromised extension. For example, malicious code could access all cookies accessible to a compromised extension that has `cookies` permission. Limiting the `host` permission to a smaller subset of webpages would decrease the attack surface.

The `cookies` permission combined with excessive `host` permissions could also introduce threats to user privacy. Access to cookies is based on the `host` permission an extension has, that is, access is allowed to any cookie that belongs to a host within the match pattern declared by the `host` permission. Hence, a match pattern of `<all_urls>` potentially means access to all user cookies. Extensions could abuse their `host` permission and access user cookies for malicious reasons such as hijacking a user's online session. Another threat scenario involves vulnerable extensions that have the `cookie` permission. Such extensions, if attacked, could elevate the privileges of malicious code and allow it access to user cookies and other reachable resources.

The dependencies between the `host` and both `tabs` and `cookie` permissions makes it important to monitor and control the specific accesses made by extensions, especially when dealing with excessive `host` permissions such as `http://*/*` or `<all_urls>`. The rationale is that extensions may need different host permissions for different types of accesses. For example, executing a script using the `tabs.executeScript` API may require certain `host` permissions, whereas reading cookies via the `chrome.cookies.get` API may require different ones. Currently, the same `host` permission is used for both purposes, which leads to unnecessary privileges and potentially unwanted accesses.

Tabs Permission: The `tabs` permission gives extensions access to the browser's win-

dows and tabs within each open window. Extensions are able to access `Tab` objects, which contain information on the tab returned such as the associated URL. hence, extensions with `tabs` permission have access to all URLs a user visits. Note that the `tabs` permission is not dependent on the `host` permission with the exception of content script execution, hence, Chrome does not prevent access to tab URLs that are not within the `host` match pattern. With access to all URLs, a malicious extension can directly analyze any URL and its query attributes, and potentially extract important information such as session IDs and OAuth request tokens. Such information can be used in compromising the user's privacy [51].

Another drawback of not bounding the `tabs` permission, is that it undermines the `history` permissions defined by Chrome. That is, extensions can generate their own history repository by keeping track of all URLs users visit. Note that the `history` permission provides additional accesses such as the methodology of reaching a certain webpage (e.g. was a URL typed, clicked, etc.), hence we only consider this a partial undermining. We improve upon the `tabs` permission within our proposed framework by allowing users to customize the URLs accessible by APIs associated to the `tabs` permission.

Other Permissions: Other Chrome permissions such as the `history` & `bookmarks` permission could also be used to gain access to URL data, hence potentially executing malicious attacks using extracted session IDs or OAuth request tokens. Such attacks may frequently fail given history and bookmark URLs are potentially old, hence contain outdated information regarding a user's session or request token. Note that both these permissions are not bounded by the `host` permissions. We also improve upon this within our framework.

5.3.2 Intrusiveness

Third party extensions that request excessive permissions can be quite intrusive. This is mainly due to the relatively course-grain nature of Chrome permissions. For example, extensions with the `tabs` permission are able to track all URLs a user visits, which in many cases is undesirable, especially in scenarios where users visit webpages of highly confidential matter, such as financial or health related webpages. The `tabs` permission also gives extensions access to the DOM, which gives it the ability to read and write data within the DOM. Such data may be highly confidential. For example, an extension with `tabs` permission can easily detect if a user has visited *https://online.wellsfargo.com/* and extract the user's balance. With additional permissions, the extension could even pass it back to a remote server. Such scenarios show the importance of giving the user the necessary controls over which webpages certain extensions have access to. Other permissions such as `history` and `bookmarks` could also reveal the browsing behaviors of users. We believe users should have the option to control the accesses associated with these permissions. With the potential threats and lack of sufficient user awareness within the Chrome extension permission model, we propose a runtime framework that monitors and informs users of extension accesses, in addition to providing them the means for controlling and customizing the permissions granted to their installed extensions.

5.4 Proposed Permission Framework

We propose and implement a runtime permission framework that allows for fine grain chrome permission monitoring and access control enforcement. The framework monitors Chrome API calls made by third party extensions and collects the data processed by these

calls. For example, when the API `chrome.windows.getAll` is called, an allocated monitor within our framework collects the information relevant to the returned browser windows, such as the set of all Tabs within each of the browser windows. Given the runtime

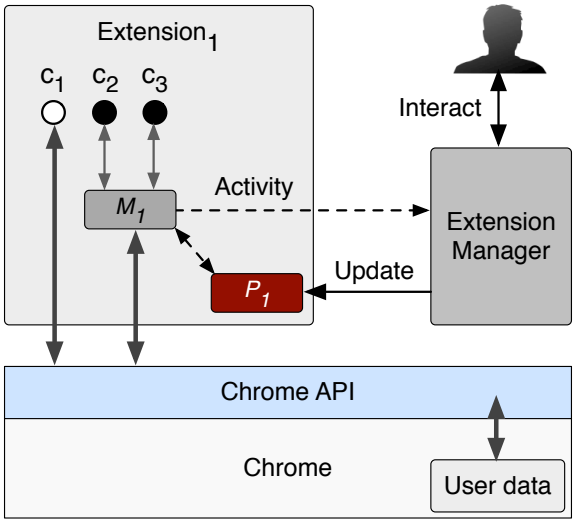


Figure 27: Framework Architecture

nature of the framework, it can inform users in realtime of the specific accesses made by extensions (e.g. which specific URLs or cookies it has accessed), it can also enforce fine-grain access control onto attempted accesses. Additionally, the proposed framework allows for users to customize extension permissions, i.e. grant/deny permissions from the original set requested by an extension. The framework consists of two main components, the extension Manager, and extension Monitor. A single Monitor is allocated for each third party extension installed on a user's Chrome browser, and has its own associated access control Policy. All Monitors report back and are managed by the framework's extension Manager. Figure 27 illustrates the overall architecture of our framework.

5.4.1 Extension Manager

Our extension Manager is the main component within our framework that allows for monitoring third party extensions. The extension manager itself is a Chrome extension with NPAPI capabilities. NPAPI access allows us to adapt the behavior of third party extensions and allow the extension manager to listen to Chrome API calls made by these extensions, in addition to enforcing fine-grain access controls on requested accesses. In the following we discuss the tasks covered by the Manager.

Adapting Third Party Extensions: To monitor API calls made by third party extensions, the manager modifies their default behavior by injecting a proposed Monitor component that reports back to the manager. Figure 27 shows the Monitor M_1 that is assigned to $Extension_1$. This is achieved by including a custom built `monitor.js` script file into the extension's bundle, then linking to it from within the extension's HTML pages such as `background.html` and `popup.html`. When building the Monitor for a specific extension, the manager can selectively choose which API calls the Monitor should monitor. This allows for optimizing the monitoring process and avoiding unnecessary checks. For example, in Figure 27, only API calls $\texttt{c}_2$ and $\texttt{c}_3$ are monitored for $\texttt{Extension}_1$. We explain the details of our Monitor component in Section 5.4.2.

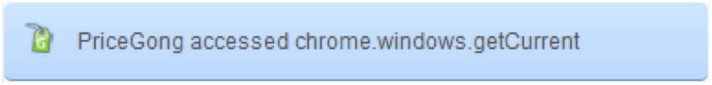


Figure 28: API Call Notification Example

API Notifications and Logging: Once a Monitor is built and injected into an extension's bundle, the Manager starts listening to incoming message calls sent by the Monitor. These

messages hold information on the Chrome API calls made by third party extensions. Using this information, the manager is able to keep users aware of the extension activities by notifying them in real time of the API calls made as seen in Figure 28. The Manager also logs all accesses for future reference and are accessible via the Manager's UI.

Fine-Grain Permission Customization: The Manager allows for users to customize the access control policy for each installed extension. Users are given fine-grain controls over the permissions granted to extensions and are provided with a simple user interface to do so as seen in Figure 29. There are mainly two types of permission controls provided:

1. Permission-based: These controls allow users to deny or allow a certain permission as a whole. Doing so prevents any API associated with the permission from executing. For example, users can choose to deny the permission `cookies` for an extension which will block all cookie associated Chrome APIs from executing.

2. Host-based: These controls allow/deny extensions from accessing certain hosts via the APIs of a certain permission. That is, we keep track of a *permission-to -host* dictionary that has all the hosts blocked for each permission of an extension. For example, a user could prevent an extension with `tabs` permission from accessing a Tab that is associated with a certain host such as `online.wellsfargo.com`. We provide host-based controls for the `tabs`, `cookies`,`history`, and `bookmarks` permissions. Host-based controls allow for decreasing the effect of excessive `host` permissions and the potential threats discussed in Section 5.3.

Users are also given the option to fully enable/disable certain extensions.

Extension Policy: Each third party extension is allocated a `policy.js` file which rep-

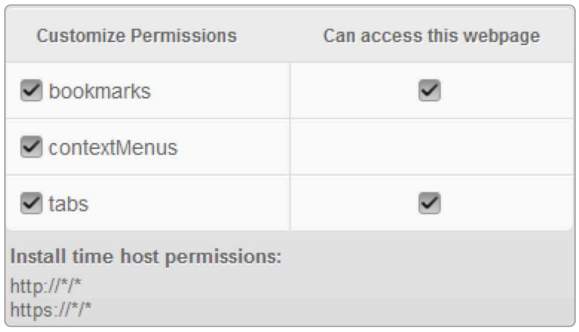


Figure 29: Permission Customization

resents its access control Policy. The policy contains the fine-grain decisions made by users via the *Permission-based* and *Host-based* controls. That is, it contains a set of denied Chrome permissions in addition to a set of denied *permission-to-host* values. This Policy is used by an extension's Monitor to make the proper access control decisions whenever a certain API call is detected. Any customizations made by the user are immediately registered by the Manager and written into the extension's Policy. Note that the Policy represents a negative access control list (ACL^{-}), hence if a Chrome permission or *permission-to-host* value does not exist within the Policy, it is considered allowed, otherwise it is denied. Also note that the `policy.js` is embedded within an extension's bundle. Figure 27 shows the Policy P_1 that is assigned to $Extension_1$.

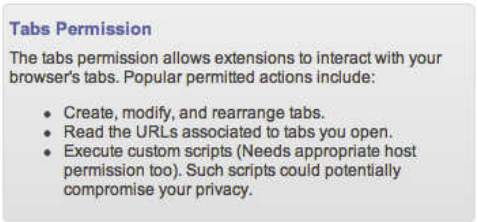


Figure 30: `tabs` permission details

Permission Details: The Manager finally provides users with a detailed description on each of the requested permissions. The detailed description for a specific permission also

contains a set of examples on popular accesses that map to the Chrome APIs associated to a permission. We manually prepared the descriptions and examples. We evaluate the effectiveness of these detailed descriptions in our user study as explained in Section 5.6. Figure 30 shows the detailed description for the `tabs` permission.

5.4.2 Extension Monitor

An extension Monitor is a custom built JavaScript file (`monitor.js`) that we use to monitor the activities of third party extensions. When a Monitor is created for a specific extension, it is assigned a set of API methods to monitor. These APIs are assigned by our extension Manager to suit the permissions requested by extensions. For example, if an extension requests the `cookies` permission, the monitor would be asked to monitor the corresponding cookie API methods:`chrome.cookies.`[`getAllCookieStores, get, getAll, remove, set, onChanged`]. Note that the Manager could select a subset of these APIs, but we monitor all associated APIs by default.

The Monitor is also assigned a Policy (`policy.js`) which it uses in making access control decisions on the API calls it detects. When relevant API calls are detected by a Monitor, the following steps occur:

1. The Monitor intercepts the API call, i.e. the execution of the API runs through the Monitor. It then informs the Manager of this call.
2. An access control decision is made on the API call. This is decided based on two factors. First, the Chrome permission the API is associated to. If this permission is in the ACL^{-} of the Monitor's Policy, the decision is rendered as Deny. Second, the host used within the API (if applicable). If a *permission-to-host* value is found for the

associated permission of the API, the decision is rendered as Deny. If either factors render a decision of Deny, then the final decision is Deny, otherwise it is Allow.

3. If the Policy decision retrieved is Allow, the Monitor executes the API call and returns the relevant results. Otherwise, if the decision is Deny, then the API is blocked and if appropriate returns an empty result (Some extension required an empty result to not break).

Figure 31 illustrates the previous access control process flow. Note that in cases of APIs that

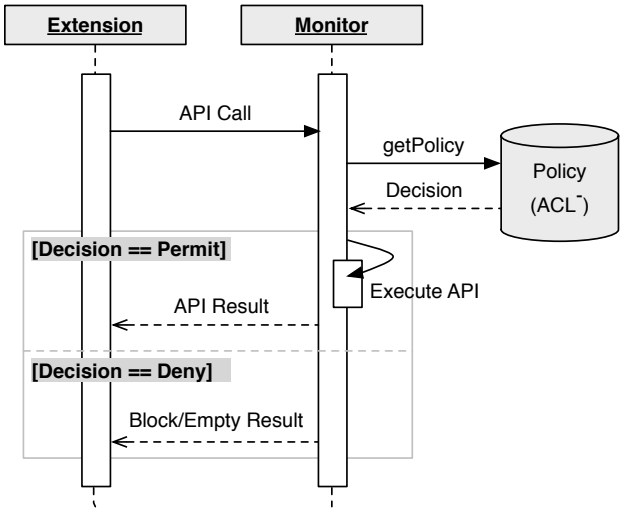


Figure 31: Access Control Sequence

do not specify a specific host value such as `chrome.windows.getAll`, the Monitor will filter the return values to not include any results associated with a *permission-to-host* value. For example, if the user has denied the host `online.wellsfargo.com` and the `chrome.windows.getAll` API results includes Tab objects associated with this host, then the results returned will exclude these Tabs.

5.5 Evaluation

The framework was evaluated on a Windows 7 machine with a 2.4GHz i3 CPU, 4GB of RAM, and was running the Chrome browser version *16.0.912.75*. In our evaluation, we studied the 100 "top rated" Chrome extensions as listed on the official Chrome Web Store at the time of evaluation. The extensions covered all categories on the Chrome Web Store.

5.5.1 Implementation

To evaluate the proposed framework, we implemented the framework as a Chrome extension with NPAPI capabilities and used the FireBreath NPAPI Framework [16] to develop the `dll` plugin used for the extension. For parsing the `manifest.json` files of each extension, we used the Cajun JSON parser. Our Monitor component of the framework was implemented using the FunMon2.js function monitor [62], which allowed us to monitor API calls from within an extension's `monitor.js` file. We used Chrome's message passing APIs to establish the connections between the Manager and Monitor components, specifically the `onRequestExternal. addListener` and `sendRequest` APIs.

When users install our implemented extension, they are required to restart their browser to initiate the adaptation process on their installed extensions. At this point, the framework starts the monitoring process and access control enforcement. The main user interface was implemented via the extension's browser action and its `popup.html`. The browser action button shows the user the number of recent API notifications. The `popup.html` will display the recent notifications and the permission customization controls as seen in Figure 28 and 29. Users can also see a detailed activity log when clicking the Activity button of an extension, and can choose to enable/disable the extension. Finally, `popup.html` shows

users the list of originally requested host permissions.

5.5.2 Permission Requests

In Section 5.3, we discussed the `host` permissions requested by the the evaluated extensions as seen in Table 2. Table 3 shows the list of permissions (excluding `host`) requested by the evaluated extensions and the frequency of each. It also shows the permissions supported by our framework. We notice that the `tabs` permission is the most popular followed by the `contextMenus` and `cookies` permissions.

Permission	Frequency (100)	Supported
`tabs`	77	YES
`contextMenus`	22	YES
`cookies`	11	YES
`notifications`	10	NO
`unlimitedStorage`	9	NO
`bookmarks`	6	YES
`plugin`	4	NO
`management`	4	YES
`idle`	4	YES
`geolocation`	2	NO
`history`	2	YES
`proxy`	1	YES
`clipboardWrite`	1	YES

Table 3: Frequency of Requested Permissions

From the 100 extensions, we analyzed the combinations of `tabs` and `host` permissions requested. As discussed in Section 5.3, with both these permissions, extensions could represent a potential threat on user privacy. We found that 6.5% of extensions with the `tabs` permission have requested a `<all_urls>` `host` permission, 5% with `*://*/*`, 49% with `https://*/*`, and 60% with `http://*/*`. Whereas, 12% have either requested a specific host or ones with wild card subdomains. We also found that 11% have no `host` permissions. Note that the percentages do not add up to 100% because of extensions that

use multiple host match patterns.

5.5.3 Real World Evaluation

Using our proposed runtime framework, we were able to successfully monitor and enforce our fine-grain permission controls in real time. This included all APIs for supported permissions within our framework. We discuss unsupported permissions in Section 5.5.3.2. To evaluate our framework on real world extensions, we installed and used the 100 top rated extensions with our framework in place. We manually analyzed the JavaScript code of each extension to make sure our usage covered all execution paths. The framework was successful in monitoring the APIs excluding those of unsupported permissions. Note that the framework was also capable of supporting event listeners, which was achieved via monitoring the callback functions of events.

5.5.3.1 Performance Evaluation

As a runtime framework it was important to measure the monitoring overhead introduced when adapting third party extensions to our framework. We measure the time to execute some of the popular APIs with and without our framework in place. These APIs were popular amongst the evaluated extensions. Table 4 shows the results of the evaluation when our framework is disabled and enabled. We believe the overhead is acceptable for most chrome extension functionalities, and will not interfere with the usability of extensions.

5.5.3.2 Coverage and Limitations

The proposed framework was able to successfully monitor and enforce fine-grain access controls onto 87% of the evaluated extensions. It failed in cases where extensions had

API	Disabled	Enabled (ms)
`tabs.onUpdated.addListener`	0.45 ms	2.7 ms
`tabs.sendRequest`	2.2 ms	6 ms
`cookies.set`	2 ms	4.5 ms
`cookies.get`	1 ms	4 ms

Table 4: Framework monitoring overhead for popular API calls

unusual manifest files, that is, not strictly following the traditional manifest guidelines. It also failed in cases where JavaScript errors occurred within an extension's code. As part of our future work, we will further enhance the framework's compatibility and error handling in such non-traditional cases.

The framework is also limited to which Chrome permissions it can monitor and control. These permissions are mostly related to APIs that run outside the context of Chrome, e.g. `plugin` APIs, and HTML5 APIs. The supported include: *background*, *contentSettings*, *experimental*, *fileBrowserHandler*, *geolocation*, *notifications*, and *unlimitedStorage*. Note that our framework only enforces the *Host-based* controls on the following permissions: *tabs*, *bookmarks*, *cookies*, and *history*. These are the permissions we believe are most relevant to webpages a user visits. As part of our future work, we will further investigate support for additional permissions.

5.6 User Study

To evaluate our proposed browser extension we conducted a user study that compares the Standard permission discovery method (By visiting an extension's detail page on the Chrome Web Store) with our own browser extension REM. Participants in the study performed a number of tasks related to third party Chrome extensions and answered a number of questions on these tasks. The study was approved by UNC-Charlotte IRB (Protocol

#12-02-50).

5.6.1 Methodology

The study participants were recruited from UNC-Charlotte and were all UNC-Charlotte students. Each participant was supplied with a $10 Amazon gift card. We recruited a total of 20 participants to start the study, of which 18 successfully completed the study and 2 dropped out. Of the 18 participants, 11 were females and 7 were males. 88.2% of the participants are at least familiar with Chrome extensions. Participants where given a brief introduction to REM's and to the existing Standard methods, and were also given a few minutes to familiarize themselves with both techniques. We then performed a within-subjects study comparison in which participants use either the Standard method or REM for performing the study tasks at first, then use the other method for performing the same tasks once again. Assigning a method (REM or Standard) to users was random, and the order of the methods assigned was counter balanced.

5.6.1.1 Study Tasks

Participants were given 8 different tasks and were asked to determine whether performing a certain action was permitted by a third party Chrome extension. For these tasks, participants could answer with: Yes, No, or Uncertain. Note that for each task a participant had to answer in regards to four different third party extension.The tasks were categorized into Social Networking related tasks and Online Shopping related ones. For each category participants performed 4 different tasks. Examples of such tasks are illustrated in Figure 32.

Category	**Task**
Social Networking	Do the installed browser extensions have permission to read your private posts on social sites you visit?
Online Shopping	Do the installed browser extensions have permission to read your history of visited product pages?

Figure 32: Example Tasks

5.6.1.2 Study Results

To evaluate the performance of participants on tasks, we considered two measures: 1)Response correctness, and 2)The time to finish a task measured in seconds. In Figure 33 we summarize the different time intervals for finishing correctly answered tasks. Notice that we consider only the correctly answered tasks as we are interested in the time it takes to correctly determine permitted actions among third party Chrome extensions.

One can notice an overall higher accuracy rate when using REM, in addition to an overall lower time-to-task intervals. For example, participants were able to answer 30 tasks correctly within a time interval of 0-25 seconds using REM, whereas with the Standard method they were able to answer 12. Surprisingly, even when REM was the first tool option used by participants, it was still able to perform relatively better than the Standard method.

To measure the significance of these results, we performed a t-test on the accuracy rate of participants. In Figure 34 we report the mean accuracy with standard deviation for all 8 tasks when using the Standard method vs. REM. Note that the accuracy rate was significant in tasks Social_1, Social_4, Shopping_1, and Shopping_4 with a p-value $p < 0.05$.

In a post survey, participants were asked to assess our proposed browser extension REM and the Standard method using three Likert scale questions. Participants responded to each of the following statements on a scale from one (strongly disagree) to seven (strongly

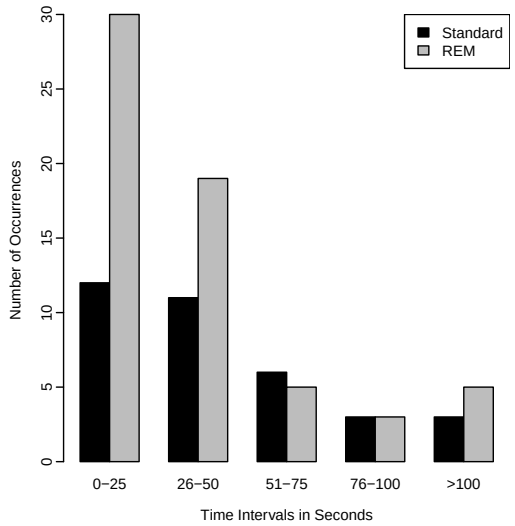


Figure 33: Time distributions for correctly answered tasks

Task	**Standard** (μ, σ)	**REM** (μ, σ)	**p-value**
Social_1	(0.0, 0.0)	(0.294, 0.469)	**0.01003**
Social_2	(0.47, 0.51)	(0.64, 0.49)	0.13469
Social_3	(0.70, 0.469)	(0.70, 0.469)	0.5
Social_4	(0.11, 0.33)	(0.41, 0.50)	**0.02787**
Shopping_1	(0.0, 0.0)	(0.41, 0.50)	**0.002048**
Shopping_2	(0.235, 0.437)	(0.352, 0.492)	0.16609
Shopping_3	(0.176, 0.392)	(0.235, 0.437)	0.33417
Shopping_4	(0.235, 0.437)	(0.58, 0.50)	**0.014459**

Figure 34: T-test Task Accuracy.

agree).

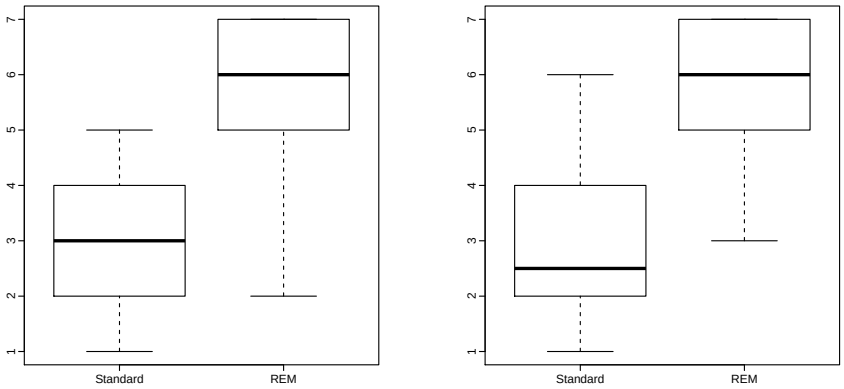
S1: I am satisfied with the tool

S2: I was able to easily identify the permissions requested by each third party Chrome extension.

S3: I was confident in determining the permitted actions for installed third part Chrome extensions.

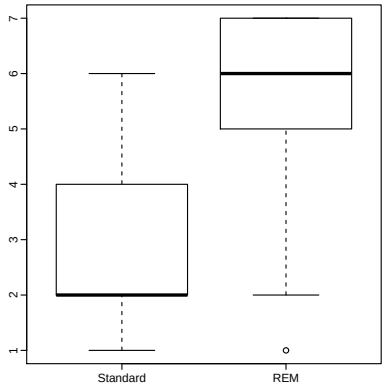
Figure 35 illustrates the user responses using boxplots. The black band in the middle of a box indicates the median. From the responses we observed that REM was rated signifi-

cantly higher ($p < 0.05$) for all three statements.



(a) Q1: Overall Satisfaction

(b) Q2: Identifying Permissions



(c) Q3: Confidence

Figure 35: Summary of Likert-Scale user responses

CHAPTER 6: VISUALIZED-BASED AND ASSISTED POLICY ANALYSIS

Performing SELinux policy analyses can be difficult, due to the complexity of the policy language and the sheer number of policy rules and attributes involved. For example, the default policy on most SELinux-enabled systems, has over 1,500,000 flat rules, involving over 1,780 *types*. Simple analyses between *types* can result in a large amount of data, which is poorly presented to administrators in existing analysis tools. Furthermore, administrators are required to add new policy rules on a regular basis, which can potentially compromise the security of a system, if the consequences of adding such rules are nfot well-understood. We propose and implement a policy analysis tool "SEGrapher" that addresses the above challenges. SEGrapher visually presents analysis results as a simplified directed graph, where nodes are *types*, and edges are corresponding rules between *types*. Graphs are generated via a proposed clustering algorithm that clusters *types* based on their accesses. Clusters provide an abstraction layer that removes undesired data, and focuses on analysis attributes specified by the administrator. Furthermore, SEGrapher assists administrators in evaluating the risks associated with custom policy modules, based on a proposed similarity approach that analyzes new rules within these modules. Visual cues are also provided to notify administrators of various levels of potential risks.

6.1 Related Work

Well known SELinux policy analysis tools include APOL [67], SLAT [47], PAL [59], and Gokyo [30]. Tresys Technology developed the APOL tool, which is used to analyze SELinux policies. It provides a wide range of features including domain transition analysis, direct and transitive information flow analysis, and type relationship analysis. APOL requires a strong understanding of SELinux policies and the involved attributes, and requires a fair set of skills to perform proper policy analyses. Results in APOL are text-based, and in many cases unmanageable due to large result sets. SLAT (Security Enhanced Linux Analysis Tool) represents a policy as a directed graph, where nodes are security-contexts and edges as the permissions on certain object-classes. The focus of SLAT is on information flow, which can be detected by traversing the policy graph. PAL (Policy Analysis using Logic-Programming) uses a logic-programming approach for analyzing SELinux policies. It follows the same model as SLAT, but provides a more extensive query set to admins. Similar to SLAT, PAL does not provide visualized analysis results, and is not able to discover inherent relations between multiple types, but is rather limited to answering direct queries.Both SLAT and PAL require a strong understanding of SELinux to generate strong queries that result in meaningful results.

Jaeger et al. [30], developed a tool called Gokyo, mainly used for checking the integrity of a proposed trusted computing base (TCB) for SELinux. Integrity checks ensure that no types outside the TCB can write to types within the TCB, and no types inside the TCB can read from those outside of it. Gokyo uses a graphical access control model for representing policies. Gokyo is limited to the proposed TCB and does not provide “on the fly” policy

analysis, nor does it allow admins to interact with the resulting analysis results.

Xu et al. [73], proposed a visualization-based policy analysis framework for analyzing security policies using semantic substrates and adjacency matrices. The framework allows admins to run visualization-based queries on a policy base to find possible policy violations. However, their framework is limited to a small set of queries, and the visualization results can be difficult to interpret and understand.

MITRE [48], developed the Polgen tool, which provides semi-automated policy generation for new applications. It relies on observing an application's system calls, and inferring a new policy. Polgen is well suited for new applications, but could require long observations to generate robust policies. Polgen doesn't utilize existing policy decisions in inferring the new policy. Existing decisions are a valuable source for identifying appropriate new policies.

6.2 SELinux Policy Analysis

Let T be the set off all types within a SELinux policy P, O the set of all object-classes, and A the set of all permissions. We propose a policy analysis tool "SEGrapher" which allows for visualizing policy analysis results, by modeling a policy as a directed graph. Given a policy P, SEGrapher builds a directed graph G_p, where a node in G_p maps to a specific SELinux type, and an edge (out-edge) maps to the set of all AV allow rules R_{ij} connecting a type t_i (subject-type) to a type t_j (object-type). Figure 36 illustrates a simple graph of three types, t_1 (subject-type), t_2 (object-type), and t_3 (object-type). The figure shows the corresponding AV rules R_{12} for t_1 and t_2 with two allow rules, and R_{13} for t_1 and t_3 with one allow rule.

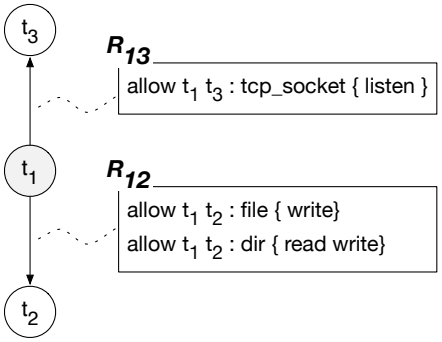


Figure 36: Allow rules for subject-type t_1 and object-types t_2 and t_3.

SEGrapher uses G_p to generate a directed focus-graph G_f representing desired analysis results, that is, G_f will indicate the accesses and relations amongst SELinux types analyzed by admins. G_f is driven by a set of inputs that are checked against AV rules (edges) in G_p. These inputs are controlled and provided by admins and include the following:

1. Focus Types T_f: A set of types $T_f \subseteq T$ which is the focus of the policy analysis and the basis of extracting the focus-graph G_f from G_p. An out-edge in G_p is added to G_f if the source-node (subject-type) of this out-edge exists in T_f.
2. Focus Object-Class o_f: An object-class $o_f \in O$, which is used to filter the out-edges that already satisfy the focus-types T_f. SEGrapher allows admins to ignore checking for o_f, hence o_f will be replaceable by any object-class in O.
3. Focus Permissions A_f: A set of permissions $A_f \subseteq A$, which are used to further filter the out-edges that already satisfy both T_f and the focus object-class o_f. SEGrapher also allows admins to ignore checking for A_f, hence A_f will be equal to A.

With the provided T_f, o_f, and A_f, an out-edge in G_p with an AV allow rule set R_{fn} is added to G_f if for any $r_i \in R_{fn}$ the following conditions are *all* true:

1. The subject-type for r_i exists in T_f.

2. The object-class for r_i = o_f

3. A_f exists within the permissions for r_i.

For example, let $T_f = \{t_1\}$, $o_f = dir$, and $A_f = \{write\}$. When applying these inputs onto the graph in Figure 36, a new focus-graph G_f is generated as illustrated in Figure 37. Note that, only out-edges with AV rule sets fulfilling the above conditions make it to G_f.

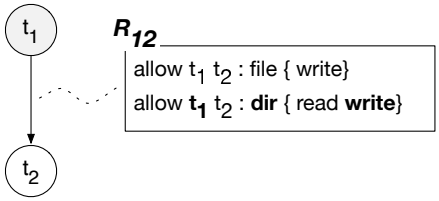


Figure 37: Filtered allow rules for t_1 and object-type t_2 as an edge in G_f

With 1,517,130 AV allow rules, 1,785 types, 47 object-classes, and 167 different permissions, the full SELinux reference policy graph is infeasible to analyze at once. Even when applying the analysis inputs T_f, o_f, and A_f, a resulting focus-graph G_f can be difficult to analyze. In many cases, simply analyzing a single focus-type can result in a large number of AV allow rules, hence a dense focus-graph G_f. For example, to analyze the *read* accesses of the Samba Server [58] on directories within an SELinux-enabled Linux system, let $T_f = \{smbd\_t\}$, $o_f = \{dir\}$, and $A_f = \{read\}$ where `smbd_t` is the subject-type (domain) corresponding to the Samba Server. This analysis results in 1,048 AV allow rules, hence a dense G_f of 1,048 edges and 1,049 nodes. If we add a second type `ftpd_t` (FTP Server) to T_f, and run a new analysis, we'll find that the number of edges in G_f almost doubles to 2,095, leading to a very dense graph, whereas the number of nodes increases just to 1,052. This is due to the fact that both `smbd_t` and `ftpd_t` have a large overlap in the object-types they access, i.e. their out-edges share a large set of end nodes within G_f.

Observation 1. *Many subject-types in SELinux have a large overlap of object-types that they access. In some cases they access the exact set of object-types, and in other cases there is a hierarchical relation between the sets accessed.*

Based on Observation 1, we define the following terms and relations between types t_i and t_j in T_f:

Definition 8. *(Object-Type Set) The object-type set $To_i \subseteq T$ for type t_i is the set of object-types in all AV allow rules, where an AV rule's subject-type is t_i. That is, the set of all types that t_i can access.*

Definition 9. *(Matching Types) Types t_i and t_j are* matching *if their respective object-type sets T_{o_i} and T_{o_j} are equal. Formally,*

$$t_i \Re_m t_j \iff (T_{o_i} = T_{o_j})$$

Definition 10. *(Hierarchical, Parent-Child Types) A* parent-child *relation between types t_i (parent) and t_j (child) exists when t_i's object-type set T_{o_i} is a proper superset of t_j's object-type set T_{o_j}. Formally,*

$$t_i \Re_h t_j \iff (T_{o_i} \supset T_{o_j})$$

Definition 11. *(Overlapping Types) Types t_i and t_j are* overlapping *if their respective object-type sets T_{o_i} and T_{o_j} overlap and neither $t_i \Re_m t_j$ or $t_i \Re_h t_j$ holds . Formally,*

$$t_i \Re_o t_j \iff (T_{o_i} \cap T_{o_j} \neq \phi) \wedge \overline{t_i \Re_m t_j} \wedge \overline{t_i \Re_h t_j}$$

Definition 12. *(Disjoint Types) Types t_i and t_j are* disjoint *if their respective object-type sets T_{o_i} and T_{o_j} are disjoint. Formally,*

$$t_i \Re_d t_j \iff (T_{o_i} \cap T_{o_j} = \phi)$$

These relations can assist in discovering other interesting relations between types t_i and t_j in T_f.

Note that our focus is not on one-to-one type relations, i.e. can $t_i \in T_f$ access $t_j \in T_f$, but on more interesting relations that exist between t_i and t_j which can eventually lead to simpler policy configurations and an easier analysis process. One-to-one relations between t_i and t_j can still easily be identified from the relations above. In SEGrapher we uniquely visualize the focus-types T_f, this makes it easy to identify them and to identify any one-to-one relations that may exist between them.

Based on the defined relations $\Re_m$,$\Re_h$,$\Re_o$, and $\Re_d$, and our higher goal of discovering new relations, we propose a clustering algorithm in section 6.2.1 that utilizes and exposes existing relations between types in T_f. By exposing these relations and building a cluster-based focus-graph reflecting these relations, the algorithm is able to visually simplify focus-graphs, hence simplify the policy analysis process.

6.2.1 Type Clustering

We propose and implement a clustering algorithm that utilizes the relations $\Re_m$,$\Re_h$,$\Re_o$, and $\Re_d$ identified above. Given focus-types T_f, object-class o_f, permissions A_f, and an edge-reduction threshold τ_e, we extract existing relations from a policy graph G_p and generate a set of clusters C where each cluster $C_i \in C$ becomes a node within a new cluster-based focus-graph G_f.

The process of generating G_f is detailed in Algorithm 1. The algorithm starts by initializing a set of cluster nodes from the object-type sets of the focus-types T_f. Lines 3 and 4 create a new cluster node C_c for each of the focus-types $t_f \in T_f$, and a new edge between

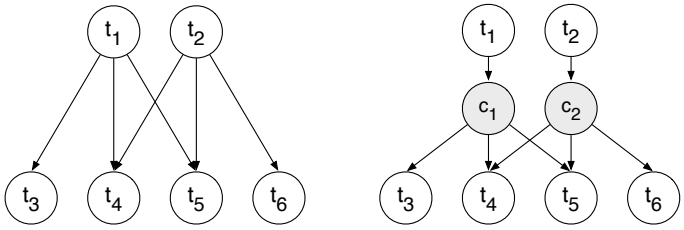
Algorithm 1: Generate Clustered Policy Focus-Graph

input : Policy graph G_p, focus-types T_f, object-class o_f, permissions A_f, and threshold τ_e
output: Clustered Focus-Graph G_f

```
1  Initialization: C ← {} ; // Candidate Cluster Nodes
2  foreach t_f ∈ T_f do
3      create new cluster node C_c;
4      add edge e(t_f, C_c) to G_f;
5      foreach node t_n ∈ OutNodes(t_f, G_p) do
6          R_fn = AV allow rule for edge e(t_f, t_n) in G_p;
7          if R_fn satisfies o_f and A_f then
8              add edge e(C_c, t_n) to G_f;
9      insert C_c into C;
10 while optimization possible do
11     for i ← 0 to size(C) do
12         for j ← 0 to size(C) do
13             outnodes_i = OutNodes(C_i, G_f);
14             outnodes_j = OutNodes(C_j, G_f);
15             if outnodes_i = outnodes_j then
16                 MergeMatching(C_i, C_j, G_f);
17             else if outnodes_i ⊂ outnodes_j then
18                 MergeSuperset(C_i, C_j, τ_e, G_f);
19             else if outnodes_j ⊂ outnodes_i then
20                 MergeSuperset(C_j, C_i, τ_e, G_f);
21             else if outnodes_i ∩ outnodes_j ≠ φ then
22                 MergeOverlap(C_i, C_j, τ_e, G_f);
```

t_f and C_c is added to G_f. On lines 6 and 7, the AV allow rule corresponding to each edge between t_f and its out-nodes in G_p is evaluated against the given o_f and A_f. If o_f is the same as the AV rule's object-class, and A_f is within the AV rule's permissions, then a new edge from the new cluster C_c and the out-node is created in G_f. Each new cluster is then stored into C, at line 9. Figure 38 shows an example of the initialization process (assuming all AV rules are satisfy o_f and A_f).

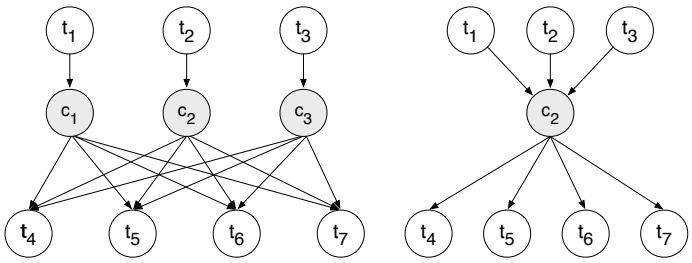
Lines 11 to 22 of Algorithm 1, involve discovering potential relations between pairs of



(a) Before Initialization (b) After Initialization

Figure 38: Initialization of new node clusters for focus-types t_1 and t_2

focus-types, where each focus-type is represented by its corresponding cluster from the initialization phase, that is, each cluster represents a type's object-type set. At line 15 of Algorithm 1, it checks if the relation $\Re_m$ holds. In this scenario, Algorithm 2 is used to merge the object-type sets into one set. Figure 39 illustrates this process. Note that the number of both clusters and edges decreases, hence simplifying the resulting G_f.



(a) Before Optimization (b) After Optimization

Figure 39: Clusters with matching object-types

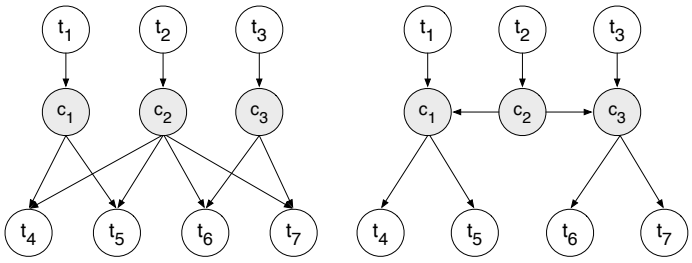
Algorithm 2: MergeMatching

input: Cluster Nodes C_1 & C_2. Focus-Graph G_f

1 **foreach** *edge* $e(t, C_1) \in G_f$ **do**
2 add edge $e(t, C_2)$ to G_f;
3 remove edge $e(t, C_1)$ from G_f;
4 remove C_1 from C

At lines 17 and 19 of Algorithm 1, it checks if the relation $\Re_h$ holds. In this scenario,

Algorithm 3 is used to establish a parent-child relationship within G_f. This is achieved by removing the out-edges of a parent cluster that point to the object-type set of the child cluster, then pointing the parent cluster to the child cluster. Figure 40 illustrates this process. Note that the edge-reduction threshold τ_e is passed to Algorithm 3, which allows it to measure the feasibility of establishing the parent-child relation. That is, before Algorithm 3 makes any changes to G_f, it checks if the resulting reduction in edge numbers is greater than τ_e. The edge reduction for an $\Re_h$ relation is equal to (the number of out-edges of the child cluster – 1).



(a) Before Optimization (b) After Optimization

Figure 40: Clusters with superset object-types (parent-child)

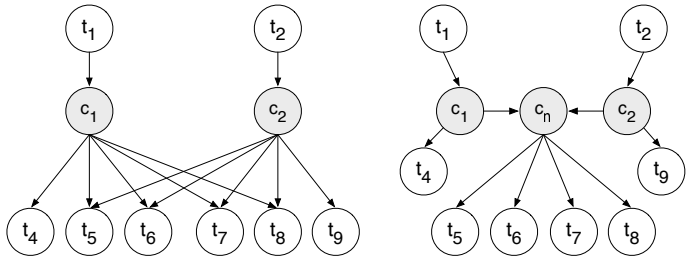
Algorithm 3: MergeSuperset

input: Cluster Nodes C_1 & C_2. Threshold τ_e, and Focus-Graph G_f

1 **if** C_1 *and* C_2 *satisfy* τ_e **then**
2 $T_o \leftarrow$ `OutNodes` (C_1, G_f);
3 **foreach** *node* $t_n \in T_o$ **do**
4 remove edge $e(C_2, t_n)$ from G_f;
5 add edge $e(C_2, C_1)$ to G_f;

At line 21 of Algorithm 1, it checks if the relation $\Re_o$ holds. In this case, Algorithm 4 is used to extract the overlapping object-types, and creates a new cluster that points to the overlap. Figure 42 illustrates this scenario. The edge-reduction threshold τ_e is passed to

Algorithm 4, which allows it to measure the feasibility of establishing the $\Re_o$ relation. That is, before Algorithm 4 makes any changes to G_f, it checks if the resulting reduction in edge numbers is greater than τ_e. The edge reduction for an $\Re_o$ relation is equal to (the number of overlapping out-edges – 2). Also note that for this scenario, the number of clusters increases by 1. In our implementation, we find that the increase of clusters for a reasonable edge-reduction τ_e, is effective from a visualization point of view.



(a) Before Optimization (b) After Optimization

Figure 41: Clusters with overlapping object-types

Algorithm 4: MergeOverlap

input: Candidate Clusters C_1, C_2, and threshold τ_e

if C_1 *and* C_2 *satisfy* τ_e **then**
  create new cluster node C_o;
  $T_o \leftarrow$ `OutNodes` $(C_1, G_f) \cap$ `OutNodes` (C_2, G_f);
  foreach *node* $t_n \in T_o$ **do**
    add edge $e(C_o, t_n)$ to G_f;
    remove edge $e(C_1, t_n)$ from G_f;
    remove edge $e(C_2, t_n)$ from G_f;
  add edge $e(C_1, C_o)$ to G_f;
  add edge $e(C_2, C_o)$ to G_f;

Algorithm 1 continues to run until no more feasible relations are discoverable.

The results from applying Algorithm 1 are effective in both discovering interesting relations between focus-types, and in simplifying the visualization of analysis results.

6.3 Assisted Policy Analysis

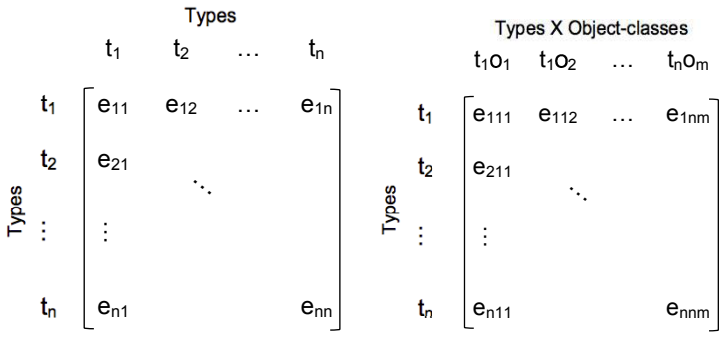
Providing clustered visual analysis results to admins allows for a clearer understanding of existing type relations, and provides a layer of abstraction that isolates unnecessary analysis data, hence focusing on types that matter the most. But, admins do not only deal with existing policy rules, but are also required to add new rules for certain types on a regular basis. This occurs when a certain service Adding new rules can potentially involve risks of compromising the security of a system, either by adding rules that are too permissive, or by adding rules that are completely unnecessary [54, 74]. It is important to provide a mechanism that analyzes the risks associated with introducing new policy rules.

We propose an assisted analysis mechanism that is part of SEGrapher. The proposed mechanism provides analysis data on newly introduced rules, which can guide the admins in making better policy management decisions.

6.3.1 Similarity-Based Model

Let R_{new} be the set of new AV allow rules for an existing type t_i, and R_{old} the set of all existing AV rules for all existing types T in the policy P. Our approach determines a set of similar types $T_s \in T$, based on the existing rules for t_i. To measure the similarity between types, we construct a *feature-vector* for each type $t_i \in T$, based on its accesses in P. To capture the accesses of each type in T, we generate two types of access-matrices, each capturing different levels of granularity.

1. M_T: A matrix that captures the accesses between each pair of types $(t_i, t_j) \in T \times T$, regardless of the object-classes or permissions involved. An entry $e_{ij} \in M_T$ indicates whether t_i is allowed ($e_{ij} = 1$) or denied ($e_{ij} = 0$) access to t_j. That is, if there exists



(a) M_T matrix (b) M_{TO} matrix

Figure 42: Feature Vectors

a rule $r_i \in R_{old}$ that allows t_i access to t_j, then $e_{ij} = 1$, regardless of the object-class or permissions for r_i. Figure 42(a) illustrates the M_T access-matrix for P.

2. M_{TO}: A matrix that captures the accesses between a type $t_i \in T$ and pairs $(t_j, o_k) \in T \times O$, where O is the set of all object-classes in P. These accesses are regardless of the permissions involved. An entry $e_{ijk} \in M_{TO}$ indicates whether t_i is allowed ($e_{ijk} = 1$) or denied ($e_{ijk} = 0$) access to t_j given the object-class o_k. That is, if there exists a rule $r_i \in R_{old}$ that allows t_i access to t_j, and its object-class is equal to o_k, then $e_{ijk} = 1$, regardless of the permissions for r_i. Figure 42(b) illustrates the M_{TO} access-matrix for P.

Based on the access-matrices M_T, and M_{TO}, we define the feature-vectors V_T, and V_{TO} respectively. Each feature-vector on a type t_i is represented by the ith row of its respective access-matrix. For example, the type t_1 has two feature-vectors: V_{T_1} ($1st$ row of M_T), and V_{TO_1} ($1st$ row of M_{TO}).

The similarity between any two types t_i and t_j is based on how similar their feature-vectors are, e.g. types that have identical accesses within P, will have identical feature-

vectors. To calculate the similarity $sim(t_i, t_j)$, we use the Pearson correlation coefficient which is widely used for similarity measures [25, 45]. $sim(t_i, t_j)$ represents the similarity between the feature-vectors of t_i and t_j. Equation 7 shows the Pearson correlation similarity value between t_i and t_j, using their corresponding V_T feature-vectors. The value of $sim(t_i, t_j)$ is between -1 and 1, where a -1 indicates a reverse correlation, a 1 indicates a perfect correlation, and 0 indicates no correlation. Types with a Pearson correlation coefficient value closest to 1 are the most similar, and are referred to as the *nearest-neighbors*.

$$sim(t_i, t_j) = \frac{\sum_{k=1}^{n}(V_{T_{i_k}} - \overline{V_{T_i}})(V_{T_{j_k}} - \overline{V_{T_j}})}{\sqrt{\sum_{k=1}^{n}(V_{T_{i_k}} - \overline{V_{T_i}})^2}\sqrt{\sum_{k=1}^{n}(V_{T_{j_k}} - \overline{V_{T_j}})^2}} \tag{7}$$

SEGrapher determines the nearest-neighbors for type t_i of the newly introduced rules R_{new}, by applying two stages of filtering:

1. Stage 1: At this stage, SEGrapher calculates the similarity values $sim(t_i, t_j)$ based on feature-vectors of type V_T, which only takes type-to-type accesses into consideration, and disregards object-classes. Once similarity values are calculated, nearest-neighbors are selected such that their $sim(t_i, t_j)$ value is larger than the threshold τ_1, which is set by admins within SEGrapher.
2. Stage 2: Let T_n be the set of nearest-neighbor types resulting from Stage 1. SEGrapher generates a new M_{TO} matrix based only on T_n rather than T. Similarity values $sim(t_i, t_j)$ are then calculated based on the feature-vectors V_{TO} from the new M_{TO}. T_n is then filtered to only contain types with a $sim(t_i, t_j)$ value larger than the threshold τ_2, which is also set by admins within SEGrapher.

6.3.2 Nearest-Neighbor Rule Classification

After applying our similarity model onto the subject-type t_i of the new rules R_{new}, we are able to identify the nearest-neighbors set T_n. Types in T_n have similar accesses to t_i, and are used as a measure of how risky the rules R_{new} are, compared to the nearest-neighbors' rules R_n. That is, rules R_{new} could be considered safe if the admin observes similar rules in R_n. SEGrapher allows admins to easily observe rules in R_n by classifying them into different access-classes based on the object-types T_{new}, object-classes O_{new}, and permissions A_{new} accessed within the rules of R_{new}.

For a rule $r_j \in R_n$, let t_j be its object-type, o_n its object-class, and A_n its permissions. We define the following access-classes on r_j:

- Non-Matching: Rule r_j belongs to this class if $t_j \notin T_{new}$, $o_n \notin O_{new}$, or $A_n \not\subset A_{new}$. This class is considered the strongest notification of potential risks. SEGrapher visually highlights these rules in red color, to grasp the admin's attention.
- Less Permissive: Rule r_j belongs to this class if $t_j \in T_{new}$, $o_n \in O_{new}$, and $A_n \subset A_{new}$. This class is also a strong notification of potential risks, because there are shared accesses, but not as permissive as rules in R_{new}. This class is visually highlighted in orange color.
- Overlapping: Rule r_j belongs to this class if $t_j \in T_{new}$, $o_n \in O_{new}$, and $A_n \cap A_{new} \neq \phi$. This is also considered a strong notification of potential risks, because there are shared accesses, but not the same as rules in R_{new}. This class is visually highlighted in yellow color.
- More Permissive: Rule r_j belongs to this class if $t_j \in T_{new}$, $o_n \in O_{new}$, and $A_{new} \subset$

A_n. Rules in this class are more permissive than R_{new}, which indicates that rules of R_{new} are potentially less risky. These are highlighted in green.

- Matching: Rule r_j belongs to this class if $t_j \in T_{new}$, $o_n \in O_{new}$, and $A_{new} = A_n$. This class reflects rules with identical accesses to those of R_{new}. These are highlighted in green color.

6.4 Design and Implementation

We implement our proposed clustering algorithm and assisted custom policy analysis module in a tool we call "SEGrapher". SEGrapher is based on the Java JDK 1.6, and uses the APIs provided by SETools [68] for parsing SELinux policies. Its graph drawing is based on an extended version of the open source visualization toolkit Prefuse [31].

6.4.1 Visualization and Interactivity

SEGrapher's GUI as shown in Figure 43, contains two main panels. First, the left panel which allows the admin to control the analysis attributes, such as focus-types, object-classes, and permissions. It also has the controls for starting the analysis, and searching for types within resulting focus-graphs. Second, a right panel which shows the resulting focus-graphs of the analysis. The right panel is also where the results of an assisted-policy analysis appears.

6.4.2 Focus-Graphs

The components of a focus-graph are visually differentiated to provide for easier policy analysis.

- Focus-type Nodes: Focus-types are shown as green nodes within the graph. SEGra-

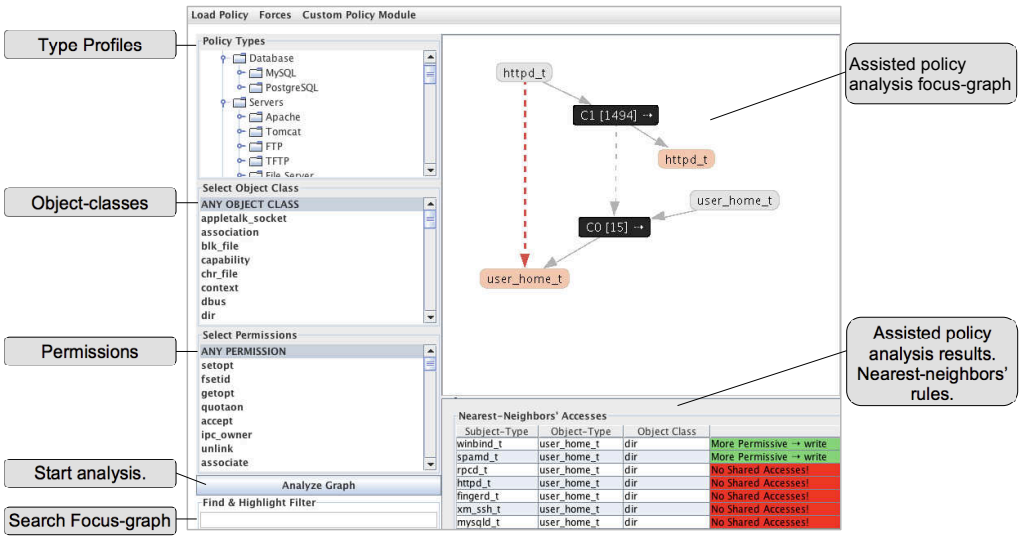


Figure 43: SEGrapher Interface

pher also creates a new version of a focus-type in cases where it also plays the role of an object-type. The reasoning behind this is to provide a simpler focus-graph with less cycles, in cases where focus-types access other focus-types.

- Object-type Nodes: Object-types are shown as orange nodes in the focus-graph. Object-type nodes are hidden by default, as they are not the focus of the analysis. In cases where an object-type is also one of the focus-types, it is by default expanded and visible.
- Cluster Nodes: The proposed clustering approach in Section 6.2.1 results in cluster nodes that become part of the focus-graph. A cluster node is shown in black color, and shows a label which indicates the number of object-type nodes it points to. Admins can also expand/hide object-type nodes for a cluster node, by double-clicking on the cluster node. Figure 44 shows the cluster node `C1_0` with 13 expanded object-type nodes.

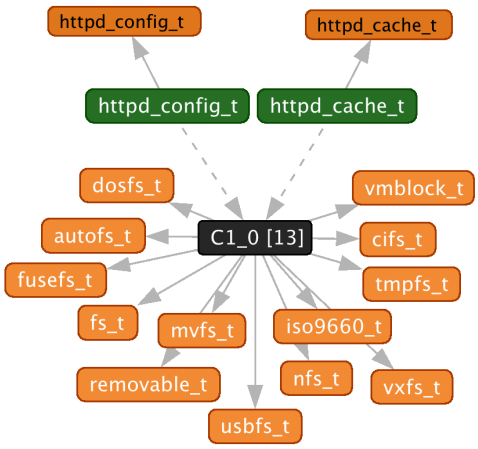


Figure 44: Overlapping relation (httpd_config_t $\Re_o$ httpd_cache_t)

An out-edge from a node n_i to n_j indicates that n_i can access the type n_j (for the specified object-classes and permissions). If n_j is a cluster node, then n_i can access all the object-type nodes for the cluster node n_j, and all object-type nodes for clusters pointed to by n_j. For example, in Figure 45, the type `httpd_t` can access all object-type nodes for cluster `C1` and `C0`, whereas the type `httpd_tmp_t` can only access nodes of `C0`. Note that edges between clusters are visually differentiated as a dashed line.

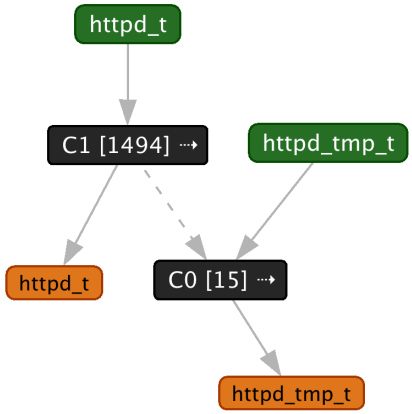


Figure 45: Hierarchical relation (httpd_t $\Re_h$ httpd_tmp_t)

6.4.3 Policy Analysis

To start a policy analysis, first, the admin loads a policy into SEGrapher, which is then parsed into a graph and stored into memory for future analysis. Second, the admin needs to select a set of focus-types to be analyzed, and can optionally select which object-class, and permissions to be used for filtering policy AV rules. The left panel of SEGrapher, as seen in Figure 43, shows some of the object-classes and permissions provided.

Focus-Types SEGrapher allows admins to select a set of focus-types from a set of profiles we define. These profiles allow for a more intuitive method of selecting types according to their functionality, rather than searching for a specific type from within a large list of types (e.g. SELinux targeted-policy has over 1,780 types). For example, an admin can easily find the type `httpd_t` within the profile *Apache* which itself is within the profile *Servers*. Other profile examples include *Databases*, *Mail*, *Introsion Detection*, etc. Figure 43 shows some of the profiles SEGrapher provides.

Once the admin decides on the analysis attributes, she/he can start the analysis. Following our proposed clustering algorithm in Section 6.2.1, SEGrapher produces a focus-graph reflecting the analysis results.

Figure 45 shows a focus-graph for focus-types `httpd_t` and `httpd_tmp_t`. This focus-graph illustrates a hierarchical relation (`httpd_t` $\Re_h$ `{httpd_tmp_t`), that is, `httpd_t` has access to all object-types that `{httpd_tmp_t` has access to. This is reflected through the cluster nodes `C1` and `C0`, where `httpd_t` points to `C1` which in turn points to `C0`, whereas `{httpd_tmp_t` only points to `C0`.

Another example of a focus-graph is shown in Figure 44, which shows an overlapping

relation (`httpd_config_t` $\Re_o$ `httpd_cache_t`). The overlapping accesses between `httpd_config_t` and `httpd_cache_t` are clearly captured within the cluster `C1_0`.

6.4.4 Assisted Policy Analysis

SEGrapher provides admins the ability to load their own custom policy modules. These modules can either be ones resulting from a tool such as audit2allow [32], or manually written by admins themselves and loaded as a text file. Once a custom module is loaded, SEGrapher analyzes the new AV allow rules within the module, and applies our proposed assisted policy analysis approach in Section 6.3. Consider the following AV rule within a loaded custom module:

```
allow httpd_t user_home_t : dir {write}
```

For this rule, SEGrapher will first, find the nearest-neighbors types for `httpd_t`, using our approach in Subsection 6.3.1, and based on the nearest-neighbors' thresholds τ_1 and τ_2 that are set by the admin. The nearest-neighbors' rules are then classified into their appropriate access-classes. Figure 46 shows the resulting nearest-neighbors' rules classifications, with their corresponding color-codes. Note that the rule with subject-type `mysqld_t` shows a strong potential risk of adding the new suggested rule, whereas the last rule in the figure shows low risk. Finally, SEGrapher generates the focus-graph for the types involved in

Nearest-Neighbors' Accesses

Subject-Type	Object-Type	Object Class	Classification
httpd_t	user_home_t	dir	Non-Matching Accesses!
mysqld_t	user_home_t	dir	Non-Matching Accesses!
winbind_t	user_home_t	dir	More Permissive → write

Figure 46: Assisted policy analysis results. Classification of existing rules.

the new custom module, in this case the types `httpd_t` and `user_home_t` as shown in Figure 43.

6.5 User Study

In order to evaluate the effectiveness and usability of SEGrapher we conducted a user study comparing it to the de-facto SELinux policy analysis tool “APOL”. Participants in the study go through a number of tasks related to SELinux policy analysis, and then complete a questionnaire on these tasks. The study was approved by UNC-Charlotte IRB (Protocol #12-05-18).

6.5.1 Methodology

The study participants were recruited from UNC-Charlotte and other corporations. They included both graduate students and IT professionals. We recruited a total of 19 participants who all successfully completed the study and the accompanying survey. Of the participants, 63.1% were in the Information Security field, 15.8% were specialized in Computer Networking, 5.3% in Computer Graphics & Visualization, and 15.7% from other fields. 5.26% of the participants change their operating system configuration on a daily basis, 15.8% weekly, 47.4% monthly, and 26.3% never. 10.5% of them configure their operating system security policy on a weekly basis, 52.6% monthly, whereas 36.8% never do so. Participants where given an introduction to both APOL and SEGrapher and were familiarized with their user interfaces. We then performed a within-subjects study comparison in which participants go through a number of policy analysis tasks using both tools. The order of using each tool was randomized and counter balanced.

6.5.1.1 Policy Analysis Tasks

The analysis tasks in the study involved 6 main tasks as seen in Figure 47. The tasks were performed both on APOL and SEGrapher. The target SELinux policy that was analyzed was the SELinux reference policy in targeted mode.

Analysis Task
Browse the policy components (types, object classes, and permissions).
Locate types belonging to Apache.
Identify the rules between `httpd_t` and `ftpd_t`.
Identify if the type `httpd_t` has `write` permission on `ftpd_t`.
Identify relations between the set of types accessed by `httpd_t` and `httpd_tmp_t`.
Identify the policy rules among `httpd_t`, `mysqld_t`, and `postgresql_t`.

Figure 47: Policy Analysis Tasks

6.5.2 Study Results

After completing the analysis tasks, the participants were asked to complete a questionnaire which covered 5 main aspects of using APOL and SEGrapher. The 5 aspects include: *Ease of Use*, *Overall Satisfaction*, *Browsing Policy Components*, *Composing Analysis Queries*, and finally *Policy Type Interconnectivity*. For each of the aspects we perform a Wilcoxon Signed Rank test (paired by participant and $p < 0.05$) to observe the significance of using SEGrapher vs. APOL. The results are summarized in Figure 48.

	APOL (μ, σ)	**SEGrapher** (μ, σ)	**p-value**
Ease of Use	(2.736, 0.871)	(4.578, 0.507)	0.00001526
Overall Satisfaction	(2.84, 0.95)	(4.63, 0.59)	0.00001526
Browsing Policy Components	(2.89, 1.19)	(4.47, 0.51)	0.0003662
Composing Analysis Queries	(2.736, 0.933)	(4.57, 0.692)	0.01562
Policy Type Interconnectivity	(2.89, 1.19)	(4.47, 0.51)	0.00001526

Figure 48: SEGrapher vs. APOL

6.5.2.1 Ease of Use

Participants were asked to rank the ease of using each of APOL and SEGrapher using a Likert Scale from 1 to 5, where 1 is *Very Complicated* and 5 is *Very Easy*. We observed that SEGrapher was ranked significantly higher than APOL with (Z=-3.79, p=0.00001526, r=0.614).

6.5.2.2 Overall Satisfaction

Participants ranked their overall satisfaction of APOL and SEGrapher using a Likert Scale from 1 to 5, where 1 is *Strongly Disagree* and 5 is *Strongly Agree*. We observed that SEGrapher was ranked significantly higher than APOL in user satisfaction with (Z=-3.8, p=0.00001526, r=0.616).

6.5.2.3 Browsing Policy Components

Participants ranked their satisfaction with browsing the policy components on APOL and SEGrapher using a Likert Scale from 1 to 5, where 1 is *Strongly Disagree* and 5 is *Strongly Agree*. We observed that SEGrapher was ranked significantly higher than APOL with (Z=-3.3779, p=0.0003662, r=0.547).

6.5.2.4 Composing Analysis Queries

Participants ranked their satisfaction with composing analysis queries within APOL and SEGrapher using a Likert Scale from 1 to 5, where 1 is *Strongly Disagree* and 5 is *Strongly Agree*. We observed that SEGrapher was ranked significantly higher than APOL with (Z=-3.7376, p=0.0000305, r=0.606).

6.5.2.5 Policy Type Interconnectivity

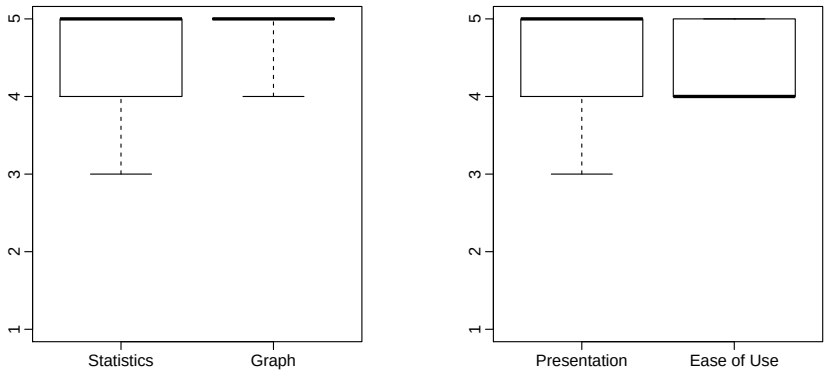
Participants ranked their satisfaction with identifying policy type interconnections using APOL and SEGrapher on a Likert Scale from 1 to 5, where 1 is *Strongly Disagree* and 5 is *Strongly Agree*. We observed that SEGrapher was ranked significantly higher than APOL with (Z=-3.796, p=0.00001526, r=0.615).

6.5.3 Assisted Policy Analysis

Participants also went through 3 tasks related to SEGrapher's assisted policy analysis module. These tasks are listed in Figure 49. From these tasks we evaluate the participants' satisfaction level with SEGrapher's assisted policy analysis module's *ease of use*, *presentation*, *resulting statistics*, and *resulting risk analysis graph*. Participants rate their satisfaction with each aspect on a Likert scale from 1 to 5, where 1 is strongly disagree and 5 is strongly agree. Figures 50(a) and 50(b) illustrate the responses using boxplots. In summary participants were overall satisfied with the various elements in the assisted policy module.

Analysis Task
Identify the existing policy rules similar to the new ones.
Identify the number of *Non-Matching*, *Less Permissive*, *Overlapping*, *More Permissive*, and *Matching* policy rules.
Identify the overall risk of adding the new rules.

Figure 49: Assisted Policy Analysis Tasks



(a) Satisfaction with resulting statistics and analysis graph

(b) Satisfaction with presentation and overall ease of use

Figure 50: Summary of Likert scale responses

CHAPTER 7: CONCLUSIONS

This chapter reiterates and clearly defines the contributions of this dissertation work, and also discusses potential future paths for extending upon this research.

7.1 Contributions

In this work we have proposed and implemented a set of policy management frameworks which are motivated by the need to guide and enhance the overall policy management process. The frameworks are mainly based on two techniques: recommendations and clustering.

First, we propose an enhanced version of the Sun PDP engine which improved policy evaluation performance by orders of magnitude. This was possible by analyzing previous access control request data, and the policy structure. We then adapted the policy to suit various scenarios of high access control requests.

Second, a recommendation-based open authorization framework that was incorporated within a browser (Chrome and Firefox) extension called FBSecure. The framework extends upon the existing OAuth mechanism and provided the following: 1) Recommendations per requested permissions, which are based on the collaborative community decisions, 2) Fine-grained control over the privacy attributes requested by third party applications.

Third, we propose a framework for guiding users towards enhancing their policy decisions on third party browser extension permissions. The framework provides fine-grained

controls over the requested permissions in addition to extended permission descriptions. We conducted a user-study to evaluate the effectiveness of the framework, which showed a significant improvement in the user awareness towards the permissions requested by extensions. The framework is also capable of monitoring the accesses made by third party extensions at run-time.

Finally, we propose “SEGrapher”, a visualized-based policy analysis tool for SELinux policies. SEGrapher uses a clustering technique that clusters SELinux *types* based on their policy accesses. Using these clusters it is able to present simplified analyses results in the form of a directed graph. The cluster-based results provide a powerful approach for discovering inherited relations between various SELinux policy types. SEGrapher also has the ability to measure the potential risks of adding new policy rules. The risks are based on previous knowledge of the policy and discovering the similarities between the types in the new rules and those already within the policy.

7.2 Future Work

This dissertation contributes a number of frameworks for enhancing the policy management process for both administrators and users. In the following we discuss future work that could further improve upon this research.

7.2.1 Recommendation-based Open Authorization

The proposed recommendation-based open authorization can further be integrated with existing social networking sites. This integration can enrich the recommendation models by utilizing extended user data (e.g. profile information and friendship network). This would also reduce the effect of cold start recommendations. Another potential path would

involve incorporating the functionalities of the framework directly into the browser and adopt a wider range of social sites. Further user studies would verify the effectiveness of such integration.

7.2.2 Third Party Browser Extension Policy Management

The current state of the framework allows for fine-grained permission controls and run-time monitoring of extension accesses. To further improve the framework, recommendation models such as those in Section 4 should be adopted. Next to the permission descriptions provided in the framework, users would also be able to utilize recommendations based on the community inputs.

The monitoring of extension accesses was mainly based on Chrome API calls. Accesses out of the Chrome API scope could still possibly occur. To mitigate the possibility of so, additional methods of detecting extension access should be adopted, e.g. static analysis of extension source code could potentially help, in addition to identifying certain attack paths or unsafe javascript methods.

As Chrome extensions already have a dedicated settings window within Chrome itself, it would be interesting to study the potential integration of the framework into Chrome's existing settings window rather than being a stand alone extension. This could change the users perception towards customizing permissions and enhance their experience.

7.2.3 SELinux Policy Management

The proposed tool "SEGrapher" can be further improved by adding better user interaction capabilities with the resulting analysis graphs. Such interactions could provide faster feedback on specific relations amongst the analyzed policy types. For example, clicking on

an edge could provide extra details on a relationship.

The risk analysis module within SEGrapher could be extend upon to accommodate new types that do not already exist within the policy. This might require some additional information to be provided by the administrator in an effort to better understand the new types. SEGrapher could provide a set of usage profiles that can be assigned to newly introduced rules and from those profiles infer possible recommendations. Such profiles could be generated by analyzing and monitoring the behaviors of different applications within various domains. For example, a profile could be generated for we servers in general, hence any new rules that are assigned to a web server could potentially be compared to its associated profile. Finally, additional user studies could be conducted to further evaluate the effectiveness of SEGrapher.

REFERENCES

[1] Alessandro Acquisti and Ralph Gross. Imagined communities: Awareness, information sharing, and privacy on the facebook. In *Privacy Enhancing Technologies*, pages 36–58, 2006.

[2] Alessandro Acquisti and Ralph Gross. Imagined communities: Awareness, information sharing, and privacy on the facebook. In *Privacy Enhancing Technologies*, pages 36–58, 2006.

[3] Gediminas Adomavicius and YoungOk Kwon. *In Recommender Systems Handbook: A Complete Guide for Research Scientists and Practitioners*, chapter Multi-Criteria Recommender Systems - Forthcoming. Springer, 2010.

[4] Sruthi Bandhakavi, Samuel T. King, P. Madhusudan, and Marianne Winslett. Vex: vetting browser extensions for security vulnerabilities. In *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, pages 22–22, Berkeley, CA, USA, 2010. USENIX Association.

[5] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. Protecting browsers from extension vulnerabilities. *17th Network and Distributed System Security Symposium*, 2010.

[6] Andrew Besmer, Jason Watson, and Heather Richter Lipford. The impact of social navigation on privacy policy configuration. In Lorrie Faith Cranor, editor, *SOUPS*, volume 485 of *ACM International Conference Proceeding Series*. ACM, 2010.

[7] Dr. Carrie and E. Gates. Access control requirements for web 2.0 security and privacy. In *Proc. of Workshop on Web 2.0 Security & Privacy (W2SP 2007)*, 2007.

[8] Shan Chen and Mary-Anne Williams. Towards a comprehensive requirements architecture for privacy-aware social recommender systems. In *APCCM '10: Proceedings of the Seventh Asia-Pacific Conference on Conceptual Modelling*, pages 33–42, Darlinghurst, Australia, Australia, 2010. Australian Computer Society, Inc.

[9] Mohan Dhawan and Vinod Ganapathy. Analyzing information flow in javascript-based browser extensions. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, pages 382–391, Washington, DC, USA, 2009. IEEE Computer Society.

[10] Catherine Dwyer, Starr Roxanne Hiltz, and Katia Passerini. Trust and privacy concern within social networking sites: A comparison of facebook and myspace. In *Proceedings of the Thirteenth Americas Conference on Information Systems (AMCIS 2007)*, 2007. Paper 339.

[11] Facebook. Facebook Press Room. *http://www.facebook.com/press/info.php?statistics*, 2011.

[12] Lujun Fang and Kristen LeFevre. Privacy wizards for social networking sites. In Michael Rappa, Paul Jones, Juliana Freire, and Soumen Chakrabarti, editors, *WWW*, pages 351–360. ACM, 2010.

[13] Adrienne Felt and David Evans. Workshop on web 2.0 security and privacy. oakland, ca. 22 may 2008. privacy protection for social networking platforms, 2008.

[14] Adrienne Porter Felt, Kate Greenwood, and David Wagner. The effectiveness of application permissions. In *Proceedings of the 2nd USENIX conference on Web application development*, WebApps'11, Berkeley, CA, USA.

[15] Adrienne Porter Felt, Kate Greenwood, and David Wagner. The effectiveness of application permissions. In *Proceedings of the 2nd USENIX conference on Web application development*, WebApps'11, pages 7–7, Berkeley, CA, USA, 2011. USENIX Association.

[16] FireBreath. FireBreath. `http://www.firebreath.org/`.

[17] Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich, and Michael Carl Tschantz. Verification and change-impact analysis of access-control policies. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 196–205, New York, NY, USA, 2005. ACM.

[18] FriendCameo, Inc. FriendCameo. *http://friendcameo.com*, 2010.

[19] David Goldberg, David Nichols, Brian M. Oki, and Douglas Terry. Using collaborative filtering to weave an information tapestry. *Commun. ACM*, 35(12):61–70, 1992.

[20] Kiran K. Gollu, Stefan Saroiu, and Alec Wolman. A social networking-based access control scheme for personal content. *Proc. 21st ACM Symposium on Operating Systems Principles (SOSP '07). Work in progress*, 2007.

[21] Ralph Gross and Alessandro Acquisti. Information revelation and privacy in online social networks. In *Proceedings of the 2005 ACM workshop on Privacy in the electronic society*, WPES '05, pages 71–80, New York, NY, USA, 2005. ACM.

[22] H. Hamed, A. El-Atawy, and E. Al-Shaer. Adaptive statistical optimization techniques for firewall packet filtering. In *INFOCOM 2006: Proceedings of the 25th IEEE International Conference on Computer Communications*, pages 1–12, April 2006.

[23] Hazem Hamed and Ehab Al-Shaer. Dynamic rule-ordering optimization for high-speed firewall filtering. In *Proceedings of the 2006 ACM Symposium on Information, computer and communications security*, pages 332–342, New York, NY, USA, 2006. ACM.

[24] Michael Hart, Rob Johnson, and Amanda Stent. More content - less control: Access control in the Web 2.0. *Web 2.0 Security & Privacy*, 2003.

[25] Jonathan L. Herlocker, Joseph A. Konstan, Al Borchers, and John Riedl. An algorithmic framework for performing collaborative filtering. In *Proceedings of the international ACM SIGIR conference*, SIGIR '99, pages 230–237, New York, NY, USA, 1999. ACM.

[26] Jonathan L. Herlocker, Joseph A. Konstan, Loren G. Terveen, and John T. Riedl. Evaluating collaborative filtering recommender systems. *ACM Trans. Inf. Syst.*, 22:5–53, January 2004.

[27] Hitachi Software. Seedit: Selinux policy editor `http://seedit.sourceforge.net`.

[28] Graham Hughes and Tevfik Bultan. Automated verification of xacml policies using a sat solver. In *Proceedings of the Workshop on Web Quality, Verification and Validation (WQVV 07)*, pages 378–392, 2007.

[29] iOpus. iMacros. `http://www.iopus.com/imacros/chrome/`.

[30] Trent Jaeger, Reiner Sailer, and Xiaolan Zhang. Analyzing integrity protection in the selinux example policy. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, pages 5–5, Berkeley, CA, USA, 2003. USENIX Association.

[31] Jeffrey Heer . Prefuse (Java). *`http://prefuse.org`*.

[32] Justin R. Smith, Yuichi Nakamura, and Dan Walsh. audit2allow. *`http://linux.die.net/man/1/audit2allow`*.

[33] Patrick Gage Kelley, Paul Hankes Drielsma, Norman Sadeh, and Lorrie Faith Cranor. User-controllable learning of security and privacy policies. In *AISec '08: Proceedings of the 1st ACM workshop on Workshop on AISec*, pages 11–18, New York, NY, USA, 2008. ACM.

[34] Kernel Trap. SELinux vs. OpenBSD's Default Security. *`http://kerneltrap.org/OpenBSD/SELinux_vs_OpenBSDs_Default_Security`*.

[35] Haruka Kikuchi, Dachuan Yu, Ajay Chander, Hiroshi Inamura, and Igor Serikov. Javascript instrumentation in practice. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 326–341, Berlin, Heidelberg, 2008. Springer-Verlag.

[36] Vladimir Kolovski and James Hendler. XACML policy analysis using description logics. *Submitted to Journal of Computer Security (JCS) available at http://www.mindswap.org/∼kolovski/KolovskiXACMLAnalysisJCSSubmission.pdf*, 2008.

[37] Vladimir Kolovski, James Hendler, and Bijan Parsia. Analyzing web access control policies. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 677–686, New York, NY, USA, 2007. ACM.

[38] Hsin-Hsien Lee and Wei-Guang Teng. Incorporating multi-criteria ratings in recommendation systems. In *IRI'07*, pages 273–278, 2007.

[39] Ming Li, Benjamin Dias, Wael El-Deredy, and Paulo J. G. Lisboa. A probabilistic model for item-based recommender systems. In *Proceedings of the 2007 ACM conference on Recommender systems*, RecSys '07, pages 129–132, New York, NY, USA, 2007. ACM.

[40] Alex X. Liu, Fei Chen, JeeHyun Hwang, and Tao Xie. Xengine: a fast and scalable xacml policy evaluation engine. In *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 265–276, New York, NY, USA, 2008. ACM.

[41] R. S. Liverani and N. Freeman. Abusing Firefox Extensions. In *Defcon*, July 2009.

[42] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association.

[43] G. A. Di Lucca, A. R. Fasolino, M. Mastoianni, and P. Tramontana. Identifying cross site scripting vulnerabilities in web applications. In *Proceedings of the Web Site Evolution, Sixth IEEE International Workshop*, pages 71–80, Washington, DC, 2004. IEEE Computer Society.

[44] LWN.net. Quotes of the week. *http://lwn.net/Articles/179829/*.

[45] Matthew R. McLaughlin and Jonathan L. Herlocker. A collaborative filtering algorithm and evaluation metric that accurately model the user experience. In *Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '04, pages 329–336, New York, NY, USA, 2004. ACM.

[46] Philip L. Miseldine. Automated xacml policy reconfiguration for evaluation optimisation. In *Proceedings of the 4th International Workshop on Software Engineering for Secure Systems*, pages 1–8, New York, NY, USA, 2008. ACM.

[47] MITRE . SELinux Analysis Tools (SLAT). *http://www.mitre.org/tech/selinux/*.

[48] MITRE. Polgen: Guided auto-mated policy development. http://www.mitre.org/tech/selinux.

[49] T. Moses. Extensible access control markup language (XACML). *Technical Report, OASIS*, 2003.

[50] Mozilla Add-Ons Blog. How many Firefox users have add-ons installed? 85%! http://blog.mozilla.com/addons/2011/06/21/firefox-4-add-on-users/.

[51] OAuth. Security Advisory:2009.1. http://oauth.net/advisories/2009-1/.

[52] OAuth 2.0. The OAuth 2.0 Protocol. *http://tools.ietf.org/html/draft-ietf-oauth-v2-10*, 2010.

[53] N. Ramakrishnan, B.J. Keller, B.J. Mirza, A.Y. Grama, and G. Karypis. Privacy risks in recommender systems. *Internet Computing, IEEE*, 5(6):54 –63, 2001.

[54] Red Hat, Inc. Red Hat SELinux Guide, Chapter 8. Customizing and Writing Policy. *http://docs.redhat.com/docs/en-US/Red_Hat_Enterprise_Linux/4/html/SELinux_Guide/selg-section-0120.html*.

[55] Paul Resnick, Neophytos Iacovou, Mitesh Suchak, Peter Bergstrom, and John Riedl. Grouplens: an open architecture for collaborative filtering of netnews. In *CSCW '94: Proceedings of the 1994 ACM conference on Computer supported cooperative work*, pages 175–186, New York, NY, USA, 1994. ACM.

[56] J. Riedl. Personalization and privacy. *Internet Computing, IEEE*, 5(6):29 –31, 2001.

[57] Ronald Rivest. On self-organizing sequential search heuristics. *Commun. ACM*, 19(2):63–67, 1976.

[58] Samba. Samba Server. *http://www.samba.org/samba*.

[59] Beata Sarna-Starosta and Scott D. Stoller. Policy analysis for security-enhanced linux. In *In Proceedings of the 2004 Workshop on Issues in the Theory of Security (WITS*, pages 1–12, 2004.

[60] Security Enhanced Linux. http://www.nsa.gov/research/selinux.

[61] Mohamed Shehab, Anna Cinzia Squicciarini, and Gail-Joon Ahn. Beyond user-to-user access control for online social networks. In *Proceedings of the 10th International Conference on Information and Communications Security*, ICICS '08, pages 174–189, Berlin, Heidelberg, 2008. Springer-Verlag.

[62] Stephen W. Cote. FunMon2.js. http://www.imnmotion.com/documents/html/technical/dhtml/funmon.html.

[63] Xiaoyuan Su and Taghi M. Khoshgoftaar. A survey of collaborative filtering techniques. *Adv. in Artif. Intell.*, 2009:4:2–4:2, January 2009.

[64] Sun XACML Policy Engine. http://sunxacml.sourceforge.net/guide.html.

[65] Mike Ter Louw, Jin Lim, and V. Venkatakrishnan. Enhancing web browser security against malware extensions. *Journal in Computer Virology*, 4:179–195, 2008.

[66] The Chromium Blog. A Year of Extensions. http://blog.chromium.org/2010/12/year-of-extensions.html.

[67] Tresys Technology. APOL. *http://oss.tresys.com/projects/setools*.

[68] Tresys Technology. Setools: Policy analysis tools for selinux `http://oss.tresys.com/projects/setools`.

[69] V. N. Venkatakrishnan, Prithvi Bisht, Mike Ter Louw, Michelle Zhou, Kalpana Gondi, and Karthik Thotta Ganesh. Webapparmor: a framework for robust prevention of attacks on web applications. In *Proceedings of the 6th international conference on Information systems security*, ICISS'10, pages 3–26, Berlin, Heidelberg, 2010. Springer-Verlag.

[70] Vincent Danen. Introduction to SELinux: Don't let complexity scare you off. *http://www.techrepublic.com/blog/opensource/introduction-to-selinux-dont-let-complexity-scare-you-off/2447*.

[71] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. 2 edition, 2005.

[72] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, Berkeley, CA, USA, 2006. USENIX Association.

[73] Wenjuan Xu, Mohamed Shehab, and Gail-Joon Ahn. Visualization based policy analysis: case study in selinux. In *Proceedings of the 13th ACM symposium on Access control models and technologies*, SACMAT '08, pages 165–174, New York, NY, USA, 2008. ACM.

[74] Yuichi Nakamura. SELinux Policy Editor(SEEdit) Administration Guide 2.1. *http://seedit.sourceforge.net/doc/2.1/tutorial/node9.html*, February 2007.

[75] Yuchen Zhou and David Evans. Protecting private web content from embedded scripts. In *Proceedings of the 16th European conference on Research in computer security*, ESORICS'11, pages 60–79, Berlin, Heidelberg, 2011. Springer-Verlag.