# Attribute-Based Access Control for Distributed Systems

by

David J. B. Cheperdak
B.Sc., University of Victoria, 2011

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTERS OF SCIENCE

in the Department of Computer Science

Your file Votre référence
ISBN: 978-0-499-26384-1

Our file Notre référence
ISBN: 978-0-499-26384-1

Attribute-Based Access Control for Distributed Systems

by

David J. B. Cheperdak
B.Sc., University of Victoria, 2011

Supervisory Committee

Dr. Y. Coady, Co-Supervisor
(Department of Computer Science)

Dr. S. Neville, Co-Supervisor
(Department of Electrical and Computer Engineering)

Dr. P. McGeer, Co-Supervisor
(Department of Computer Science)

**Supervisory Committee**

## ABSTRACT

Securing information systems from cyber attacks, malware and internal cyber threats is a difficult problem. Attacks on authentication and authorization (access control) is one of the more predominant and potentially rewarding attacks on distributed architectures. Attribute-Based Access Control (ABAC) is one of the more recent mechanisms to provide access control capabilities. ABAC combines the strength of cryptography with semantic expressions and relational assertions. By this composition, a powerful grammar is devised that can not only define complex and scalable access control policies, but defend against attacks on the policy itself. This thesis demonstrates how ABAC can be used as a primary access control solution for enterprise and commercial applications.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

## 1.1 Introduction

Securing information systems from cyber attacks, malware and internal threats is a difficult problem. Cyber security is a multi-variable function of complexity correlated to a given information system. As complexity of a given information system increases, so can the distribution of potential attack vectors. In this respect, cyber security can be modeled as a mapping of an attack vector on a given resource to zero or more defensive mechanisms for that resource. An attack vector denotes the attack path of an entity $A$ may take to gain access to a given resource.

Many different approaches have arisen to deal with various attack vectors within information systems. Such approaches include firewalls, intrusion detection systems, intrusion prevention systems, virus scanners, deep packet inspection techniques, access control mechanisms, VPN and real time monitoring. Each approach emphasizes a particular range and type of potential attack vectors. Additionally, the architecture, technology and requirements governing an information system dictates or imposes stringent requirements on a given cyber defense approach. Specifically, securing individual components defines a significantly different problem than a homogenous Cloud Computing architecture. In particular, Cloud Computing architectures, in their diverse forms, derive several unique properties. These include but are not limited to, multi-tenancy, dynamic tenancy, multiple operational domains, shared infrastructure and policy-defined federation requirements. Such properties require access control mechanisms that similarly facilitate these properties. This thesis focuses on a particular element of cyber security. Access control mechanisms for distributed architectures. In particular, this thesis demonstrates that an access control technology known as Attribute-Based Access Control (ABAC) can be used as a primary means for authorization and authentication in corporate and enterprise distributed architectures. This thesis contains two case studies that demonstrate ABACs capacity to subsume such distributed architectures authorization and authentication requirements. The first case study analyzes the capacity of ABAC to subsume the role of legacy access control mechanisms in a legacy distributed system. The second case study analyzes ABACs capacity to fully encompass a modern distributed architectures formal access control requirements. It should be noted that when subsuming legacy access control mechanisms, there is a distinct relationship between legacy access control requirements and those of a modern distributed architecture. This difference is emphasized by the technology, iterative architecture and scalability requirements of a modern distributed system.

## 1.2 Thesis Claims

I make *one* claim which my thesis validates:

Claim: This thesis demonstrates that Attribute-Based Access Control (ABAC) can be used as a primary authorization and authentication mechanism (access control) for legacy or modern enterprise systems.

The Claim, as detailed in this thesis, will be demonstrated *empirically* and *qualitatively* through analysis of two case studies. The first case study investigates how ABAC can be integrated into a open source legacy architecture. The second case study investigates how ABAC can be integrated into a proprietary modern architecture.

### 1.2.1 Importance of Claims

Access control is a fundamental aspect of any given information system. Information is considered an asset whether it be medical, financial, personal or some other form of valuable data. However, authentication and authorization (access control) of an information system in most cases has either serious limitations or weaknesses. Attribute-Based Access Control (ABAC) can overcome many of these weaknesses by reducing potential attack vectors down to one. This attack vector is the the management and protection of X.509 certificates. Additional capabilities inherent to ABAC include the capability to factorize policy from mechanism and policy from implementation. Factorization ensures that policy remains scalable and manageable as a computer architecture scales. ABAC also has capacity to facilitate federation. Federation is the mechanism by which multiple *exclusive* infrastructures or systems may interact or share data in a secure, trusted way. With the growth of Big Data, the need for a system to facilitate management of large collections of data while maintaining fine grain access control is essential. These capacities among others are explored throughout this thesis.

The Claim as noted in this thesis implies:

1. Distributed infrastructures can:
    - federate with other infrastructures;
    - enable policy to scale as the infrastructure scales;
    - provide fine grain access control policies;
    - decouple access control policy from implementation;
    - decouple policy from infrastructure;
    - factorize policy to infrastructure.

Thus, ABAC can ultimately *provide access control capabilities that model the Cloud Computing paradigm.* Other access control mechanisms can subsume a similar set of capabilities as ABAC but have a different attack surface.

## 1.3 Outline

This section provides a map of the thesis as follows:

**Chapter 1** contains a review of Access Control technologies and fundamental principles inherent in authorization and authentication.

**Chapter 2** contains the first case study that investigates the feasibility of integrating ABAC into a legacy architecture.

**Chapter 3** contains the second case study that investigates the feasibility of integrating ABAC into a modern architecture.

**Chapter 4** provides high level analysis of the case studies conducted in this thesis and concluding inferences.

## 1.4 Introduction

This thesis is divided into four chapters. These chapters are as follows. Chapter 1 provides a comprehensive review of application, principle and technology behind access control systems. Chapter 1 is divided into several primary sections. The first major section provides an overview of authorization. The second section provides an overview of authentication and technologies that provide authentication capabilities. The third section provides an overview of technologies that provide both authorization and authentication capabilities. The fourth and final section provides a detailed introduction to Attribute-Based Access Control (ABAC) including technology and grammar components.

Chapter 2 provides the first case study that investigates the feasibility of integrating ABAC into a legacy architecture. This chapter is divided into several sections. The first section provides an introduction to the legacy architecture PlanetLab. The second section details the analysis of the legacy architecture. The third section details the approach to integrate ABAC into this legacy architecture. The fourth section details the evolution of legacy policy into an ABAC grammar. The fifth and final section details quantitative and qualitative results.

Chapter 3 provides the second case study that investigates the feasibility of integrating ABAC into a modern architecture. This chapter is divided into several sections. The first section details the core objectives and requirements detailed by the modern architecture. The second section details the modern architecture under study. The third section details the approach and design of integrating ABAC into a modern architecture. The fourth section details the evolution and design of an ABAC policy. The fifth and final section details quantitative and qualitative results.

Chapter 4 provides a high level overview of the case studies explored in this thesis. Additionally, contrast between approaches for these two case studies are evaluated with respect to the claims defined within this thesis. Additionally, conclusions to the findings in these studies is detailed.

## 1.5 Technology Introduction: Access Control

Access Control is a defensive mechanism against one type of attack vector. This attack vector encompasses the set of actions and operations required to gain access to a information system resource that an entity does not or should not have access to. It is important to note that actions and operations that result in erroneous access to an information system's resources are not necessarily malicious. It is possible that, by fault in a system, access can be granted to an invalid non-malicious user. This can happen if an access control policy is inappropriately defined or a failure occurs. However, appropriate access control design measures can be taken to eliminate such occurrences.

*Access Control* defines a mechanism and policy that controls how and which entities may access a particular resource. It should be noted that entities can refer to both human clients and other information systems. Access Control can incorporate different mechanisms to facilitate assertion of access to a particular resource. Generally these mechanisms incorporate the notion of *Credentials* to denote a key to unlock a gate to a particular resource. Access Control can be partitioned into two primary functions, Authorization and Authentication.

## 1.6 Authorization

Authorization is the process and function of delegating resource access rights to entities [36]. It also is the process by which an access control policy for a particular resource is created. In most cases a policy continues to evolve as the given information system evolves. Thus the structure of an access control policy directly correlates to its manageability. A policy can take on many structures and be defined as several distinct forms. These forms include but are not limited to Role Based, Resource Based and Claims Based policy definitions.

### 1.6.1 Role-Based

Role-Based policies are generally concerned with the delegation of a role to a given entity within the Policy [11]. This pairing of role and entity defines an authorization entity within the system. In the general case, a role is defined as a bundle of capabilities. This aggregation of capabilities simplifies delegation of rights to a particular entity.

### 1.6.2 Resource-Based

Resource-Based policies associate entity credentials directly with a given resource [26]. By this policy type an Access Control List (ACL) can be defined for each resource which denotes explicitly which may access the resource and in which way. This policy type differs from Role Based policies as each user is explicitly given access instead of delegating access to a Role.

### 1.6.3 Claims-Based

Claims-Based policies add additional layers of abstraction to policy layers through intermediate steps [27]. An example of policy layers can be elevating privileges in a Linux operating system through *sudo* before authenticating against a database and its given ACL.

### 1.6.4 Authorization for Cloud Infrastructures

These three policy types are not always ideal for a given system. This thesis will demonstrate that a hybrid approach of all three types is necessary to construct a policy for a large scale distributed system using ABAC technologies. Authorization and policy design is only one factor of Access Control. The second aspect of Access Control is Authentication.

## 1.7 Authentication

Authentication is the means by which an entity confirms a truth or fact. This fact or truth is often correlated to proof of identity which is essential to asserting Authorization. As a side note, a common attack vector on Access Control systems is forgery or a related attack on the identity of a given trusted entity within a system. Thus many different Authentication systems exist, each designed to mitigate attack types based on identity.

### 1.7.1 Types

There are three primary types of Authentication. The first type of Authentication is identity through proof. The second type of Authentication is attribute comparison. The third type is third party assertion. Identity through proof is a mechanism by which an entity presents both a request for access and a proof of access. Such a mechanism can be expressed as the case when entity X desires to access the resource OPERATION at entity Y. Entity X provides Y a token that asserts trust of X and permission to access resource OPERATION. In the case of the attribute type comparison, the entity Y that asserts access control over the resource OPERATION requires comparison against expected attributes of a requesting entity X. In this case, attributes such as IP Address, MAC address, Domain Name can be evaluated to assert a proof that entity X has access to resource OPERATION controlled by Y. The third type of authentication, third party assertion, specifies that a mutually trusted third party is delegated the privilege to assert truth of identity on behalf of both entities X and Y. This type of authentication is particularly important in web applications and federated access control. One example of this situation is when two web services need to share data but do not wish to reveal the nature of their Authorization structures or given policies. In this context, the trusted third party acts as a negotiator or intermediary. Another aspect of Authentication, which is independent of the means of which trust of identity is asserted, is Identity Management.

### 1.7.2 Identity Management

Identity Management or (IdM) describes the capacity and mechanism to manage the identities of entities within a given system [29]. Identity Management is defined by organizational policies for one or more distributed systems. IdM is critical to ensuring Access Control mechanisms are effective at protecting resources from unauthorized access. For example, if an human user of the system is demoted within an organization, a corresponding change to the Identity within the system should also occur to reflect a new organizational role. Similarly, in the event that an employee leaves an organization, the account of that employee and the corresponding identity should be revoked from the system. Failing to perform the appropriate change in Identity within the system can expose a new attack vector. This thesis does not address the issue of Identity Management with respect to ABAC, as this scope far exceeds the intent to demonstrate that ABAC can act as a primary mechanism of Access Control in distributed systems for enterprises and corporations. As IdM is largely dependent on the practices and internal policies of an organization, the variants of potential systems are unbounded. This thesis does identify fundamental principles concerned with IdM with respect to ABAC credentials for future IdM implementations.

### 1.7.3 Mechanisms

Authorization policies and Authentication mechanisms are realized in implementation as Access Control mechanisms. Mechanisms are functional systems that provide the means that Authorization and Authentication can be bound to resources in a given system. Mechanisms encompass the software that performs assertions, provide the means of communicating and implement analytical processes on access control policies. These mechanisms harness a wide variety of technologies and are implemented into different mechanisms.

### 1.7.4 Technology and Mechanisms

As described previously, there exists a wide range of attack vectors on information systems. There are also a wide range of attack vectors on Access Control systems. Many different approaches have been taken to eliminate potential attack vectors while ensuring capacity to provide an effective Access Control solution. Incorporating the previous notions of Authorization and Authentication, three primary classes of Access Control mechanisms are defined. These classes are Indirect Assertion, Direct Assertion and Hybrid Assertion. Prominent mechanisms that fall into these classes are discussed in the next section.

## 1.8 Indirect Assertion

Web services are a unique case as they emphasize requirements for simple, minimalistic, access delegation across user level facilities. Access Control systems such as OAuth and Shibboleth are mechanisms that have evolved to satisfy these requirements. However, to define complex fine grain complex policies, alternatives must be considered.

### 1.8.1 Shibboleth

Shibboleth [34] is a mechanism for single-sign on for information systems. Shibboleth is a federated identity-based authentication and authorization infrastructure based on Security Assertion Markup Language (SAML). The Shibboleth architecture is divided into two primary components, Identity Provider and Service Provider. Shibboleths primary Access Control mechanism involves a several step process. This process first involves an entity being redirected from a resource to a Shibboleth Identity Provider. This entity then authenticates against the Identity Provider by which a SAML authentication assertion is returned by the Identity Provider to the Service Provider which is then consumed. Although Shibboleth provides a mechanism for federated access, a primary weakness in this mechanism is attacks directly on the Identity Provider or the Service Provider. A legitimate request can be redirected to a potentially malicious Identity Provider that steals entity credentials [35]. Additionally, as Shibboleth requires external authentication request using HTTP, XML and SAML standards, many web based attacks are prevalent such as XML encryption assertions in transit.

### 1.8.2 OpenID

OpenID [33] incorporates a similar authorization and authentication architecture to Shibboleth. OpenID has seen considerable adoption in large organizations such as Google, Yahoo and PayPal. OpenID comprises several components. These components are requesting entity, relaying party (RP) and OpenID Identity Provider (OP). The RP is the resource that seeks to validate the requesting entities credentials and the OP is the mediator between the entity and the RP. This form of Access Control can be viewed as delegation. In this context the requesting entity delegates rights to the OP to act on its behalf to carry out some operation or access a resource being the RP. The OP then returns results of the operation or request to the requesting Entity. The reliance on delegation to the OP is a major security flaw in OpenID technology. OpenID is particularly vulnerable to phishing attacks by which a user is directed to a fake OP [16]. OpenID also suffers from major privacy and security flaws [3]. In particular, as a requesting entity delegates rights to the OP to act on its behalf, the OP has potentially unrestricted access to private data at the RP.

### 1.8.3 OAuth

OAuth [19] is a technology once again comparable to Shibboleth and OpenID. OAuth foundationally is built on an open standard of delegated authentication. OAuth differs from OpenID in one fundamental way. OAuth utilizes tokens to delegate access where as OpenID uses certificates that contain private information. The use of tokens creates capacity for pseudo-authentication or obfuscating one's identity. However, OAuth like OpenID is also susceptible to several serious attacks [17]. Similarly to OpenID, phishing attacks are prevalent at Identity Providers. Other weaknesses include lack of data confidentiality and brute force attacks against certain token types.

## 1.9 Direct Assertion

### 1.9.1 DAC

One of the first approaches to provide direct assertion is Discretionary Access Control (DAC) [32]. DAC enables access to resources objects based on identity of the client and the group the client belongs to. However, DAC lacks ability to accurately describe the association between client and resource; the lack of meta-data associated with function can create security holes and result in overly complex policies [12].

### 1.9.2 MAC

Similar to Discretionary Access Control, Mandatory Access Control (MAC) [24] defines access control lists and attributes present in a given system. However, MAC does not allow delegation of these privileges to anyone but the administrator of the Access Control policy. Unlike DAC, MAC allows security administrators to define domain wide Access Control policies. Although DAC and MAC are effective access control solutions, they do not necessarily facilitate mechanisms of federate access, negotiation, dynamic scalability and modularly, all of which are critical to distributed architectures like Cloud Computing.

## 1.10 Hybrid Assertion

Chadwick [4] [6] first bound attributes to X.509 certificates. This precedent led to the development of Role-Based Access Control (RBAC), an attempt to solve problems with scalable access control for large distributed infrastructures [5] [1] [20] [2].

### 1.10.1 RBAC

Role-Based Access Control originates from the efforts to minimize complexity of a given policy. RBAC ties cryptographic permission attributes directly to a cryptographic role. Additionally, efforts have been made over the years to further evolve RBAC capabilities and mechanisms [28] [39] to scale with distributed infrastructures such as attribute hierarchies [21]. The role abstraction layer imposed limitations on the capabilities of distributed federated architectures. For example, federates must maintain the same record of complex role definitions to facilitate inter-federate member access. However, primary problems associated with RBAC include difficulty mapping roles across a federated domain. Organizations would have to agree upon common attributes.

### 1.10.2 ZBAC

The mechanism authoriZation Based Access Control (ZBAC) [18] attempts to address issues relating to the difficulty of reaching cross organization agreements on shared attributes or roles. In this context, ZBAC tends to be more asymmetric when compared to other Access Control mechanisms. This is accomplished by moving the Policy Decision Point (PDP) into the requester domain instead of the resource domain. In this sense, authorization is based on the authentication in the users domain before a request is made. This can be considered *preauthentication*. The preauthentication credential is then submitted along with the request to the intended resource. However, as RBAC requires mutual understanding of roles, ZBAC requires mutual understanding of federated agreements. ABAC, however, does not require any mutual agreement. An organization may choose to informally or formally announce what roles are present in the system along with what services are offered. It is then left up to the requesting system to map the given syntactic role definition to a given policy implementation. Additional details concerning these unique ABAC properties are defined later in Section 5.

### 1.10.3 ABAC

Attribute-Based Access Control (ABAC) binds attributes and roles directly to a client credential. David W. Chadwick et al. first bound authorization information to X.509 [6] that has resulted in cross domain authorization. However, when considering silo-based federated architectures, each silo system may implement a unique set of policies; authorization in testbed A with role X may not be valid in testbed B. Therefore, ABAC establishes the means for unique federate policies while maintaining cohesive cross federate roles. ABAC expands authorization capability for federated distributed architectures. ABAC provides an effective means for access control in rapidly evolving systems [38], can fully encapsulate existing policy definitions [25] and provide a means for trust negotiation while preserving privacy [25]. ABAC also facilitates extended attribute hierarchies for confidentiality, scalability, fine grained access control in Cloud storage, and aggregation of complexity. ABAC has also been shown to enable multi-domain access control supporting the inter-system mapping of roles and attributes. Additional properties of ABAC include concurrent and adaptable policies based on risk. These capabilities and properties provide the basis for replacement of legacy policies and access mechanisms in multi-domain distributed, federated infrastructures. ABAC has been integrated within distributed infrastructures such as ProtoGENI [31] and DETER's Federation Architecture [7].

ABAC incorporates a formal logic of authorization [8] within a cross-platform implementation [10]. It allows testbed operators to state access policies as a collection of attributes, provides a logic engine that uses these policies to make decisions and produces a record of the reasoning used to arrive at those decisions in human- and machine-readable forms. Policy statements enable a precise translation of roles between administrative domains across a federation, e.g. a supervising research scientist attribute in one domain may translate to project leader in another.

## 1.11 ABAC Introduction

### 1.11.1 Attribute-Based Access Control

ABAC policies are expressed in Role-based Trust-management (RT0) logic [22] which is a first order predicate logic that can be translated into datalog and are cryptographically signed. RT0 logic provides the foundation for many of ABACs logical grammar. The following section provides introductory material to RT0 logic and unique credential and policy properties.

### 1.11.2 RT0 Logic

RT0 logic and associated language are rooted in two fundamental works of William H. Winsborough et al, credential chain discovery [23] and credential graphs [37]. Credential chain discovery or (CCD) is the process by which access control decisions are made by finding one or more paths from the resource requestor to the authority of that resource. Four possible states occur from a given access control query. The first is a failure at the first delegate chain link, the second a partial chain where one or more attributes match. The third state is a single successful chain link and the fourth and final is when more than one successful chain has been created. Each of these states can result in a distinct response to the requestor based on the requirements defined within the system. Credential graphs are the resulting structure of one or more credential policies that may be queried given a set of query attributes and a requestors identity credential. These fundamental properties define the reasoning logic required to formalize access control policies.

### 1.11.3 RT0 Syntax

The RT0 declarative logic and language are defined by two primary constructs called entities, also referred to as principals, and roles. Roles and entities are coupled in the form of ENTITY.role where the dot notation signifies binding of the role to the given entity. The dot also semantically expresses the entity's *ownership* of the given role. Through this mechanism, a role may be delegated to other entities within the system. RT0 incorporates four primary kinds of credentials: membership, role-based membership, link-based membership and intersect membership. Membership is the direct association of an external entity to the role owned by another entity. This can be denoted as RESOURCE.role ← REQUESTOR. Similarly, the role-based membership describes a trust relationship between two entities where anything trusted by the first entity should be trusted by the second. This can be denoted as RESOURCE.role ← REQUESTOR.role. In this case any requestor that is delegated a REQUESTOR role also has access to the RESOURCE through the role mapping. Link-based membership describes the mechanism of delegation within RT0 logic. This form of credential delegation can be illustrated through the following example. If entity A trusts B and entity B trusts C, then A should also trust C, therefore C has the credential to access A through implicit delegation of authority. This can be denoted in the form of RESOURCE.role ← RESOURCE.role1.role2, where REQUESTOR1 ← RESOURCE.role1 and REQUESTOR2.role2. The final credential type, membership intersection, denotes the mechanism by which members of RESOURCE.role result in the intersection of members of each entity role present in a given system. This mechanism can largely be viewed as credential aggregation of one or more roles to a given requesting entities. Alternatively described, a requesting entity would require partite sets of credentials to be delegated access to a role owned by another entity.

These credential membership mechanisms provide one of the primary components of ABAC aside from Credential chain discovery and Credential graphs.

### 1.11.4 Credential Chain Discovery

Credential chain discovery as mentioned is the mechanism by which an access control policy may be reasoned about by the entity that controls a resource. Given the set of RT0 credentials, there are three primary types of queries that can be made over credential chain relationships. These queries are illustrated as follows. The first query type is described, given a role denoted r and an entity E, determine if E is an element of *Role* →+ Entity delegations. Alternatively put, for a given set of linking roles associated with Entity E, is a role denoted r associated with entity E through the credential mechanisms illustrated in RT0 logic. The second query type is expressed in the follow manner: given a role denoted r, obtain all members in set S with the matching role r. The third query time is expressed as follows: given an entity E, determine all roles r associated with E. These three query types are powerful mechanisms that enable complex RT0 and ABAC access control systems to reason about credential based policies.

### 1.11.5 Credential Graphs

As previously described, a credential graph is a directed graph that represents the relationships between two or more attribute certificates and one or more attribute certificates bound to entity certificates. Each node in a credential graph C represents a role expression and every edge in the graph is denoted as a credential edge. One or more role credentials can be attributed to an entity credential. Credential edges are a super set of semantically related derived edges, which represent the semantic relationship between roles within the credential graph. Derived edges are the super path or the direct path between semantically linked role credentials within the credential graph. For each derived edge there exists one or more support sets. Each support set defines the set of paths between two credentials linked by the derived edge. Credential graphs mathematically support closure properties and have been proven for soundness [37].

### 1.11.6 Search Algorithms

Credential graphs and chain delegation only provide the framework at which policy can be derived for authorization. However, to provide the capacity to authenticate against an RT0 policy, mechanisms of searching and querying the policy must be established. For a credential graph C that contains a subset of entities E, role names R, edges e and a set of delegate chains DC, there exists subsets of derived edges that assert a mapping between E and some R by a path of a set of e. RT0 provides three different search or query algorithms over a credential graphs credential chain between different query elements. The first of these search algorithms is Backward Search. Backward Search takes a given credential role in the graph and extracts the delegate chains that link to credential entities in the policy set. The return result is a set of entities that, through some credential chain CD, has a derived path to the given query credential. The second search algorithm is Forward Search. Forward Search takes a given entity credential E and follows credential chains to locate all credential roles associated with the E through both credential edges. From the resulting set a derived edge can be created to assert the relationship between E and a given specific credential role r. The third and final search algorithm is defined as Bidirectional Search. Bidirectional Search takes an entity E and a credential role r and performs a Forward Search and Backward Search to define an explicit credential chain between E and r. Bidirectional Search can thus be used to perform authentication for a given Entity and a desirable role within the system. For example, for a user Alice972348234 which is an arbitrary unique code for this user and a role defined as UVIC.user, a Bidirectional Search is initiated with these parameters. If a delegate chain exists between the entity code Alice972348234 and UVIC.user, then we assert that this users entity code has a derived edge between the Entity credential Alice972348234 and the role UVIC.user. Authentication is thus possible through the combination of credential graphs, delegate chains and query algorithms.

### 1.11.7 ABAC Overview

Attribute-Based Access Control (ABAC) is the mechanism that implements much of the research by William H. Winsborough et al. and David W. Chadwick et al. [4] ABAC also builds upon the RT0 properties of credential graphs, delegate chains and search algorithms. In more practical terms, ABAC is an authorization and authentication system that supports delegate-based authorization, auditing, interoperability between implementations and mechanisms to control how much information is revealed by requestors and granters.

Like RT0, ABAC incorporates two primary types of credentials, Principals and Attributes. In this context, Principals are comparable to Entities in the RT0 model and Attributes subsume the role credentials. The primary distinction between role credentials and attributes is the layer of abstraction. More specifically, roles define a specific nature of existence, to represent roles, whereas an attribute can take on any meaning. The generic nature of attributes significantly expands the expressive power of ABAC to define complex policies not limited to roles.

#### 1.11.7.1 Principals

Principals may be representative of a single user or an organization such as a corporation. Principals, through credential chains, can exert ownership or control over one or more attributes. For example, a principal can be the subject of authorization or the granter and assert of rights of a requesting principal. Thus to devise a new attribute that semantically represents some element of an access control policy, a principal must create a new attribute. The principal that creates this attribute retains ownership of it. Once an attribute has been created, this attribute may be delegated to another principal. This process of delegation is comparable to giving rights to a user to access systems owned by the principal who created that certificate. This can be illustrated by the following example. A principal denoted UVIC, creates an attribute User in the semantic definition UVIC.User. In this case User is the attribute and UVIC is the principal that created the attribute User. It should be noted that the dot in UVIC.User asserts ownership of the User attribute. A user David is considered another principal denoted DAVID. Therefore, to create a delegation must take place. This is described by DAVID $\leftarrow$ UVIC.User, where this means that the principal credential DAVID is delegated the rights of a UVIC.User. ABAC supports many delegation types additional to Simple Delegation as seen in the previous example. Delegation will be described in greater detail in a later subsection. Attributes discussed in the next subsection provide the foundation by which more complex policies and delegation mechanisms can be discussed.

#### 1.11.7.2 Attributes

Attributes provide the foundation for building and deriving policies. Every attribute that is created must be created by a principal. An attribute credential or certificate is stored in an attribute certificate file with the extension .der. Within this attribute certificate a signature based relationship is defined. An example of such a signature based relationship can be seen in the following example: a811a3ab98a3437 40b2eb4f07e31200cf2bc9178.GetVersion ← a811a3ab98a343740b2eb4f07e31200 cf2bc9178.AllAPI. The leading elements that comprise the credential have been left out for brevity.

As observed, there exists a certificate signature extended by the attribute principal defined semantic definition GetVersion following by a signature key and the principal attribute AllAPI. The two definitions are related using an arrow "←". When this attribute is loaded into the ABAC access control libraries, the semantic definitions of the attributes, *GetVersion* and *AllAPI*, are the vertexes or nodes of the credential graph. Additionally the prefix for each semantic definition is asserted by a principal.

In this case a811a3ab98a343740b2eb4f07e31200cf2bc9178 is a signature of a principal that can be replaced with the principal name UVIC. Thus we obtain a comparable relationship UVIC.GetVersion ← UVIC.AllAPI which in turn describes the relationship; if a principal is delegated the attribute UVIC.AllAPI then this principal also has access to UVIC.GetVersion. This relationship is created by simple delegation defined as UVIC.GetVersion → UVIC.AllAPI, or simply UVIC delegates the rights or chain to the attribute UVIC.AllAPI for UVIC.GetVersion. UVIC can also delegate the rights to a principal, in this case a user who desires to access UVIC.GetVersion. This delegation is described asa811a3ab98a343740b2eb4f07e31200cf2bc9178.AllAPI ← 8544f78944576fcd91090d31003cb86c61c9559f. In this case the user has a signature 8544f78944576fcd91090d31003cb86c61c9559f and has to AllAPI. Substituting semantic definitions we get, UVIC.AllAPI ← DAVID, or David has access to UVIC.AllAPI.

This results in the comparable delegation, UVIC.AllAPI → DAVID, or UVIC delegates the UVIC.AllAPI rights to David. This set of relationships creates the delegate chain, UVIC.GetVersion ← UVIC.AllAPI ← DAVID. Finally incorporating

Table 1.1: Principal Attribute Association.

| | | |
|---|---|---|
| Direct Assignment | U has attribute AM.ListResources | AM.ListResources ← U |
| Delegation | All principals with attribute AM2.ListResources have AM1.ListResources | AM1.ListResources ← AM2.ListResources |
| Linked Delegation | Any principal P with the AM2.Linked attribute can assign the AM1.ListResources attribute by assigning the P.ListResources attribute | AM1.ListResources ← (AM2.Linked).ListResources |

RT0 logic principals we get the derived edge UVIC.GetVersion ← DAVID. These delegate mechanism provide the foundation for describing policies in ABAC. However, delegation is the foundational element to create policies suitable for distributed architecture.

#### 1.11.7.3 Delegation

Delegation provides the power to describe complex associations between ABAC credentials and formally define robust policies. More particularly, delegation is key to policy properties such as scalability and modularity. A well-defined policy incorporates delegation to minimize coupling between attributes through implicit delegation relationships. This in turn reduces the number of credential edges that must be managed in a given policy as the policy evolves over time. ABAC supports three primary kinds of delegation, two of which have already been covered. These types of delegation are Direct Assignment, Delegation or Simple Delegation and Linked Delegation. These types of delegation can be viewed in the Table below.

Linked Delegation provides a powerful mechanism of scalability and modularity in ABAC policy design. This form of delegation and the resulting link in the credential graph has the capacity to bundle large segments of policy together through implicit linking. More specifically, to consider the Linked Delegation example above, any principal with AM2.Linked is automatically delegated the rights ListResources. The credential graph does not have edges between these attributes and instead the edge is created dynamically at query time. The power of linked attributes can be further expanded by the example of doubly-linked attributes.

For example, (AM1.Linked).ListResources $\leftarrow$ (AM2.Linked).ListResources defines an attribute relationship where two disjoint segments of policy are combined without explicit credential edges. Linked Delegation allows for permissions to be aggregated at runtime instead of explicitly defined. This enables policy to be partitioned into sets and managed independently from one another.

#### 1.11.7.4 Mechanism

As mentioned ABAC policies are expressed in RT0 logic [22] which is a first order predicate logic that can be translated into datalog and are cryptographically signed. Additionally, relationships between attribute certificates and attribute signatures provide the capacity to form comprehensive access control policies. However, for ABAC to subsume the role as the primary access control mechanism in a distributed architecture, it must be supported with an interoperable and flexible implementation. ABAC is implemented into the open source libabac. The library libabac can be considered a generic mechanism such that it provides only the capacity to facilitate credential graphs, chain delegation, search and credential verification and validation.

#### 1.11.7.5 Implementation

The implementation of libabac can be separated into several primary components. These components are strongSwan, a C-based runtime library and two wrappers in Python and Java. The Python wrapper is automatically generated using Simplified Wrapper and Interface Generator (SWIG). The Java wrapper is a custom written wrapper that augments the C runtime libraries. Cryptographic functionality associated with credential validation and creation is delegated to the strongSwan open source Linux IPSEC libraries. This library provides the foundation for integration and interoperability with Python, Java and C-based architectures.

ABAC through its grammar, credential graphs, credential chains, delegation, search capabilities and library interoperability provides the functionality to satisfy all formal and informal requirements necessary to be used as a primary means for authorization and authentication in corporate and enterprise distributed architecture. The following two case studies demonstrate this capacity. The first case study analyzes the feasibility of integrating ABAC into a legacy architecture. The second case study analyzes the feasibility of integrating ABAC into a modern architecture.

# Chapter 2

# Case Study 2: ABAC Integration for Open Source Architectures

## 2.1 PlanetLab SFA Introduction

PlanetLab [30] is a modern, extensible and scalable infrastructure for the deployment of distributed systems that has been in continuous operation since 2002. Today it numbers over 1000 nodes at more than 300 sites worldwide. In order to scale, PlanetLab adopted a federated architecture in 2007. PlanetLab is an ideal candidate to evaluate the feasibility of integrating ABAC into a legacy architecture.

PlanetLab exhibits a distributed federated architecture and functionality resides in distinctly-defined silos. This case study segments analysis into several sections. In Section 2.1.1, the Slice-Federated Architecture is defined and analyzed. In Section 2.2, requirements for legacy access control mechanisms are defined and elicited. In Section 2.3, ABAC libraries and mechanisms are integrated into the SFA. In Section 2.4, the work required to transition legacy policy to ABAC policy is described and in Section 2.5 results and analysis of these processes are defined.

### 2.1.1 Slice-Federated Architecture

The Slice-Federated Architecture (SFA) layer is the legacy framework that enables PlanetLab to form a federated architecture built on the notion of mutual trust. An understanding of this legacy architecture is critical for evaluating the method, mechanisms and abstraction required to successfully migrate a legacy architecture to ABAC. Analysis of the SFA layer is divided into two parts. The first part involves evaluating inter-module and inter-server architecture for communication, trust and functionality dependencies. The second is evaluating existing access control policies for trends, correlation and potential hurdles for implementing an ABAC driven policy. To initiate the investigation of the feasibility of integrating ABAC into a legacy architecture such as the SFA, a general overview of the PlanetLab SFA is given.

PlanetLab's SFA architecture is divided into four primary server archetypes that each implement a different set of communication, functionality, and trust dependencies. These server archetypes are Slice Manager (SM), Aggregate Manager (AM), Registry Manager (RM) and Component Manager (CM). We note that the legacy CM has subsumed a previous server archetype denoted as the Node Manager (NM).

The archetypes in the SFA model correlate to common Cloud Computing platform archetypes. The SM acts as an inter-federate proxy that mediates operation requests to AMs; the SM is comparable to a proxy and shares functionality with a Eucalyptus Cloud Controller. The AM manages the allocation and mediation of resources that exist in a federate member's deployment; the AM is comparable to a Eucalyptus Cluster Controller. The RM in the SFA model acts as a domain registry for virtual and physical resources; the RM is comparable to a credential database. The RM exists as a Slice- and credential-driven access and policy database. The CM exists as a set of control resources that directly interface with node resources; the CM is comparable to the Eucalyptus Node Controller. Based on this initial comparison, this study investigates pre-existing legacy access control mechanisms and requirements.

### 2.1.2 Defining Legacy Access Control Mechanism Requirements

The SFA utilizes a legacy authentication and authorization module defined within the Python package sfa.trust. The package *trust* further defines eighteen Python modules and schemas used in credential validation. This package contains a total of 3365 lines of Python code and 672 lines of XML schemas used for credential validation. These numbers do not include additional modules present throughout the legacy code base that provide auxiliary authentication capacity in support of the trust package.

Two management interfaces have been introduced and annexed onto existing SFA policy, the command line SFI and its GUI wrapper Sface. The client interfaces with the SM, AM, NM and RM through Sface or SFI to instantiate and control Slice- or Sliver-based resources. SFI authorizes client-based requests by querying the RM.

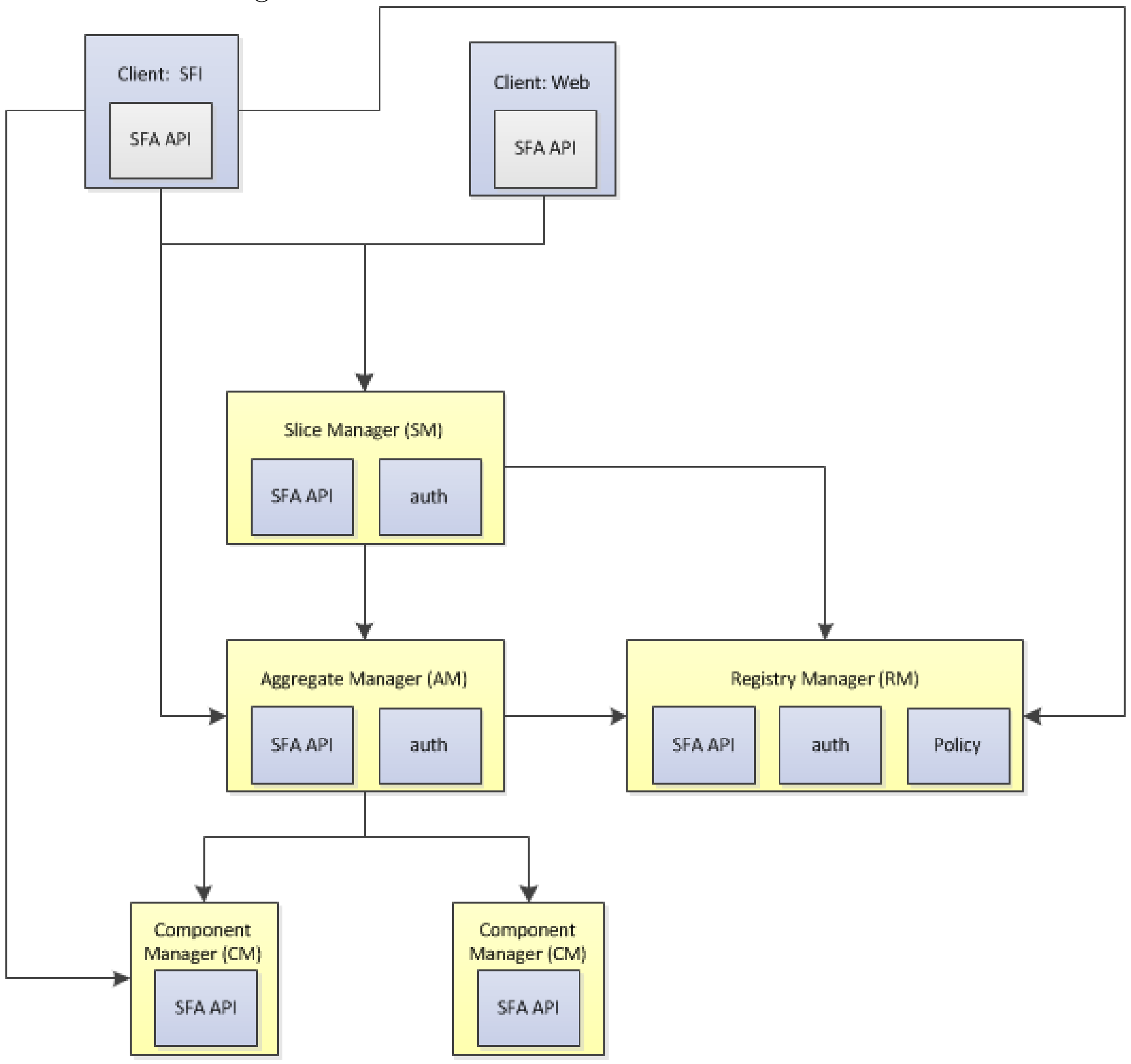
When federating two testbeds, the RMs may be accessed from either testbed by the AM, SM and SFI. Similarly, the SMs may interface with Managers found in the other testbed. The inter-Manager complexity increases proportionally to the number of sites that are involved in federation. To integrate into the SFA, an ABAC library must be developed to abstract deployment and role based dependencies into attribute relationships. Similarly, an abstraction is made for communication dependencies between modules into attribute relationships in ABAC policies. Trust is established between components by use of ABAC policy attributes and use of attribute hierarchies.

By abstracting these elements into ABAC policy, the legacy SFA's access control mechanisms can be decoupled from implementation. Specifically, policy can be factorized among each Manager (component) in the architecture. Thus each Manager within the system can exert a certain level of component level authority over the resources each provides. Furthermore, legacy access control architecture in the SFA also incorporates centralized access control per operational domain. The term *operational domain* encompasses the notion of some owner of a given resource within the distributed system. PlanetLab is a federated architecture; many computational resources and owners of those computational resources are shared within a trusted environment. Thus, the notion of trust is enforced through use of databases that replicate data among each site in which a set of PlanetLab resources are deployed. This can be seen in Figure 2.1 which illustrates both the legacy PlanetLab SFA architecture and the mechanisms involved in access control.

As observed in Figure 2.1, policy is centralized at the Registry Manager. The centralization of policy at the RM creates a dependency relationship between core infrastructure SFA architecture components and the Registry Manager. Simply put, dependency in architecture inhibits scalability and is a potential bottle neck between access control calls. A bi-directional trust relationship is also formed between components (Managers) in the architecture illustrated in Figure 2.1. Each Manager in the SFA architecture also establishes mutual trust between architecture components. For example, a specific SM denoted UVIC.SM explicitly trusts a AM denoted UVIC.AM. This explicit trust relationship can also put unnecessary constraints on other federated components, and in turn requires $n$ edges between $m$ child components. In this case study, the factorization of these trust relationships is demonstrated using the notion of mutual *distrust*.

Finally, each Registry Manager must retain data for every other *operational domain* within the federated architecture. Alternatively viewed, for each deployment

Figure 2.1: PlanetLab Access Control Architecture



within the SFA owned and operated by a given organization, a database must be kept up to date for every other federated entity within the PlanetLab distributed architecture. It is apparent how possible discrepancies and flaws in maintaining these policy sets may expose security flaws or vectors to be exploited. For example, there may exist discontinuity between users within PlanetLab; a user may be invalidated in one PlanetLab deployment but may be a legitimate user in another deployment. Similarly, one deployment may have record of an instantiation of resources in one deployment but not another. This problem can be defined within the scope of a *blanket policy*, or a policy that governs all resources in a distributed system. However, PlanetLab is a federated architecture and therefore *blanket policies* are not feasible and eliminate all notions of federation. To reiterate, federation is built on the notion of *sharing* resources for the purpose of gaining capacity. Many academic institutions use the federation capabilities of SFA to join small clusters of resources together to gain access to a much larger set of resources. Thus each institution may desire to impose different policies, within different countries and different legal frameworks. ABAC by its inherent set of capabilities, can facilitate federation, scalability, reconfigurability and fine grain policy without extensive management of localized databases. Section 2.3 of this study provides an in-depth analysis of these capacities. However, to understand how a legacy system such as the SFA can adapt an ABAC access control mechanism and policy, the architecture first has to be understood. The next section, Section 2.2 provides this analysis and introduction.

## 2.2 Legacy Architecture Design

As noted previously, the PlanetLab SFA is comprised of five primary Managers or Components. These components are: (1) Slice Manager, (2) Aggregate Manager, (3) Registry Manager, (4) Component Manager and (5) Client Interfaces including but not limited to Sface and SFI. Each of these Manager's exhibit two notions of tight coupling: tight coupling between components themselves and tight coupling between access control mechanism and component.

Tight coupling is problematic from an architecture perspective for two major reasons. The first reason is that it inhibits scalability of the architecture. The second reason is that it can inhibit architecture evolution. As technology, client requirements, system requirements, resource requirements and services evolve, so must the system architecture.

In particular the SFA API exhibit tight coupling of both access control mechanism and inter-component dependencies. Specifically, access control or authorization and authentication checks are performed both client-side and server-side. This duplication adds no security benefit as XML-RPC is used over an SSL tunnel. This duplication only imposes a limitation to system evolution and scalability. Upon more investigation it can be observed that both client and server verify SSH key credentials supplied through SSL XML-RPC, verify the key to sliver association and verify certificates. Both client and server also verify permission to execute a designated action and validate user GID.

ABAC eliminates this duplication by factorizing policy from access control mechanism by bundling proof of authorization with the client credential. The communication and structural hierarchies and hard-coded policy and legacy access mechanisms are abstracted into an ABAC nomenclature to enable the mapping of roles between federate entities and client credential to policy. However, more detail about this process and the work completed to integrate ABAC into a legacy architecture such as ABAC is defined in Section 2.3. Analysis of access control policy is the final step to formalizing understanding of the legacy SFA architecture.

### 2.2.1 Defining Legacy Access Control Policy Requirements

PlanetLab authorization policies are expressed as a series of specific terms that are used within the scope of this paper. These terms are, role, relationship and policy. With regards to the term role, role-based authorization has been studied and has be seen to be effective at access control [13] [14]. A role within PlanetLab is as an aggregate of capabilities. It should further be noted that PlanetLab roles may contain subsets of capabilities of other roles. A relationship is the mapping of a role to a capability. The term policy describes a collection of relationships. These relationships are many to many.

These three terms define the context by which policies in PlanetLab are created. To illustrate tight coupling within the given legacy access control framework, a user case is explored. This use case involves a client attempting to instantiate a sliver within PlanetLab. As defined, the AM validates the client credential and correct association between client and a slice allocated through the AM. The request to instantiate a sliver through the SM, results in a request being placed at the RM for authorization. Once validated the request then propagates back to the AM which again validates credentials by sending a request to the RM. Finally, the AM places a request at NMs to instantiate the request for resources.

As described, three primary aspects of legacy access control can be observed: (1) the access policy is spread over multiple Managers and domains, (2) different federates can implement different access policies, (3) roles can be mapped between federates resulting in a communication dependency.

To further illustrate the potential for an inter-dependency relationship the following example is provided, testbeds (A and B) that are federated. If a user from testbed A instantiates on testbed B, the SM of A must make a request to the RM of B to authorize the user. Client credentials must be transmitted across the federate and conform to expected credential standards of testbed B. This process forces static behavioral interfaces between federate members.

As noted previously and in addition to distinct Manager archetypes, PlanetLab's SFA layer also implements a set of distinct user roles with specific capacities and permissions within the federate environment. These roles are User, Principal Investigator, Administrator and Technical Contact. These roles implement two sets of distinct capabilities depending on the access context. A client may access PlanetLab resources through the SM or the Slice Federated Interface (SFI) command line tool. The SM Manager acts as a proxy on behalf of the client request. By this functionality, access is less restrictive as request specific parameters are obtained from the RM. The SFI tool is parameterized by the Client and therefore requires additional access control checks to be performed. These checks can verify access control relationships such as the association of a client with a specific resource. However, as the credential databases per each PlanetLab operational domain may differ, so can role definitions or the privileges associated with each role.

ABAC once again has the inherent capacity to subsume the association of a syntactic role definitions with multiple semantic meanings. This process is described in Section 2.3.

## 2.3 ABAC Policy

As mentioned, PlanetLab's existing access control policy is bound to the access control implementation by a functional means. Additionally, there exists explicit trust relationships between Managers based on function and role. The tight coupling of policy implementation and bi-directional trust relationships are each eliminated by one or more properties of ABAC. First, access control is separated from policy through the ABAC libraries that were developed and integrated into the legacy SFA layer. Additionally, ABAC provides access control capabilities by abstracting policy to a logic layer in memory over a generic access control mechanism. Secondly, inter-component dependencies are decoupled through the factorization of policy.

To implement an ABAC policy, existing architectural policy must first be formally defined. Once a formal policy has been devised, it can be abstracted, designed and expressed in ABAC grammar. The process and construction of the ABAC grammar for the PlanetLab SFA described in this section. However, a more rudimentary aspect to ABAC policy design is actually the factorization of policy among components, or more generally, the decoupling of components.

PlanetLab components or *Managers* each perform a specific role. The NM manages node resources. An AM manages NMs for a site or SFA deployment. The SM manages access to PlanetLab resources and acts as a proxy to other Managers. The RM manages credentials and policy for users, slices, components and authorities. Therefore, the *role* of a given Manager is the primary starting point for factorization. More specifically, each Manager can be viewed as an agent working on behalf of another Manager or Client. This inherent autonomy thus implies the delegation of an identity credential. ABAC relies on two kinds of credentials Principals and Attributes. In this case each Manager is delegated a X.509 certificate Principal credential. A Principal credential enables that entity to delegate and assert rights solely owned by that Principal. In this case, delegating Principal credentials to Managers enables each Manager can assert authority over the resources and services defined by its role definition. Similarly, by enabling the assertion of policy at a resource level owned by a specific Manager entity, tight coupling is eliminated. Tight coupling is eliminated because the authentication and authorization requirements defined within the policy of the Manager are no longer entity specific. A Manager may assert any number of requirements for authorization and authentication such as asserting membership to the federation or another role. In this way an Aggregate Manager may assert that only Slice Managers may access restricted resources. As long as the requesting client credential contains the PlanetLab.SM attribute in the signature portion of the certificate, access is granted. This relationship can be defined simply as Credential $\leftarrow$ PlanetLab.AM. Although this in principle can provide the needed policy requirements to facilitate federation and decouple components, this approach is by no means complete. The following sections detail the process of decoupling access control from policy and components from each other.

### 2.3.1 The Transition from Legacy To ABAC Access Control Mechanisms

The first stage to the integration and formalization of access control policy is defining the critical components of legacy architecture required to integrate ABAC access control libraries. With most legacy systems, extensively modifying modules, components or libraries can have significant and serious consequences. Resulting outcomes of such drastic modifications use near irreparable and costly damage of the legacy architecture. Therefore, great care is taken to identify a minimal dependency graph of modules that are involved in access control. Encapsulating the mechanisms within pre-existing API calls reduces impact to architecture. The first portion of code that was identified as the primary access control method was the *checkCredentials(*)*(* denotes a set of parameters) method which is utilized in almost all client/server API and related APIs. The checkCredentials method is modified to support ABAC related access control calls. Specifically, *authorize(*)* method is placed inside of the checkCredentials method and can be toggled on or off using the PlanetLab SFA configuration files. However, this method is not the only identified library that utilizes and requires access control. Table 2.1 provides a more comprehensive view of the changes within the legacy architecture required to facilitate the integration of ABAC.

Many of the observed modifications in the SFA architecture as noted in Table 2.1 define more utilitarian aspects of ABAC access control. A majority of the complexity associated with ABAC access control is subsumed into a custom Python module that provides an interface and facilities for ABAC. This Python *wrapper* interfaces to the C library libabac through a SWIG interface. The ABAC wrapper may be instantiated as an independent module within the legacy SFA allowing for the same access control object to migrate through various layers of encapsulation within SFA while maintaining a local unified policy of a given component. The libabac API has been further expanded within the Python wrapper to support a robust, fully encapsulated independent module. Details relating to the provided API can be observed in Table 2.2.

Additional facilities provided within the ABAC wrapper include algorithms to provide conversion between certificate credentials, auditing, logging, maintaining access

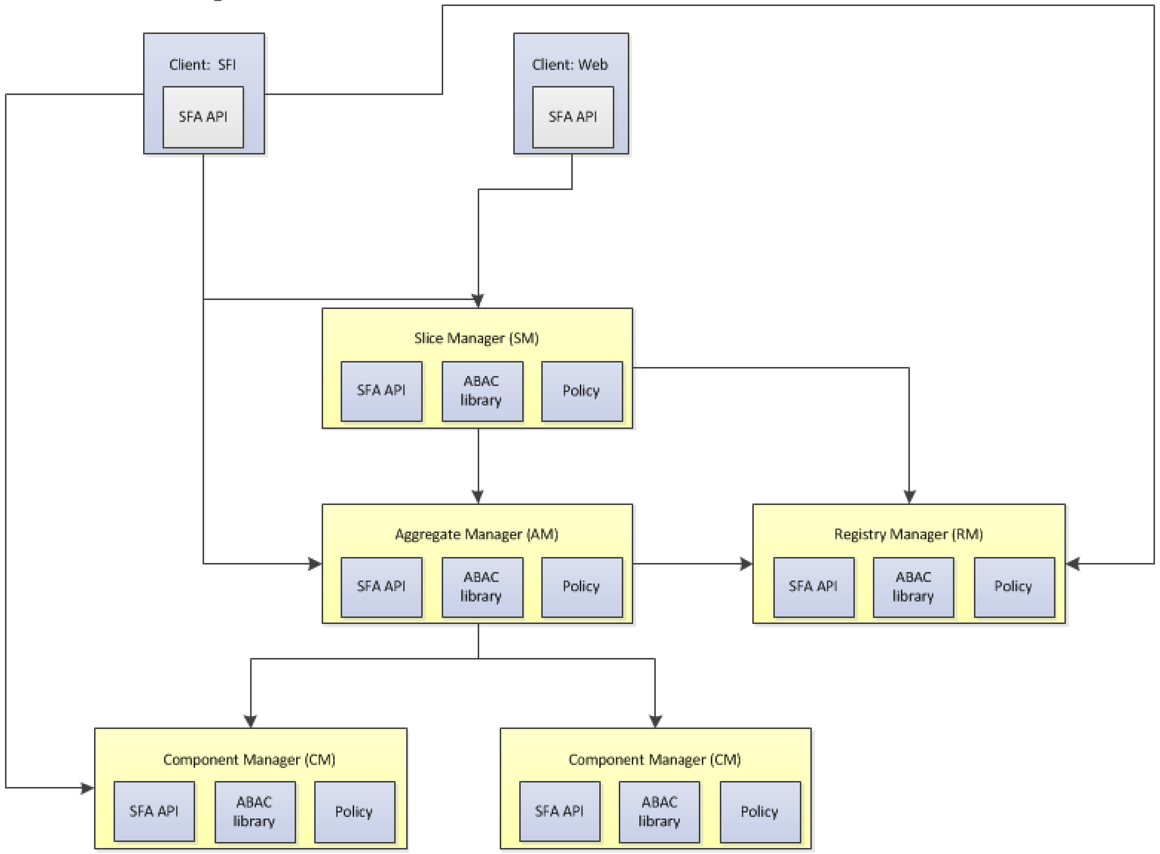
Table 2.1: Modified Modules and Objects.

| **Package/Module** | **Object** | **Modification** |
|---|---|---|
| sfa.client.sfi.py | class level | Modified several API calls to obtain ABAC credentials. |
| sfa.plc | SfaAPI | Modified generic API object to load ABAC credentials. |
| sfa.trust.auth.py | Auth | Primary integration point of ABAC. Added new ABAC object to encapsulate functionality. |
| sfa.util.api.py | BaseAPI | Modified BaseAPI object to load ABAC configuration and load ABAC credentials. |
| sfa.util.config | Config | Modified the Config object to support ABAC variables and resources. |
| sfa.config.gen-sfa-cm-config.py | class level | Extended default configuration parameters to incorporate default ABAC configuration and directories. |

Table 2.2: libabac Python Wrapper

| **libabac API** | **Python API** |
|---|---|
| query(role, principal) | authorize(self, role, principal) |
| Context(context) | cloneContext(self, context) |
| load_id_chunk(identity_chunk) | load_identity_chunk(self, identity_chunk) |
| load_id_cert(identity_file) | load_identity_cert(self, identity_file) |
| load_attribute_chunk(attribute_file) | load_attribute_cert (self, attribute_file) |
| load_directory( key_store) credentials() | load_directory(self, key_store) get_credentials(self) |
| Attribute(self, identity1, attribute, identity2) (and additional API calls) | create_attribute(self, identity1, attribute, identity2) |
| ID(id) | createIdentity(self, id) |
| load_id_chunk(identity_chunk), load_attribute_chunk(attribute_file) | load_attribute(self, identity, attribute) |

Figure 2.2: PlanetLab ABAC Access Control Architecture



control context and managing policy credentials. Furthermore, the external ABAC library is comprised of three primary Python modules. These are abac_manager.py, abac_wrapper.py and abac_logger.py. The abac_manager module implements credential management functions such as delegating new credentials to new clients. The abac_logger module implements access control logs for all valid, partially valid and invalid access control attempts. Finally, the abac_wrapper implements core access control API away from the libabac implementation. The functions provided through this API abstract complexities managing ABAC credentials such as authorization context, attribute certificates and identity certificates. The modules described exist external from the SFA implementation. This ensures that the legacy architecture and the access control mechanism can evolve independently of their respective dependents. This is critical, as policy can evolve independently from implementation requiring little or no modification of the legacy architecture over time. To provide a general overview of the modification to the architecture, the revised SFA ABAC architecture is defined in Figure 2.2.

As observed, policy is decentralized to each Manager in the SFA architecture. Similarly, each Manager has the capacity to assert the SFA API at each Manager location. For example, the Aggregate Manager can validate a requesting Slice Manager using its own policy and ABAC library. More specifically, any Slice Manager, if asserted by the appropriate attribute credential, can access resources governed by a given AM. This assertion is described as PlanetLab.SM $\leftarrow$ AM.AllAPI. Formally, this describes that the SM belonging to PlanetLab is delegated access to AllAPI governed by AM. The AM in this scenario may control which SM the AllAPI attribute may be delegated to. This use case provides only an example of a policy and access control scenario. Now that policy and implementation are delegated and integrated among Managers (components), an ABAC policy can be defined for each Manager reflective of their given role.

## 2.4 ABAC Policy Integration

Design of an ABAC policy for a federated architecture is divided into four phases. In phase one, PlanetLab policies are subsumed into an ABAC scalable policy through the creation of attribute hierarchies using a formal policy descriptor. This phase is defined as the mapping of Roles to Capabilities. In phase two, policy is factorized among Managers to eliminate tight coupling. This phase is described as the mapping of Capabilities to Resources. Phase three defines implicit delegation to form inter-system trust and attribute capacities for federation. This phase is defined as the Inter-Federate Mapping of Roles. The final phase involves integration of the proposed policies with the SFA architecture. This phase is defined as the Separation of Access Control Mechanism from Policy occurs.

Throughout this section policies are defined in RT0 logic and ABAC grammar. ABAC policies are illustrated in tables that define attribute delegations. Delegations can be read from right to left in first order predicate logic detailed in Section 1.11. Delegations are in the form of column1$\leftarrow${column2}. Each element of the column2 set is directly delegated to the element in column1.

### 2.4.1 The Mapping of Capabilities to Resources

SFA Managers in PlanetLab each implement a set of operations defined as API. Each Manager archetype (SM, AM, RM and CM) establishes relationships of explicit trust

Table 2.3: PlanetLab Manager Attributes

| **Actor** | **Delegated Assertions** |
|---|---|
| **(Principal: PlanetLab)** | **and Attributes** |
| PlanetLab.SM | AM.AllAPI |
| PlanetLab.AM | RM.AllAPI, CM.AllAPI, PlanetLab.SFI |
| PlanetLab.RM | RM.GetVersion, RM.resolve, RM.list, SM.GetVersion, SM.get_credential |
| PlanetLab.CM | No Delegations |
| PlanetLab.SFI | PlanetLab.SM, PlanetLab.AM, PlanetLab.RM, PlanetLab.CM |

between other SFA Managers. For example, a set of CMs establish an explicit trust relationship with an AM and vice versa.

These explicit trust relationships can be decoupled to allow for the SFA architecture to scale in an Ad Hoc manner. This is achieved by factorizing policy among each federate manager.

Access control policy is separated into bipartite sets to factorize these relationships of trust. The first set specifies access and the second set specifies control. Every operation at a specific Manager is under that manager's explicit control. Every operation that may be carried out in the federation is delegated by another Manager. The policy design results in Managers evaluating requestor federation membership status instead of maintaining explicit relationship of trust between specific Managers. Similarly, Manager credential access can be delegated as membership access or explicit access to distinct Managers. The factorization of policy can be observed in Table 2.3.

An example of the above factorization is defined in the following examples. A Slice Manager defined as SM1 asserts ownership of the operation CreateSliver through the Attribute SM1.CreateSliver. This assertion thus implies that an entity would require the specific attribute delegation SM1.CreateSliver to access operations at SM1. Similarly, if the same entity tried to access SM2 with the attribute SM2.CreateSliver, the access call would fail as the specific assertion was not made.

The next subsection addresses translating SFA's federate policy into ABAC driven federation mechanisms.

Table 2.4: PlanetLab Manager Attributes

| **Actor Principal** | **Delegated Assertions and Attributes Attributes** |
|---|---|
| PlanetLab | Federated |
| GENICloud | Federated |
| Orca | Federated |
| ProtoGeni | Federated |

### 2.4.2 Inter-Federate Mapping of Roles

The federation of the SFA architecture is achieved through implicit trust relationships and by the inter-federate mapping of roles. The creation and delegation of a common attribute to each federate member enables the extensibility of access control policy. By the explicit mutual trust of a common attribute, implicit trust can be dynamically negotiated using RT0 logic to form a mapping of first tier attributes or roles to another federate member's first tier attributes. These implicit delegate relationships are defined in Table 2.3.

The inter-federate delegations are pair-wise implicit delegate relationships of federate trust. These delegations are in the form of PlanetLab.User $\leftarrow$ (GENICloud.Federated).User. Similarly for GENICloud, there exists the relationship GENICloud.User $\leftarrow$ (PlanetLab.Federated).User. Thus, policy extensibility can be achieved without the requirement to define explicit trust relationships.

### 2.4.3 The Mapping of Roles to Capabilities

As seen previously in Section 2.2 PlanetLab incorporates four primary roles. Additionally, the context of integration also ensures that the semantics of role definitions are decoupled from syntactic definitions. The SFA legacy implementation defines a set of capabilities associated with a role. These capabilities are defined as operations that may be called over HTTPS and XMLRPC. Operations may evolve independently of semantic role definitions. Policy reconfigurability is enabled by decoupling role from operation through attribute hierarchies. Within the policy definition aggregate, attributes are defined such that they provide a common interface for roles and operations (capabilities). Attribute hierarchies can be considered as layers of internal semantic relationships or as a set of layered hashmaps. The hashmap keys are the attribute identifiers and the corresponding value can be either a capability or

Table 2.5: PlanetLab Role Attributes

| **Actor (Principal: PlanetLab)** | **Delegated Assertions and Attributes** |
|---|---|
| PlanetLab.User | SM.AllAPI |
| PlanetLab.TechnicalContact | No Delegations |
| PlanetLab.PrincipalInvestigator | PlanetLab.User |
| PlanetLab.Admin | PlanetLab.User |

another key to another hashmap. The attribute hierarchy abstraction allows for both role and operations to evolve independently as the infrastructure evolves. In the case of the SFA layer, two-tiered attribute hierarchies are defined. The first tier is a role syntactical definition and the second is an aggregate of potential capabilities.

The first tier of attribute hierarchies can be viewed in Table 2.5 and Table 2.6. PlanetLab implements a single quaternary set of syntactic role definitions. As observed in Section 2.2, these syntactic definitions are responsible for mapping SM and SFI relative roles. To devise the ABAC role mechanisms, semantic and syntactical definitions of roles that the SFA layer implements are elicited and abstracted them to a cryptographic data structure. This data structure is also considered as a basket of capabilities in the ABAC policy set.

SM relative role attributes are a minimal attribute set that directly correlate to legacy role definitions. As seen in Table 2.5, tier two aggregate hierarchies are delegated to the first tier role attribute definitions.

However, SFI relative roles implement a disjoint attribute hierarchy set such that some roles are directly bound to the SFI client credential as seen in Table 2.6. In this way complexity is balanced between over abstraction and manageability of policy.

Two tier attribute aggregate sets are defined to facilitate the dynamic evolution of operations and capabilities in the SFA architecture. These attributes can be observed in Table 2.7. This table illustrates how sets of operations are aggregated to one of four aggregate attributes: SM.AllAPI, AM.AllAPI, CM.AllAPI and RM.AllAPI. It should be noted that all attributes are asserted by the corresponding SM, AM, CM and RM principals that correlate to a SFA Manager. This policy structure is one component to facilitate the factorization of policy.

These ABAC attributes comprise the core abstraction of legacy access control policy into an ABAC grammar. ABAC attributes also decouple access control mechanism from implementation and components from each. However, in Section 2.5

Table 2.6: PlanetLab Attribute Hierarchies

| **Actor (Principal: PlanetLab)** | **Delegated Assertions and Attributes** |
|---|---|
| SFI.User | SM.AllAPI, AM.AllAPI, CM.AllAPI, RM.GetVersion, RM.get_credential, RM.resolve, RM.list, RM.create_gid, RM.update, RM.remove |
| SFI.TechnicalContact | SM.GetVersion, SM.ListResources, SM.ListSlices, AM.GetVersion, AM.ListResources, CM.GetVersion, CM.ListResources, RM.GetVersion, RM.get_credential, RM.resolve, RM.list, RM.create_gid, RM.update, RM.remove |
| SFI.Admin | SM.AllAPI, RM.AllAPI, AM.AllAPI, CM.AllAPI |
| SFI.PrincipalInvestigator | SM.AllAPI, RM.AllAPI, AM.AllAPI, CM.AllAPI |

Table 2.7: PlanetLab Operation Attributes

| **Actor (Principal: PlanetLab)** | **Delegated Assertions and Attributes** |
|---|---|
| SM.AllAPI | SM.GetVersion, SM.ListResources, SM.CreateSliver, SM.RenewSliver, SM.DeleteSliver, SM.SliverStatus, SM.ListSlices, SM.get_ticket, SM.start_slice, SM.stop_slice, SM.reset_slice |
| AM.AllAPI | AM.GetVersion, AM.SliverStatus, AM.CreateSliver, AM.RenewSliver, AM.start_slice, AM.stop_slice, AM.reset_slice, AM.DeleteSliver, AM.ListResources, AM.get_ticket |
| CM.AllAPI | CM.GetVersion, CM.SliverStatus, CM.start_slice, CM.stop_slice, CM.DeleteSliver, CM.reset_slice, CM.ListSlices, CM.redeem_ticket, CM.ListResources |
| RM.AllAPI | RM.GetVersion, RM.get_credential, RM.resolve, RM.list, RM.create_gid, RM.register, RM.update, RM.remove |

Table 2.8: Impact: Depth and Breadth of Execution Traces

| **Managers** | **Legacy** | **ABAC** | **% Complexity Reduction** |
|---|---|---|---|
| SM | 309 | 3 | 99.03% |
| AM | 309 | 3 | 99.03% |
| RM | 309 | 3 | 99.03% |
| CM | 309 | 3 | 99.03% |

the ABAC policy design is evaluated with respect to impact, scalability, modularity, reconfigurability and extensibility.

## 2.5 Impact

Impact on the legacy infrastructure must be minimized to ensure existing functionality is preserved over its lifecycle. The criteria for impact is formalized by evaluating depth and breadth of execution traces through the authorization logic per Manager. The execution metric is devised by adding subsequent function calls together. If a function call contains multiple sub-functions, the resulting recursive function call evaluation is added. For example, if a function call contains two functions of depth two, the higher order function takes on a value of eight. The same code trace method is conducted on the ABAC integration but do not count calls to the external ABAC libraries that reside outside of the SFA implementation. It should be noted that the original access control code has been subsumed into the ABAC libraries. The results can be observed in Table 2.8.

As observed from Table 2.8, the ABAC implementation reduces number of access control calls in the legacy code base by 99.03%. This code is subsumed into the ABAC external access control libraries. By this result it can observe that impact is approximately 0.97% of authorization logic per Manager. From this result it can asserted that the ABAC integration subsumes complexity of authorization logic into a shared library. It should be noted that the primary evaluation is impact on the legacy architecture and not dependencies on external modules. By this result it can asserted that the ABAC implementation minimizes impact on the legacy architecture.

Table 2.9: Reconfigurability: Change in Credential Set Size

| | Policy Size (bytes) for N Managers | | | | |
|---|---|---|---|---|---|
| **Manager** | **0** | **5** | **10** | **15** | **20** |
| SM | 28021 | 30492 | 32963 | 35434 | 37906 |
| RM | 33427 | 35877 | 38327 | 40777 | 43227 |
| CM | 27068 | 29544 | 32021 | 34498 | 36974 |
| AM | 26584 | 29073 | 31562 | 34051 | 36540 |

### 2.5.1 ABAC Policy Scalability and Modularity

The integration of ABAC into a legacy architecture and creation of a new policy is evaluated by the two methods. In the first method reconfigurability is evaluated based on the change in size of the credential set that comprise the ABAC policy when new operations are added or the inclusion of a new role. In the third method extensibility is evaluated by analyzing the change in size of the ABAC credential set when new federate members join the federation. For extensibility two cases are considered. The first case assumes that each new Manager has an identical policy set as to a previous Manager in the same class. In the second case, each Manager is assumed to have a unique or distinct policy compared to other Managers in the same class. These two cases can be considered boundary cases.

If the rate of policy set size scales linearly or sub-linearly, then the complexity of the ABAC policy also scales linearly. Thus if policy can maintain linear growth as complexity increases then the policy could remain maintainable. Specifically, policy must remain maintainable for factors of reconfigurability, scalability and extensibility. Each of these factors are evaluated in this section. Finally, it is considered that the lower the percent change in trace length, the less legacy code was touched and the greater separation of ABAC access control mechanism from legacy infrastructure.

### 2.5.2 Reconfigurability Evaluation

Reconfigurability of the ABAC policy defined in this study is illustrated through the increase in credential policy size. Increase in complexity of the policy is denoted through the percent change relative to the core structure of the policy. The results can be observed in Table 2.9.

As observed, the results illustrate linear growth in policy complexity as the infrastructure scales linearly. However, it can be noted that the rate of growth for the

Table 2.10: Extensibility: Change in Credential Set Size

| | Policy Size (bytes) for N Unified Managers | | | | |
|---|---|---|---|---|---|
| **Manager** | **0** | **5** | **10** | **15** | **20** |
| SM | 2542 | 5084 | 7626 | 10168 | 12710 |
| RM | 2542 | 5084 | 7626 | 10168 | 12710 |
| CM | 2542 | 5084 | 7626 | 10168 | 12710 |
| AM | 2542 | 5084 | 7626 | 10168 | 12710 |
| Normalized Difference | 2542 | 1016.8 | 762.6 | 677.9 | 635.5 |
| Percent Growth | - | -60% | -22.67% | -11.11% | -6.67% |

SM is 6.52%, the RM 5.67%, the CM 6.69% and the AM 6.81%. It is observed that as the number of operations per manager doubles, the complexity of the policy only increases by an average of 6.42%. From these results, it can be asserted that the ABAC policy enhances reconfigurability of a legacy distributed infrastructure.

### 2.5.3 Extensibility Evaluation

Evaluation of the ABAC policy to enable extensibility is illustrated by analyzing the change in policy set size by adding new members to the Federation. Analysis is divided into two scenarios that can be considered boundary cases. The first boundary case assumes that the addition of the new Manager implements the same policy as all existing managers and thus the policy certificates can be reused. The Second scenario, assumes every new Manager implements a mutually exclusive policy to all Managers already within and added to the system. In this case, policy certificates cannot be reused and increases the credential graph complexity in the ABAC policy.

For these calculations the creation of new implicit trust relationships required to form the mapping of roles across federate members is evaluated. The results for the first part can be observed in Table 2.10 and the second part of the evaluation in Table 2.11.

As observed in Table 2.10 the complexity of the ABAC policy grows at a rate less than a linear growth as new Members join the federate and existing Managers trust the new federate Manager. In this scenario, attribute certificates in the credential chains can be shared among each Manager. This results in a policy size increase of 508.4 bytes per manager. In Table 2.11 the complexity of the ABAC policy grows as

Table 2.11: Extensibility: Change in Credential Set Size

| | Policy Size (bytes) for N Distinct Managers | | | | |
|---|---|---|---|---|---|
| **Manager** | **0** | **5** | **10** | **15** | **20** |
| SM | 5614 | 28492 | 79332 | 155592 | 257272 |
| RM | 5614 | 28492 | 79332 | 155592 | 257272 |
| CM | 5614 | 28492 | 79332 | 155592 | 257272 |
| AM | 5614 | 28492 | 79332 | 155592 | 257272 |
| Normalized Difference | 5614 | 5698.4 | 7933.2 | 10372.8 | 12863.6 |
| Percent Growth | - | 1.50% | 39.22% | 30.75% | 24.01% |

a function of multiple mutually exclusive policy sets. In this scenario, it is assumed that each new Manager implements its own policy and thus no attribute certificates in the credential chains can be shared. Thus, it can be observed that the linear growth of 5084 bytes is attributed to implementing a complete new policy per manager.

When comparing Table 2.10 to Table 2.11, the shared attributes reduce the required policy size by 90% per manager. These results demonstrate that for each new distributed infrastructure that, join the federation, the rate of complexity increase is less than an the *m:m* mapping of explicit trust relationships. By these results, it can be asserted that the ABAC policies improves extensibility of the legacy distributed architecture.

## 2.6 Conclusion

This case study investigated the feasibility of integrating ABAC into a legacy architecture. As demonstrated, it *is feasible* to integrate ABAC into a legacy architecture. Furthermore, ABAC, through its inherent properties and capacities, can enhance a legacy architecture by factoring policy from implementation. Additionally, it can be observed that components may be decoupled from each other to form mutually untrusting entities. This in turn results in a more scalable, modular and reconfigurable architecture where access control places no limits upon the architecture and were policy may evolve independently from access control mechanism. Therefore, it is concluded that ABAC is a suitable mechanism for fully subsuming the access control requirements of an enterprise legacy architecture. The question still remains if ABAC can be a suitable mechanism for fully subsuming the access control requirements of a modern architecture that is continually evolving, growing and changing access control requirements. The investigation of this question and a case study is found in Chapter 3.

# Chapter 3

# Case Study 2: ABAC Integration for Proprietary Architectures

## 3.1 Introduction

One primary challenge associated with proprietary distributed systems is intellectual property rights. In particular, many proprietary and open source licenses are incompatible. For example, the Mozilla Public License and the Apache License are incompatible with GPL-2. These licensing restrictions may make adopting some access control mechanisms into open source systems and proprietary systems difficult within existing legal frameworks. To illustrate this issue in a relevant context, the following case study investigates the integration of Attribute-Based Access Control (ABAC) into a proprietary commercial distributed system known as the GeoAnalytical Grid Engine (G2E).

The G2E framework is a proprietary framework that implements licensing such that any modifications or changes to the code base results in a transfer of ownership from the software library to be integrated to the owner of the distributed system. This inherent legal constraint is incompatible with the open source licensing of the ABAC libraries. Therefore, the full access control features embedded within the ABAC grammar and library must be exposed to this framework without ever modifying the architectures implementation. This restriction may significantly inhibit scalability, modularity and manageability of the architecture and the access control framework. The question is, how can ABAC be integrated into such a system while satisfying these legal requirements? This is by no means an easy question to answer; however logically the core requirement that ABAC must be integrated as a separate system from the code base. Research and legal analysis have demonstrated that when interfacing systems and components through APIs, legal constraints of licenses do not interfere [15]. The distributed system is then considered as a collective work and is treated as separate modules under copyright law. Therefore, the requirement is that ABAC must be integrated as a separate module from the distributed system in question and provide no direct hook into running processes of that system. This solution solves the problem of licensing restrictions but raises issues of scalability, modularity and manageability remain. This case study investigates how ABAC libraries and policies can be adaptive to satisfy these requirements in a proprietary system.

Section 3.2 describes the proposed access control architecture. Section 3.3 investigates mechanisms to support inter-component communication and proposed API to facilitate the architecture. Section 3.4 describes a set of proposed policies required by the G2E framework to support corporate customer requirements. Section 3.5 provides an overview of of the qualitative results based on this case study. Finally in Section 3.5 conclusions are made based on the results from this case study.

## 3.2 G2E Introduction

### 3.2.1 Access Control Architecture

As noted in the previous case study, ABAC is a certificate-driven authorization and authentication system that relies on attribute certificate chain delegation to establish relationships between contextual symbols that are asserted through certificates. Policies are constructed by assigning semantics to each attribute certificate and then inter-connecting these attribute certificates to establish relational referencing between semantics. The result is a set of attributes linked through certificate assertions that describe or relate to a set of policy requirements for a given distributed system. In this context, policy is a data map over a generic library that facilitates policy querying, credential verification and credential validation.

The first case, found in Chapter 2, demonstrated that it is indeed feasible to integrate ABAC into a legacy architecture. As noted, ABAC has the potential to subsume all access control requirements of a modern architecture. However, to assert this claim, similar analysis is performed in this case study. It should be noted that this case study differs from the case study in Chapter 2 in one fundamental way:

the PlanetLab architecture was an open source project and the G2E architecture is proprietary. This problem is outsides of technical feasibility issues. Specifically, it has been demonstrated that many open source licenses are compatible with each other. However, in general, open source licenses (apart from academic licenses such as Free BSD) are incompatible with proprietary licenses. This case study defines methods such that technical challenges of integrating ABAC into a modern proprietary architecture can be overcome and how licensing restrictions for many enterprise systems can be addressed.

### 3.2.2 Avoiding Licensing Restrictions

For many years licensing restrictions in open source and proprietary software applications have been researched. One solution proposed by Dr. German et al. to overcome issues of licensing restrictions is through component interfaces [15]. More specifically, if two software components can be interfaced through an API and in some cases distributed as separate modules, the resulting aggregation is not considered a derivative work. Not being classified as a derivative work is necessary to ensure there is a clean separation of licensing terms. Based on this core principle, the formulation of an approach to integrating ABAC into G2E materializes.

To integrate ABAC into a proprietary system, the facilities and capacities of ABAC must be implemented as an external component. As seen in the first case study in Chapter 2, the ABAC libraries and policy can be directly integrated into the SFA framework. However, when considering the G2E architecture, this is not possible. Hence, component specific or localized access control is not feasible. Therefore, all complexity associated with policy design, deployment, and querying must be done in a separate component that may be located either remotely or locally. The formal design process is described in Section 3.3. Unlike the first case study, the G2E architecture is an architecture under constant evolution. Furthermore, it has been designed with no mechanism or system to facilitate policy, access control or client management. Therefore, the G2E architecture provides an excellent starting point to evaluate the integration of ABAC into modern architecture from the ground up. The next section, Section 3.3, details the process to design this access control architecture.

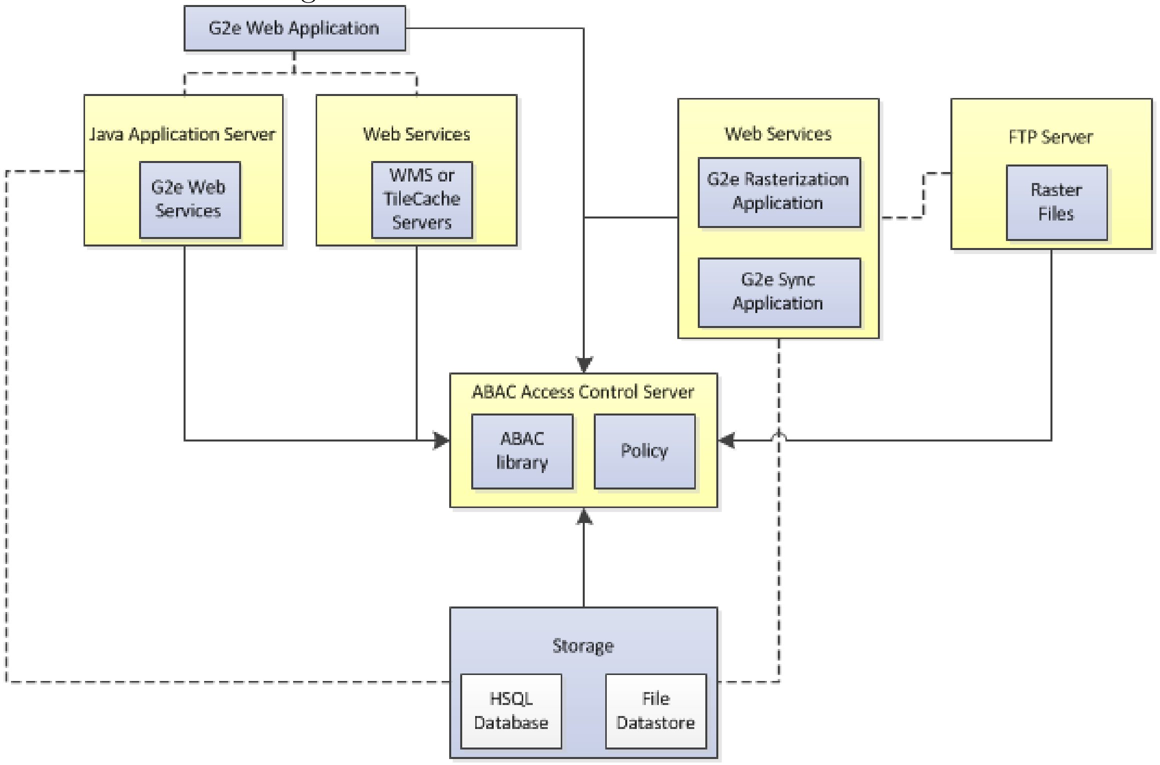
## 3.3 G2E Architecture and Design

Access control designs for the G2E architecture can be classified into two primary categories: User Context Access Control (UCAC) and Server Context Access Control (SCAC). From previous work integrating ABAC into distributed architectures, SCAC has demonstrated greater potential for scalability by factorizing the access control context. UCAC aggregates policy to a centralized location which is usually an authentication server, and SCAC distributes policy to each component within a distributed architecture. By distributing policy between each component, policy is factorized, resulting in smaller policies to manage and a distribution of demand for access control resources.

To illustrate these concepts in a more concrete way, consider a website that requires username and password authentication. This website provides a single authentication page for all of its components and features. Furthermore, this website employs a database that contains all registered users of the site. Therefore, when a user authenticates against the credential store or database, the entire data set is analyzed. Once authenticated the user may access any of the website resources. This form of access control is classified as UCAC. An example of UCAC is as follows, consider a set of web services where each component(web service) or feature has an authentication page. If a user desires to access the upload file resource, the user authenticates with this resource. Once authenticated, the component generates a context token denoting the users credentials within the system. If this user accesses a different resource, the authentication token is passed to this new resource and the user is authenticated. Furthermore, each component within the architecture can harness this same functionality to authenticate among themselves to assert such mechanisms as mutual trust. Server contextual assertion is critical for establishing federated systems. Federated systems are such that mutual trust can be established between potentially untrusting components through certificate verification and validation. These features also enable servers to act as a proxy on behalf of a user through delegation similarly to the PlanetLab case study.

ABAC may be used as either a UCAC or SCAC authentication architecture but is better suited for SCAC due to properties in its grammar and implementation. Therefore, each component of the G2E architecture will authenticate against the ABAC access control server asserting both identity and permission to carry out a specific operation. This prevents a hijacked component from carrying out malicious

Figure 3.1: G2E Access Control Architecture



operations that vary from intended operation. For example, the Java Server in G2E which houses Rasterization and Sync applications will have to authenticate against the Storage server's policy to assert they have permission to write to the database. This prevents a compromised G2E Web Server from running queries against the database. To illustrate this design, the G2E architecture is defined in Figure 3.1.

As observed in Figure 3.1 each of the G2E components are decoupled from one another. Therefore, an ABAC driven policy would likewise be minimized and decoupled from bi-directional trust relationships for each components API. The proposed ABAC implementation is partitioned into three components: (1) ABAC-Server, (2) ABAC-Proxy and (3) ABAC Client Interface.

The ABAC-Server is the primary access control server in the proposed architecture. This server provides one primary interface that utilizes HTTPS and a RESTful request/response API. The ABAC-Server is responsible for maintaining access control policies for a given *access control context* and performing access control queries using the ABAC libraries. An access control context is an instance of ABAC and associated utility libraries that facilitate authentication and authorization requests. Specifically, every component in a given architecture can request or *register* with the server to request space to maintain an access control policy for the requesting component. A clean RESTful API is provided to integrate with this server and is described later in Section 3.3.2.

The next major component in the ABAC architecture is the ABAC-Proxy. The ABAC-Proxy can either be directly integrated within a proprietary architecture if licensing restrictions allow, or it can be instantiated on a server (local or remote). The ABAC-Proxy contains all necessary libraries and utility functions to easily formalize HTTPS RESTful requests to the ABAC-Server. Additionally, the ABAC-Proxy provides libraries for processing an API response. It should be noted that the ABAC-Proxy is not required to interact with the ABAC-Server. Any mechanism that implements HTTPS RESTful requests may make calls to the ABAC-Server. However, utilizing the ABAC-Proxy simplifies implementation through a clean API and prevents direct attacks on the ABAC-Server. It should be additionally noted that the ABAC-Proxy also supports an XML-RPC interface over SSL. The XML-RPC interface over SSL is utilized when the proxy is run as a separate module or externally to an architecture such as G2E. A third module called the ABAC Client Interface forms XML-RPC communication architecture with the ABAC-Proxy from within a proprietary architecture such as G2E.

The ABAC Client Interface is a lightweight module that implements a basic XML-RPC interface to the ABAC-Proxy. The ABAC Client Interface (ABAC CI) is licensed under an academic license and can be freely incorporated into any architecture whether open source or proprietary. The ABAC CI supports an API that is mirrored by the ABAC-Proxy and the ABAC-Server to maintain consistency in functionality throughout the access control architecture.

Finally, these three architectural components that comprise the ABAC access control framework for G2E can be arranged in numerous configurations based on the demand and requirements of a given architecture. The first of these configurations can be observed in Figure 3.2.

The architectural configuration observed in Figure 3.2 illustrates a centralized access control server. Each component, Database Server and two G2E-GUI instances communicate with the same access control server. Each component instantiates the ABAC CI object which then communicates with a local ABAC-Proxy instance that runs in the same physical server. Each ABAC-Proxy interfaces with the ABAC-Server. The ABAC-Server constrains a unique Access Control Context (ACC) for each registered component. Concerns become apparent relating to scalability as more

Figure 3.2: G2E ABAC Configuration: Centralized

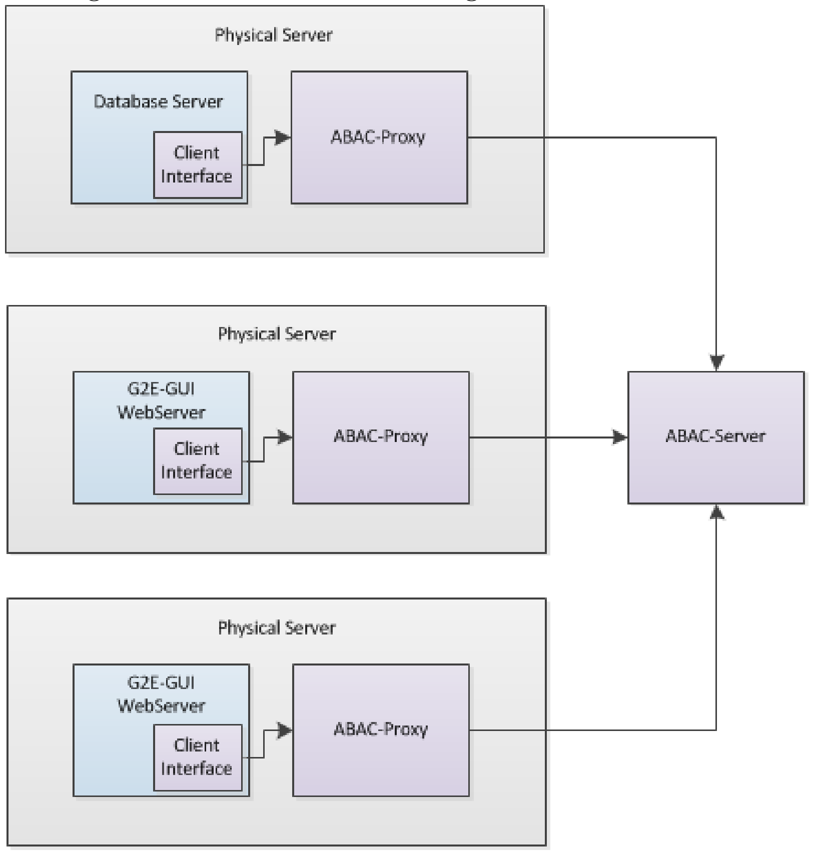
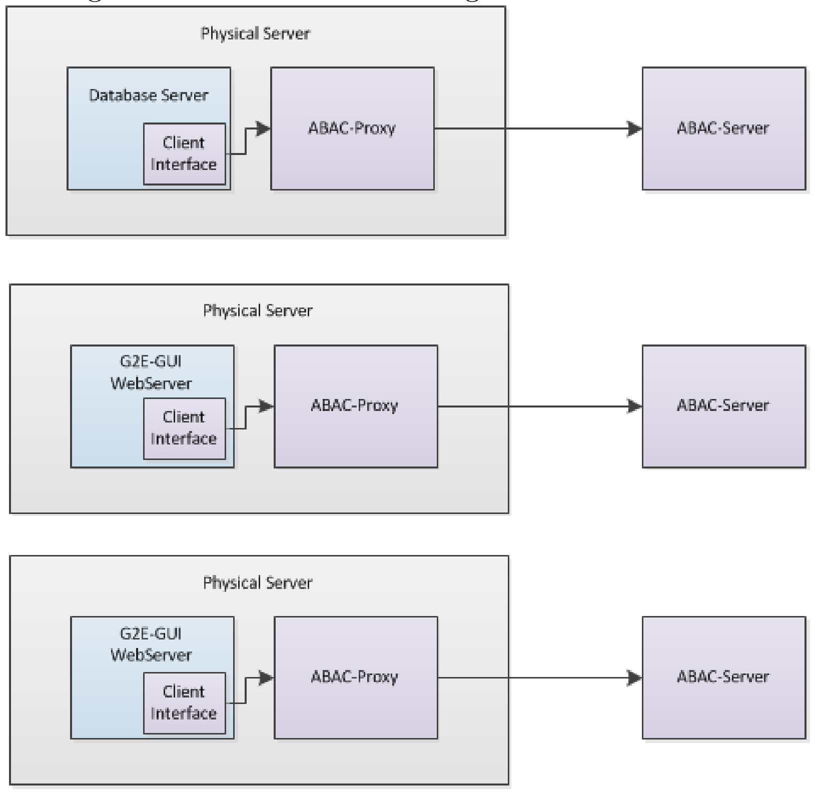


Figure 3.3: G2E ABAC Configuration: Distributed

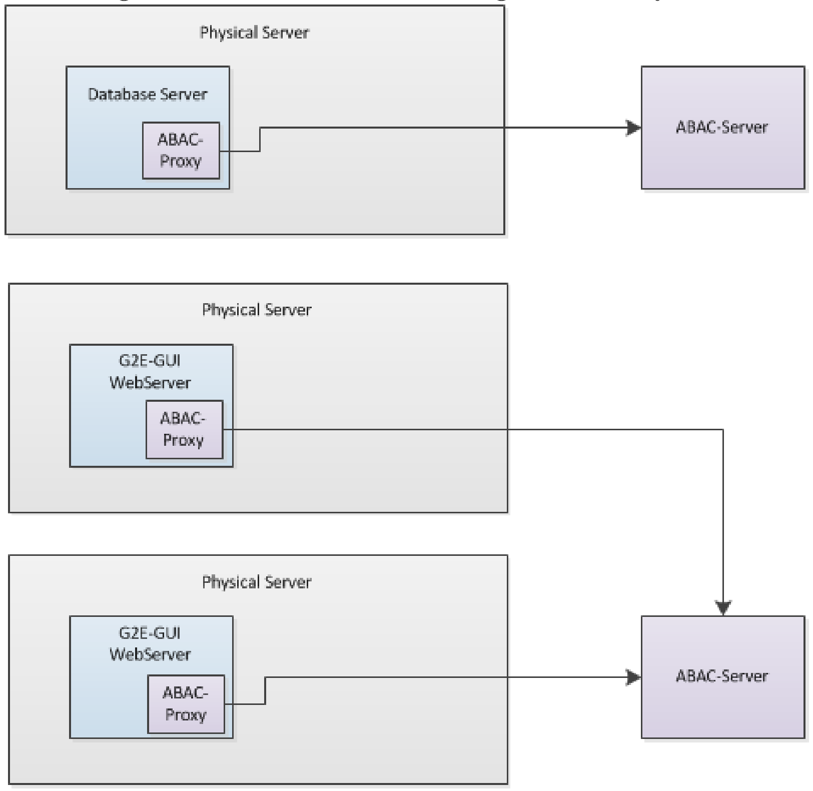


components are added. This architectural concern can be eliminated by following the next architectural design in Figure 3.3.

As observed in Figure 3.3, each component places requests to a unique ABAC-Server. This architectural design demonstrates that scalability can be achieved by decoupling access control clients from the access control server. It should be further noted that the factorization of policy implemented in case study 1 in Section 2 is maintained through ACCs. One concern that arises over the architecture proposed in Figure 3.3 is demand for compute resources. Specifically, as the architecture scales so does the demand for physical servers. By factorizing policy among many ABAC-Servers, the potential for wasting compute resources is high. Therefore a third access control architecture is proposed as noted in Figure 3.4.

Figure 3.4 illustrates a hybrid approach based on the last two configurations, where any component may chose to register with any given ABAC-Server. Thus an architecture can be designed to satisfy any architectural requirements. Additionally noted in Figure 3.4, the mechanism by which the ABAC-Proxy can be directly integrated into

Figure 3.4: G2E ABAC Configuration: Hybrid



the G2E architecture is also illustrated. However, this configuration is only provided for demonstration purposes and is not feasible due to licensing restrictions.

One final aspect of the ABAC architecture not depicted in the diagrams above is the locality of the ABAC-Proxy. In all architecture above, the ABAC-Proxy resides on the same physical machine as the client component of the G2E architecture. The ABAC-Proxy may be relocated to any physical and remote server as long as the ABAC CI correctly points to the right proxy and the ABAC-Proxy points to the right ABAC-Server.

Based on the architecture as described, a more detailed analysis of Figure 3.1 can be investigated. In Figure 3.1, dotted lines denote the functional dependency between components and the solid arrow lines denote functional dependency on a given authentication request. The authentication flow is illustrated by the following example. If the Java Application Server desires to make a query on the Storage Server, the Java Application Server (JAS) makes a request through the Storage Server API. The Storage Server obtains the Java Application Servers credential through operation request. The operation identifier and the JAS credential are submitted to the ABAC Access Control Server (ACS) with the Storage Server credential. The ACS queries the ABAC policy and a response is given back to the Storage Server if the JAS has permission to run that particular query. In addition to the denoted access control flow example, user credentials are similarly processed. For example, if a user makes a request to query for some data, the user submits their credentials at the G2E Web Application along with their desired request. As in the previous example, the G2E web application obtains a response from the ACS if the user has permission to execute the requested operation. Once the request has been validated the JAS acts on the user's behalf to execute the query. The JAS authenticates with the Storage Server as per the first example. These examples illustrate how policy is factorized among each component. Specifically, when the G2E Web Application authenticates against the ACS, only the policy for the G2E Web Application is analyzed. The JASs policy is similarly factored from the G2E Web Application. Therefore, each component's policy may be modified without being concerned about functional policy constraints between other components.

The final significant alteration to the G2E architecture to facilitate ABAC, is modifying the graphical user interface (GUI). To integrate ABAC into the G2E GUI, the architecture was analyzed and several major modifications were implemented. The G2E GUI component was analyzed for the API used to interact with other components within the architecture. The identified API or interface was the Query Java class. This class facilitated a generic mechanism for formulating queries on the G2E data set which included satellite images. The satellite images contained metadata that described the image contents including region, location and classification. The Query class additionally provided a mechanism such that multiple requests could be combined using Boolean logic including, but not limited to, $AND$,$OR$ and $NOT$. This metadata proved essential for associating request data with an access control policy. Therefore, the servlet that handles query processing was modified to include an instance of the ABAC CI module that makes authentication requests based on the given query composition.

If a response from the ABAC CI *authenticate* method returns true then the algorithm proceeds to server the request. If the *authenticate* method returns false, an exception is thrown based on the given request criteria. In addition to location-based request criteria, the primary stakeholder in the G2E project also requested for authorization based on User location and subscription status. The formal policy as requested is detailed in Section 3.4.

The final primary modification to the G2E system was the notion of binding a Query action to a client credential. Therefore a login/logout process was implemented. In the rudimentary form of this mechanism, a client may upload an ABAC certificate into the G2E framework. When a query is made, the client credential of the user currently logged in is passed to the authentication method servlet side for authentication based on the given query parameters. The next section investigates choice of technology and mechanisms related to Inter-Component Communication.

### 3.3.1 Inter-Component Communication

Several core technologies have been considered as a medium to facilitate communication between G2E components and ABAC access control facilities. These technologies are Remote Procedure Call (RPC) and HTTPS. Both core technologies enable communication over an SSL tunnel. In the case of RPC, an SSL connection is established to the access control server through a socket; a request and parameters are serialized. The remote end de-serializes the request and parameters, and immediately carries out the subsequent requested operation. In the case of HTTPS, requests are made through POST requests in the HTTP protocol through an SSL tunnel established with a web server. The HTTP request is managed according to the implementation of the protocol server-side. Both methods provide the necessary mechanisms to facilitate inter-component communication; however RPC has one primary limitation. This limitation is a reliance on client side implementation of the RPC framework. With regards to the licensing restrictions, any modification to the code base of G2E transfers ownership of these modifications to the distributed architecture. Therefore, any client-side RPC mechanism would come under the ownership of the G2E architecture. The corresponding RPC client libraries can be developed under the G2E architecture license if required. If more than one distributed architecture utilizes the ACS, then each client may implement a different RPC mechanism utilizing a different technology.

HTTPS however provides an important contrast to these concerns. Specifically, HTTPS requests can be formed with minimal effort client-side and be easily requested through utilities such as Curl, scripts or integration with the Java runtime in the case of G2E. HTTPS has been chosen as a primary medium for access control requests. In this case, REST or a RESTful architecture has been chosen as a design paradigm for subsequent requests. This REST style has been chosen because of its innate notion of loose coupling, client architecture, resource-driven semantics and stateful representation. This is particularly important in the context of access control. In particular, each operation and corresponding access control request denotes a transition within the client state. This notion of state fits well with the ABAC notion of negotiation. In particular, ABAC allows an authenticating client to make subsequent access control request using different credential sets. Furthermore, a client may append different credentials to the request to assert different types of access within the system. Comparably, REST allows for a client to assert an increasing degree of refinement with regards to credential choice with respect to an access control resource. For example, a client can make a request comparable to: ACS/operation/base_credential /server_specific_credential/operation_specific_credential. The corresponding design of a REST-driven architecture combined with ABAC access control mechanisms is detailed in subsequent sections.

The technology chosen to facilitate this functionality is as follows. Python has been chosen as the language of preference because of its wide interoperability with many distributed systems. With respect to previous work integrating ABAC into PlanetLab, Python was the language of choice, therefore many of the required libraries and features are already implemented. To facilitate web-based requests Django is used due to its stability and wide-scale deployment. Django also provides a desirable

View-based architecture to facilitate potentially high volume access control requests. Finally to enable a REST-based style, the Django REST Framework is used. The Django REST Framework overcomes some limitations with comparable Django REST projects such as Tasypie and Piston. Then the ABAC wrapper is integrated into the Django server context which in turn loads the libabac library written in C. The ABAC wrapper is written in C and enables an interface to C through SWIG. The libabac libraries utilize strongSwan for cryptographic functions and features. Requests are made through HTTPS utilizing a RESTful style.

### 3.3.2 API

The ABAC Access Control Server (ACS) implements a set of APIs that defines key characteristics of how clients may interact with related resources. These APIs and their associated descriptions are defined in the Table 3.3.2.

These API denote core functionality presented to servers and clients within a distributed architecture. All of the complexity associated with access control resides within the policy map within the ABAC context. This enables a clean API to be presented as described above and reduces complexity. The next section details the manageability and scalability of the ABAC Access Control Server with respect to the G2E architecture.

Based on this in-depth analysis of the G2E architecture, the subsequent section details the formal policy designed to support policy requirements of the primary stakeholder in the G2E project.

## 3.4 Formal Client Access Control Specification

### 3.4.1 Policy

The following formal policy defines the set of all policy elements and their relationships with respect to the formal access control requirements requested by Biosphere Management Systems Inc. This section is partitioned into two policies, the Client Access Control Specification (CACS) and the Architecture Access Control Specification (AACS).

The CACS defines the formal policy that controls and facilitates authorization and authentication of human actors within the system. This formal policy further specifies

Table 3.1: ABAC Server REST API

| **API** | **Description** |
|---|---|
| RegisterContext(UUID, credential, type ) | A server can use this API to register its identity with the ABAC Access Control Server. This operation creates an access control context by which it can subsequently query the policy against a credential set. The UUID is a unique identifier of the server, type denotes associated meta data for logging, and the credential denotes the principal credential of the system. |
| LoadPolicy(UUID, credential, policy) | A server may use this API to load a policy governing a set of resources controlled by the server referenced by the associated UUID. The UUID parameter denotes the unique identifier of the server context instance, the credential parameter denotes the server's previously registered credential, and the policy denotes the set of attributes that comprise the servers policy. The credential in this API is used both to associate a policy with the server credential context and to authorize the request. The policy additionally contains attributes that have semantic operations bound to them within the certificates. |
| Authenticate(UUID, credentials, operation) | The Authenticate API is the primary mechanism by which both clients and servers are authenticated within the context of another server. In this case, the parameter UUID denotes the unique identifier of the server that owns the access control policy in context, the credentials attribute stores both the requesting client or server and the credential of the owner of the policy, and the operation parameter denotes the requested operation of the client or server. |
| DeletePolicy(UUID, credential) | Similar to the LoadPolicy operation, the DeletePolicy API will delete all policy attributes associated with a given server context. This enables a server to load a new policy in place of the old policy or a policy maybe deleted to take the server out of commission. Once again the UUID denotes the unique identifier of the server context and the credential parameter is the credential associated with the access control context of the server. |
| UnRegisterContext(UUID, credential) | UnRegister API deletes a server's access control context from the ABAC server. This API is used if a server is decommissioned or removed from operation. UUID is the unique identifier of the access control instance and the credential parameter is the credential of the server associated with the access control context. |

Table 3.2: ABAC Server REST API

| API | Description |
|---|---|
| RegisterUser(UUID, credential, role) | RegisterUser API delegates a role specified by the role parameter and delegates the role to the user credential specified by the credential parameter. The UUID is a unique identifier that references a unique access control instance. The delegated role must exist within the UUID context specified. The delegate role is only valid within the specified UUID access control instance. |
| UnRegisterUser(UUID, credential, role) | UnRegisterUser API removes a role specified by the role parameter and delegates the role to the user credential specified by the credential parameter. The UUID is a unique identifier that references a unique access control instance. The un-delegated role must exist within the UUID context specified. |

filtering mechanisms on a variety of access control attributes. Many different mutually exclusive attribute sets are evaluated sequentially. Failing one access control query will result in an authentication failure. This authentication mechanism can further be expanded to support multi-criterion logging and feedback mechanisms. For example, if a client or human actor fails at authenticating based on resident country but belongs to the correct association, then an authentication exception will be thrown specifying an incorrect delegation of credentials.

AACS encompasses mechanisms of access control solely within a federated Cloud infrastructure. AACS enables policies to be further partitioned by separating policy that governs data from policy that governs architecture. The AACS policy protects infrastructure components at an inter-procedural level. This prevents compromised system components from interacting with other components in ways not defined by the enforced policy.

The AACS and the CACS policies are defined within this section. Each policy includes an *Index* field and a *Hierarchy Link* field. Every table has a key and using the Hierarchy Link field, the policy map can be followed to another policy table. By this method, both a formal policy and ABAC attribute hierarchies can be modeled. When considering the case study presented in Chapter 2, a formal policy was already created so the method only required abstracting a preexisting policy into an ABAC grammar. This differs from present case study where a modern architecture does not have a formal access control policy. Instead, creation of both a formal policy and an ABAC policy is required.

### 3.4.2 Client Access Control Specification

The Client Access Control Specification defines the policy that governs user interaction with the G2E architecture. Any user registered with the system is associated with a set of metadata that describes properties and permissions of that user.

This metadata can include elements such as location, organization, subscription status and role. Similarly, data (satellite data) is also associated with metadata. By binding metadata and user data into a policy, an ABAC bi-directional search can be performed to evaluate if a user has access to the specified data resource.

The primary stakeholder for the G2E project has set two primary requirements: (1) the association of users with subscription status and data with subscription information and, (2) the association of users with geo-location and data with geo-location.

Requirement (1) enables data to be partitioned based on a subscription model to the service. Requirement (2) allows for users to be restricted from potentially sensitive data based on their association with a geo-location. The policy specification in this section describes the implementation of requirement (1) and (2) into an ABAC grammar and formal policy specification.

Table 3.3 defines the set of system roles associated to users within the G2E architecture. Each role is associated with a subscriber status following the Hierarchy Link column. Each user is also associated with geo-location metadata such as *CanadianCitizen* or *RussianCitizen*.

Table 3.4 defines the subscriber status that is bound to the client credential. Each client credential may be bound to one subscriber status. This policy specification satisfies requirement (1).

Table 3.5 defines a mapping of subscription and access classes to data within the G2E architecture. The access level then maps to the policy map layer associated with a subscription status. This policy specification satisfies requirement (1).

Table 3.6 defines a set of attributes that are bound to an entity credential. This policy specification satisfies requirement (2).

Table 3.7 defines a set of geo-location data that may be bound to the client credential and the data. This policy specification satisfies requirement (2).

Table 3.8 defines a set of organization-based metadata that may also be bound to a client credential and data within the G2E architecture. This policy specification satisfies requirement (2).

Table 3.9 defines the primary data (satellite image) quality bound to a subscriber status. Thus, subscriber status can determine the quality of data a user may access. This policy specification satisfies requirement (1).

Table 3.10 defines a set of account classifications that enable a degree of user management. For example, if a user is awaiting approval to gain access to a set of resources, the account can be temporarily locked.

Table 3.11 defines a set of attributes that are bound to the dataset in the G2E application that allows filtering on any give set of attributes. This policy specification satisfies requirement (2).

Table 3.12 defines a set of operations within the G2E architecture that a user can perform. This set of operations can be expanded as the architecture evolves. Only one operation within the architecture was defined: *Query*. A Query operation was defined that requires multi-variable access control logic to be evaluated for successful

Table 3.3: System Roles (KEY = SR)

| **Index** | **Role** | **Description** | **Hierarchy Link** |
|---|---|---|---|
| 0 | User | An entity within the system that has basic access throughout the system. This user only has read access when allowed. Access is not limited to specific systems. | SS.1.(0-3), EA.(0-3), ES.(0-3) |
| 1 | Manager | An entity within the system that has read access and limited write access. This role is a Superuser compared to a User. A Manger only has write access to a given project the manager is associated with. | SS.1.(0-3), EA.(0-3), ES.(0-3) |
| 2 | Director | An entity within the system that has read and write access. This role is a Superuser compared to a Manager. Additionally noted, a Director only has write access to a given project the Director is associated with. | SS.1.(0-3), EA.(0-3), ES.(0-3) |
| 3 | Intelligence | An entity within the system that has unlimited read access. This entity also has the capacity to delete and modify images in a limited and controlled way. | SS.1.System, EA.(0-3), ES.(0-3) |
| 4 | TechnicalContact | An entity within the system that has read access. This entity role also has the ability to restart infrastructure. | SS.1.System, EA.(0-3), ES.(0-3) |
| 5 | Administrator | Is the Superuser within the system and has no restricted access. | SS.1.System, EA.(0-3), ES.(0-3) |

Table 3.4: Subscriber Hierarchy (KEY = SS)

| **Index** | **Subscriber Status** | **Description** | **Hierarchy Link** |
|---|---|---|---|
| 0 | UnSubscribed | (Subscribed0) | This attribute identifies the entity within the system is unsubscribed. This attribute is mutually exclusive with SS.1 and SS.2 |
| 1 | Subscribed | (Subscribed1) | This attribute identifies the entity within the system is subscribed with basic service. This attribute is mutually exclusive with SS.0 and SS.2 |
| 2 | Premium | (Subscribed2) | This attribute identifies the entity within the system is subscribed to premium services. This attribute is mutually exclusive with SS.0 and SS.1 |
| 3 | Contributor | (Subscribed3) | This attribute can be delegated in addition to (0-1) to denote the entity makes contributions to the the data store. Additional filtering may occur on this attribute; for example, because a contribution has been made, temporary access can be delegated as a monetary reward or enticement. |
| 4 | System | (Subscribed4) | This attribute identifies the entity requesting access to a particular resource is a system entity. This can include personnel such as a TechnicalContact in SR.1 and priority access can be delegated. |

Table 3.5: Access Class Map (KEY = ACM)

| **Index** | **Access Level Mapping** | **Subscriber Status** | **Description** | **Hierarchy Link** |
|---|---|---|---|---|
| 0 | Public | SR.(0 -5) | Anyone can access this resource. | DL.(0-1) |
| 1 | Subscription | SR.(0 -5) | Paid subscribers | DL.(0-2) |
| 2 | Premium | SR.(0 -5) | Premium subscribers | DL.(0-3) |
| 3 | Private | SR.(2,3,5) | Only intelligence agencies, an Administrator or directors of a specific project may view this data. | DL.(0-2) |
| 4 | Restricted | SR.(3,5) | Only intelligence agencies and the Administrator may view this data. | DL.(4) |
| 5 | Classified | SR.(3,5) | Only intelligence agencies and the Administrator may view this data. | DL.(4) |

Table 3.6: Entity Attributes (KEY = EA)

| **Index** | **Entity Attribute** | **Description** | **Hierarchy Link** |
|---|---|---|---|
| 0 | Country | The country(ies) the entity is affiliated with. | CC.(0-2) |
| 1 | Organization | The organization(s) the entity is affiliated with. | ORG.(0-2) |
| 2 | Status | The current status of the user within the system (active, locked). Active status would imply the entitys account is not locked. A locked account would imply the entitys account is locked and unable to access elements within the architecture. | ES.(0-3) |
| 3 | Role | Denotes an associated role defined in SR. | SR.(0-5) |

Table 3.7: Country Code (KEY = CC)

| **Index** | **Country** | **Description** | **Hierarchy Link** |
|---|---|---|---|
| 0 | Canada | Canada | NULL |
| 1 | USA | United States of America | NULL |
| 2 | Russia | Russian Federation | NULL |

Table 3.8: Organization (KEY = ORG)

| **Index** | **Subscriber Status** | **Description** | **Hierarchy Link** |
|---|---|---|---|
| 0 | None | No organization | NULL |
| 1 | BC Forest Ministry | BC Forest Ministry | NULL |
| 2 | Russian Forest Ministry | Russian Forest Ministry | NULL |

Table 3.9: Data Layering (KEY = DL)

| **Index** | **Data Imagery Layer** | **Description** | **Hierarchy Link** |
|---|---|---|---|
| 0 | VeryLow | Low resolution imagery (50 km x 50 km) | SS.0 |
| 1 | Low | (Primary) Low resolution imagery (25km x 25 km) | SS.0 |
| 2 | Medium | (Primary) High resolution imagery by square kilometer or by hectare (small number of samples only) | SS.1 |
| 3 | High | (Primary) High resolution imagery by sq kilometer or by hectare (medium number of samples only) | SS.(2,3) |
| 4 | VeryHigh | High resolution imagery by square kilometer or by hectare (high number of samples) | SS.(4,5) |

Table 3.10: Entity Status (KEY = ES)

| **Index** | **Status** | **Description** | **Hierarchy Link** |
|---|---|---|---|
| 0 | Active | The account is active and can be used. | NULL |
| 1 | Review | The account is currently under review, waiting for authorization. | NULL |
| 2 | Locked | The account has been locked and unable to be used. | NULL |
| 3 | Deleted | The account has been deleted and no longer exists. | NULL |

Table 3.11: Metadata Classification (KEY = MC)

| **Index** | **Subscriber Status** | **Description** | **Hierarchy Link** |
|---|---|---|---|
| 0 | Country | Filter access control based on country. This attribute is compared to the entity trying to access a resource. | NULL |
| 1 | Organization | Filter data based on organization in control over the region. This attribute is compared to the entity trying to access a resource. | NULL |
| 2 | Province/State (Partition) | Filter data based on organization in control over the partition. This attribute is compared to the entity trying to access a resource. | NULL |
| 3 | Forrest (Region) | Filter data based on organization in control over the region. This attribute is compared to the entity trying to access a resource. | NULL |

Table 3.12: Entity Operations (KEY = EO)

| **Index** | **Status** | **Description** | **Hierarchy Link** |
|---|---|---|---|
| 0 | Query | Run a query on the dataset | NULL |
| 1 | QuerySubscribed | Run a query on the dataset if Subscribed | NULL |
| 2 | QueryPremium | Run a query on the dataset if Premium User | NULL |
| 3 | QueryUnSubscribed | Run a query on the dataset if UnSub-scribed | NULL |

authentication against the data set. Three attributes were defined to subsume the multi-variable query logic. These attributes can be observed in Table 3.12.

The Client Access Control Specification ensures that a client interacting with the G2E architecture is only able to access the data restricted by requirement (1) and (2). The next aspect of the policy specification for the G2E architecture is the inter-component access control specification.

### 3.4.3 Architecture Access Control Specification

The Architecture Access Control Specification defines the policy by which components in the G2E architecture may interact with one another. This policy specification also defines a set of infrastructure level commands system administrators may use for interacting with the architecture. These policy attributes only define a basic set of operations but can be expanded as the architecture evolves.

Table 3.13 defines the policy specification for the Authentication Proxy. Operations within this policy can be delegated to the ABAC Client Interface for added security.

Table 3.14 defines the policy specification for the ABAC-Server. This policy is delegated to the Authentication Proxy to ensure that only a given Authentication Proxy has access to the ABAC-Server. This access control constraint prevents other components or entities within the architecture from directly querying the ABAC-Server.

Table 3.15 defines the access control policy for the G2E Web Application. As

Table 3.13: Authentication Proxy (KEY = AP)

| Index | Attribute | Description | Hierarchy Link |
|---|---|---|---|
| 0 | Register | Registers an ABAC context with the ABAC-Server | NULL |
| 1 | LoadPolicy | Loads an ABAC policy into a given ABAC context at the ABAC-Server | NULL |
| 2 | Authenticate | Authenticates a user credential against the ABAC policy in a given context stored within an ABAC context at the ABAC-Server | NULL |
| 3 | DeletePolicy | Deletes a policy from an ABAC context stored at the ABAC-Server | NULL |
| 4 | UnRegister | Deletes an ABAC policy and the ABAC context from the ABAC-Server | NULL |

Table 3.14: ABAC-Server (KEY = ABACS)

| Index | Attribute | Description | Hierarchy Link |
|---|---|---|---|
| 0 | Register | Registers an ABAC context with the ABAC-Server | AP |
| 1 | LoadPolicy | Loads an ABAC policy into a given ABAC context | AP |
| 2 | Authenticate | Authenticates a user credential against the ABAC policy in a given context stored within an ABAC context | AP |
| 3 | DeletePolicy | Deletes a policy from a stored ABAC context | NULL |
| 4 | UnRegister | Deletes an ABAC policy and the ABAC context | AP |

Table 3.15: G2E Web Application (KEY = G2E-WA)

| **Index** | **Attribute** | **Description** | **Hierarchy Link** |
|---|---|---|---|
| 0 | None | None | G2E |

Table 3.16: G2E Java Application Server (KEY = G2E-JAS)

| **Index** | **Attribute** | **Description** | **Hierarchy Link** |
|---|---|---|---|
| 0 | Query | Query G2E data | G2E-WA |

observed, no policy exists for access control as this is the entry point for the G2E application. The G2E Web Application is the component that facilitates login/logout and authentication processes. However, an ABAC policy can be added to this component if requirements demand this capability.

Table 3.16 defines the access control policy for the Java Application Server. The only operation currently supported by the G2E architecture is the *Query* operation. This operation is delegated to the G2E Web Application which ensures only the Web Application can execute this operation.

Table 3.17 defines the access control policy for the G2E Cache Services component. Once again only the Web Application may query cached G2E data.

Table 3.18 defines the access control policy for the Rasterization service in the G2E architecture. The Rasterize operation is delegated to the FTP-Server. However, it should be noted that this operation is *optional* as the FTP Server must join the federation to enable utilization of ABAC.

Table 3.19 defines the Store operation for the G2E Sync Application. The Store operation is delegated to the Rasterization Application. Thus, only the Rasterization Application may store rasterized images within the G2E architecture.

Finally, Table 3.20 defines the access control policy for the G2E Database Application. As illustrated, there are two operations in this policy specification: Insert and Query. The Insert operation is delegated to the G2E Sync Application and the

Table 3.17: G2E Cache Services (KEY = G2E-CS)

| **Index** | **Attribute** | **Description** | **Hierarchy Link** |
|---|---|---|---|
| 0 | QueryCached | Query cached data | G2E-WA |

Table 3.18: G2E Rasterization Application (KEY = G2E-RA)

| **Index** | **Attribute** | **Description** | **Hierarchy Link** |
|---|---|---|---|
| 0 | Rasterize | Initiate rasterization of data files | FTP-Server |

Table 3.19: G2E Sync Application (KEY = G2E-SA)

| **Index** | **Attribute** | **Description** | **Hierarchy Link** |
|---|---|---|---|
| 0 | Store | Stores rasterized data into the G2E-Database Application | G2E-RA |

Table 3.20: G2E Database Application (KEY = G2E-DA)

| **Index** | **Attribute** | **Description** | **Hierarchy Link** |
|---|---|---|---|
| 0 | Insert | Adds data to the G2E-Database Application | G2E-SA |
| 1 | Query | Queries the data in the G2E-Database Application | G2E-JAS |

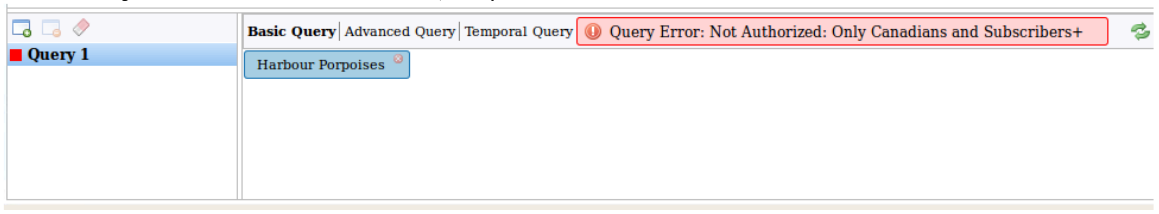
Query operation is delegated to the G2E JAS component. This policy ensures only the Sync Application may write to the Database and only the JAS may read. An additional policy layer can be added to this policy specification to support a *limited write* operation to allow the JAS component to update metadata. However, as the G2E architecture is currently designed, such an operation is not supported.

The policy specification defined in this section demonstrates how restrictions can be enforced at the infrastructure level between components. Furthermore, it should be observed that these access control restrictions are not bi-directional. Specifically, each infrastructure level access control requirement can be delegated to any component or any number of components. This ensures that as new components of the same type are added, policy is automatically delegated to each new component. Thus scalability, modularity and extensibility can be achieved by factorizing policy among each component. This result is comparable to that found in Chapter 2. The primary difference between each ABAC system is distribution of the ABAC policy. In the legacy system, ABAC policy was centralized at each component. In the modern G2E architecture the factorization of policy is maintained in an access control context stored at the ABAC-Server.

## 3.5 Results

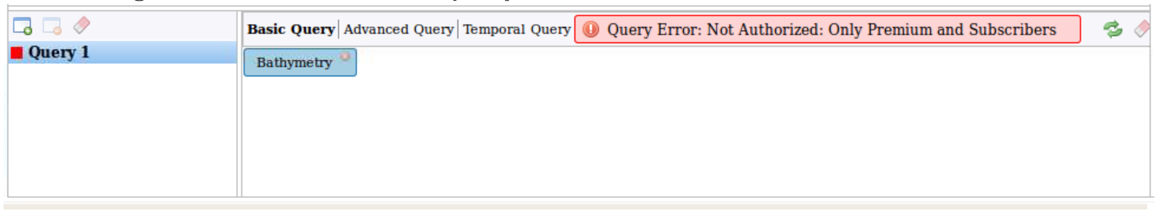
One challenge associated with evaluating a given implementation and architecture is metric choice. However, this case study investigates if it is possible to integrate ABAC into a modern architecture. It has been demonstrated that ABAC was successfully integrated into the G2E architecture and ABAC grammar has fully subsumed primary stakeholder policy requirements. Thus, the fact that ABAC can be used as a primary access control mechanism for a modern architecture is evident by the creation, construction and design of the components, policy and mechanisms as outlined

Figure 3.5: G2E ABAC Query Restriction: Canadian and Subscribers+



Query Error: Not Authorized: Only Canadians and Subscribers+

Figure 3.6: G2E ABAC Query Restriction: Premium and Subscribers



Query Error: Not Authorized: Only Premium and Subscribers

in this study. Evaluation is conducted using three methods: (1) proof of concept, (2) performance and scalability metrics and (3) stakeholder review.

### 3.5.1 Proof of Concept

Proof of concept evaluates three primary principles required for successful integration of ABAC into a modern architecture. These principles are: (1) ABAC can be integrated into a proprietary system with licensing restrictions; (2) ABAC can facilitate access control and fully encompass access control policy of a modern architecture into an ABAC grammar; and (3) ABAC policy can be combined with mechanism to provide fine grain control over large data sets. Method (1) is demonstrated by existence of a working architecture; (2) is demonstrated by a working policy defined within this case study; and (3) is demonstrated by evaluating use case models. An example of these use case models can be seen in Figure 3.5. Figure 3.5 describes access control requirements based on location and subscription status and Figure 3.6 describes access control decisions based only on subscription status. Each of these figures represent a screen capture of a working G2E architecture where multiple users may login and make requests using the Query structure.

As observed in Figure 3.5, access to geographical information on Harbour Porpoises is limited to only Canadians who are Subscribers or Premium users denoted by the +. In this scenario a Russian user is attempting to access restricted data.

As observed in Figure 3.6, access to geographical information on Bathymetry is limited to only Subscribers or Premium users. In this scenario, there is no geo-location restriction imposed upon the data, only subscriber restrictions.

### 3.5.2 Performance and Scalability

In this section the performance and scalability of the proposed ABAC access control solution created for G2E is analyzed. For ABAC to be a suitable primary access control mechanism integration of ABAC should have minimal performance impact on the architecture. Performance and scalability of ABAC and related access control server architecture can be partitioned into two categories. These categories are (1) credential operations and (2) architecture performance.

ABAC library and credential performance has been closely analyzed at Deterlab [9]. From the observed experimental results, ABAC policies grow linearly as the complexity of the policy increases. Deterlabs demonstrated that fairly complex policies remain sustainable for a high number of authentication and authorization requests in a second. Specifically, for a credential policy of two hundred credentials, ABAC can support approximately 30,000 valid queries and approximately 2,000,0000 invalid queries in one second. It should be noted that for repeated queries the cached valued of the query result is stored to prevent re-querying the entire credential chain. These results demonstrate how ABAC core libraries and credential systems can easily scale of a high number of users per ABAC access control architectures. However, to truly determine the performance and scalability of ABAC as a primary access control system, the centralized access control system must be analyzed.

As mentioned previously, the access control architecture created for G2E consists of several components: ABAC Access Control Server, Access Control Proxy and Access Control Client. Evaluation of the ABAC access control architecture is divided into several segments. The centralized ABAC Access Control server is first analyzed using the RESTful API. Each API is evaluated over a 30 second interval and then averaged on a one second interval. It should be noted that all experiments were conducted using within a virtual machine. It should be noted that running the ABAC Access Control architecture on a physical server may see an improvement in performance.

The experimental results of each Access Control server API is detailed in Table 3.5.2.

The average number of operations per second is detailed in Table 3.5.2. It should

Table 3.21: ABAC Access Control Server Performance

| **API** | **Average Number of Operations per Second** |
|---|---|
| RegisterContext | 1.01 |
| LoadPolicy | 4.6 |
| Authenticate | 202.3 |
| DeletePolicy | 198.67 |
| UnRegisterContext | 200.56 |

be noted that all experiments were conducted two virtual processors, 2048 Megabytes of RAM and a 8 Gigabyte virtual hard drive. The hard drive is in VMDK format and the VM is run in VMWare Player. The host computer runs an AMD Phenom II X6 1055T processor clocked at 2.8 GHz, has 8 Gigabytes of RAM and has a 2 Terabyte Western Digital Caviar Black.

Each operation was evaluated using a Python client program that comprised the correct data and policy sets to utilize the ABAC server. Each Python program attempted to carry out a maximum number of operations within a 30 second period. Each Python script was run 10 times and the number of operations were averaged across all runs. The results of these experiments can be observed in Table 3.5.2.

Despite the experiments running in a virtual machine, the observed results demonstrate the ABAC server's capacity to support a high number of concurrent users. It should be noted that many ABAC servers can be deployed and load balanced to satisfy demand for authorization and authentication.

### 3.5.3 Stakeholder Review

Evaluation of proof of concept Section 3.5.1 and scalability Section 3.5.2 are critical for evaluating the feasibility of ABAC being used as a primary access control mechanism for enterprise Cloud systems. However, for ABAC to be a success in commercial enterprise architectures the decision makers and stakeholders must be convinced of ABAC's functionality, usability, manageability and requirements. Primary stakeholders can have varying perspectives that differ at times drastically from technical analysis. To investigate and affirm the support of enterprise and corporate customers in ABAC-driven solutions, a survey has been devised. This survey was designed to evaluate the success of satisfying primary stakeholder requirements of integrating ABAC into a modern architecture. The questions in this survey were carefully created to determine the confidence in ABAC being a suitable and primary access control mechanism for a modern distributed architecture under continual evolution. The Table 3.5.3 illustrates the findings of the survey.

The survey illustrated in Table 3.5.3 details the responses from two critical project stakeholders: the owner and primary enterprise stakeholder and the Platform Architect responsible for the design and development of the G2E system. It should be noted that the two stakeholders belong to different organizations by which the G2E

Table 3.22: G2E ABAC Survey

| **Survey Question** | **Project Stake-holder** | **Platform Architect** |
|---|---|---|
| In your opinion, can ABAC describe the access control policies required for a modern software system? | 10 | 8 |
| In your opinion, can ABAC access control mechanisms can satisfy the requirements of a modern software system? | 9 | 7 |
| In your opinion, to what degree does ABAC have advantages over traditional access control mechanisms? | 9 | 7 |
| In your opinion, how confident are you that ABAC can evolve with a modern architecture and continually satisfy system requirements? | 8 | 7 |
| In your opinion, how confident are you that ABAC can scale as a system scales? | 8 | 6 |
| In your opinion, how easy are ABAC policies to manage? | 7 | 6 |
| In your opinion, can ABAC be used as a primary access control mechanism in a modern software system? | 7 | 8 |

system was developed under contract to the for the primary stakeholder. It should also be noted that the level of expertise from a system design standpoint may differ.

When analyzing both sets of responses it can be observed that from the empirical proof of concept, there is confidence that ABAC satisfies essential requirements to fully subsume enterprise access control responsibilities. However, a larger survey pool would increase the validity of these results.

# Chapter 4

# Evaluation, Analysis and Conclusions

## 4.1 Claims and Analysis

This thesis has endeavored to assess the claim that ABAC can fully subsume the requirements of enterprise architectures. For ABAC to satisfy these requirements the following core requirements must be satisfied: (1) policy requirements must be fully supported by ABAC Grammar, (2) access control mechanisms must be interoperable with enterprise architecture, (3) scalability, manageability and reconfigurability must be satisfied and (4) ABAC must support both legacy and modern evolving infrastructures.

To answer these four requirements two case studies were conducted. Each case study sought to identify requirements (1), (2) and (3) in terms of (4), legacy and modern architectures. As observed in the legacy architecture case study in Chapter 2, legacy architectures can benefit from ABAC's capacity to factorize policy. As observed in the modern architecture case study found in Chapter 3, modern architectures can incorporate licensing restrictions and experience continual evolution. Once again ABAC was demonstrated to overcome both challenges by factorizing policy to a single server and providing a minimal interface for authorization once again abstracting policy complexity away from implementation.

In both case studies, (1) was demonstrated by the formalization of ABAC grammars that fully support the policy requirements of the primary stakeholder. In the legacy architecture case study, the formal policy was already defined. In the modern architecture case study, policy was under continual evolution as the architecture evolved. In both cases, an ABAC grammar was implemented and integrated into both architectures.

In both case studies, (2) was demonstrated by development of required ABAC libraries and integration of these software libraries into each respective architecture. In the case of the legacy architecture, the modules were integrated directly into each Manager. No licensing conflicts were observed as the licenses were compatible. In the case of the modern architecture, licensing conflicts were evident but such constraints were overcome by means of engineering a component-based access control architecture (ABAC-Server architecture). In both cases fundamental principles of decoupling policy from mechanism and components from each other was achieved, although by different approaches. In the case of the legacy architecture, policy resided on each physical computer. In the case of the modern architecture, policy was factorized in the Access Control Context (ACC).

In both studies, (3) was demonstrated by achieving operability and allowing for direct use and demonstration. In both cases, the policy, access control mechanism, and inter-component interaction operated within the constraints of the formal specifications and stakeholder requirements. However, the integration of ABAC also resulted in distinct modifications to architecture capacity and evolution. Specifically, ABAC enabled the factorization of policy, the abstraction of policy from mechanism, expanded capacity to federate and the factorization of implementation from policy. Each of these unique capacities of ABAC directly correlate to expanded capacity of the systems ABAC was integrated into.

## 4.2 Conclusions

As demonstrated in this thesis, ABAC can subsume all access control requirements of the architectures described in these case studies. It has also been demonstrated that architectures similar those outlined in these case studies can benefit from ABAC's grammars and interoperable access control mechanisms. ABAC eliminates many inadequacies associated with access control mechanisms found in Section 1.5 with respect to the architectures assessed in this thesis. ABAC instead provides an expressive grammar for constructing policies and applications to enterprise architectures and a generic library that can be integrated into two classes of access control architectures as defined in this thesis. If architecture engineering requirements demand a secure, scalable, reconfigurable and manageable access control architecture, then Attribute-Based Access Control can rise to this challenge.

# Bibliography

[1] *15th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE 2006), 26-28 June 2006, Manchester, United Kingdom*. IEEE Computer Society, 2006.

[2] *2006 International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM 2006), 26-29 June 2006, Buffalo, New York, USA, Proceedings*. IEEE Computer Society, 2006.

[3] Antepedia. Openid. `http://www.antepedia.com/detail/p/306232.html`, 2013.

[4] David W. Chadwick. *Understanding X.500 - the directory*. Chapman and Hall, 1994.

[5] David W. Chadwick and Alexander Otenko. The permis x.509 role based privilege management infrastructure. *Future Generation Comp. Syst.*, 19(2):277–289, 2003.

[6] David W. Chadwick, Alexander Otenko, and Edward Ball. Role-based access control with x.509 attribute certificates. *IEEE Internet Computing*, 7(2):62–69, 2003.

[7] deterlab. Abac access control for fedd. `http://fedd.deterlab.net/wiki/FeddABAC`, 2012.

[8] deterlab. Abac authorization control model and discussion. `http://groups.geni.net/geni/wiki/TIEDABACModel`, 2012.

[9] deterlab. Abac performance. `http://abac.deterlab.net/wiki/PerformanceData`, 2012.

[10] deterlab. deterlab. http://abac.deterlab.net/wiki/DocumentationRT2, 2012.

[11] DMFT. Role based authorization profile. http://www.dmtf.org/sites/default/files/standards/documents/DSP1039_1.0.0.pdf, 2008.

[12] Stephen Dranger, Robert H. Sloan, and Jon A. Solworth. The complexity of discretionary access control. In *Proceedings of the 1st international conference on Security*, IWSEC'06, pages 405–420, Berlin, Heidelberg, 2006. Springer-Verlag.

[13] David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. Proposed nist standard for role-based access control. *ACM Trans. Inf. Syst. Secur.*, 4(3):224–274, August 2001.

[14] D.Richard Kuhn David F. Ferraiolo. Role-based access control (rbac): Features and motivations. 1995.

[15] Daniel M. German and Ahmed E. Hassan. License integration patterns: Addressing license mismatches in component-based development. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 188–198, Washington, DC, USA, 2009. IEEE Computer Society.

[16] Jeff Goldman. Barracuda labs warns of openid phishing attacks. http://www.esecurityplanet.com/network-security/barracuda-labs-warns-of-openid-phishing-attacks.html, 2013.

[17] Internet Engineering Task Force (IETF). Oauth 2.0 threat model and security considerations. http://tools.ietf.org/html/rfc6819, 2013.

[18] Alan H. Karp, Harry Haury, and Michael H. Davis. From abac to zbac: The evolution of access control models. Technical report, Hewlett-Packard laboratories, 2009.

[19] Barry Leiba. Oauth web authorization protocol. *IEEE Internet Computing*, 16(1):74–77, 2012.

[20] Herbert Leitold and Evangelos P. Markatos, editors. *Communications and Multimedia Security, 10th IFIP TC-6 TC-11 International Conference, CMS 2006, Heraklion, Crete, Greece, October 19-21, 2006, Proceedings*, volume 4237 of *Lecture Notes in Computer Science*. Springer, 2006.

[21] Jin Li, Qian Wang, Cong Wang, and Kui Ren. Enhancing attribute-based encryption with attribute hierarchy. *MONET*, 16(5):553–561, 2011.

[22] Ninghui Li, John C. Mitchell, and William H. Winsborough. Design of a role-based trust management framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 114–130. IEEE Computer Society Press, May 2002.

[23] Ninghui Li, William H. Winsborough, and John C. Mitchell. Distributed credential chain discovery in trust management (extended abstract). In *Proceedings of the Eighth ACM Conference on Computer and Communications Security*, pages 156–165. ACM Press, November 2001.

[24] Peter A. Loscocco, Stephen D. Smalley, Patrick A. Muckelbauer, Ruth C. Taylor, S. Jeff Turner, and John F. Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. In *In Proceedings of the 21st National Information Systems Security Conference*, pages 303–314, 1998.

[25] Uche M. Mbanaso, G. S. Cooper, David W. Chadwick, and Seth Proctor. Privacy preserving trust authorization framework using xacml. In *WOWMOM* [2], pages 673–678.

[26] Microsoft MSDN. Authorization. `http://msdn.microsoft.com/en-us/library/ff649821.aspx`, 2008.

[27] Microsoft MSDN. Claims based authorization using wif. `http://msdn.microsoft.com/en-us/library/ff649821.aspx`, 2011.

[28] Tuan-Anh Nguyen, Linying Su, George Inman, and David W. Chadwick. Flexible and manageable delegation of authority in rbac. In *AINA Workshops (2)*, pages 453–458, 2007.

[29] OpenGroup. Identity management forum. `http://www.opengroup.org/tech/idm/`, 2013.

[30] PlanetLab). Planetlab. `http://www.planet-lab.org/`, 2013.

[31] ProtoGENI. Distributed identity and authorization mechanisms. `http://groups.geni.net/geni/wiki/ABAC`, 2012.

[32] Lili Qiu, Yin Zhang, Feng Wang, Mi Kyung, and Han Ratul Mahajan. Trusted computer system evaluation criteria. In *National Computer Security Center*, 1985.

[33] David Recordon and Drummond Reed. Openid 2.0: a platform for user-centric identity management. In *Proceedings of the second ACM workshop on Digital identity management*, DIM '06, pages 11–16, New York, NY, USA, 2006. ACM.

[34] Internet2 Mailing List Service. I2-news: Internet2 releases privacy-preserving web authorizing software. `https://lists.internet2.edu/sympa/arc/i2-news/2003-07/msg00000.html`, 2003.

[35] Shibboleth. Securityadvisories. `https://wiki.shibboleth.net/confluence/display/SHIB2/SecurityAdvisories`, 2013.

[36] Microsoft TechNet. Authentication vs. authorization. `http://technet.microsoft.com/en-us/library/cc759647%28v=ws.10%29.aspx`, 2005.

[37] William H. Winsborough and Ninghui Li. Protecting sensitive attributes in automated trust negotiation. In *Proceedings of ACM Workshop on Privacy in the Electronic Society*, November 2002. To appear.

[38] Eric Yuan and Jin Tong. Attributed based access control (abac) for web services. In *Proceedings of the IEEE International Conference on Web Services*, ICWS '05, pages 561–569, Washington, DC, USA, 2005. IEEE Computer Society.

[39] Gansen Zhao, David W. Chadwick, and Sassa Otenko. Obligations for role based access control. In *AINA Workshops (1)*, pages 424–431, 2007.