2021-08-09

The Abacus: A New Approach to Authorization

Jacob Aaron Jess Siebach
Brigham Young University

The Abacus: A New Approach to Authorization

Jacob Aaron Jess Siebach

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

School of Technology

Brigham Young University

ABSTRACT

The Abacus: A New Approach to Authorization

Jacob Aaron Jess Siebach
School of Technology, BYU
Master of Science

The purpose of this thesis is to investigate the implementation of digital authorization for computer systems, specifically how to implement an efficient and secure authorization engine that uses policies and attributes to calculate authorization. The architecture for the authorization engine is discussed, the efficiency of the engine is characterized by various tests, and the security model is reviewed against other presently existing models. The resulting efforts showed an increase in efficiency of almost two orders of magnitude, along with a reduction in the amount of processing power required to run the engine.

The main focus of the work is how to provide precise, performant authorization using policies and attributes in a way that does not require the authorization engine to break domain boundaries by directly accessing data stores. Specifically, by pushing attributes from source domains into the authorization service, domains do not require the authorization service to have access to the data stores of the domain, nor is the authorization service required to have credentials to access data via APIs. This model also allows for a significant reduction in data motion as attributes need only be sent over the network once (when the attribute changes) as opposed to every time that the engine needs the attribute or every time that an attribute cache needs to be refreshed, resulting in a more secure way to store attributes for authorization purposes.

Keywords: authorization, RBAC, ABAC, domain-driven design, attribute-based authorization

ACKNOWLEDGEMENTS

My priorities in life are God, Family, and Country, and thus I will acknowledge each of them in that order. I begin with gratitude to my Father in Heaven, for without His Almighty Hand, I never would have undertaken the roads that led me to where I am today. Countless times He has inspired me in the software, relationships, and knowledge that enabled this thesis to come forth. Everything of good report that I have produced is because of His influence in my life.

To my beautiful Bride, Rebecca, and to my wonderful Children: thank you for donating the many hours that I might have spent with you to the endeavor of my furthered education. I hope that the fruits of the time that we spent apart as I worked on my degree bring blessings into the lives of all mankind. Words cannot express how I look forward to the coming evenings and days that I will get to spend with each of you individually, and all of us together as a family.

I am grateful to the Founders of this Nation, and to the Framers that gave us the Constitution that protects our liberties and affords us the opportunities to learn and grow. In that vein, I would be a remiss student to not recognize the patient, abundant efforts of Justin Giboney to assist in my quest for a Master's degree. Without the guidance and direction received from Justin, I never would have published any papers in my post-undergraduate career. Thank you, Justin, for your work with me, and for promoting Cybersecurity education at BYU.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

Every existing computer system has rules governing which identities are allowed to perform certain tasks or view specific data within that system, even if the rule is that anyone with access to the device is allowed to use it. These rules are called authorization policies. An organizational unit that provides a specific business purpose is called a domain, and domains use authorization policies to safeguard the data within them. Numerous commercial and custom systems in the world today use roles and groups to control authorization, but these have proved to lack the fine-grained control needed (thus requiring additional developments) [1], are prone to role explosion [2], and are often difficult to keep in sync with who should be allowed to have access [3].

In the past twenty years, several enterprise systems have been created to allow organizations to control authorization via authorization policies that rely on data attributes instead of roles or groups, such as Axiomatics [4], Next Labs [5], and Plain ID [6]. For a policy to grant authorization, the system using the policy needs access to the attributes of the user requesting authorization. Many current systems get these attributes by directly accessing the database tables where the attributes are stored. While this access method may allow the authorization system to get the current value of the attribute at run-time, it poses numerous security and domain-boundary issues, among which are tight coupling of the authorization system to the domains, the ability for a malicious actor to utilize the authorization system as a pivot into production databases, and increased authorization latency.

The research question was asked, “How do we solve the security issues exhibited by modern authorization systems while maintaining domain boundaries and increasing performance?” This thesis investigates and proposes an authorization methodology that respects domain boundaries by decoupling the authorization domain from the source domains, thus significantly increasing security. It also gives evidence that the new model enables increased speeds of up to two orders of magnitude from existing systems.

2 BACKGROUND

2.1 Defining Terms

An organizational unit that provides a specific business purpose is called a **domain**. In the modern age, most people use software in some form to accomplish the business purpose of their domain. Domains are at the heart of every computer system, storing data and enabling the business functions of the organization. Said Eric Evans, "Every software program relates to some activity or interest of its user. That subject area to which the user applies the program is the domain of the software" [7, p.2]. Domains are areas that control a specific set of data for an organization, e.g. HR, engineering, or customer support. The domain is the final authority on the data for the business function that it defines. For instance, the HR domain has the final say of employee status, engineering is responsible for the files for mechanical and digital systems, and customer support is the authority on the status of outstanding customer request tickets.

Because "domain" is such an overloaded term in the industry, it is specifically meant within this thesis to refer to a business unit that is responsible for one thing. Each business domain has a defined boundary that allows interaction with other domains. A web domain could actually have several business domains beneath it: the domain of the website presentation, the domain responsible for goods purchased on the site, and/or the domain of a customer support portal, among other things.

Every domain is accessed by one or more **identities**. "Digital identity is the unique representation of a subject engaged in an online transaction" [8, p. iv]. The identity could identify a user, computer system, physical device, network block, or anything else that may need to be referenced or interact with the domain.

When dealing with an individual face-to-face, it is easier to ascertain if the person is the identity that they claim to be, especially if they are known to you. Contrast this with computer systems where there is no physical interaction between a user and the verifying system; a user could impersonate any identity without a way to certify the identity of the user [9]. **Authentication** is the process of making reasonably sure that the identity presented to a computer system is indeed controlled by the entity that it represents within the system [8]. User authentication is usually accomplished through something that the user *knows* (like a username and password) and/or something that the user *has* (such as a smart phone). Some systems also authenticate through something that the user *is*, meaning a physical characteristic like fingerprints or retinal patterns.

Authorization and access control have been conflated to mean the same things for decades, which I believe to stem from two major causes: 1) Historical use of the term "access control" in software systems has referred to what I herein describe as authorization and access control, and 2) past research has not presented a reason previously to fully disjoin the two terms. To be precise in the use of these terms, I define **access control** to be the ability of an identity to reach a resource, whereas **authorization** is the permission of an identity to perform a function on a resource. The differences in these two terms are most easily illustrated with the simple example of a safe deposit box, secured by a lock and key. The ability to enter the room containing the box is having access. For our purposes, anyone who has been given the key to open the lock has the authorization to enter the box.

Imagine that the said safe deposit box sits inside of a bank vault, secured by an iron fence that you cannot pass. Because you are unable to reach the box, you do not have access to it; the access control mechanism has barred you. In this instance, authorization is irrelevant; neither possession of the key nor lack of it change the fact that you cannot access the box. Now imagine that the fence is open and you have access to the room with all of the safe deposit boxes in the bank. You can touch the various boxes, but without proper authorization, i.e. having the key, you cannot get inside any of them. What if you were standing in Provo with the key to a box that was in a bank in London? You would have the authorization required to enter the box, but you would not have access to it.

The distinction between authorization and access is subtle, but important. Access does not imply authorization, nor does authorization imply access. Both must be met for an identity to perform the intended function on the target resource: access control must permit an identity to get to the location of a resource and authorization must be granted for the identity to interact with the resource.

In a computer system, the access control may permit an identity to access the list of employees in a company, but the authorization component may only permit the viewing of employees in a specific department or with a certain job. Conversely, the identity may be authorized to view all of the employees, but the access control mechanism may prevent the identity from using a specific service to get that data. Access control allows an identity to get to a set of data, but the authorization is used to filter what data within that set is available to the calling entity.

All authorization is based on **policies**. Take the example of a child asking to eat dessert: the authorization response will depend on what policy or policies the parents have created. There could be a single policy, "If you eat all of your dinner, then yes." This implies that the default is to deny the request *unless* the policy conditions are met. It is also possible for the opposite to occur; permit by default yet *deny* a request if the policy conditions are fulfilled.

These authorization questions also exist in the world of technology. Within computer systems, the policies can be extremely complex when trying to determine if a user is authorized to perform a specific task. This thesis presents a new model for authorization that simplifies policies and the calculating of authorization decisions. By reducing policy options that lead to computationally intense situations, such as allowing policies to include other policies, which may result in circular references, creation and calculation of policies is greatly simplified.

2.2 Understanding Domain Driven Design

With terms defined, it is essential to understand domain-driven design (DDD), as it one of the central tenants of the research question in protecting the domain boundaries. Several books have been written on DDD principles [7, 10-11], and recent literature has addressed the difficulties in implementing DDD in microservice design [12-13].

2.2.1 Central Data Store

Every domain records the relevant, necessary data for its operation in its Central Data Store (CDS). This store can be in any structure or format that is best suited to the needs of the domain and the decisions of its developers, be it a relational database, a no-SQL system, a graph database, a flat file, or some other configuration.

Domains should not permit access into the Central Data Store (CDS) by any system outside of the domain. Granting access to the CDS by external entities prevents the domain from 1) owning the definition of its data, 2) modifying the CDS to meet its changing needs, and 3) controlling the business requirements to access the data. Understanding the nuances and importance of these statements is so vital that each of these must be discussed in detail.

2.2.1.1 Owning Data Definitions

Data alone does not have meaning without some sort of business description of it. If a data field is named "date" and contains the value "26-Apr-2005", we know that it is a date, but without an association to additional information we cannot tell if the data refers to a birthday, the date that an item was purchased, the occurrence of an error in the system, or some other information. The domain owners are responsible for the *definition* of the data: what it means, and how it is to be used.

Let us imagine a field called "isEmployee" in the CDS that contains a Boolean value, and every identity in the system has a value for "isEmployee". To find out if an identity is an employee of the business, it may be tempting to just check the "isEmployee" value, but there may be significantly more required than just the value of "isEmployee" to answer a particular question: it could be that "isEmployee" is a field that gives access to the building, or it needs the value of the "status" field as well to know if someone is an *active* employee, or maybe "isEmployee" refers to customers that are Federal employees for contract purposes.

Now suppose that an external system needs to know if an identity is an employee of the domain. The developer of the external system could see the "isEmployee" field and write a process to grab that value and return it to the external system. By so doing, the domain is not controlling the definition of that field, because the external system is not required to perform the business checks to see that the identity truly is an employee of the domain. Over time, external systems using the data fields in the CDS may not only misinterpret the definition of the data, they may also use the same fields *differently* from each other, resulting in multiple domains having various definitions of the same data.

Field interpretation has been a problem at BYU for many years: if you ask any two domains on campus what the definition of a "student" is, the answers will differ. Each department uses the same underlying data store, but because there is no central domain that enforces the *definition* of the data, each department reports their answer separately. An unintended side-effect of this is that no effective comparisons can be made between departments for purposes of enrollment, budget, or sometimes even classroom use!

The solution to the data definition problem is to create APIs that return data in a business context. This will be discussed further in section 2.2.2 Application Programmable Interfaces.

2.2.1.2 Modifying the CDS for Business Needs

All businesses must evolve in order to maintain success over time, and this change includes the CDS for its domains. When the business practices change, the underlying data structures much be adapted to match the business needs. It is thus essential that the CDS of a domain be free to modification as required.

Unfortunately, some systems allow entities, external to the domain, to execute directly against the CDS. By so doing, any change in the CDS may necessitate a change in the external systems, otherwise those external processes will break when they call for data that is either renamed, moved, or no longer present. Allowing external systems to operate directly against a domain's CDS introduces extremely tight coupling, for the external system must not only know the *technology* of the CDS, but the *structure* of the data within the CDS as well.

The cure for this issue is to prevent all external systems from receiving CDS access within a domain. Instead, defined interfaces allow those systems to receive the data required while permitting a change in the underlying format, organization, location, and even age, of the data. These interfaces could be APIs, data lakes, or other appropriate devices.

2.2.1.3 Controlling Business Requirements to Access Data

For any domain, the business has the right and the responsibility to determine what identities are permitted to access which data. This is extremely important in the current climate of digital privacy regulations and the constant threat of data breaches. As such, the business needs to determine the policies that govern each part of the data that it makes available to users, systems, and even its employees.

A common policy in many organizations is that an employee is authorized to view certain data. If an external system is granted access to the CDS containing the information for employee viewing, the domain has no way of enforcing the restriction that those accessing the data are indeed employees. There may be agreements in place between the domain and the external system to ensure that only domain employees are getting the data, but there is no way to enforce this at run-time from the domain's perspective: the external system opens a connection to the CDS, reads the data, and the domain is left to hope that the business policies of the domain are followed. The domain could get around this by creating a database user for each unique entity requesting access, but this brings its own challenges and is not recommended.

The risk that the business policies for data access are violated is greatly increased when an external system *copies* the data retrieved from the domain. Now, instead of contacting the domain for the data, it is sometimes possible for external systems to contact the copier of the data and retrieve the information there. Copying of data, with lack of enforcement of the original business policies, results in data breaches, and this very thing has happened at BYU.

Supposed that the external system is granted CDS access to a domain and copies data, but agrees to, and actually implements, the same policy logic for letting others access that data as does the source domain. What happens when the changing needs of the source domain require a change in the authorization policies? Now there is a potential for the two domains to be out-of-sync, and a user may be able to access data from one domain while being denied by the other. This causes security holes and prevents complete auditing of who has access to what data.

Ensuring that data access is only authorized according to the domain policies, external systems should never be given access to the CDS of a domain.

2.2.2 Application Programmable Interfaces

Where necessary, every domain should expose its data externally through the use of well-defined interfaces. To be well-designed, an interface must state what it accepts and what it returns. The definitions should include the technology used (e.g. REST HTTP requests), the format of the request, and the format of the returned information.

A common form of inter-domain traffic occurs via Application Programmable Interfaces (APIs). An API is simply a well-defined interface, that allows domains to request, transfer, and exchange data. In his book on API development [14], Sascha Preibisch gives a most excellent example of an API is a standard United States electrical outlet. The interface is well-defined: outlets accept 1) two-prong, unpolarized plugs, 2) two-prong, polarized plugs, 3) or three-prong plugs, and the outlet returns 120V RMS alternating current. Each side of this interface is only concerned about what it needs for its own business processing. The domain of the electrical grid does not care if the user is charging a phone, running a vacuum cleaner, or heating a room; it maintains its business logic of recording the amount of electricity used and providing power. On the user side (where the plug is at), not a thought is given to the electrical generators, substations, distribution networks, or even which breaker the outlet is routed through. Both domain and user do what they need to and go about their business.

Why are APIs an invaluable part of software domains? APIs are useful to 1) prevent access to the data stores of the domain and provide an abstraction layer for data transfer, 2) maintain the domain business logic and data definitions, and 3) preserve authorization to domain data and functions. In conjunction with the reasons laid out in the above Central Data Store section, these reasons will be enumerated more fully below.

2.2.2.1 Providing a Data Abstraction Layer

In order to avoid the security flaws and tight coupling that comes by tying a domain to the CDS of another, any two domains that need to utilize information between them should do so through domain APIs. The API provides a data abstraction layer that external systems can build upon without being affected by changes inside of the domain. When domain "A" publishes an API that domain "B" wishes to use, the definition of the API for domain A is static. Because the API is only an interface, A can modify the underlying structure and representation of the data without needing to inform B of changes.

Granted, for large changes in business logic, data presentation, or modifications to what data sets may be returned from a domain, new APIs should be created to reflect the updated/changed business needs of the domain. Once an API is presented to outside systems and is in use, it should not change. If something must change, then the API should officially be deprecated and the subscribers to the API should be notified.

2.2.2.2 Maintaining Business Logic

If external systems can write directly to a domain CDS, then the business logic of the domain is not protected. For instance, take a case where students can register for classes. If applications are written to go straight to the CDS of the enrollment system, then those applications could bypass the business logic that governs who can or cannot add a class, and for what reasons. Class sizes, course prerequisites, or financial holds may not be verified. To ensure that the business logic of the domain is executed properly, all applications should be required to use the domain APIs for class registrations. So doing will enforce that the business logic is checked during every attempt to add, update, delete, or view data in the CDS.

As a side note, the execution of business logic within an API does not prevent an application from attempting the same business logic. It is common for web pages to be developed with the business knowledge of a domain. This allows for the business logic to be checked in a real-time manner, allowing the user to receive feedback if something is amiss. Such checks should *not* be considered a substitute for building the business logic into the API layer, however, as in some instances it is possible to bypass the web page or modify things in the presentation layer, leaving *only* the API to ensure that the business logic is properly executed.

2.2.2.3 Preserving Authorization to Domain Data and Functions

APIs should generally be protected by an access manager that provides access control to authenticated identities, but the APIs should call the authorization service. It is possible that a user may gain access to a system without the use of the access manager, or it may be possible that the access manager has been misconfigured. As a good rule of security, authorization should always be checked and validated immediately before data is returned to any identity.

2.2.3 Policies Protect Domains

No matter how systems interact with a domain, the data is protected by policies, even if the policy is that all accesses are permitted. General Moore's Medallion, named after Brent Moore, a Systems Architect at BYU, shows the idea that no matter what is available within a domain, all components should be protected by a unified set of policies. One of the subtleties of this diagram is that the policies that protect the CDS, APIs, etc. are the *same* policies: if the same data can be obtained through multiple means, then the same policy should be used to protect the same data, no matter where it is accessed.

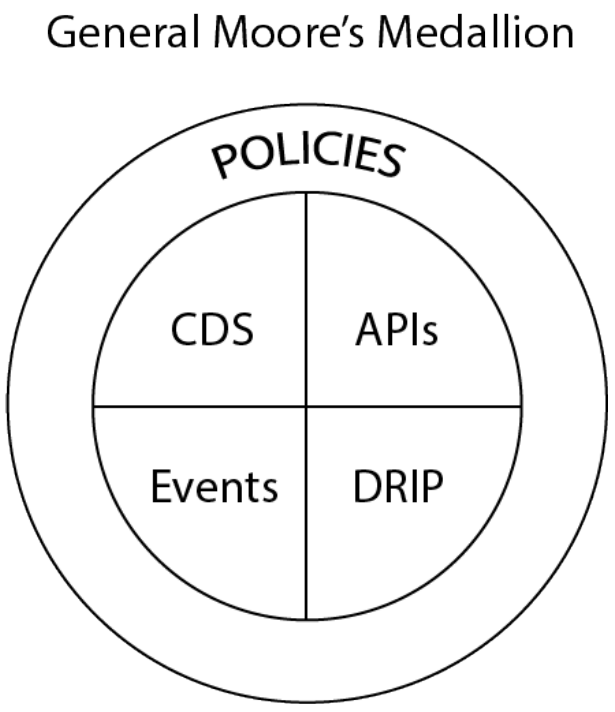

Figure 2-1: General Moore's Medallion. Shows that all components of a domain should be protected by authorization policies.

2.3 Historical Authorization Models

2.3.1 Pre-ABAC Designs

When large mainframe systems were first created, the United States government and military provided much of the funding for the initial research. In 1983 the first Department of Defense Trusted Computer System Evaluation Criteria (TCSEC) document was published, providing for various divisions of protection required for systems. The 1985 version of the TCSEC contained two specific access control schemas: Discretionary Access Control and Mandatory Access Control [15]. Role-Based Access Control was introduced later [16], followed by Attribute-Based Access Control [17].

Mandatory Access Control (MAC) and Discretionary Access Control (DAC) are often discussed as contrasts one with another. The fundamental tenant of MAC is that a user with access to a resource cannot delegate that access to another identity. The MAC methodology is still utilized by military security systems today as malicious actors cannot escalate privileges within the system without compromising the system administrator account directly. Users in a DAC system can delegate access that they have to other users. These two paradigms can be used in concert with each other, to some extent, where the MAC policies set boundaries on the limits of access that can be granted by the DAC components.

While MAC and DAC components worked well for military systems requiring strict regulation and enforcement, it was found that there was a need for expanded capabilities for the commercial and industrial sectors. This new model came in the form of Role-based Access Control (RBAC) [16]. This model provided for each resource to be protected by a permission. Each permission was associated with one or more roles, and users were assigned as members of roles. When a user attempted to access a system, the authorization engine checked for a path from the user to the permission required, through the roles.

The abstraction of roles allowed system administrators to change what roles granted which permissions without requiring the group administrators to modify memberships in the group. A person in the “Faculty” group might have an “Employee” role and an “Instructor” role. Should management decide that faculty should also have access to library resources, the “Library” role could have the “Faculty” group added as a member, and now anyone within the “Faculty” group would automatically receive permission to use library resources. This enabled access without requiring group administrators to know what roles governed which permissions.

While RBAC development simplified the maintenance required for system administrators [18], it led to an unintended consequences of “role explosion” [2]. With MAC an administrator had to manually assign users to resources, but with RBAC the administrator had to assign users to roles. Oft times this resulted in the administrators creating individual roles for specific permissions. Unfortunately, this methodology has resulted in lots of confusion in the industry, conflating what was intended for simplifying user assignments to permissions with what presently is [19].

If a role is nothing more than a binary access condition, then access is solely dependent upon the role, which negates valuable attributes such as employment status, job function, or other such possibilities. For instance, it may be valuable to have a user be an employee of an institution in order to grant them authorization to look at the management hierarchy for that organization. If the role of “Employee” grants that permission, and the user has the role, then they get access. An issue arises when the user has the role but is *not* an employee—they still get access, which breaks the intended model. A new method requiring more than just binary access was required.

2.3.2 Attribute-Based Access Control

Attribute-Based Access Control (ABAC) was developed to fill the role (pun intended) of allowing for real-time attributes from domains to be used when determining access. One of the best definitions of ABAC comes from the Executive Summary of the NIST publication on the subject, stating:

> "The concept of Attribute Based Access Control (ABAC) has existed for many years. It represents a point in the space of logical access control that includes access control lists, role-based access control, and the ABAC method for providing access based on the evaluation of attributes. Traditionally, access control has been based on the identity of a user requesting execution of a capability to perform an operation (e.g., read) on an object (e.g., a file), either directly, or through predefined attribute types such as roles or groups assigned to that user. Practitioners have noted that this approach to access control is often cumbersome to manage given the need to associate capabilities directly to users or their roles or groups. It has also been noted that the requester qualifiers of identity, groups, and roles are often insufficient in the expression of real-world access control policies. An alternative is to grant or deny user requests based on arbitrary attributes of the user and arbitrary attributes of the object, and environment conditions that may be globally recognized and more relevant to the policies at hand. This approach is often referred to as ABAC." [17, p. vii]

Much of the power in ABAC is the ability to rely on multiple pieces of data in order to calculate an authorization decision, as opposed to a single binary check for the permission within

RBAC. Carrying the RBAC specification further, Attribute-based Access Control attempts to provide an implementation pattern which uses policies and attributes. ABAC and RBAC can be used together to develop a robust authorization system [20-21]. While RBAC is considered access control for the purposes of this paper, it is possible to utilize a role within an ABAC system as an attribute that is checked by an ABAC policy, in conjunction with other attributes. In this way, it can be verified that a user has the role of "Faculty" while also verifying their active employee status from the HR domain.

The NIST publication on ABAC defines several different components for an ABAC system to operate [22]. For many instances, the process to get data from a domain begins with a user or other system that makes a request to the domain (see Figure 2). The domain verifies that the caller is approved for such data, then returns it. Several modern authorization systems modify this by placing a Policy Enforcement Point (PEP) before the domain. The PEP is responsible for calling the Policy Decision Point (PDP) which calculates the authorization decision, and if approved, the PEP passes the calling request to the domain. The domain gets the data and returns it to the PEP. The PEP may then filter the data, based on the authorization policies, before returning it to the caller. Here is the normal flow of modern ABAC systems:

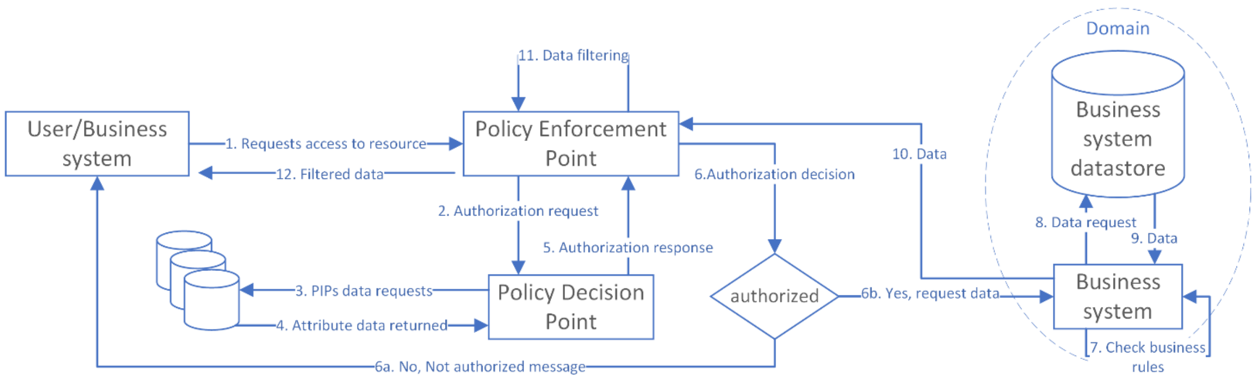

Figure 2-2: Modern ABAC authorization flow.

The flow of normal ABAC authorization, as shown above in Figure 2-2, can be broken down into the following twelve steps.

1. A user or system requests access to a resource.
2. The PEP takes the request, determines who/what is making the call, and sends a request to the PDP for authorization.
3. PIPs request data from other domain stores.
4. Attribute data is returned to the PDP.
5. The PDP calculates the authorization and returns a response to the PEP.
6. a. If the result is “Deny”, then the PEP is directed to return a “Not Authorized” message.
   b. If the result is “Permit”, then the request is forwarded to the domain.
7. The domain checks the business rules to see if it should send an error to the client or return the requested data.
8. If the business rules check out, the domain queries its CDS to get the data.
9. The CDS returns the relevant data to the domain.
10. The domain returns the data to the PEP.
11. The PEP may filter the data based on various authorization configurations.
12. The PEP returns the authorized data to the caller.

The above model is not the only model; there are many iterations on this theme. Some of the works based on ABAC are the Attribute-Rule ABAC (AR-ABAC) [23], restricted Hierarchical Group ABAC (rHGABAC) [24], Multi-Tenant ABAC (MT-ABAC) [25], and Lightweight Static and Dynamic ABAC (LSD-ABAC) for mobile environments [26].

While the original access control systems ran on mainframes, modern implementations recognize the importance for utilizing cloud architectures to run ABAC engines [27-28]. Some groups have designed ABAC implementations to protect cloud resources [23, 25], and many researchers have also attempted to use ABAC for IoT governance [29-30].

The transition from RBAC to ABAC is significant enough that numerous papers have been published on the subject [31–33]. RBAC need not exist separately from ABAC, though; the ABAC model can utilize roles and access control lists within its policies [34]. Various groups have developed their own policy grammars, created various approaches to obtaining attributes [35], and attempted to mine policies from attributes in access logs [36–38], while others have developed systems with feedback loops and machine learning to detect unusual access attempts [39]. Research has also progressed in attempting to extract ABAC policies from natural language descriptions [40].

One of the main standards for ABAC is XACML [41]. The XACML 3.0 standard utilizes XML to describe policies and policy sets. While XACML has the ability to express many different policy implementations, it is plagued with technical complexity [42]. This complexity has resulted in many attempts to make policies easier for the layman to understand [43], as well as attempts to simplify XACML policies [44-45]. Another, more subtle but dangerous problem with XACML is the ability to hide attributes from policies, possibly resulting in inaccurate decision responses [46].

Modern authorization systems built with the XACML standard include Axiomatics [4] and Plain ID [6], but there are other systems that use their own policy grammars such as Next Labs [5] and Open Policy Agent (OPA) [47]. The difficulty with these systems is that they break domain boundaries in their approach to obtain attributes, and/or the policy languages used are not simple enough to allow business owners to write the policies. As an example, Axiomatics will require their clients to purchase connectors that are designed to go directly into data stores to retrieve data for calculation authorization decisions. As an example with grammar difficulties, Rego, the language for writing policies in OPA is very expressive, but it requires significant technical development skills to develop policies with it.

The above problems led to the research question of how to create a system that protects domain data while making authorization policies for that data simple and concise. The research started with the idea of using the ABAC model to provide authorization in a way that does not tie directly to domain data stores and yet provides all of the attributes necessary to make an authorization decision. While simplicity was also important to the work, the development of a policy grammar was determined to be too much for the purpose of this research in the end, and the focus was placed solely upon providing authorization without the authorization engine crossing domain boundaries.

3 METHODOLOGY

"Digital identity is hard. [8, p. iv]" Just as accurately maintaining digital identity is difficult, authentication and access control also have their own challenges associated with them. The purpose of this thesis is not to attempt these items, but to approach authorization within the context of a known identity that has been properly authenticated and granted access to an endpoint. With these constraints, research could be performed for developing a performant authorization engine that respects domain boundaries.

While domains have been discussed at length previously, it is important to note that this research paper deals specifically with an authorization domain. The **authorization domain** is defined as a domain separate from the other business domains within an organization, retaining its own CDS, APIs, etc. The business of the authorization domain is the storage of authorization policies used by the other domains within an organization and the calculating of authorization decisions at the request of those other domains.

Behind the authorization domain is the **authorization system**. It is the authorization system that allows domains to craft authorization policies, and it provides a way for the domain to make attributes available for use in those policies. The authorization system is also responsible for an engine that calculates authorization decisions from the given policies and attributes. As a domain, the authorization domain protects its own CDS by providing APIs that the domains use for policy/attribute creation and maintenance, and it has its own policies governing who can update these items. The focus of this thesis is the research performed on creating the authorization system, specifically the development of a performant engine that can calculate authorization decisions from policies provided by other domains *without* crossing the boundaries of those domains in order to obtain the requisite attributes.

Specifying what was to be created was critical to the success of the project. Several different ideas and investigations allowed a concrete set of requirements to be formed, as listed below. The idea for a novel authorization system started with Doug Walker, a Systems Architect at BYU. He suggested that I build a simple authorization engine to gain some experience in the space. In his words, "We'll learn what we don't know." From that simple engine, the ideas for a full authorization system came forth, which are the foundation of this research.

One of the high-level concepts for the research was that all domains should document their authorization policies in the authorization system, even domains that have access to all of the data required for authorizations within itself. In other words, HR APIs should document their policies completely in the authorization domain, even if all of the required attributes for the policies only come from HR (employment status, etc.). By having all policies documented in one place, management and auditing becomes significantly simpler, along with the benefit of ease of maintenance should a policy need to change. The research should promote distillation of authorization into explicit policies.

3.1 Technical Guiding Principles of the Abacus

It was known that the authorization domain should not reach into the CDS of a source domain to obtain its attributes, but that alone was insufficient to guide the direction of the research. As the specifications for developing an authorization system were evaluated with industry professionals, the following key points in the subheadings below stood out as a framework of how to design the system. These specifications gave boundaries to the scope of the project, as well as a list of challenges to overcome.

3.1.1 Cloud-Native Distributed System

In the modern world of microservices, cloud architecture, and distributed systems, the new authorization domain being developed needed to support these types of systems while following the tenants of domain-driven design. It needed to be designed as a cloud-native application, providing authorization in a Software-as-a-Service (SaaS) setup. It should be highly available, given that the authorization engine is an essential component of infrastructure, thus necessitating a distributed architecture.

3.1.2 No Engine External Connections

Due to the emphasis on security and respect to domain boundaries, the design of the authorization engine should require no outside connections to other business domains. The only things that it should access are the various components required to make it run as part of its own domain of authorization. This means that the run-time authorization engine should have all of the information that it needs when it is invoked. Such a requirement does not preclude other components of the authorization domain from having some means of obtaining the authorization-important attributes without the run-time of the authorization engine, but how attributes would be updated was a major issue to be investigated.

3.1.3 Binary Decision Responses

One of the main considerations of authorizations is avoiding data leakage. From a security standpoint, if the authorization engine holds certain authorization-important attributes, then none of that data should be presented to other domains. The authorization engine should only respond with “Permit” or “Deny”, (i.e. it only answers yes/no questions). If the answer is “Deny”, then the API should return an HTTP 403 equivalent without a whole lot of other information—a 403 basically means "go away". If the answer is “Permit” then the domain should continue its processing of the request.

It was determined to provide only binary responses for a few reasons. First, as previously stated, was to avoid leaking data. If a domain or a malicious actor were able to query the authorization engine and have data returned, then it would be possible to have data disclosed that the owning domain does not want others to see. By removing data disclosure, domains can provide data for authorization without other domains actually being able to see the data that is used! Additionally, if the engine does not return data then the privacy of any user information within the authorization system is protected.

A second reason for binary responses is to avoid the possibility of data mining. Consider the case of a malicious actor who wants to obtain data about an individual or group of individuals. In a system that returns data, it may be possible for a malicious actor to craft several different queries and send them to the engine to return various pieces of data for one or more IDs. By returning only permit or deny, the caller cannot mine any of the authorization data because 1) nothing is returned, and 2) the policies are already existing and cannot be modified at run-time to produce data sets.

3.1.4 Authorization Questions Built from Domain Query Logic

A system will call an API for a business domain, and the business domain will then send an authorization request to the authorization domain. The questions that are presented to the authorization engine should be built by the queries to the business domain. By this it is meant that there are two components: 1) the URL request to the business domain (meaning the API query parameters), and 2) the data required by the authorization engine in the form of authorization-important attributes.

In order for the authorization engine to perform its duties, it requires certain data, such as the Subject and Target of the request, the ID of the client application making the calls, plus the policies that need to be checked. There could also be other data, such as the IP address of the Subject, the time of the request, and so forth. Because the authorization domain is queried by the business domain after the user has authenticated, some of this data could come from attributes provided by the authentication system.

The second group of data is the information gathered from the request to the business domain. Assuming a basic REST request as the simple case, the method lets the domain know what type of operation is being performed (e.g. GET, PUT, etc.) Additionally, data can come from the URL itself; the domain should not be looking inside of the body of the HTTP call to gather info required to ask for authorization, nor should it be gathering information from databases or other sources. Additionally, the system calling the authorization engine may pass data included in the URL (API parameters, etc.) in the authorization request.

3.2 Plans for Authorization System Implementation and Testing

Once the ideas and requirements had been fleshed out, the next phase was the creation of the system. It would require several iterations as new breakthroughs were discovered. After the system had been implemented and proved stable, it would be imperative that the system be tested for durability, reliability, and viability. The goal was to build a system that could be utilized by the BYU Office of Information Technology (OIT), providing authorization decisions to the APIs managed by OIT. This provided data from real-world operations, along with feedback from the software engineers using the system.

In addition to implementation in real systems, a battery of tests was designed for characterizing the efficiency of the system. This allowed for data to be obtained with specific requests under controlled conditions, as opposed to the unpredictable nature of requests from the real-world usage. The data allowed for comparisons against existing authorization products, proving or disproving the ideas set forth herein this thesis.

4 FINAL DESIGN OF THE ABACUS

This section of the thesis is dedicated to explaining the high-level design of the authorization engine created for this project, named “the Abacus”. The name comes from its design at the outset: the Attribute-Based Authorization Control University System. Later sections will describe the technical development of the Abacus, the experience of integrating the Abacus with real systems, and the testing done in a closed system to determine the efficiency of the Abacus.

To obtain better insights into the problems of authorization, the design and implementation of the Abacus was performed in close collaboration with the System Architects at BYU. With over 100 years of experience in the historical and modern issues surrounding system design, authorization, integration, and more, the Architects were an excellent sounding board for ideas and direction. Not only did they provide real-world instances of present problems, giving me use cases to consider, they also had ideas that challenged the current understanding of authorization. On more than one occasion they presented an idea of how to proceed, but it was not until hours of discussion had passed that I saw the nuances of what they were attempting to describe and the value of one design over another. Their assistance and guidance were invaluable to make the Abacus possible.

4.1 Authorization Flow

Desiring to provide increased security and efficiency to systems through an authorization engine, the existing ABAC model was evaluated. This evaluation led to a modified model, placing the authorization *behind* the business system. While access to the business system may still be controlled by an API manager, authorization for the specific data requested would be handled by the Abacus.

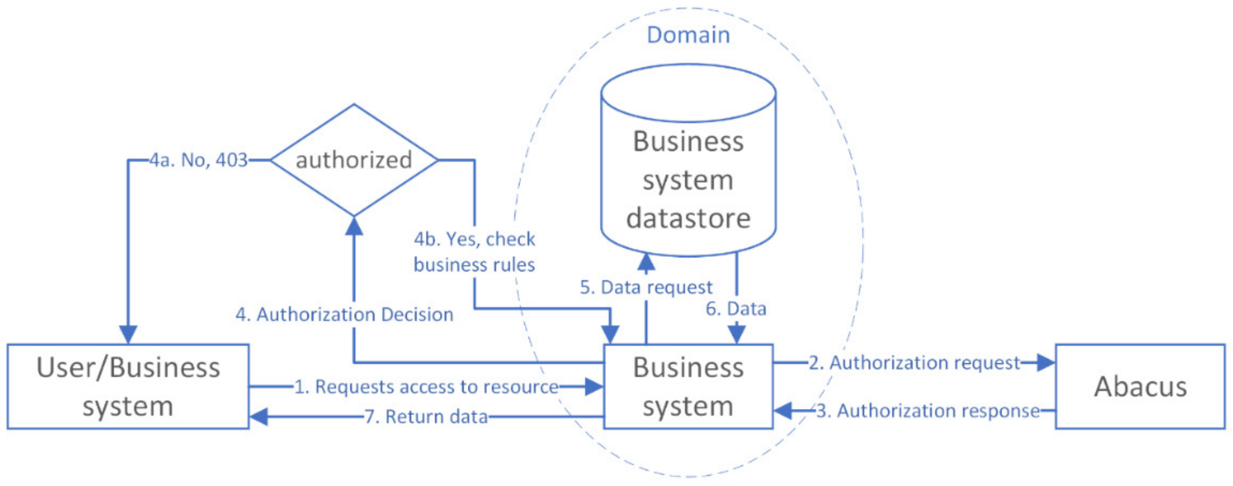
Domain
Business system datastore
authorized
4a. No, 403
4b. Yes, check business rules
5. Data request
6. Data
4. Authorization Decision
User/Business system
Business system
Abacus
1. Requests access to resource
7. Return data
2. Authorization request
3. Authorization response

Figure 4-1: Authorization flow for the Abacus.

The Abacus flow is shown above (see Figure 3) and described below. It sits inside of the same network as the business systems, and the architectural design is that *only* systems inside of this network can invoke the Abacus. This provides an additional layer of security since only trusted systems can invoke the Abacus for authorization decisions. The data flow of Abacus consists of 7 steps:

1. A user or system requests access to a resource.

2. The domain takes the request, determines who/what is making the call, and sends a request to the Abacus for authorization. The request to the Abacus will be designated as "the Request" when discussing this object in the future.
3. The Abacus calculates the authorization and returns a response to the domain. The response from the Abacus will be designated as "the Response" for the rest of this paper.
4. a. If the result is "Deny", then the domain returns a "Not Authorized" message.

   b. If the result is "Permit", then the domain checks the business rules to see if it should send an error or the requested resource.
5. If the business rules check out, the domain queries its CDS, with whatever business rules it requires, to get the data.
6. The CDS returns the relevant data to the domain.
7. The domain returns the data to the caller.

4.2 Policy, Set, and Authorization Check Designs

While policies form the heart of authorization, there are other abstracted layers that are important as well. For this thesis, "Policies" are the individual items that return a Boolean value. Collections of Policies are organized into "Sets", also known as policy sets, meaning a mathematical set of Policies that represent an action on a resource. Finally, a collection of Sets can be used to create an authorization check, or "Check", meaning an authorization check that is used when a domain client calls the Abacus to check authorization for a specific endpoint. Collectively these objects are called "PSA items", referring to "Policies", "Sets", and "authorization checks"—shortened to "Checks" for brevity herein this paper.

To help understand the distinctions between PSA items, an example case will be provided that the subsections herein will reference. Suppose that a user wants to update a phone number in a system. The user is allowed to make the update if 1) the phone number is assigned to the user, 2) the phone number is within a department and the user is the manager of the department, or 3) the user is an administrator over the phone system.

4.2.1 Policies

The fundamental idea of an authorization system is the question, “Can *x* perform *y* on *z*?” This question boils down to the evaluation of policies, statements which return a Boolean answer. Within the Abacus, a Policy represents an individual case of the possibilities in which the authorization will be permitted. For the given example above, each of the three cases are individual Policies. Each Policy is self-contained, meaning that it has no references, and thus no knowledge, of other PSA items.

A Policy uses various operators to perform comparisons between data. The compared data can be of two types: literal values such as the number 237 or the string “Active”, and dynamic values that come either from the Request object or from the attribute cache. With these constraints, a Policy can compare a property of the Request to a specific value, or it can check that the user has a certain attribute.

4.2.2 Sets

A Set is comprised of a set of one or more Policies, meaning that there are no duplicate Policies within the Set. A Set is assigned a decision of either “permit” or “deny”. If any Policy in the Set evaluates to “true” then the decision is returned to the caller. Should no Policy evaluate to “true”, then the inverse of the decision is returned. This allows Sets that require a condition to be met (standard policies) in order to return permission, but it also provides for policies that will permit *unless* a certain condition is detected (blocking policies).

A Set represents an action on a resource. As such, each Policy within the Set represents one way in which the policy decision can be fulfilled. It is recommended that each Set be named representing an action on a resource. For the given example, the Set described could be named "Can Update Phone Number" and be assigned a decision of "permit". This tells any person looking at the Set that should any of the Policy conditions pass, then the identity attempting to update the phone number is authorized for the action.

4.2.3 Checks

There are occasions when a domain wants to obtain multiple authorization decisions for the same identity. Instead of querying the Abacus repeatedly, the several Sets can be combined into a Check. A Check is set of one or more Sets. When a check is listed in a Request, the engine will evaluate each Set within the Check and return the authorization decision for each Set. This reduces the overhead of the domain sending multiple requests to the Abacus, and it significantly cuts down processing time within the engine since the same Request properties are used in every Policy evaluation.

4.3 Request and Response Objects

The Abacus has a very simple design for its invocation API and its response. The design was developed to make it as easy as possible for developers to integrate the Abacus into new or existing systems. Both the Request and Response consist of JSON objects with various properties. JSON was chosen as an implementation requirement as JSON is ubiquitous in web technologies today, and many tools are available for working with it quickly and easily.

4.3.1 Request Object Properties

When calling the Abacus, the Request must contain four specific properties: “subject”, “target”, “client”, and “check”. The subject is the system identifier of the identity that is attempting to do something. The target is specific to the Request: it could be the system identifier of a specific identity, the identifier for a specific object within a resource, or it may be set to “null” if it is not needed. The client is the system identifier for the client application that is making a request to the domain. This is useful because there may be multiple types of applications that request a resource from a domain, and it may be important to know which application the subject is using to attempt the function.

In addition to the required properties, any other properties that the domain sees fit to send may be included in the Request. This allows for Policies to be created with specific properties that it evaluates from the Request. Specifically, URL query parameters used to invoke the business domain APIs may be passed to the Abacus, and the Policies can check for authorization based on those parameters. This was a design decision based on feedback from the Architects, as they felt that it would allow for more dynamic policy design within the Abacus.

4.3.2 Response Object Properties

The Abacus will evaluate every Set within the Check requested in the Request. The Response object contains a property for each Set evaluated, the name of the property being the name of the Set, and the value of that property is the authorization decision returned, either “Permit” or “Deny”. In the event that something unexpected occurred in the Abacus, the value could be “Error”, at which point the domain should issue the request again, in the event that the error was due to some temporary state of the engine or network hiccup.

4.4 Overall System Design

The Abacus was designed with three major components: the authorization engine (akin to the ABAC PDP), the attribute cache, and a database for storing policies and attributes. Each component is crucial to the overall operation of the system; specifics are described below.

4.4.1 Persistent Storage

All PSA items must be stored in a non-volatile location. It was decided that a relational database would be the most effective means of accomplishing this task, given that there are many tools to integrate with database, make backups, and recover systems. The database would also be used to store a copy of all existing attributes known to the Abacus. By storing a copy of the attributes in the database, multiple instances of the cache could be instantiated with full records of the attributes. Additionally, if there was ever a concern that a cache was out-of-sync, it could be flushed and reloaded with the full set of attributes from the database.

4.4.2 Attribute Cache

As seen in the design above, the Abacus does not call out to domains for authorization-important attributes as it processes requests. This requires that the attributes be cached with the Abacus. When an authorization request is made, the Abacus checks its attribute cache to see what the value of the requested attribute is. If the attribute is not found, then the attribute is assumed to not exist.

The decision to only store needed attributes allowed for a smaller cache than a system that requires all attributes to be stored. For example, when using the Abacus at BYU, half of the 40,000 IDs may be employees of the university: 10,000 full-time employees, adjunct faculty, and university associates, while the other 10,000 may be students that work for the university part-time. Instead of requiring all of the IDs to have an attribute about employee status, only half of the IDs will require that attribute, saving considerable space.

4.4.3 Engine Design

In order to make the engine fast, it was decided that the engine should ingest all active PSA items upon initialization. Each item is stored in a compressed, optimized format for the engine. Whenever a Request comes into the Abacus, the engine already has the objects necessary for the evaluation.

Upon receiving a Request, the Abacus loops through each Set in the Check that is specified in the Request. As an optimization, every Policy that does not require attributes from the cache (these are called "fast" Policies) is evaluated first, and if a policy returns "true" then the Set is given the authorization decisions that is associated with is. Should none of the fast policies provide a decision for a set, then the engine makes a call to the cache for all of the required attributes in the remaining Policies for that Set (also called "slow" Policies), and then the slow Policies are evaluated. The efficiency of this model is borne out by the results in Section 6.2.5.2 which discusses a test of fast Policies.

4.4.4 Attribute Updates

The most novel contribution of the Abacus is the idea to use an event-based system for updating attributes, leading to significant performance gains and the respecting of domain boundaries. The NIST ABAC proposal and many commercial offerings discuss the possibility of caching data [17, 48]. These publications admit that cached attributes may provide a security risk if the attributes are stale, but this consideration comes from the fact that these systems expect the authorization domain to query the source domains for updated attributes. There are discussions about the proper time-to-live (TTL) for cached attributes, and the variability of the TTL depending on the criticality of the attribute [49].

With the design of the Abacus, the onus is on the source domains to keep the attributes updated. This reduces network traffic as the authorization engine does not need to continually poll domains for attribute values. Another effect is increased security, since the attribute only needs to be sent across the network once, at the time that the attribute changes. Because the attributes are only updated when they change, all attributes are persisted in the cache with no TTL requirement.

4.5 Advantages of Proposed Flow

There are several advantages of the architectural design that the Abacus makes use of through updating attributes with evented frameworks. These improvements are described below.

4.5.1 Security

The most important factor in the architectural design of the Abacus is the focus on security. Unlike other models, the Abacus *never* reveals data to calling systems. By restricting answers to "Permit", "Deny", or "Error", systems can use attributes without having access to the data! There are no worries about compliance issues, data sharing agreements, or anything else that typically accompanies systems that can share data.

By requiring domains to *push* attributes instead of the Abacus reaching into data stores, the Abacus is not a valid attack vector for gaining access to domains or their data. Where other systems require an authorization user with access to most (if not all) data, the Abacus has no such requirement. Domain boundaries remain intact and secure, and the Abacus is not tightly coupled to the data store. Since attributes only need to be pushed when they change, the attribute data only traverses the network *once*, instead of every time that the authorization engine queries a domain data store. This reduced data motion reduces the risk of interception on the network.

Since domains are responsible for maintaining the attributes and the definition of them, there is no need to store sensitive data inside of the Abacus. For instance, take a policy that requires the user to be over twenty-one years of age. Where other systems commonly approach this by storing a birth date and asking, “Is the current date >= birth date?”, it is proposed that the Abacus store an attribute called “Over 21”. If the domain that knows the birthdate detects that the user is now over 21, then they should push that updated attribute to the cache. This ensures that the cache never contains any data that could be used by a malicious actor to any real detriment, and it distills the authorization question down to its fundamental statement, which, for this example, is “Is the user 21 or older?”

4.5.2 Consistency

For domains that have multiple points of data access, it is unfortunately common to have policy variance between systems that protect the same pieces of data. The problem arises when one system is updated while the other is not, allowing a user to invoke the two endpoints and get *different* authorization responses. For instance, if there is an API that returns a username, as well as an event that provides the username of a new account, then both should use the same policy to ensure that both grant the same authorization to the data, but too often there are discrepancies; one might deny the authorization while the other permits it.

Why does this policy inconsistency occur? Often it is a function of authorization embedded in the code of systems, which means that if the business changes the authorization for one piece of data, the code has to be updated directly, requiring developer resources, planned outages for upgrades, and so forth. If two domain endpoints can return the same data and the business changes the authorization policy, a failure to update both endpoints simultaneously can result in discrepancies, *especially* if one of the endpoints never gets updated!

The Abacus solves the issue of policy consistency by abstracting the authorization to a central system, e.g. the authorization domain. The novelty of Policies within the Abacus is that multiple Sets can reference the same Policy without needing to recreate it between Sets, allowing business owners to know for sure that the exact same definition is utilized across multiple endpoints. The business can set the policy once and ensure that every system relying on that Set will receive the same decision in the same way. This is different from the XACML 3.0 standard, which allows policies and policy sets to be included within each other, because of the simplified structure of PSA items within the Abacus. Some previous literature on policy consistency has erroneously concluded that a central policy system would be a bottleneck [50, p.75], though this was due to the method in which the authorization in the literature was implemented.

4.5.3 Efficiency

The most time-intensive part of any cloud system is time spent waiting on network requests. For instance, when executing a request to the Abacus, which is hosted in the Oregon region of AWS, the entire request from Provo, UT may take anywhere from 68 ms to 135 ms, depending on network conditions. When reviewing the request logs inside of the Abacus, actual engine execution time ranges from microseconds to a few milliseconds, depending on the number and complexity of policies in the request. This means that, on average, the network is fifty times slower than the actual time to calculate authorization decisions! The design of the Abacus reduces the network traffic for real-time requests to a minimum.

The first optimization is the caching of attributes. Where other systems request attributes from domain data stores or from domain APIs, the run-time engine of the Abacus works with the local cache of attributes that were previously pushed to it. This means that the network time required for other systems to obtain attributes is significantly diminished in the Abacus.

A similar optimization for the Abacus is the storage of Policies, Sets, and Checks *inside* of the engine. Any system that requires the querying of a data store to get an authorization policy incurs time on the network, as well as the time to parse the policy to ensure its validity. Since the Abacus parses all policies on initialization and stores them in memory, efficiency in retrieval is unparalleled. Additionally, the policies are referenced through a series of pointers, making everything fast and compact within the engine.

4.5.4 Reliability

When an external domain goes down, any domain that relies on that domain to answer its authorization question would normally be hung as well, but with *caching* of attributes, domains can go offline at will without affecting the answers that are returned by the authorization system. Some might say, "But what if the attributes change while the domain is offline?" Remember: since the domain is offline, then the attributes of that domain *cannot* change, since the domain is unavailable to change anything. It is up to the *domain* to check for changes while it was down, and then push those changes accordingly: the authorization system is not expected to know the business of the domain, only respond to it.

This decoupling of attribute availability from the source domains makes it easier for the domains to perform needed maintenance. While the domains will still need to inform users of upcoming periods of unavailability, other domains that rely on the said domain for authorization attributes will have no need to coordinate, since the attributes that they need are still available within the Abacus. This further reduces inter-domain dependencies and helps prevent catastrophic failure to all of an organization's systems when a critical domain goes offline.

Another example of reliability in the Abacus is the cloud-native nature of its design. It can be deployed in multiple locations around the world, with auto-scaling capabilities, to handle large volumes of traffic without a reduction in efficiency. Also, because of the ease in deploying instances of the Abacus, the data could easily be sharded, according to the desires of the admins.

4.5.5 Visibility

Lack of visibility is one of the main issues with systems that rely on directory services or groups to grant authorization. Directory services or group management tools most often do not have a way to know which systems make use of what directories or groups. This is all too common in industry, and the pain caused by this void of knowledge is well illustrated by a story from the BYU Office of Information Technology.

Years ago, BYU had a group within its group management system that represented part-time employees of the university. The government changed the definition of part-time employee, which necessitated the creation of new groups, and the migration of existing systems to use the new groups. Unfortunately, there was no way to tell who was using the old group. To figure out what systems needed to be updated, the engineer in charge of the change would remove the old group early every Monday morning. Once a couple of people had called to complain that their systems were no longer working, the engineer would restore the old group and spend the week helping those systems migrate to the new groups. After six months of effort, all systems were using the new groups.

Contrast this with explicit policies available in the Abacus: the policies state exactly what attributes are required from which domains, which allows a simple query to see exactly which systems rely on what attributes. If a domain needs to change the definition of an attribute, an administrator of the Abacus can easily enumerate what systems need to change, based on the users of that attribute.

Another visibility factor is the logging that occurs when requests are made to the Abacus. Each request is logged for auditing purposes and for gathering metrics. This allows operators to find out how often requests occur, which systems are requesting authorization the most, and by extension, which attributes are most valuable to the entire organization. All of this data allows domains to make informed decisions when considering changes to systems.

4.5.6 Domains Own Attributes and Definitions

One of the main problems that exist in systems which utilize cross-domain attributes is the definition of those attributes. If multiple domains use the same data *differently*, then how can the definition of the data be preserved? If systems use a group management tool that does not describe the exact conditions required to be a member of the group, then domains (or even different systems within a domain!) might assume the definition to be something other than what was intended by the group creator. For example, a group named "Students" might be referenced by one domain to mean "undergraduate students", while a different domain might interpret it to mean "full-time students", while yet a third system operates on the group as if it includes anyone that is currently taking a class.

There must be governance to ensure that attributes are used properly, yet an additional problem occurs when multiple domains are allowed to update the same attribute. An organization with this data model is bound to step on its own toes if there are conflicting sets of rules that permit updating.

With the Abacus, only the domain owning the attributes pushes them, and thus they own the definition. This prevents multiple domains from changing the same attribute. Anyone expecting to use an attribute from a domain must contact the domain owner to know what the business definition of that attribute is, how it is updated, etc. An attribute owner can record the specific definition of an attribute so that when others want to use it, they know precisely what it entails and its intended function.

5 DEVELOPMENT OF THE ABACUS

The development of the Abacus occurred in four main stages. During this process, numerous conversations were held with industry professionals to gain insight into what use cases such a system would need to handle. Multiple versions of the engine were then created, each building upon the work of the former, combined with additional information from technical experts, resulting in the present form. The final version was deployed and utilized in production systems at BYU to evaluate its viability.

5.1 Minimum Viable Product

This first design of the engine was to be the "minimum viable product" or "MVP", a simple engine that accepted an authorization request, processed a basic policy, and returned an authorization decision. This task was completed in three months, providing the following data.

It was decided that the MVP would be compliant with XACML 3.0 [41] in order to leverage an existing standard and allow integration with existing toolsets. After studying the specifications, it was decided to base the MVP on the XACML standard, using the JSON extension [51]. At that point, the engine was written using the spec as a guide, but it was not implemented with XML. It would return the several different types of authorization responses, including "Permit", "Deny", and "Indeterminate". It had some of the functions listed in the spec available, but not all.

When the MVP was complete, several things became apparent. First, XACML was deemed to be too cumbersome, too verbose, and too complex for users to understand. One of the fundamental tenants of the Abacus is simplicity. XACML requires so much to specify even a simple attribute because it tries to be everything to everyone instead of focusing on one specific model and proceeding forward with it. A new model for attributes would need to be created.

One of the difficulties of the XACML standard is the ability to reference other policies and policy sets from within policies and policy sets; doing so enables the possibility of circular references. Circular reference checking is a pain to implement, and should there be a problem with it, the system would crash as it recursively calls through a loop.

Another issue was the possibility of returning an “Indeterminate” response. Existing systems all have the ability to make a determination, so an authorization engine should be able to do the same. The issues with various types of “Indeterminate” responses within policies made it difficult to simplify the system.

5.2 Designs for Version Two

When designing the second version of the Abacus, XACML 3.0 was decided to be too complex and verbose to be used for the project. It was decided that all policies, requests, and responses would be implemented with JSON as it was simpler, less complex, and more commonplace than XACML. This also allowed easy integration into the codebase.

The second design point was to prevent circular references. For simplicity, it was decided that Policies can reference attributes, but they cannot use other Policies or Sets, and Sets can only include Policies. This means that there are no circular checks required, and it makes it much easier for users to keep track of what Policies are part of a Set.

From a business perspective, most domains do not want those from other domains to have the ability to modify their policies. This led to the creation of two different types of policies: global policies, and domain policies. Global policies are created only by the administrators of the Abacus but are includable in any Set. Domain policies can only be modified by the admins of the domain to which they belong. This format also allows Policies and Sets to be namespaced by the domain name, preventing naming collisions between domains.

The policy engine exists as a piece of infrastructure, requiring that it execute as fast as possible. To avoid checking a data store for policy rules on every invocation, the Policies and Sets were designed to be pulled into the engine upon its initialization. When a request comes into the Abacus, it only has to check its internal memory space to find the Policy. Additionally, because the initialization verifies that the policy grammar is correct, the run-time never has to check the Policy for proper construction. Policy retrieval and execution is far more efficient than if a database were called on every engine invocation. Should a Policy or Set need to be changed, the change can be pushed to the Abacus, it can be verified, and then stored in the internal memory space if all is well.

5.3 Designing Version Three

In meeting with various software development groups, questions were raised about allowing for multiple Sets to be returned by one authorization request. After discussing various options, it was determined that an extra layer of abstraction should be added to the authorization request. The authorization check was decided to be added to the request. The Check is a mapping of Sets to a single Check. When a Request references a Check, every Set within that Check will be evaluated and a decision will be returned for it. The calling application can then check the decision of each Set in the response instead of calling the engine multiple times.

Implementing the Check required adding more code to the engine and another table to the data store, but it significantly reduced the execution time when evaluating multiple Sets for the same request data. Even when adding up to five Sets within a Check, the engine showed little appreciable difference for calculating one Set versus evaluating multiple.

The most significant time delay occurs with network traffic, so to minimize that, the engine was given two main optimizations: 1) Policies that require no attributes are evaluated first, and if the rule conditions are met, the decision is returned immediately. This saves a call to the attribute cache. 2) Any policies requiring attributes have those attributes included in a single request to the attribute cache, meaning that *at most* there is one trip across the network to the attribute cache. These optimizations allow the engine to operate in the sub-millisecond time frame for evaluating decisions.

5.4 Version Four Development

It was during the development of version three that Brent Moore presented to me the idea of pushing attributes into the cache from the source domains. This became the most valuable point of the whole project, and an interface was created to allow domains to push attributes to the Abacus. This used a simple AWS lambda that would verify that the invoker was authorized to push attributes of the submitted type (by making a call to the Abacus), and if permitted, the attribute would be updated for the listed identity. This process was tested with thousands of updates every day, and it worked well.

5.5 Discussions with Professionals

In the course of this research project, several professionals from OIT were consulted. Some of the valuable things that I learned are listed below.

5.5.1 Wisdom from Doug Walker

Doug Walker has been an employee at BYU for over forty years, and he has seen development with the original BYU mainframe up through the current cloud systems. His depth and breadth of knowledge allowed me to consider many use cases that I had not previously thought of when designing the Abacus.

One of Doug's important points was the idea of how to phrase an authorization question. Because of the binary nature of responses from the Abacus, Doug would frequently state, "What is the question being asked that gives the 'yes' or 'no' answer?" By looking at the authorization in this light it sometimes changes how authorization is approached.

Another contribution from Doug was the idea of "authorization-important attributes". When a database is queried for data, and that data is checked for authorization, it becomes difficult to separate the data from the attributes that truly matter for authorization. This goes back to the example of using a birthdate to see if an individual is twenty-one or older. From a developer's perspective, they would query the database for the birthday of the individual and then check if the date, compared against today, makes the person at least twenty-one. By looking at authorization-important attributes, the birthday is unnecessary; just the attribute of "isAtLeast21". This distillation process is subtle but absolutely required to truly find the attributes that domain policies rely upon.

5.5.2 More with Brent Moore

As stated previously, Brent Moore is the Chief Engineer over the System Architects at BYU. When Brent proposed the idea of pushing attributes into the Abacus, the research had progressed to require the Abacus to call domain APIs to get the data needed for Policy evaluation. Brent pointed out that this still requires the tight coupling of the authorization system to the domain, since the authorization system needs to know the specifics of what data to query for and how to transform that data into the attributes needed for the policies. He then discussed the advantages of letting the domains push fully-transformed authorization-important attributes, including the decoupling of the authorization and source domains, decreased latency, and a removed requirement for the authorization system to poll for attribute updates.

5.5.3 Research with Greg Richardson

Greg Richardson is the Identity Architect at BYU. After publishing my first paper [52], Greg was invited to read it and provide feedback. Through those discussions, Greg pointed out that many of the issues with modern authorization systems is because they rely on a bad data model. RBAC itself can be used as a sub-component of an ABAC system, but because many modern systems do not use a level of abstraction where users are placed in groups, groups are given roles, and roles are attached to permissions, the common issue of group explosion occurs. Similarly, poor authorizations are made because organizations do not start by reviewing their data model before they attempt to create authorization policies, which leads to difficult, long-term, entrenched difficulties in creating granular authorization rules.

5.5.4 Conversations with Alan Karp

While attending the Internet Identity Workshop (IIW), a conference held twice a year in Mountain View, California, I met Alan Karp, who has worked in authorization systems for much of his career [18]. Alan had experience performing research for military systems, and his favorite comment to me was, “Chaining and attenuation are critical to security. Without chaining, every private in the army is saying, 'Yes, sir, Mr. President.' Without attenuation, that private ends up with permission to launch nukes.” [53] This phrase highlights the importance of utilizing a dynamic authorization system. While the ideas of chaining and attenuation were not evaluated as part of this research project, they are important considerations for authorization and merited inclusion in this section for future reference.

5.6 Fulfilling the Guiding Principles

The Guiding Principles of the Abacus mentioned in Section 3.1 required various design breakthroughs to fulfil the objectives of 1) building a cloud-native, distributed system, 2) preventing external connections from the Abacus to other systems, 3) returning only binary responses from the authorization engine to prevent data leakage, and 4) construct authorization questions from domain logic. Accomplishment of each of these items and the research breakthroughs are listed below.

5.6.1 Cloud-Native Design

The first step to making a cloud-based solution is to pick a cloud provider. Amazon Web Services (AWS) was chosen because of its maturity, available tools, and the experience available with OIT. All components were designed to be deployable via Handel, an Infrastructure-as-Code platform developed within OIT. This deployment strategy allowed separate instances of the entire stack for the Abacus to be deployed in multiple regions, with multiple domains, or even to have two entire instances running adjacent to each other to provide redundancy for critical systems that require agreement from two systems.

The Abacus was designed with the capability to run multiple instances of the authorization engine behind a load balancer. If the organization needs to scale up or down the number of engine instances, it is simple through AWS to enable auto-scaling rules that will automatically increase or decrease the number of instances based on usage. Also, AWS provides the ability to set the number of running instances, which is useful for times of known load increases (such as the first week of a semester at BYU).

5.6.2 Authorization Autonomy

Besides connections into domain data stores, many existing systems also leverage the ability to call domain APIs to obtain the attributes needed for authorization decisions. The problem with this setup is that the *authorization engine* is required to know about the domains and their attributes. It is much more effective and efficient to have the domains know how to get their attributes into the authorization attribute store than to require the authorization domain to write connectors into every other system.

The major breakthrough was the idea to *push* attributes to the Abacus from the domains whenever an authorization-important attribute changed. Brent Moore, Chief Engineer over the System Architects at BYU was the one to present this idea. Since the domains own the definition of their authorization-important attributes, then the domains know when enough has changed to require an update in the authorization domain. Pushing of authorization-important attributes decouples the authorization domain from the other business domains, allowing for a truly distributed authorization architecture.

5.6.3 Separating Authorization from Business Logic

The design objective to only return binary responses revealed the need to differentiate between *authorization logic* and *business logic.* What, then, are the differences? **Authorization logic** is that component that when evaluated to false prevents the user from function execution, halting the user outright. **Business logic** consists of the internal checks within the function that are required to complete function execution, and if failed, allow for some sort of specific error returned to the user about what failed and why.

When first gathering use cases from business domains, it was common for people to try to push all of the business logic into the authorization policies. This led to instances where authorization requests for a user to a domain could result with a “Permit” answer for one function request, yet a “Deny” would be returned for the same function when provided with a different set of parameters. These inconsistencies led to a deeper research of what actually constitutes authorization and what does not, leading to the following revelation. If the answer is “Deny”, then the API should return an HTTP 403 equivalent without a whole lot of other information—a 403 basically means "go away". If this is not an acceptable answer to return to the client/user, then it is probably not authorization logic but business logic instead.

As an example, consider a case of a student attempting to enroll in a class. There is business logic specific to certain classes stating that a student may not be allowed to enroll for a particular class unless he is in one of the required majors. Should a student without one of the requisite majors attempt to enroll, simply telling the student to "go away" means that the student will end up calling someone for an explanation of why he can't add the class but his best friend can. Instead, we very much want to return an error code and a great explanation of the issue. Clearly, this check of required major should be in the business logic of the domain API and not in the authorization policy.

On the other side, let us assume that someone who is not a student, yet with an identity and valid authentication credentials at the university, attempts to register himself for a class. The authorization logic could require that the individual is indeed a student, and if not, he should work in the office of the registrar or be a college counselor that works with students and their schedules. If none of these policies are met, then the individual should be met with a “Not Authorized” message and be given no further information.

Determining this separation of authorization and business logic was critical to determining how best to approach the development of authorization Policies and Sets.

5.6.4 Authorization Questions from Domain Requests

The XACML 3.0 spec provides for the HTTP method to be part of the request to an authorization engine. While developing the Abacus it became apparent that instead of following the standard design of sending in a method and a resource separate from each other, authorization should be performed as an action on a request. Since the business domain accepts the initial request from the caller and then sends a Request to the Abacus, the domain can modify an HTTP method to be something more meaningful. For instance, if the user is trying to start an oven via an IoT protocol, there is no good REST method or CRUD operation that maps well to this function. With the Abacus, however, there could be a Set called “Start Oven”. The Abacus does not care what method is used to invoke the domain. By making calls to the Abacus method-agnostic, it can service ANY type of authorization request without being locked to HTTP. This makes it forward-compatible with new technologies that may emerge in the future.

Because the Request object enables any property to be added to it, which can subsequently be used to provide data to Policies, domains are not restricted in the data that they can send. Another unexpected benefit of this is that when data is provided to the engine, not every property sent in the Request need be used.

6 IMPLEMENTATION AND PERFORMANCE TESTING RESULTS

6.1 Real-World Implementation of the Abacus

The Abacus was designed to be as simple as possible for everyone to use, both the business owners and the development teams behind them. It was deployed for use at BYU, running in “silent pilot mode” for a system used by the Department of Continuing Education (DCE) and within one of the Identity systems. In this configuration, The Abacus was invoked for every authorization check that the system required. The system would also check its hard-coded authorization logic. If there was an error, with The Abacus, or if there was ever a discrepancy between the decisions returned by the built-in code and the decision from The Abacus, then it would be logged and reported. This section discusses the experiences of how The Abacus was used by DCE and the Identity Team.

6.1.1 Deployment

The Abacus was designed to be cloud-native, allowing it to take advantage of autoscaling, high-availability, and distributed architectures available through cloud providers. An “Infrastructure as Code” platform called Handel was chosen to simplify this deployment process. For the initial development, and for the actual production implementation, Handel was used to create the stacks in AWS. This ensured that the Abacus deployed with the proper components every time, and that the various pieces of architecture were able to communicate with each other.

Deploying the components of the Abacus with Handel was simple, but it took about forty-five minutes for the entire stack to become operational. Most of this time results from the instantiation of the database, which can take over half an hour to become operational. Once the was deployed and running, a simple SQL script was executed to create the database tables required for system operation.

6.1.2 Policy, Set, and Check Creation

While effort was taken to simplify policy creation, a user interface was never developed as part of this work. As such, all policies, sets, and checks require manual creation. Policies only take a few minutes to create, at most, while sets and checks only involve creating a list of the policies and sets included within them, respectively. This means that a domain can implement its policies in under an hour, by hand, and from scratch.

The two Policies required for DCE were relatively straightforward and simple. One policy required no attributes, and the other policy only required one. There were only two Sets needed, one requiring the Policy with the attribute, and the other Set used both policies. These took only a few minutes to create. The Checks required for DCE were simple, but numerous. There were forty-nine different API endpoints in the DCE system, requiring forty-nine different checks (one for each endpoint). Thirty-four of these Checks called one Set, while the remaining fifteen Checks called the other Set.

The reasoning behind implementing a different Check for every endpoint is simple: while they all use one of two Sets, the abstraction of a Check 1) provides visibility into what is being executed as all calls to the Abacus are logged, 2) it allows Sets to be added or removed from the Check without breaking the code that is calling the Abacus with a given Check, and 3) it allows an endpoint to completely change its authorization without affecting the other Sets or Checks. If endpoints A, B, and C all called the same Check, should the business owner decide to change authorization for one of them, they may be tempted to go change the Check, which results in unintended changes to the other two endpoints. With each endpoint calling a separate Check, the business owner can modify the authorization for specific endpoints without any concern of changing other authorization calls, a problem that occurs quite often in modern ERP systems that use roles or groups to manage authorization. Additionally, this abstraction is much like the original RBAC spec where roles were originally designed to be hidden from the users, with roles being the clearinghouse between groups and permissions.

The policies required for the Identity systems were much more complex. Sets required multiple policies with varied attributes. In fact, it was an excellent exercise to see the difference in authorization requirements for the two systems. Surprisingly, while the policies were difficult to understand, once they were decomposed into individual, separate policies (instead of large conjunctions), they were rather straightforward. There was some difficulty in deciding *how* to represent certain attributes: the issue being a result of the data model used for these systems.

Surprisingly, by far the most challenging problem with PSA creation was not the technical aspect, but the paradigm shift required in the minds of the business owners. People had become accustomed to thinking, "Person *X* is allowed to do *Y* if they are in group *Z*." I often asked, "*WHY* is Person *X* in group *Z*? What are the attributes that determine if a person is in that group?" It was difficult to elicit a response that truly got to the attributes that could be used to programmatically determine if someone was authorized for a function.

These varied experiences showed that it is easier to implement authorization in a new domain with domain-driven design and a proper data model than it is to retool a system with a convoluted data model. Additionally, it takes work to help people understand how to approach authorization from a true attribute-based perspective.

6.1.3 Endpoint Implementation

With the Abacus deployed and the Policies, Sets, and Checks for DCE and the Identity systems implemented, the last thing to do was to modify the domain code to call the Abacus for its authorization decisions. A simple POST request is all that is needed to query for authorization. The actual body for the request changed multiple times over the development of the Abacus, but it became simpler every time. Eventually the payload ended with only four required properties as described in Section 4.3.1.

A wonderful experience during the implementation phase occurred while working with Brian Wheelhouse, the software engineer assigned to implement the code to call the Abacus and check the response in the DCE system. Brian was shown how to call the endpoint for the Abacus, with each API referencing its own check based on what operation was to be performed. It only took a couple of hours for Mr. Wheelhouse to write a function for his code that abstracted everything besides the Check to be referenced and call that function from the endpoint paths for his API. Once he had finished he asked what else needed to be done, and upon learning that his work was complete he replied: “That was it? That was easy!” Not only was this a testament to the simplicity of implementing the Abacus within other systems, it also generated satisfaction to those involved with the project.

6.1.4 Real-World Tests

It was vital that the Abacus be checked for accuracy against the hard-coded authorization logic in the DCE system. While running in silent pilot mode, an alert system was created that would tell Mr. Wheelhouse when there was a discrepancy between the hard-coded logic in the DCE endpoints and the responses from the Abacus. In the beginning there were a couple of issues, but they were discovered to be faults in the hard-coded logic and not the Abacus! Everything was running correctly on 17 December 2019.

All told, The Abacus has run for over seventeen months in silent pilot with DCE without a single engine error, and at the time of this writing, DCE is still invoking The Abacus. With the addition of the calls from the Identity Team applications, the Abacus consistently received more than 34,000 requests, sometimes up to 38,000 requests, per day. Even more impressive is the fact that the authorization engine was only running on a single AWS t2.micro instance the entire time, which has only 1 vCPU and only 1 gigabyte of RAM, without any problems! This proves the stability of the Abacus in real enterprise systems. The potential for scaling and stability is unmatched, especially when utilizing larger hardware as described in the next section.

6.2 Performance Test Results

To prove the efficiency of the system, several tests were designed to run the engine through various policies and configurations to give us hard numbers on how fast the system would operate. One of the major commercial entities that offers software in the authorization space tests their deployments with one million user IDs and thirty million attributes corresponding to those IDs. For testing architecture, the commercial entity used five instances of their authorization engine (PDP), fronted by a load balancer.

It was determined to create a similar setup for the Abacus to allow testing against a known baseline. In addition to generating users and attributes, a series of Policies, Sets, and Checks would need to be written against those attributes so tests could be executed. The architecture used to run the tests would also need to be as close to the commercial model in order to provide an accurate test.

6.2.1 Hardware Configuration

The architecture required an elastic beanstalk configuration of five instances of the authorization engine, all serviced by a load balancer. While the commercial system uses five instances of storage, it was determined that only one instance of a cache for the Abacus would be required. The commercial baseline utilized hardware with 8 vCPUs and 30 gigabytes of memory for each instance of their engine and data stores. For the Abacus, the engines were placed on t3.2xlarge processors which have 8 vCPUs and 32 gigabytes of memory. The cache was set to run on a cache.m3.2xlarge processor which also has 8 vCPUs and 32 gigabytes of memory.

6.2.2 Generation of Users and Attributes

Attributes were generated by creating a script that produced one million IDs and thirty million attributes to associate with them. The script pushed the generated information into the cache before the test battery. The user IDs were simply the numbers 1 through 1000000. The attributes were then generated in a deterministic fashion. Attribute descriptions and frequencies are listed in the sections below.

The attribute generation process resulted in 7.25 million sets of attributes to store 30 million values. It required 3.01 gigabytes of disk space to store the attributes, meaning that it only requires one gigabyte to store 10 million attributes in the Abacus! Also of note, the attributes only consumed 14.37% of the available memory in the cache. This means that the Abacus can store almost 200 million attributes on a system with 32 gigabytes of memory.

6.2.2.1 Gender

Every ID had a gender. Three would be male, then the next three would be female, and so forth. There were 1,000,000 gender attributes generated.

6.2.2.2 Employee Status

Only 1 in 4 IDs were given an employee status. Of those with this attribute, 60% were given 'A' to represent active status, 30% were given 'R' to indicate retired status, and the remaining 10% were given 'T' for terminated status. There were 250,000 employee status attributes generated.

6.2.2.3 Graduate Degree

Only 1 in 8 IDs were given a graduate degree, either a value of "Masters" or "Ph.D". All graduate degrees also created and undergraduate degree attribute with a value of "Bachelors" for the ID with the graduate degree, but these will be counted below. Only even-numbered IDs would have graduate degrees. There were 125,000 graduate degree attributes generated.

6.2.2.4 Undergraduate Degree

Every other ID was given an undergraduate degree attribute, but they were given to the odd-numbered IDs to avoid collisions with the undergraduate degree attributes created for the even-numbered IDs. The values were "Associates" or "Bachelors" and changed back and forth after each one. There were 500,000 undergraduate degree attributes generated by this logic, plus an additional 125,000 attributes generated by the graduate degree function, resulting in a total of 625,000 undergraduate degree attributes generated.

6.2.2.5 Clubs

The clubs attribute was different than other attributes. 3 out of 4 IDs received at least one club attribute, but an ID could receive up to three clubs. The first ID would get one club, the next two, the third would get three, and then it would go back to one club. This means that for every three IDs that received a club attribute, or for every four IDs (since the last one was skipped), there would be six attributes generated. The available options were put in an array, and an iterator would simply move through the clubs to avoid possible duplicates (if the values were randomly selected). The possible club values were “Art”, “Sci-Fi”, “Tech”, “Bookbinding”, “Movie”, “Running”, and “Mining”. There were 1,500,000 club attributes generated.

6.2.2.6 Music

2 out of 4 IDs received a music attribute, but it was the middle two that received it, the first and last in a group being skipped. Available music attribute values were “Voice”, “Guitar”, “Piano”, “Composition”, and “Percussion”. There were 500,000 music attributes generated.

6.2.2.7 Random

There were three different random attribute groups generated, named “random1”, “random2”, and “random3”. Each set was randomly generated using the Node.js “crypto” library. For each set, seven different attribute values were randomly generated. The “random1” group had values that were four bytes in length, the “random2” group had values that were six bytes in length, and the “random3” group had values that were eight bytes in length.

The reason for generating random values in this manner was two-fold. First, these attributes were created in order to get to the requisite number of required attributes in the system. Not every attribute stored in the system was to be used in policies, so having large groups of randomly generated values did not take away from the characterization tests. Secondly, for tests where it was intended that values would not match, using these attributes was effective, since there was a low probability that two IDs would share matching values.

Each ID received three random attributes with seven values per attribute. This resulted in a total of 21,000,000 random attributes generated.

6.2.2.8 Virtues

Every ID was given the same five virtue attributes, namely “light”, “liberty”, “love”, “hard work”, and “charity”. By ensuring that these were the same for every ID, it was known that there would always be a match when comparing the attribute. There were 5,000,000 virtues attributes generated.

6.2.2.9 Total Attributes

The attributes used in this test were “gender”, “employee status”, “graduate degree”, “undergraduate degree”, “clubs”, “music”, “random1”, “random2”, “random3”, and “virtues”. The number of each and total is listed in the table below.

Table 6-1: The names and number of attributes generated for the characterization tests.

Attribute Name	Total Number of Occurrences
Gender	1,000,000
Employee_Status	250,000
Graduate_Degree	125,000
Undergraduate_Degree	625,000
Clubs	1,500,000
Music	500,000
Random1	7,000,000
Random2	7,000,000
Random3	7,000,000
Virtues	5,000,000
Total	30,000,000

6.2.3 Test Policies, Sets, and Checks

6.2.3.1 Policies

Ten Policies were used for the tests, each chosen for a specific use.

Table 6-2: Test policies used in engine characterization.

Policy Name	Policy Description
Self-service	Subject ID is the same as the Target ID
Employee	Subject has the "employee_status" attribute with a value of "A"
Clubmates	Subject and Target both have the attribute "clubs" with a matching value
Musical	Subject has the "music" attribute with any value
Bachelor	Subject has the "undergraduate_degree" attribute with a value of "Bachelors"
Graduate	Subject has the "graduate_degree" attribute with any value
Random1	Subject and the Target have the "random1" attribute with a matching value
Random2	Subject and the Target have the "random2" attribute with a matching value
Random3	Subject and the Target have the "random3" attribute with a matching value
Virtues	Subject and the Target have the "virtues" attribute with a matching value

The Self-service policy was designed specifically to test the efficiency of the engine in a situation that requires no attributes. The Employee and Bachelor policies allow testing for an attribute with a specific value. The Clubmates policy was important for testing for a matching attribute between two identities. The Musical and Graduate policies allowed the evaluation of the engine function that checks to see if an attribute exists for an identity with any value. The

Random policies made it possible to test the matching attributes with values of different lengths with the expectation that no matches would occur. Similarly, the Virtues policy enabled a match check with certainty of matching.

6.2.3.2 Sets

The following five Sets were created for use in characterization testing.

Table 6-3: Test Sets used in engine characterization of the Abacus.

Set Name	Policies within the Set
GetClubInfoForId	Self-service, Employee, Clubmates
UsePracticeRoom	Employee, Musical
EnrollInGradClass	Bachelor, Graduate
RandomMatch	Random1, Random2, Random3
VirtueMatch	Virtues

6.2.3.3 Checks

Four Checks were used to execute the Sets needed for each battery test.

Table 6-4: Test Checks used in engine characterization of the Abacus.

Set Name	Policies within the Set
CanGetData	GetClubInfoForId, UsePracticeRoom, EnrollInGradClass, RandomMatch, VirtueMatch
CanGetClubInfoById	GetClubInfoForId
CanUsePracticeRoom	UsePracticeRoom
CanEnrollInGradClass	EnrollInGradClass

6.2.4 Test Battery Configuration and Execution

The four test Checks provided for five different configurations to stress and test the system. Each of the five tests involved executing 500 calls to the Abacus with a specific Check and a specific way of choosing the subject IDs. Each test was run three separate times to provide for variations in IDs used, and to allow a wider view of system run-times.

The script to invoke the Abacus was written in Node.js. It would generate the list of IDs to use in the requests before making the calls. This ensured that ONLY the request/response time of the calls were measured. The code recorded the time just before the first call was made, which allowed the calculation of total time to make the 500 calls to the Abacus and to calculate the run time when the last response was returned.

The log data was queried after each run to record the length of time in milliseconds that each engine request took; this calculation began when the Request was received and ended when the Response was returned. The data was compiled and stored in Appendix A.

6.2.5 Test Battery Results

For the battery of tests, the length of time that it took just to generate the 500 calls to the Abacus ranged from 164 ms to 236 ms, with an average of 209 ms per battery. The data for the test runs is listed in Appendix A, showing the length of time that each run took, the length of processing time for each individual request, and the average length of time for each request. Because of the limitations of the software, the smallest increment of time that could be measured was 1 ms. Thus, a result of 1 ms might actually range from 1 ms to 1.999 ms. As such, following timing descriptions will be referenced as between two millisecond lengths, e.g. "between 1 and 2 ms" for the example just given.

6.2.5.1 CanGetClubInfoById (Randomized)

The IDs for the Subject and Target were randomly generated for this test. As the IDs would be different, the engine would need to retrieve the “clubs” attribute for both the Subject and Target to compare them, and the “employee_status” of the Subject was also needed. This simple test pulled three attributes from the cache and evaluated them against the policies.

For this test, 57.67% of all requests took between 1 and 2 ms, while 18.87% of the requests took between 2 and 3 ms. This means that more than 75% of all requests completed in less that 3 ms. The mean run time for all 1500 requests in this battery was 1.734 ms.

6.2.5.2 CanGetClubInfoById (Self-Service)

This battery utilized the same Check as the previous battery, but ID allocation was different. The ID numbers were randomly generated, but the Subject and Target were set to the same ID, meaning that all of the requests would permit via the self-service Policy. Because the self-service Policy requires no attributes, this is the fastest test that could be executed against the engine, providing data on fast policies.

In this configuration, 83.13% of the 1500 requests took less than 1 ms to complete. 16.33% of the requests took between 1 and 2 ms to complete, and only 0.53% of the requests took longer than 2 ms. The average run time for this battery was 0.175 ms, which is admittedly inaccurate: due to the limitation of our chronotic discrimination, it is known that 0 ms is not correct. It is expected that this average is closer to 700-800 µs, if the true length of processing time were known.

Despite the limitations of the time measurement, it is clear that the evaluation of policies not requiring attributes takes place in less than 1 ms, which is an achievement in and of itself.

6.2.5.3 CanGetData

CanGetData was the Check that went over everything. It would pull all of the attributes for graduate_degree, undergraduate_degree, clubs, employee_status, music, random1, random2, random3, and virtues. This meant that over thirty attributes could be retrieved for a single identity, and those would be compared against over thirty attributes for another identity, resulting in over sixty attributes retrieved and compared. This Check took the longest to execute because of its intense nature.

Despite its complexity, 53.73% of the requests only took between 1 and 2 ms to complete. Another 31.6% took between 2 and 3 ms to finish, and reviewing the data, more than 86% of the time the request was guaranteed to finish is less than 3 ms. The average run time for all of the Sets was 1.701 ms, which is still very fast for calculating five different Sets with numerous attributes, especially with the matching functions that were part of some of the policies.

6.2.5.4 CanUsePracticeRoom

The Policies involved in this Check required the “employee_status” and “music” attributes for the Subject alone, putting it in a quicker execution bracket. With 10.67% of the requests taking less than 1 ms, 68.13% of the requests taking between 1 and 2 ms, and 16.53% of the requests taking between 2 and 3 ms, over 95% of all requests took less than 3 ms. The mean time for these requests to complete was 1.277 ms.

6.2.5.5 CanEnrollInGradClass

The last set of tests was similar to the CanUsePracticeRoom tests in that it required only two attributes for evaluation. It is not surprising then that the numbers for the two tests were very similar. This test responded in less than 1 ms for 11.13% of the requests, between 1 and 2 ms for 68.2% of the requests, and 15% of the requests between 2 and 3 ms. Thus, just under

95% of all requests were serviced in less than 3 ms. The mean time to service all requests in this test was 1.305 ms.

6.2.5.6 Hardware Metrics

During the battery of tests, the maximum CPU usage for the authorization engine was 0.311%. This figure is incredibly small. A test of 2,000 requests was attempted, just to see how well the engine could handle it, and during that time the maximum CPU usage climbed only to 0.437%. Provided that access to an enterprise testing platform, capable of executing tens of thousands of requests in quick succession, then the system may have been stressed, but these numbers show how efficient the Abacus truly is in its creation. It also shows the value in having the attributes cached instead of requiring real-time querying of attributes from domain systems.

In addition to the engine CPU, the cache CPU was also reviewed for its performance. Under the large loads of the CanGetData tests, the maximum CPU usage was only 3.5%. Remember that this is for a single instance of the cache; had there been five instances of the cache, one per engine, then this number may have been as low as 0.7%!

6.2.6 Comparison to Commercial System

As stated at the beginning of Section 6.2 and Section 6.2.1, the hardware settings for the performance tests were chosen to match the hardware used in the commercial benchmark report.

On page 11 of the commercial benchmark report, the company recorded their latency for approximately 400 transactions per second: 91 ms for the PDP, 19 ms to retrieve assets, and 3 ms latency in identity retrieval, for a total of 113 ms of latency, not counting network latency. The mean execution time for the Abacus (using 0.8 ms as the mean value for the self-service policies) was 1.363 ms. This means that the Abacus is almost two orders of magnitude *faster* than the commercial system!

Evaluating the CPU usage between the Abacus and the commercial system is also revealing. For the commercial application, approximately 400 transactions per second would utilize 82% of the available CPU across the five processors in the test. Contrast this with the CPU usage of 0.311% reported by the Abacus during its testing. The Abacus requires 0.38% of the CPU power that the commercial system does.

For additional information, when the production instance of the Abacus received 988 requests over the span of five minutes, equating to approximately 200 requests per minute, the CPU usage reached only 1% of capacity. Again, this was on a t2.micro instance with 1 vCPU and only 1 gigabyte of RAM.

After reviewing the performance tests for the Abacus, it is easy to see why the implementation in the BYU production systems was able to perform so well. Not only does the Abacus require less hardware power than the commercial system, because it can run in a reduced fashion, it enables significant financial savings to any institution that utilizes it. Running a t2.micro instance in AWS costs less than 1/13th what it costs to use a t2.2xlarge instance, and utilizing a t3.micro instance is only 1/16th the cost of a t3.2xlarge instance! Running a t2.micro instance is a whopping 1/32nd of the price of the t3.2xlarge, for a comparison between the BYU production instance and the instances used in this performance testing.

In every way evaluated, the Abacus is faster, can handle more throughput than the commercial system against which it was compared. It requires less hardware to achieve an outcome that is far superior to other offerings, and by extension is less expensive to run. The Abacus model is an excellent contender against other systems for efficiency, power consumption, and cost of operation.

7 CONCLUSION

The aim of this research was to discover and prove a way to define and provide authorization without breaking domain boundaries. The goal was to find a performant way to evaluate attribute-based policies without requiring tight coupling to domain data stores or endpoints. This was achieved by pushing attributes into the Abacus, thus decoupling the authorization service from the source domains while still allowing for attributes to be up-to-date.

A second purpose of the research was to construct an authorization system that was of a distributed, cloud-native architecture to allow for easy deployment in multiple locations. The system was to be highly-available to prevent issues with individual servers. This was accomplished using AWS technologies to create the data stores, servers, and cache needed for the Abacus. The infrastructure-as-code framework allowed multiple, identical instances of the Abacus to be deployed, and it simplified modification of configuration parameters.

Table 7-1: AWS technologies needed to realize the architecture of the Abacus, providing for attributes to be pushed to the Abacus and cached efficiently.

Component Name	AWS Service
Authorization engine	EC2 server
Attribute cache	Elasticache
Data store	RDS
Attribute updater	Lambda

To prove its efficiency and efficacy, the Abacus was implemented in a real production environment for the Department of Continuing Education at BYU. In this case, the Abacus ran for over seventeen months without a single error. A battery of tests was used to characterize the efficiency of the Abacus, proving it to be almost 100 times faster than a commercial authorization system. The Abacus can handle significantly more traffic than the commercial system without even coming close to stressing the CPU of the engine.

7.1 Future Work

7.1.1 Attribute Reconciliation

When noting the advances in the Abacus, it is important to recognize that there are challenges still to be overcome. While the pushing of attributes is an extremely effective method for keeping attributes current in the authorization domain, the issue of reconciliation was never fully addressed in this research. In order to verify that attributes were accurate between the Abacus and the source domains, database extract-transform-load (ETL) processes were developed that compared the two data stores and updated the Abacus as needed.

Using ETL mappings is against the end goal of the Abacus as it requires a system that connects to both the domain data store AND the Abacus data store. It has been suggested that an API could be created to allow domains to verify that the attributes in the Abacus are in sync with the domain. Another idea was to have the Abacus produce a data lake of all attributes for a given domain, and the domain could execute ETL processes against the data lake. These ideas have various advantages and disadvantages to them, and it is hoped that a simple, efficient solution can be found.

7.1.2 Non-Technical Difficulties

The greatest difficulties encountered in the research was changing the authorization paradigm in the minds of the business owners. Experience had taught these good people that all that was required to grant authorization to a person was to put them in a specific group. Attempting to distill groups into their core attributes was difficult for many individuals and took both time and training. In addition, some business owners did not want the responsibility of creating policies, instead desiring to have the developers responsible for those decisions. It was a challenge to help the business owners understand that the developers only implement what the business owners ask for, and that this was not a transfer of duties, but instead fully placed the responsibility where it should be. Developing a way to help business owners understand how to approach policy creation is a whole topic for future investigation.

Another bridge to cross was having the domain owners produce a specific definition of an attribute. Too often the business owners referred to attributes as database fields instead of looking at attributes as a composite of individual points of data. Additionally, various domains would often interpret the same database field to mean different things. It was work to have the domain owners determine what constituted an authorization-important attribute and how it should best be represented. It is recommended that research in this area would be beneficial in conjunction with that mentioned in the preceding paragraph.

7.2 Open Questions

The most obvious need for future work lies in the development of a user interface for the Abacus. A true investigation into the simplicity of the policy grammar developed for the Abacus was beyond the scope of this research and the thesis, but to be a tool easily used by business owners, a UI is a necessity. How will this look, and what must be done to simplify the UI to avoid problems such as natural language processing?

During the research I was approached by a product manager at BYU who asked if the Abacus could be used in conjunction with a commercial ERP system to calculate group changes. This is a most fascinating idea: whenever an attribute within the Abacus changes, that attribute could be checked against the policies that rely on it. Should there be a change in authorization, those changes could be pushed to the ERP system and its internal roles and groups updated accordingly. How would the RBAC roles be mapped to Policies and Sets in the Abacus, and what would the mechanism that detects the attribute changes look like? Developing a methodology to allow an ABAC system to update an RBAC system would make an excellent doctoral dissertation in the future.

REFERENCES

[1] Z. Yi-qun, L. Jian-hua, and Z. Quan-hai, “A General Attribute based RBAC Model for Web Service,” Jul. 2007. doi: 10.1109/SCC.2007.8.

[2] A. Elliott and S. Knight, “Role Explosion: Acknowledging the Problem,” presented at the Software Engineering Research and Practice, 2010. [Online]. Available: https://pdfs.semanticscholar.org/143e/25f527eedecdf0a4f1b11646144fdfe694d5.pdf

[3] S. Nakamura, D. Doulikun, A. Aikebaier, T. Enokido, and M. Takizaw, “Synchronization Protocols to Prevent Illegal Information Flow in Role-Based Access Control Systems,” Jul. 2014. doi: 10.1109/CISIS.2014.39.

[4] *Axiomatics*. [Online]. Available: https://www.axiomatics.com/

[5] *NextLabs*. Next Labs. [Online]. Available: https://www.nextlabs.com/

[6] *PlainID*. PlainID. [Online]. Available: https://www.plainid.com/

[7] E. Evans, *Domain-Driven Design*. Boston, MA: Addision-Wesley, 2004.

[8] P. A. Grassi, M. E. Garcia, and J. L. Fenton, “Digital Identity Guildelines.” National Institute of Standards and Technology, Jun. 2017.

[9] P. Steiner, *On the Internet, nobody knows you’re a dog*. 1993.

[10] V. Vernon, *Domain-Driven Design Distilled*. Addison-Wesley Professional, 2016.

[11] V. Vernon, *Implementing Domain-Driven Design*. Addison-Wesley Professional, 2013.

[12] R. H. Steinegger, P. Giessler, B. Hippchen, and S. Abeck, “Overview of a Domain-Driven Design Approach to Build Microservice-Based Applications,” presented at the The Third International Conference on Advances and Trends in Software Engineering, Apr. 2017.

[13] F. Rademacher, J. Sorgalla, and S. Sachweh, “Challenges of Domain-Driven Microservice Design: A Model-Driven Perspective,” *IEEE Software*, vol. 35, no. 3, pp. 36–43, May 2018, doi: 10.1109/MS.2018.2141028.

[14] S. Preibisch, *API Development: A Practical Guide for Business Implementation Success*. Apress, 2018.

[15] *Trusted Computer System Evaluation Criteria*. United States Government Department of Defense, 1985.

[16] D. Ferraiolo and R. Kuhn, “Role-Based Access Controls,” Oct. 1992, pp. 554–563. [Online]. Available: https://csrc.nist.gov/CSRC/media/Publications/conference-paper/1992/10/13/role-based-access-controls/documents/ferraiolo-kuhn-92.pdf

[17] V. Hu *et al.*, “Guide to Attribute Based Access Control (ABAC) Definition and Considerations.” NIST, Jan. 2014. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/specialpublications/NIST.SP.800-162.pdf

[18] A. H. Karp, H. Haury, and M. H. Davis, “From ABAC to ZBAC: The Evolution of Access Control Models,” *Journal of Information Warfare*, vol. 9, no. 2, Art. no. 2, 2010.

[19] G. Orwell, *Animal Farm*. London, England: Secker and Warburg, 1945.

[20] G. Batra, V. Atluri, J. Vaidya, and S. Sural, “Deploying ABAC Policies using RBAC Systems,” *Journal of Computer Security*, vol. 27, no. 4, Art. no. 4, Jul. 2019.
[21] E. Coyne and T. R. Weil, “ABAC and RBAC: Scalable, Flexible, and Auditable Access Management,” *IT Pro*, vol. 15, no. 3, pp. 14–16, May 2013.
[22] V. Hu, D. Ferraiolo, R. Chandramouli, and R. Kuhn, *Attribute-Based Access Control*. Artech House, 2017.
[23] K. Riad, Z. Yan, H. Hu, and G. Ahn, “AR-ABAC: A New Attribute Based Access Control Model Supporting Attribute-Rules for Cloud Computing,” Hangzhou, China, Oct. 2015.
[24] S. Bhatt, F. Patwa, and R. Sandhu, “ABAC with Group Attributes and Attribute Hierarchies Utilizing the Policy Machine,” in *Proceedings of the 2nd ACM Workshop on Attribute-Based Access Control*, Mar. 2017, pp. 17–28. doi: 10.1145/3041048.3041053.
[25] N. Pustchi and R. Sandhu, “MT-ABAC: A Multi-Tenant Attribute-Based Access Control Model with Tenant Trust,” in *International Conference on Network and System Security*, 2015, pp. 206–220.
[26] Y. Rahulamathavan and M. Rajarajan, “LSD-ABAC: Lightweight static and dynamic attributes based access control scheme for secure data access in mobile environment,” Edmonton, AB, Canada, Sep. 2014. doi: 10.1109/LCN.2014.6925791.
[27] E. Yuan and J. Tong, “Attributed based access control (ABAC) for Web services,” presented at the IEEE International Conference on Web Services, 2005.
[28] N. Dan, S. Hua-Ji, C. Yuan, and G. Jia-Hu, “Attribute Based Access Control (ABAC)-Based Cross-Domain Access Control in Service-Oriented Architecture (SOA),” Aug. 2012. doi: 10.1109/CSSS.2012.354.
[29] M. Hemdi and R. Deters, “Using REST based protocol to enable ABAC within IoT systems,” Vancouver, BC, Canada, Oct. 2016.
[30] S. Bhatt and R. Sandhu, “ABAC-CC: Attribute-Based Access Control and Communication Control for Internet of Things,” in *Proceedings of the 25th ACM Symposium on Access Control Models and Technologies*, Jun. 2020, pp. 203–212.
[31] Y. Zhu, D. Huang, C. Hu, and X. Wang, “From RBAC to ABAC: Constructing Flexible Data Access Control for Cloud Storage Services,” *IEEE Transactions on Services Computing*, vol. 8, no. 4, pp. 601–616, Jul. 2015.
[32] D. Brossard, G. Gebel, and M. Berg, “A Systematic Approach to Implementing ABAC,” in *Proceedings of the 2nd ACM Workshop on Attribute-Based Access Control*, Mar. 2017, pp. 53–59. doi: 10.1145/3041048.3041051.
[33] M. P. Singh, S. Sural, J. Vaidya, and V. Atluri, “Managing attribute-based access control policies in a unified framework using data warehousing and in-memory database,” *Computers & Security*, vol. 86, pp. 183–205, Sep. 2019, doi: 10.1016/j.cose.2019.06.001.
[34] D. Ferraiolo, S. Gavrila, and G. Katwala, “A System for Centralized ABAC Policy Administration and Local ABAC Policy Decision and Enforcement in Host Systems using Access Control Lists,” in *Proceedings of the Third ACM Workshop on Attribute-Based Access Control*, Mar. 2018, pp. 35–42. doi: 10.1145/3180457.3180460.
[35] P. Biswas, R. Sandhu, and R. Krishnan, “Label-Based Access Control: An ABAC Model with Enumerated Authorization Policy,” Mar. 2016.
[36] C. Cotrini, T. Weghorn, and D. Basin, “Mining ABAC Rules from Sparse Logs,” London, UIK, Apr. 2018. doi: 10.1109/EuroSP.2018.00011.
[37] D. Mocanu, F. Turkmen, and A. Liotta, “Towards ABAC Policy Mining from Logs with Deep Learning,” Ljubljana, Slovenia, 2015.

[38] S. Das, B. Mitra, V. Atluri, J. Vaidya, and S. Sural, “Policy Engineering in RBAC and ABAC,” in *From Database to Cyber Security*, vol. 11170, 2018, pp. 24–54.
[39] C. Bailey, D. W. Chadwick, and R. de Lemos, “Self-Adaptive Authorization Framework for Policy Based RBAC/ABAC Models,” Sydney, NSW, Australia, Dec. 2011. doi: 10.1109/DASC.2011.31.
[40] M. Alohaly, H. Takabi, and E. Blanco, “A Deep Learning Approach for Extracting Attributes of ABAC Policies,” Jun. 2018. doi: 10.1145/3205977.3205984.
[41] “eXtensible Access Control Markup Language (XACML) Version 3.0.” OASIS, Jan. 22, 2013. [Online]. Available: docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html
[42] B. Stepien, A. Felty, and S. Matwin, “Advantages of a non-technical XACML notation in role-based models,” in *2011 Ninth Annual International Conference on Privacy, Security, and Trust*, 2011, pp. 193–200. doi: 10.1109/PST.2011.5971983.
[43] B. Stepien, A. Felty, and S. Matwin, “A Non-technical User-Oriented Display Notation for XACML Conditions,” 2009. doi: 10.1007/978-3-642-01187-0_5.
[44] B. Stepien, S. Matwin, and A. Felty, “An Algorithm for Compression of XACML Access Control Policy Sets by Recursive Subsumption,” in *Proceedings of the 2012 Seventh International Conference on Availability, Reliability, and Security*, USA, Aug. 2012, pp. 161–167. doi: 10.1109/ARES.2012.38.
[45] M. Mejri, H. Yahyaoui, A. Mourad, and M. Chehab, “A rewriting system for the assessment of XACML policies relationship,” *Computers & Security*, vol. 97, Oct. 2020, doi: 10.1016/j.cose.2020.101957.
[46] C. Morisset, T. A. C. Willemse, and N. Zannone, “Efficient Extended ABAC Evaluation,” in *Proceedings of the 23rd ACM on Symposium on Access Control Models and Technologies*, Indianapolis, Indiana, USA, Jun. 2018, pp. 149–160. doi: 10.1145/3205977.3205980.
[47] *Open Policy Agent (OPA)*. [Online]. Available: https://www.openpolicyagent.org/
[48] Q. Wang, D. Chen, N. Zhang, Z. Qin, and Z. Qin, “LACS: A Lightweight Label-Based Access Control Scheme in IoT-Based 5G Caching Context,” *IEEE Access*, vol. 5, pp. 4018–4027, Mar. 2017, doi: 10.1109/ACCESS.2017.2678510.
[49] P. Tamilselvan, “Fine-Grained Access Control with Attribute Based Cache Coherency for IoT with application to Healthcare,” Thesis, Iowa State, 2017.
[50] S. Ioannidis, “Security Policy Consistency and Distributed Evaluation in Heterogeneous Environments,” Dissertation, University of Pennsylvania.
[51] “JSON Profile of XACML 3.0 Version 1.0.” OASIS, Oct. 12, 2017. [Online]. Available: docs.oasis-open.org/xacml/xacml-json-http/v1.0/xacml-json-http-v1.0.html
[52] J. A. J. Siebach and J. Giboney, “The Abacus: A New Architecture for Policy-based Authorization,” presented at the Hawaii International Conference on System Sciences 2021, Jan. 2021. doi: 10.24251/HICSS.2021.848.
[53] A. Karp, private communication, Oct. 2, 2019.

APPENDIX A. CHARACTERIZATION RUN DATA

Each set of tests consisted of three runs, executing 500 queries to The Abacus per Check. Data was obtained for 1) total time to complete all queries, 2) length of time per individual request, and 3) the average of lengths of individual run times.

A.1 CanGetClubInfoById (Randomized)

Table A-1: Total run time to process 500 calls for CanGetClubInfoById Check with randomized IDs.

	Run 1	Run 2	Run 3
Total run time (ms)	346	371	351

Table A-2: Execution lengths for individual CanGetClubInfoById queries with randomized IDs.

Run Time (ms)	Occurrences 1	Occurrences 2	Occurrences 3
0	26	31	31
1	266	295	304
2	89	104	90
3	43	30	31
4	41	14	13
5	19	12	8
6	7	8	7
7	5	2	7
8	4	3	1
9	0	0	3
10	0	1	2
11	0	0	1
12	0	0	1
17	0	0	1
Total	500	500	500

Table A-3: Averages of individual run times for CanGetClubInfoById with randomized IDs.

Run Time (ms)	Total	Average
0	88	5.87%
1	865	57.67%
2	283	18.87%
3	104	6.93%
4	68	4.53%
5	39	2.60%
6	22	1.47%
7	14	0.93%
8	8	0.53%
9	3	0.20%
10	3	0.20%
11	1	0.07%
12	1	0.07%
17	1	0.07%
Total	1500	100%

A.2 CanGetClubInfoById (Self-Service)

Table A-4: Total run time to process 500 calls for CanGetClubInfoById Check as self-service requests.

	Run 1	Run 2	Run 3
Total run time (ms)	478	404	527

Table A-5: Execution lengths for individual CanGetClubInfoById queries as self-service requests.

Run Time (ms)	Occurrences 1	Occurrences 2	Occurrences 3
0	421	410	416
1	77	86	82
2	1	3	2
3	1	1	0
Total	500	500	500

Table A-6: Averages of individual run times for CanGetClubInfoById as self-service requests.

Run Time (ms)	Total	Average
0	1247	83.13%
1	245	16.33%
2	6	0.40%
3	2	0.13%
Total	1500	100%

A.3 CanGetData

Table A-7: Total run time to process 500 calls for CanGetData Check with randomized IDs.

	Run 1	Run 2	Run 3
Total run time (ms)	759	921	948

Table A-8: Execution lengths for individual CanGetClubInfoById queries with randomized IDs.

Run Time (ms)	Occurrences 1	Occurrences 2	Occurrences 3
0	2	11	11
1	248	281	277
2	185	156	133
3	43	32	47
4	15	8	9
5	3	5	4
6	1	6	7
7	1	1	5
8	0	0	2
9	1	0	2
10	1	0	0
11	0	0	1
12	0	0	1
35	0	0	1
Total	500	500	500

Table A-9: Averages of individual run times for CanGetClubInfoById with randomized IDs.

Run Time (ms)	Total	Average
0	24	1.60%
1	806	53.73%
2	474	31.60%
3	122	8.13%
4	32	2.13%
5	12	0.80%
6	14	0.93%
7	7	0.47%
8	2	0.13%
9	3	0.20%
10	1	0.07%
11	1	0.07%
12	1	0.07%
35	1	0.07%
Total	1500	100%

A.4 CanUsePracticeRoom

Table A-10: Total run time to process 500 calls for CanUsePracticeRoom Check with randomized IDs.

	Run 1	Run 2	Run 3
Total run time (ms)	906	860	761

Table A-11: Execution lengths for individual CanUsePracticeRoom queries with randomized IDs.

Run Time (ms)	Occurrences 1	Occurrences 2	Occurrences 3
0	53	55	52
1	320	354	348
2	90	68	90
3	10	6	9
4	3	5	1
5	5	5	0
6	5	1	0
7	1	2	0
8	6	1	0
9	0	3	0
11	1	0	0
12	2	0	0
13	2	0	0
14	1	0	0
15	1	0	0
Total	500	500	500

Table A-12: Averages of individual run times for CanUsePracticeRoom with randomized IDs.

Run Time (ms)	Total	Average
0	160	10.67%
1	1022	68.13%
2	248	16.53%
3	25	1.67%
4	9	0.60%
5	10	0.67%
6	6	0.40%
7	3	0.20%
8	7	0.47%
9	3	0.20%
11	1	0.07%
12	2	0.13%
13	2	0.13%
14	1	0.07%
15	1	0.07%
Total	1500	100%

A.5 CanEnrollInGradClass

Table A-13: Total run time to process 500 calls for CanEnrollInGradClass Check with randomized IDs.

	Run 1	Run 2	Run 3
Total run time (ms)	877	845	960

Table A-14: Execution lengths for individual CanEnrollInGradClass queries with randomized IDs.

Run Time (ms)	Occurrences 1	Occurrences 2	Occurrences 3
0	61	52	54
1	347	355	321
2	67	69	89
3	13	9	7
4	4	5	5
5	4	2	3
6	1	2	3
7	2	2	5
8	0	1	2
9	0	1	2
10	0	1	0
11	0	1	3
12	1	0	4
13	1	0	1
17	0	0	1
Total	500	500	500

Table A-15: Averages of individual run times for CanEnrollInGradClass with randomized IDs.

Run Time (ms)	Total	Average
0	167	11.13%
1	1023	68.20%
2	225	15.00%
3	29	1.93%
4	14	0.93%
5	9	0.60%
6	6	0.40%
7	9	0.60%
8	3	0.20%
9	3	0.20%
11	1	0.07%
12	4	0.27%
13	5	0.33%
14	1	0.07%
15	1	0.07%
Total	1500	100%