5-2022

Live Access Control Policy Error Detection Through Hardware

Bryce Mendenhall
University of Arkansas, Fayetteville

Live Access Control Policy Error Detection Through Hardware

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Engineering

by

Bryce Mendenhall
University of Arkansas
Bachelor of Science in Computer Engineering, 2021

May 2022
University of Arkansas

This thesis is approved for recommendation to the Graduate Council.

Jia Di, Ph.D.
Thesis Director

Alexander Nelson, Ph.D.	Brajendra Panda
Committee Member	Committee Member

ABSTRACT

Access Control (AC) is a widely used security measure designed to protect resources and infrastructure in an information system. The integrity of the AC policy is crucial to the protection of the system. Errors within an AC policy may cause many vulnerabilities such as information leaks, information loss, and malicious activities. Thus, such errors must be detected and promptly fixed. However, current AC error detection models do not allow for real-time error detection, nor do they provide the source of errors. This thesis presents a live error detection model called *LogicDetect* which utilizes emulated Boolean digital logic circuits to provide continual feedback and error source identification. The outcome is a new error detection model allowing policy creators to identify the source of errors quickly and accurately at any stage during policy creation.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Jia Di. He has not only guided me through my undergraduate and graduate education but helped me to become a better student, worker, and person. I am grateful to have had him and his support throughout the later years of my education.

I would also like to thank the members of my committee Dr. Alexander Nelson and Dr. Brajendra Panda for their attitude towards education and for their involvement in my undergraduate and graduate careers.

CONTENTS

1. Introduction

Access Control (AC) is a widely used security measure designed to restrict access to information system resources or physical facilities. [8] Further, it defines basic permissions for users or processes within an information system based on the AC policy. An AC policy is a set of rules describing the security of the system in its entirety. Each rule within the policy designates the security of a single user or process and assigns authorized *Subjects* a permission (either *Grant* or *Deny*) to protected *Objects* and *Actions*. [7] Correctly defining an AC policy for an information system is crucial to the security of the system. Any error in the policy may not only lead to unauthorized *Subjects* accessing protected materials, but it may also lead to sensitive information leaks or complete lock outs of protected resources. Therefore, the policy creator must design a policy to protect sensitive aspects of an information system, and this policy also must be error free. When working with large information systems, there may be thousands of rules with thin margins of error. For example, the policy creator may create an illegal *Inheritance* constraint between two *Subject*s. This can lead to unauthorized users gaining access to sensitive material which they are not allowed to access. Such errors may have detrimental effects on the integrity of an information system, so faults in AC policies must be detected and swiftly correctly.

Traditional AC policy fault detection is time-consuming for larger policies, and current approaches usually require a completed policy before testing may begin. [5] Moreover, error checking in a large policy after it is created can be difficult because the addition of new rules may exacerbate the severity of existing errors. Many approaches detect errors within the policy but usually cannot inform the policy creator of where errors originate. [5]

This thesis work studies a new approach to detecting AC policy errors which is fundamentally different from traditional methods. A new method (referred to as *LogicDetect*) is developed based on real-time hardware detection to allow a policy creator to check for errors during any phase of policy creation. *LogicDetect* not only allows for real time error-detection but also shows the policy creator problematic rules as well. *LogicDetect* leverages Boolean digital logic in Field Programmable Gate Array (FPGA) emulation to detect faults. This method allows policy rules to be intuitively expressed as Boolean logic expressions, which are translated into hardware and are added into their corresponding logic circuit (LC). Four LCs are constructed to simulate the AC policy: the *Grant* LC, *Deny* LC, *Support* LC, and the *Cyclic Inheritance* LC. The two primary LCs, the *Grant* and *Deny* LC, express authorization or denial of a rule. When specifying rules, gates within the LC represent a logic operation connecting AC variables enforced by the rules. Rules are added to their corresponding LCs based on the permission rendered by the rule, either *Grant* or *Deny*. The permission of each rule is defined upon rule creation. To effectively emulate the policy in hardware, each element of a rule is represented by a single bit. A *Grant* permission is represented when all elements of a rule are changed from ‘0’ to ‘1’, so the *Grant* LC outputs Boolean logic ‘1’. The same is true for *Deny* permissions in the *Deny* LC.

The three specific faults studied in this thesis work are the *Conflict Fault,* the *Privilege Escalation Fault*, and the *Cyclic Inheritance Fault*. The outputs of the *Grant* LC, *Deny* LC, and the *Cyclic Inheritance* LC are considered when detecting the faults. When both the *Grant* LC and *Deny* LC output Boolean logic ‘1’, a *Conflict Fault* is present in the policy. When the *Cyclic Inheritance* LC output is Boolean logic ‘1’, an illegal chain of *Inheritance* constraints exists on the *Subjects* or *Roles* of a rule and an error is detected. The *Privilege Escalation Fault* is detected by the *Grant* LC and represents an illegal *Inheritance Constraint* assigning an unauthorized *Subject* access to a resource.

LogicDetect shows promising results, and throughout this work, it has consistently succeeded in error detection. *LogicDetect* breaks away from traditional methods is two key ways: due to the modeling of AC rules in hardware and simulating logic circuits, *LogicDetect* allows for a new real-time detection strategy for AC policies; additionally, traditional methods traverse through an entire policy and are computationally expensive. *LogicDetect* takes advantage of its parallel nature to emulate rules and allows the policy creator to check individual rules without having to check the entire policy, thereby saving time and resources.

2. Background

2.1 Access Control Overview

Access control is a critical aspect of many systems requiring restrictions, such as in information systems. In the context of this thesis, access control is defined as the selective restriction of access to software or computer resources. Specifically, access control is addressed in terms of information systems and comprising elements handled by a policy creator and may encompass personal, commercial, or government information systems. An access control policy is composed to define resource access by various principals (e.g., processes and users). [7] This is a vital component of keeping information systems secured, so having an appropriate access control policy for a system is crucial. However, for large systems where there are many access control policies due to the size and complexity of the system, access control is more susceptible to human error. [9] For example, with a large AC policy, there may be a rule which gives access to a process or user for a certain recourse. Even so, the AC policy creator may form another rule with that same set of principles, resulting in denial of that user or process from the same recourse. This would cause an error which is called a *Conflicting Fault*, which occurs when a user or process is both given and denied permission to access the same resource. This is just one of many errors which may arise during AC policy creation, all of which will be discussed in later sections.

2.2 Policies

At the heart of access control is the policy. A policy is the list of a variable number of rules created to protect a system. A rule is a set of enforce conditions to *Grant* or *Deny* access to a system *Object* or *Action*. There are two primary rule types in this thesis work, Attribute-Based rules and Role-Based rules, explained in Sections 2.1.1 and 2.1.2.

It is important to understand that the policies identified and designed in hardware for this thesis are not intended to control a real system but are example policies. The purpose of this research is to emulate the proposed policy to detect errors and does not impose adherence to policy rules. When simulating the policies in hardware, only one rule will be active at a time to detect faults unless otherwise required. This is a stark difference from a live system where many rules may be active simultaneously.

2.2.1 Attribute-Based Rules

The Attribute-Based (AB) rule provides the guidelines for the most basic access control and is commonly used to create Mandatory Access Control (MAC) policies. This rule requires a minimum of three inputs: *Subject*, *Object*, and *Action*. It also contains a fourth optional input named *Role*. The *Subject* describes a user within the system who is to be restricted by the rule. The *Object* describes a resource, such as sensitive documents, needing protection. The *Action* describes what the *Subject* is allowed to or not allowed to do to the protected resource. For example, consider a *Subject* named *Jason*, an *Object* named *Locked Door*, and an *Action* named *Open*. If this is a *Grant* rule, then *Jason* has the permission to *Open* the *Locked Door* (Equation 1). Conversely, if the rule is meant to deny *Jason*, then *Jason* does not have permission to *Open*. The policy creator has authority on whether a rule is a grant or deny rule. Equation 1 shows an example Boolean logic expression of an Attribute-Based rule with no *Roles*.

$$\boldsymbol{Grant = Jason\ AND\ Locked\ Door\ AND\ Open} \qquad \textbf{(1)}$$

2.2.2 Role-Based Rules

Role-Based (RB) rules act the same way as AB rules. Instead of having a *Subject* like AB rules, RB rules feature a *Role*. This can be leveraged in several different ways, from having rules for certain groups of users to modeling multi-level security policies. For example, an RB rule may be used when a rule is needed for all employees in a company. AB rules would grant each employee access to unique *Objects* or *Actions*, like permission to use an office computer. Only the *Subject* of the rule may access the corresponding *Object*. However, if the *Subjects* are given a *Role*, like *Employee,* an RB rule considers the *Roles* of all *Subject*s. The rule may allow anyone who is an employee to have access to an employee forum. Any *Subject* would be checked for the '*Employee' Role* when they attempt to login under this RB rule. This is an important distinction from AB rules, as it enables the policy maker to define one rule for many different *Subjects*.

Role-Based rules are utilized in a widely used version of access control called multi-level security (MLS), prevalently used in government systems since it allows the policy creator to create "levels" of security or clearance. Equation 2 demonstrates an MLS ruleset conversion to Boolean logic. In this example, a *Role* check is performed for the *SecretClearance Role*. Ensuring the *Subject* may be granted access if and only if the correct *Role* constraint is satisfied. Equation 2 features a single RB rule in Boolean logic.

$$\boldsymbol{Grant} = (\boldsymbol{SecretClearance\ AND\ Edit\ AND\ SecretDocument}) \tag{2}$$

2.2.3 Rule Constraints and Relationships

Rule constraints are modifications applied to both AB and RB rules. Each constraint or relationship relies on existing rules to add more complexity and meaningful access control parameters. There are seven types of constraints studied in this work throughout Section 2.1.3.1 to Section 2.1.3.7.

2.2.3.1 Role Constraint

Role constraints are treated differently than RB rules. This is a constraint placed on an Attribute-Based rule to allow the user access to RB rules. Furthermore, this constraint may be a part of an *Inheritance* relationship where *Privilege Escalation* must be weighed. *Privilege Escalation* is explained in Section 3.3.2.

2.2.3.2 Workflow Rule Constraint

The Workflow constraint ensures one or many rules must be fulfilled before a successive rule is granted. [10] For example, if one rule allows a *Subject Janice* to take a *test* on blackboard, another rule grants permission to the *TA* to *grade* the *test*. In a Workflow relationship, it can be ensured that *Janice* has *taken* the *test* before the *TA* is allowed to *grade* it. This can be represented in Boolean logic by using the first rule's *Grant* signal, *S0,* as part of the second rule's input (Equations 3 and 4). Grant signal *S1* will not be granted unless *Grant* signal *S0* is true. Further steps need to be taken for state-keeping when converting this relationship to VHDL and will be addressed in Section 3.3.5.2.

$$\boldsymbol{S0 = Janic\ AND\ Take\ AND\ Test} \tag{3}$$

$$\boldsymbol{S1 = TA\ AND\ Grade\ AND\ Test\ AND\ S0} \tag{4}$$

2.2.3.3 General Mutual Exclusion Constraint

General Mutual Exclusion (GME) ascertain whether two or more *Subjects* should not be able to access the same resource at the same time. [7] For example, if two users, *user1* and *user2*, both have access to *edit* a *document* but should not have simultaneous *edit* capabilities, a GME constraint between the two rules is required to *Deny user1* access to the *document* while *user2* is *editing* and vise-versa. The opposite rule should not be granted while the other rule is active. Equations 5 and 6 display a GME rule in Boolean logic.

$$\boldsymbol{S0 = User1\ AND\ Edit\ AND\ Document\ AND\ NOT\ S1} \tag{5}$$

$$\boldsymbol{S1 = User2\ AND\ Edit\ AND\ Document\ AND\ NOT\ S0} \tag{6}$$

2.2.3.4 N-Person Control Constraint

The N-person control constraint forces multiple *Subjects* to be *Granted* access to the same *Object* at the same time. [7] With a 2-person control constraint, two *Subjects* must be active for the rule to be *Granted*. For example, if there is a 2-person rule requiring two *keys* to be scanned concurrently to open a locked *door*, both users who possess a certified *key* must be present for the *door* to *unlock*. Equation 7 shows an example of a 2-person control rule in Boolean logic.

$$\boldsymbol{Grant = ((Key1\ AND\ Key2)\ AND\ Unlock\ AND\ Door)} \tag{7}$$

2.2.3.5 Inheritance Constraint

An *Inheritance* constraint describes a relationship between two *Subjects* or *Roles* by allowing one *Subject* or *Role* to share their corresponding rules with another. A single one-way *Inheritance* constraint demands two *Subjects* or *Roles*, the beneficiary and the tribute. The tribute shares all their corresponding rules with the beneficiary. However, with an AB rule, the beneficiary does not receive any *Role* constraints for that rule. Consider the example with two *Subjects, Bob,* who has two *Grant* rules and, *Alice,* who has no rules. *Alice* inherits from *Bob*, providing *Alice* access to *Bob's* two *Grant* rules. Equations 8 and 9 demonstrate this inheritance example in Boolean logic.

$$\boldsymbol{Grant = (Bob\ OR\ Alice)\ AND\ Edit\ AND\ Document} \tag{8}$$

$$\boldsymbol{Grant = (Bob\ OR\ Alice)\ AND\ View\ AND\ Document} \tag{9}$$

When *Inheritance* constraints are combined with *Role* constraints, the policy runs the risk of *Privilege Escalation*. This is an error type resulting in the beneficiary having access to rules when *Role* requirements are not met. This fault is considered in-depth in Section 3.3.2.

2.2.3.6 Separation of Duty Constraint

Separation of Duty (SoD) divides critical functions between *Subjects* so a single *Subject* cannot misuse the system independently. [11] There are several different types of SoD rules studied in this research: Simple Static SoD, Dynamic SoD, Simple Dynamic SoD, Object-Oriented SoD, and Operational SoD.

Simple Static SoD states that anyone with a certain role should not be allowed to execute a certain *Action*. [11] This can be implemented using an RB deny rule. Equation 10 depicts an example RB rule in Boolean logic to express Simple Static SoD.

$$\boldsymbol{Deny = Employee\ AND\ Approve\ AND\ TimeSheet} \quad \textbf{(10)}$$

Dynamic SoD ensures only one *Subject* may do one of multiple exclusive *Actions* related to a certain *Object*. [11] This type of SoD essentially allows a *Subject* to claim an *Action*. When an *Action* is claimed, access will not be *Granted* to any other *Subject* related to the same *Action*. Consider the example shown in Equation 8. There are two *Subjects, User1* and *User2.* They both have access to *Approve* and *Write Checks.* However, due to the DSoD constraint, the *Subjects* will not be able to claim the same *Action*. Furthermore, when a *Subject* claims one *Action*, such as *Approve,* they may not access the other *Action*, *Write.* Equations 10 – 13 demonstrate dynamic SoD in Boolean logic.

$$\boldsymbol{Grant1 = User1\ AND\ Approve\ AND\ Checks\ AND\ NOT\ Grant2\ AND\ not\ Grant3} \quad \textbf{(10)}$$

$$\boldsymbol{Grant2 = User1\ AND\ Write\ AND\ Checks\ AND\ NOT\ Grant1\ AND\ not\ Grant4} \quad \textbf{(11)}$$

$$\boldsymbol{Grant3 = User2\ AND\ Approve\ AND\ Checks\ AND\ NOT\ Grant1\ AND\ not\ Grant4} \quad \textbf{(12)}$$

$$\boldsymbol{Grant4 = User2\ AND\ Write\ AND\ Checks\ AND\ NOT\ Grant2\ AND\ not\ Grant3} \quad \textbf{(13)}$$

Simple-Dynamic SoD restricts the *Role*s which a user may have. In this constraint a user may not take on more than one exclusive *Roles*. [11] This model is implemented by the policy creator during policy creation.

Object-oriented SoD is the same as Dynamic SoD with the only difference being the user claims an *Object* instead of an *Action*. [11] The Boolean expression for this rule has the same structure as the expression for Dynamic SoD.

Operational SoD ensures one person cannot be a part of every step of a business task. [11] One person must draft the checks and a different individual must sign them. However, multiple people may be allowed to participate in such step as long as no one person is allowed access to every step. Consider the example outlined in Equations 14 - 17. There are two *Subjects*, *User1* and *User2*. Both *Subjects* may *Approve* and *Write* checks. However, when the *Subject* is granted access to one *Action*, they will not be granted for the other *Action.* Both *Subjects* may access the same *Action* given that neither access both *Actions.*

$$\boldsymbol{Grant1 = User1\ AND\ Approve\ AND\ Checks\ AND\ NOT\ Grant2} \qquad \textbf{(14)}$$

$$\boldsymbol{Grant2 = User1\ AND\ Write\ AND\ Checks\ AND\ NOT\ Grant1} \qquad \textbf{(15)}$$

$$\boldsymbol{Grant3 = User2\ AND\ Approve\ AND\ Checks\ AND\ NOT\ Grant1} \qquad \textbf{(16)}$$

$$\boldsymbol{Grant4 = User2\ AND\ Write\ AND\ Checks\ AND\ NOT\ Grant2} \qquad \textbf{(17)}$$

2.2.3.7 Conflict of Interest Constraint

Conflict of Interest is commonly known as the Chinese Wall model. This constraint states that if two *Objects* are conflicting and a *Subject* has access to one of those *Objects*, then the *Subject* cannot access the other *Object*. [7] For example, if a company manages security for many other firms and an employee has access to sensitive information for one of the client firms, the employees should not have access to sensitive information from any competing firms. In Equations

18 - 21, *Jason* has access to *Company A Sensitive Info* so he should be denied access to any other companies sensitive information.

$$\boldsymbol{Grant1 = Jason\ AND\ CompanyASensitiveInfo\ AND\ Read/Write} \tag{18}$$

$$\boldsymbol{Grant2 = Alice\ AND\ CompanyBSensitiveInfo\ AND\ Read/Write} \tag{19}$$

$$\boldsymbol{Deny1 = Jason\ AND\ CompanyBSensitiveInfo\ AND\ Read/Write} \tag{20}$$

$$\boldsymbol{Deny2 = Alice\ AND\ CompanyASensitiveInfo\ AND\ Read/Write} \tag{21}$$

2.3 Current Error Checking Techniques

There are several current policy verification schemes, but some of the most frequently used are Model Checking, Multi-Terminal Binary Decision Diagrams (MTBDD), Mutation Testing, Automated Combinatorial Testing, and Pseudo-Exhaustive Testing.

Model Checking is a verification technology that provides an algorithmic means of determining whether an abstract model—representing, for example, a hardware or software design—satisfies a formal specification expressed as a temporal logic formula [1]. If the specification is not satisfied, the method identifies a counterexample execution to show the source of the problem [1].

Underlying representation of access-control policies, Multi-Terminal Binary Decision Diagrams (MTBDDs) serve as decision diagrams to map bit vectors over a set of variables to a finite set of results [2].

Mutation Testing, on the other hand, exceeds where Model Checking and MTBDD fail. These aforementioned methods may not be sufficient to guard against some unexpected behaviors embedded in the AC model [1]. To address this shortcoming, the mutation test of the white box test method generates additional inputs to consider policy-related entities not covered by black box test methods [1].

Automated Combinatorial Testing is a methodology to test all t-way combinations of input parameters in at least one test [1]. This methodology excels at error checking Multi-Level Security policy models.

Pseudo-Exhaustive Testing is defined as exhaustive testing of all combinations of *Role* values on which an access control decision is dependent [1]. An advantage of this approach is its potential use to produce a complete test set for all possible rule combinations [1].

2.4 Logic Circuit Emulation

Digital logic circuits implement Boolean logic expressions. As the name implies, Boolean logic variables have the value ‘0’ or ‘1’, or false or true, respectively. In the Boolean logic family, there are three essential operations: OR, AND, and NOT. There are other operations such as XOR (exclusive OR) and NAND (inverse of AND), but these result from combinations from the three core operations. Because of the simplicity of Boolean logic, emulating Boolean circuits is extremely fast, and simulations at Register Transfer Level (RTL) or as a gate-level netlist may take just a few seconds or less. Boolean emulations may be run on a variety of software tools including ModelSim from Mentor Graphics, VCS from Synopsys, Quartus from Altera, and ISE from Xilinx, but for this research, Vivado from Xilinx was used.

Emulations are run on an FPGA board, which can be configured to run multiple hardware configurations. The flexibility of FPGAs arises from the programmable circuitry contained, such as Static Random Access Memory (SRAM) based configurable logic blocks (CLBs) and programable interconnects. These blocks create a physical array of logic gates united by the programmable interconnects, allowing a single FPGA board to implement many different logic circuits.

3. Methodology and Implementation

3.1 Overview

To effectively translate AC policies to hardware in a way to uphold the integrity of the policy, rules and constraints are designed to function individually. This exempts the rules within the policy of any dependencies on one another, meaning the policy is considered functional at every state while being defined. This is crucial to live error detection, as it allows the policy to be tested at any point even while incomplete. All hardware models designed for AC policy conversion were done in collaboration with a team member on this research work, Yatish Dubasi. The later sections in this chapter explain how each rule and constraint is designed in this manner.

3.2 Logic Circuit Layout

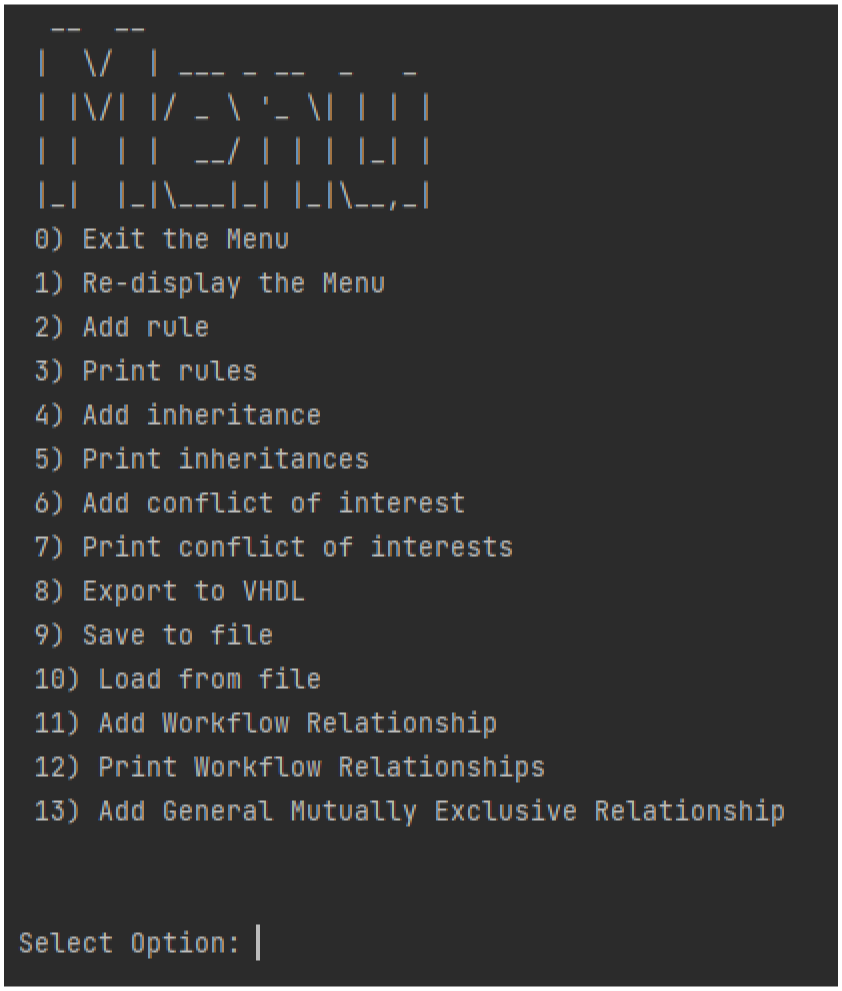
Before diving into how rules are designed and implemented, the layout of the policy must be understood. The policy consists of 4 distinct Logic Circuits (LCs) working in tandem to form the hardware model for error detection. These LCs are the *Grant* LC, *Deny* LC, *Support* LC, and *Cyclic Inheritance* (CI) LC. The *Grant* LC and *Deny* LC consist of the core elements of all *Grant* rules and *Deny* rules within the policy, respectively. The support LC consists of the logic used to support the *Grant* and *Deny* rules while also maintaining necessary dynamic states and other support logic (e.g., Workflow rules when state keeping is necessary or in *Privilege Escalation* when privilege checks are required). The final CI LC consists of hardware to check for CI errors. These LCs are not physically constrained on the FPGA and are largely used to organize rules in the script and VHDL files.

3.3 AC Conversion Script

Originally, converting AC policies to VHDL was done by hand. To streamline the process and make AC checking more accessible for policy creators, a Python script was developed to automate the conversion. The script currently allows the policy creator to automatically generate AB and RB rules with most constraints. Current constraints supported by the script are Inheritance, Workflow, Conflict of Interest, General Mutual Exclusion, N-person control constraints, and a few SoD types. After the script is functionally complete to support all constraint types, there are plans to add a user interface. This should further simplify the conversion process by allowing nearly any user to convert their policies for testing.

3.3.1 Menu Options

On script start-up, the policy creator is presented a series of prompts asking what kind of rules and constraints they would like to add. Prompts from the script startup menu are provided in Figure 1. This menu contains the options a policy creator will need to define most rule types.



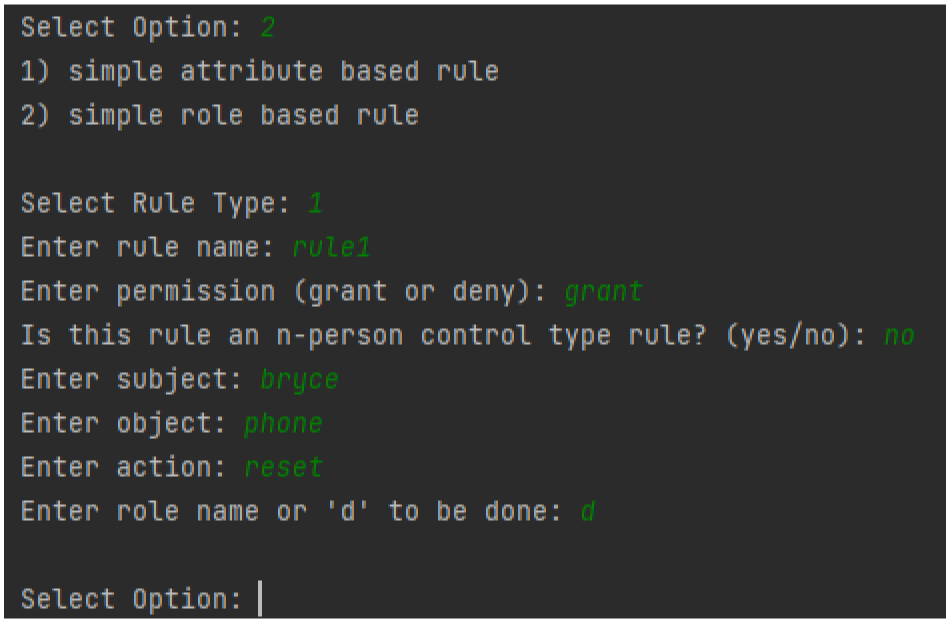
```
 __  __
|  \/  | ___ _ __  _   _
| |\/| |/ _ \ '_ \| | | |
| |  | |  __/ | | | |_| |
|_|  |_|\___|_| |_|\__,_|
0) Exit the Menu
1) Re-display the Menu
2) Add rule
3) Print rules
4) Add inheritance
5) Print inheritances
6) Add conflict of interest
7) Print conflict of interests
8) Export to VHDL
9) Save to file
10) Load from file
11) Add Workflow Relationship
12) Print Workflow Relationships
13) Add General Mutually Exclusive Relationship


Select Option:
```

Figure 1: AC Script Menu

3.3.1 Adding Attribute/Role-Based Rules

When the policy creator adds an AB rule using menu option 2, they are offered the prompts in Figure 2. Likewise, if the policy creator creates an RB rule, they are provided similar prompts: *rule name*, *Role*, *Object*, and *Action*.



```
Select Option: 2
1) simple attribute based rule
2) simple role based rule

Select Rule Type: 1
Enter rule name: rule1
Enter permission (grant or deny): grant
Is this rule an n-person control type rule? (yes/no): no
Enter subject: bryce
Enter object: phone
Enter action: reset
Enter role name or 'd' to be done: d

Select Option:
```

Figure 2: User prompts after selecting menu option 2

Rule name is a crucial field when adding rules into the policy, as the names are used to link constraints to that rule. To help keep track of added rules and their names, a *print* function is implemented in the script and is explained in Section 3.2.3.

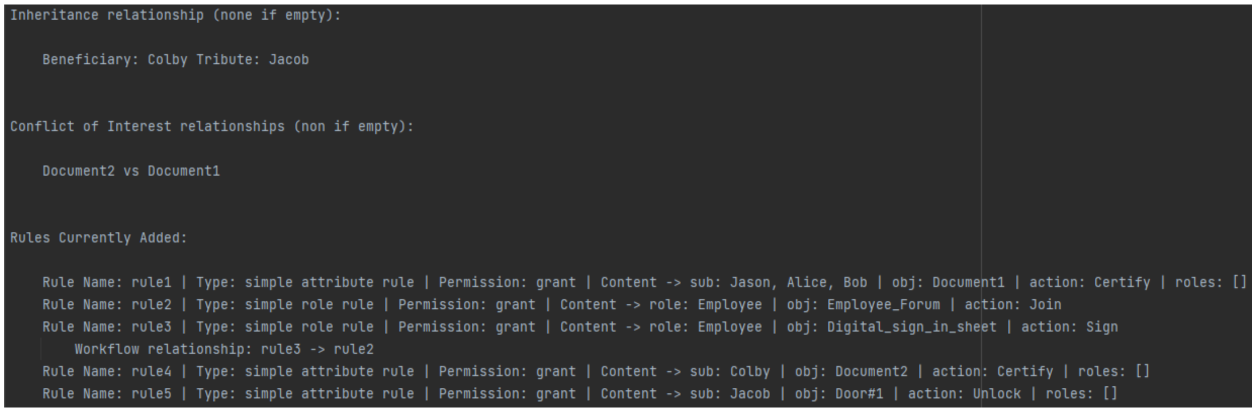
When a rule is created, it must have one of two permissions, *Grant* or *Deny*. The rules are categorized by this permission and added to the *Grant* and *Deny* LCs accordingly. The N-person constraint is one of two constraints added at rule definition. When the prompt for the N-person constraint is confirmed, the policy creator adds all the *Subjects* involved in the constraint as a comma separated list. This list of *Subjects* is stored as a single variable in the rule object and is partitioned into individual *Subjects* when converted to VHDL. The successive three AC script prompts ask for the core variables: *Subject*, *Object*, and *Action*. If the rule is constrained by an N-person constraint, the user will not be prompted to enter another *Subject*. Each variable is stored in its respective list and will be used as I/O after VHDL conversion. The *Role* constraint is also added at rule definition. Any number of *Roles* may be assigned to a *Subject* unless otherwise stated by Simple Static SoD. Although the *Role* constraint is treated as a regular variable within the rule, it is treated much differently when converting to VHDL. The rule conversion process to VHDL is elaborated in Section 3.3.5.

3.3.2 Adding Constraints

In the menu, the constraint options can be observed as “add inheritance’”, “add conflict of interest”, “add workflow”, and “add general mutually exclusive constraint”. These are options 4, 6, 11, and 13, respectively. Constraints can be added at any time if the same names are used for the rules which the constraint applies to, even when the policy is empty.

3.3.3 Print Options

Option 3 in Figure 1 allows the user to print the current rules added using the script. This selection has two sub-options: “print to file” and “print to console”. As explained in Section 3.3.1, rule names are essential in referencing rules and constraints, so it is crucial for the policy creator to remember the names of every rule. However, when dealing with large policies, this is unrealistic. Thus, the print option was created for the specific purpose of ensuring the policy creator does not mis-reference rules. Printing to the console is inefficient with large policies but is an appropriate method to quickly check rule nomenclature for smaller policies. Printing to a file is the preferred method when working with large policies. Additionally, the file can remain open as a reference while adding rules and may be refreshed to maintain a running list of rules. Figure 3 depicts the file generated when printing a policy.



```
Inheritance relationship (none if empty):

    Beneficiary: Colby Tribute: Jacob


Conflict of Interest relationships (non if empty):

    Document2 vs Document1


Rules Currently Added:

    Rule Name: rule1 | Type: simple attribute rule | Permission: grant | Content -> sub: Jason, Alice, Bob | obj: Document1 | action: Certify | roles: []
    Rule Name: rule2 | Type: simple role rule | Permission: grant | Content -> role: Employee | obj: Employee_Forum | action: Join
    Rule Name: rule3 | Type: simple role rule | Permission: grant | Content -> role: Employee | obj: Digital_sign_in_sheet | action: Sign
        Workflow relationship: rule3 -> rule2
    Rule Name: rule4 | Type: simple attribute rule | Permission: grant | Content -> sub: Colby | obj: Document2 | action: Certify | roles: []
    Rule Name: rule5 | Type: simple attribute rule | Permission: grant | Content -> sub: Jacob | obj: Door#1 | action: Unlock | roles: []
```

Figure 3: Printing policy to a file

3.3.4 Saving the Policy

Menu options *9* and *10* in Figure 1 permit the user to correspondingly save and load a policy. These options utilize the Python pickle library, which leverages binary protocols to serialize and de-serialize the Python objects, in this case the rule objects. When the file is saved, the object hierarchy is converted into a byte stream, and loading in the file is the inverse operation. However, this method of pickling is not secure, so users should only un-pickle trusted files.

3.3.5 Converting Rules to VHDL

Concerning Figure 1, menu option *8* is the most critical aspect of the script, as it converts all current rules to VHDL. The rules will be converted to VHDL in the same order they were added into the policy. Supporting Boolean logic and error checking logic are autonomously generated and placed within their corresponding LC groups. The VHDL file contains all LCs and the logic needed to test the three errors addressed in this research: *Conflicting Faults*, *Privilege Escalation Faults*, and *Cyclic Inheritance Faults*. In Section 3.4, the aforementioned errors are explained in detail.

3.3.5.1 Attribute-Based and Role-Based Rules

Attribute-Based and Role-Based rules are largely treated the same when converting to VHDL. The core inputs *Subject, Object,* and *Action* for AB rules or *Role, Object,* and *Action* for RB rules are connected to a single AND gate. For this work, the AND gate where the I/O for a single rule converges is called the *Grant gate* or *Deny gate* depending on their permission. The layout of the signals change depending on the constraints, but there will always be a *Grant* or *Deny gate* to decide the outcome. These gates are presented in Figures 4 and 5 and represent a simple schematic of how these rules translate to hardware. While the schematics in Figures 4 and 5 appear nearly identical, the difference between the two rules is the way they are emulated or constrained. RB rules may be used to add security functions, such as MLS, and they may be referenced by AB rules. However, AB rules will not affect or be referenced by RB rules. When no constraints exist to connect the two rule types, the single difference is the *Role* and *Subject*.

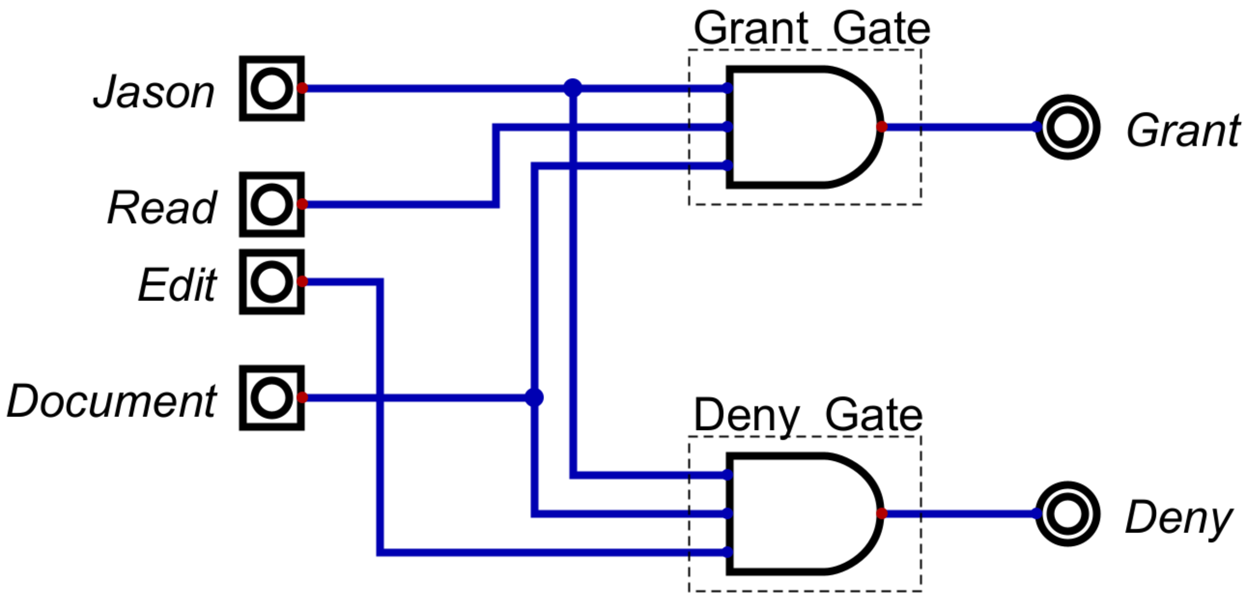


Figure 4: Attribute-Based rule schematic

The rule in Figure 4 is the hardware translation of a security measure allowing *Jason* access to *read* but not *edit* a *document* using AND gates as the *Grant* and *Deny gates*.

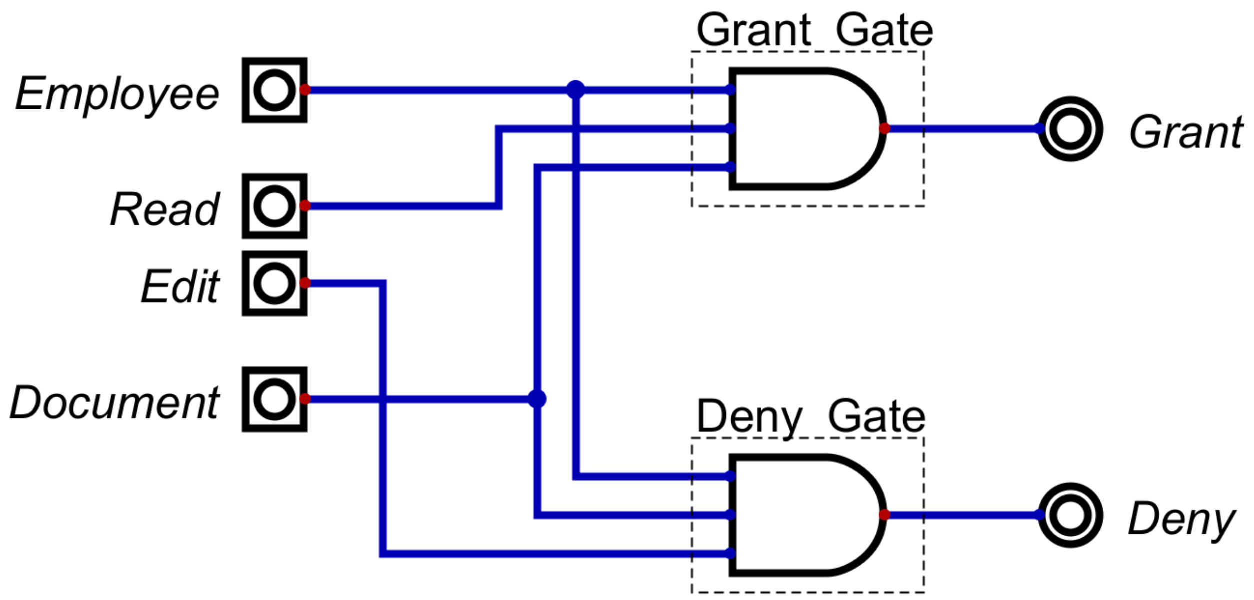


Figure 5: Role-Based rule schematic

The hardware implementation of an RB rule in Figure 5 describes a security measure allowing an *Employee* access to *read* but not *edit* a *document*. This is different from the AB rule in Figure 4 because the AB rule only *Grant* or *Denies* a single *Subject*, *Jason.* When this RB rule exists in a larger policy, any *Subject* with the *Employee Role* will be authorized to *read* but not *edit* the *document*. In a policy with both examples, *Jason* could be given the *Role Employee* through a *Role* constraint and his AB rules removed while maintaining the integrity of the policy.

3.3.5.2 Workflow Constraint

As stated in Section 2.2.3.2, when a Workflow constraint is present in the policy, there must be state keeping for this constraint to function properly. When two or more rules are constraint by Workflow, the hierarchy of rules is called the Workflow chain. The rule with higher priority in the Workflow chain must be fulfilled before a successive rule is fulfilled. However, during emulation, only one rule is flagged at a time. Therefore, the succeeding rule will not be fulfilled since it depends on the output of the preceding rule, which is no longer fulfilled. To fix this issue, an SR-flip-flop is introduced between each link along the Workflow chain. The flip-flops represent the state of the rules along the Workflow chain. Figure 6 shows an example of a Workflow constraint between two rules.

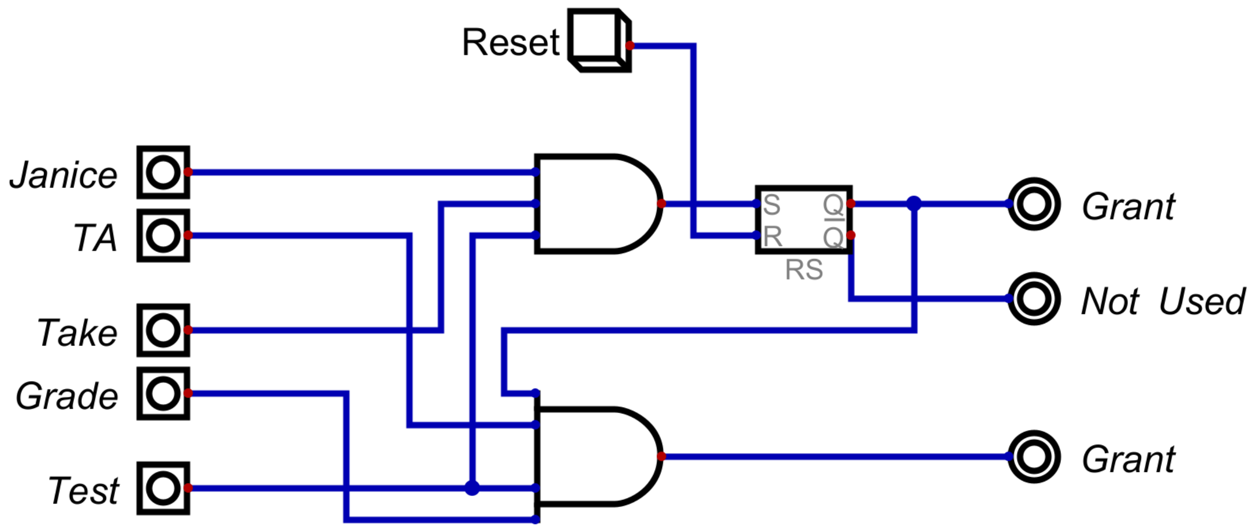


Figure 6: Example schematic of a workflow constraint between two rules

Before any rules are granted, the Workflow chain is in its first state, meaning only the first rule may be granted. When the first rule is *Granted*, its output is stored in an SR latch. A digital logic ‘1’ stored in the latch represents a granted rule, and it will hold this value until the latch is reset. The Workflow chain is now considered to be in its second state, so the second rule can be *Granted*. When the final rule is *Granted*, it is not mandatory for the flip-flops to be reset. Resetting the state of the Workflow chain is at the discretion of the policy creator during testing.

3.3.5.3 General Mutual Exclusion Constraint

A General Mutually Exclusive (GME) constraint is added when an *Object* needs to be protected from more than one *Subject* accessing it at the same time. The protected *Object* is defined when the constraint is added using the script. When the *Object* is identified and the policy converted to VHDL, all rules with the protected *Object* are included in the GME constraint. In hardware, this is expressed on the *Grant/Deny* gate of every rule involved. For all rules in the constraint, a new input is added to each *Grant/Deny* gate. These inputs represent the inversion of every other rule comprised in the constraint. Figure 7 exhibits an example GME schematic with two rules with a GME constraint on *Document.* Two *Subjects*, *User1* and *User2*, have access to *Edit* the *Document*. Since *Document* is a protected resource, only one *Subject* should be allowed to *Edit* at any given time.

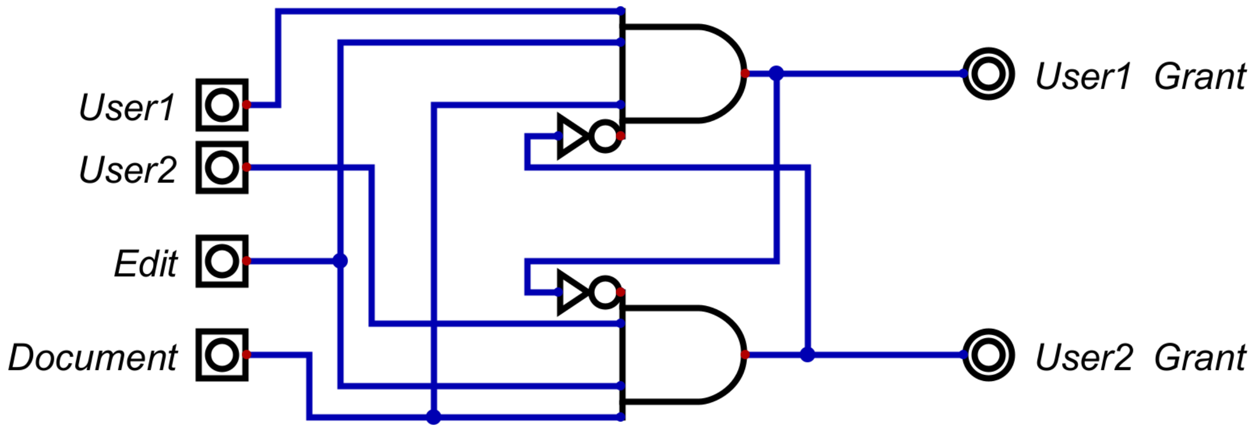


Figure 7: General Mutually Exclusive constraint schematic

3.3.5.4 N-Person Control Constraint

An N-person constraint is added during rule definition. It is a single-rule type constraint only affecting the *Subjects* in the corresponding rule. When converting an N-person constraint to VHDL, all *Subjects* involved converge at an AND gate before the *Grant*/*Deny* gate. Figure 8 presents a hardware representation of a 2-person rule. enforcing the security of a locked door needing two keys, *Key1* and *Key2,* to be unlocked. If *Key1* and *Key2* are not scanned simultaneously, the rule is not granted, and the *Door* is not *Unlocked*.

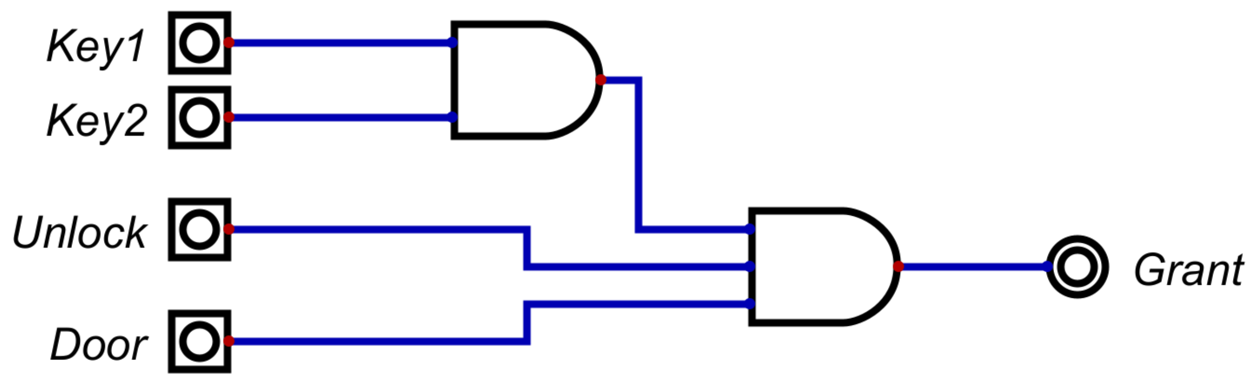


Figure 8: 2-person constraint schematic

3.3.5.5 Inheritance Constraints

To add an Inheritance constraint to a policy, two *Subjects* or *Roles* must be defined, one as beneficiary and the other as tribute. When the *Subjects* or *Roles* are identified, the beneficiary is added as an optional input to all the tribute's rules. To convert this constraint to VHDL, the tribute and beneficiary are connected by an OR gate before the grant gate. This allows either the beneficiary or the tribute to drive a *Grant*/*Deny* gate, as shown in the schematic in Figure 9. The figure demonstrates the relationship between *Bob*, the tribute, and *Alice*, the beneficiary. *Bob* has two AB rules, and *Alice* inherits from *Bob*. Therefore, *Alice* is allowed access to both of *Bob's* rules. It is important to note if the tribute, *Bob*, is constrained by a *Role* constraint, *Alice* will not inherit *Bob's Roles*.

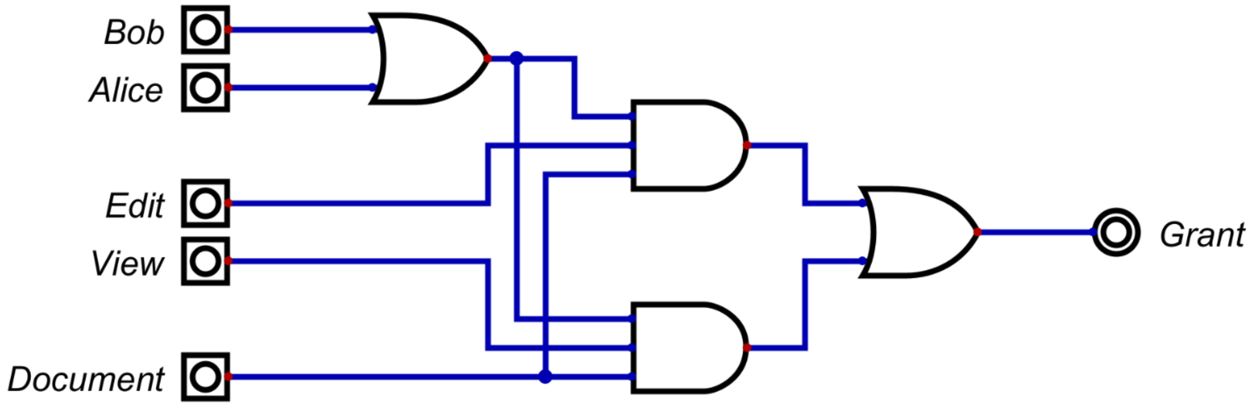


Figure 9: Basic *Inheritance* schematic

3.3.5.6 Separation of Duty

Separation of duty is not fully implemented into the script at this time. While some SoD aspects are not handled by the script, there are plans to implement full SoD support in the future. There are five separate SoD types studied in this work: Simple Static SoD, Dynamic SoD, Simple Dynamic SoD, Object-Oriented SoD, and Operational SoD.

Simple Static SoD is implemented differently than other constraints, as this SoD type has no defined relationship. Simple Static SoD is implemented with the addition of new RB *Deny* rules as opposed to adding a constraint. This will *Deny* all *Subjects* with a certain *Role* from being *Granted* a particular *Object* or *Action*. Figure 10 depicts an example hardware implementation of this SoD rule type. It demonstrates the denial of any *Subject* with the *Employee Role* to *Write Checks*. This approach is sufficient in representing Simple Static SoD in any policy.

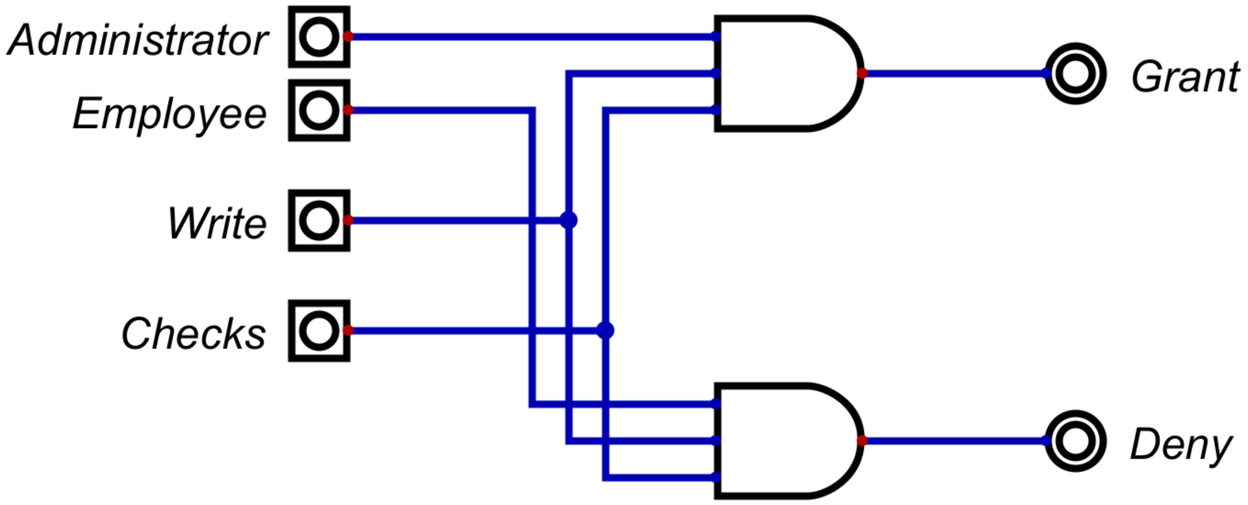


Figure 10: Simple Static SoD schematic

Dynamic SoD is one of the SoD aspects not currently addressed by the script. When defining Dynamic SoD (DSoD), the policy creator needs to identify conflicting *Actions*. Upon DSoD implementation, a *Subject* should be able to claim one conflicting *Action*. When an *Action* is claimed, the *Subject* may not be granted access to any other claimable *Action*. Furthermore, no other *Subject* may be granted access to the claimed *Action*. To realize DSoD in hardware, the state of the *Actions* must be stored. To achieve this, an SR latch is attached to the output of each rule involved in the DSoD rule set. When a rule is granted with a claimable *Action*, that *Action* is considered claimed and results in the SR-flip-flop output being set to Boolean logic '1'. The *Action* will remain in a claimed state until reset. Figure 11 offers an example schematic of a DSoD constraint where *User3* and *User4* can write or approve *EmployeeChecks*. Consistent with the DSoD definition above, when one of the *Subjects* claims one of the *Actions*, they may not access the opposing *Action*, and both *Subjects* cannot claim the same *Action*.

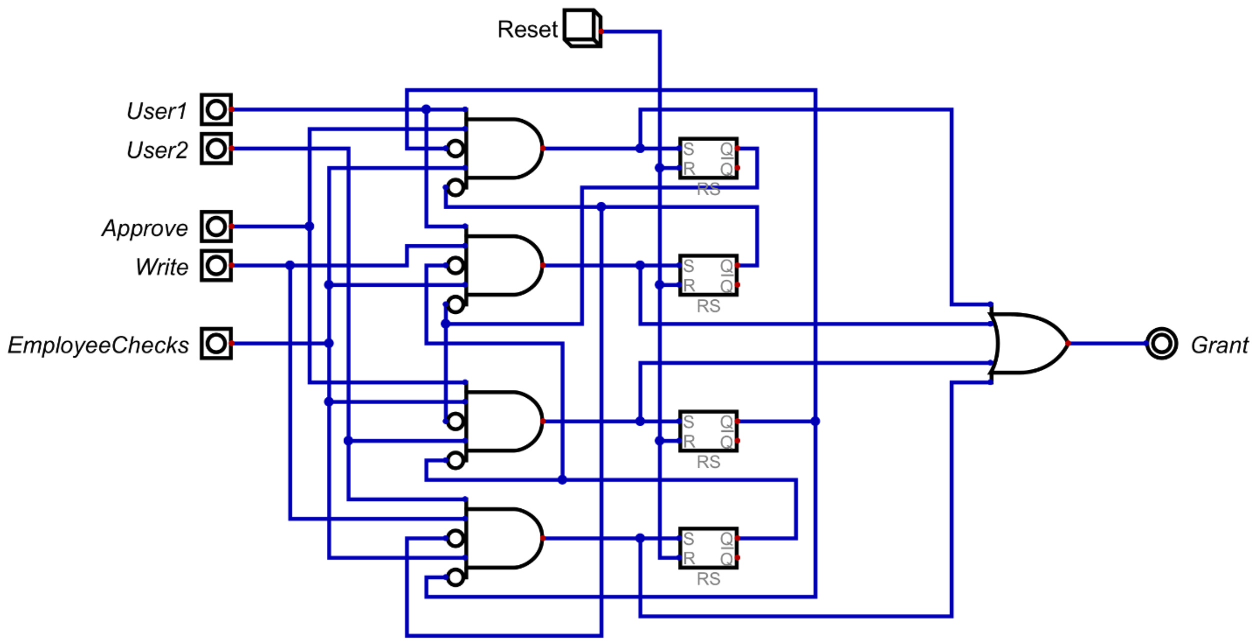


Figure 11: Example Dynamic SoD schematic

Simple dynamic SoD (SDSoD) is implemented during policy definition, meaning the policy creator must refrain from giving conflicting *Roles* to the same *Subject*. This is enough to satisfy SDSoD and does not require hardware enforcement.

Object-Oriented SoD is another type not currently addressed by the script. Object-oriented SoD is converted to hardware in the same manner as DSoD. The difference between these SoD types arises from the conflicting variable. In DSoD the conflicting variable is the *Action*, but in Object-Oriented SoD, the conflicting variable is the *Object*. A *Subject* will claim an *Object*, and when the *Object* is claimed, the *Subject* will not be able to access other conflicting *Objects*. Object-Oriented SoD requires state-keeping like DSoD. Figure 12 shows an example of Object-Oriented SoD. *User1* and *User2* may *edit* both *Document1* and *Document2*, however, when they *Edit* one document, they may not *edit* the other document. Either user may not *edit* the *document* that has been edited by the other. Given implementation similarities, this hardware approach is nearly identical to Figure 11 (DSoD), and the difference in this example derives from there being two *Objects* and only one *Action* as opposed to two *Action* and one *Object*.

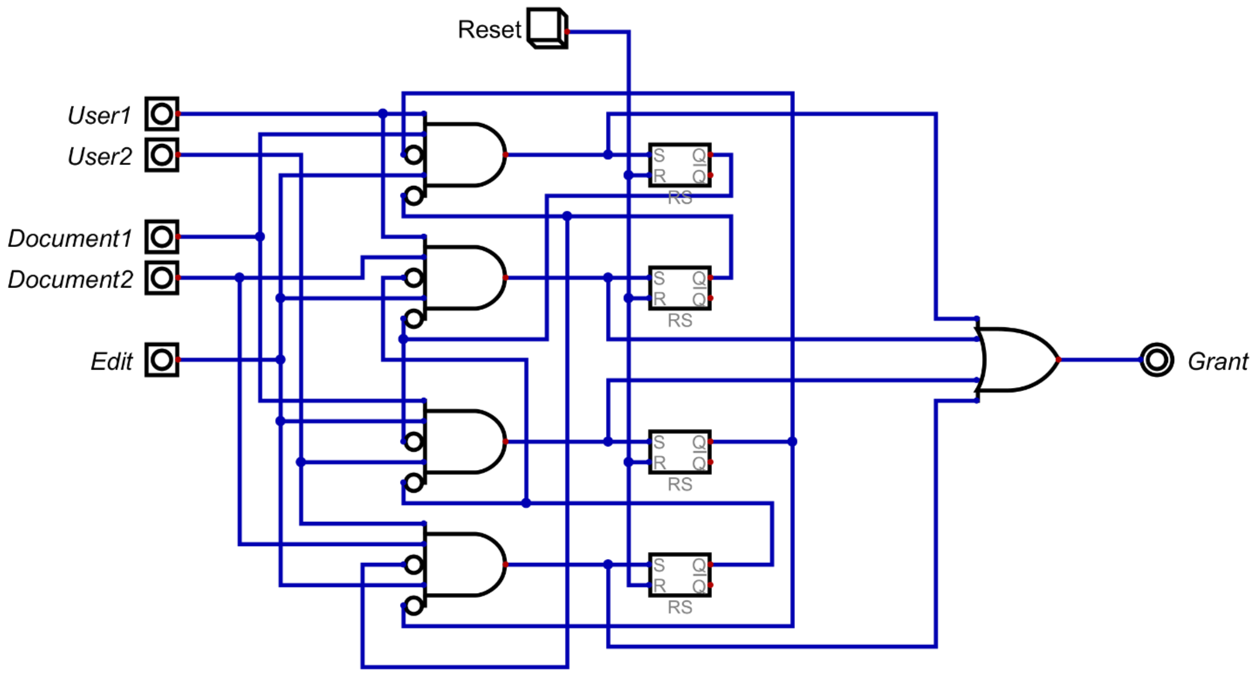


Figure 12: Object-Oriented example schematic

Respecting Operational SoD (OSoD), similarities between Object-oriented and Dynamic SoD continue. The difference concerning Operational SoD is that multiple *Subjects* may access the same *Action*, but an individual *Subject* may not access all *Actions*. This SoD type does not necessitate checking for claimed *Actions* outside of individual *Subjects'* rules as opposed to

Object-Oriented SoD and DSoD. Figure 13 demonstrates how OSoD is enforced in hardware. There are three *Subjects*: *User1*, *User2,* and *User3.* All *Subjects* may *Approve* or *Write* the *Object*, *EmployeeChecks.* However, no individual *Subject* may perform both *Actions.*

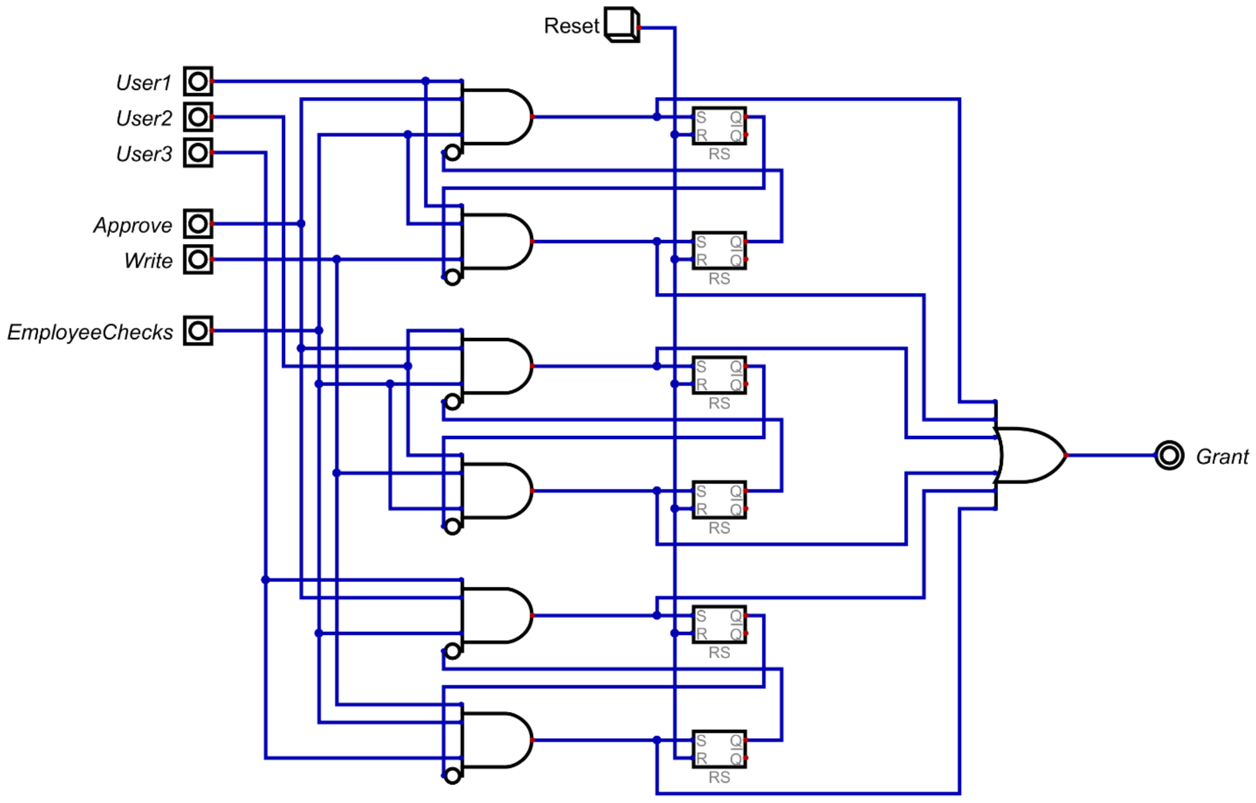


Figure 13: Example Operational SoD schematic

3.3.5.7 Conflict of Interest

When defining Conflict of Interest (COI) constraints, conflicting *Objects* must be designated. Any number of conflicting *Objects* may be selected. In a COI constraint, a *Subject* who has access to a conflicting *Object* is denied access to other conflicting *Objects*. During hardware conversion, deny rules are automatically inserted into the design. For a *Subject* who has access to an *Object* in the COI set, a *Deny* rule will be generated to disallow access to remaining conflicting *Object* in the set. Figure 14 illustrates an example COI relationship in hardware.

Company A sensitive info and *Company B sensitive info* are conflicting *Objects*. *Jason* is allowed to access *Company A sensitive info*, so a deny rule is automatically generated to deny *Jason* from *Company B sensitive info.*

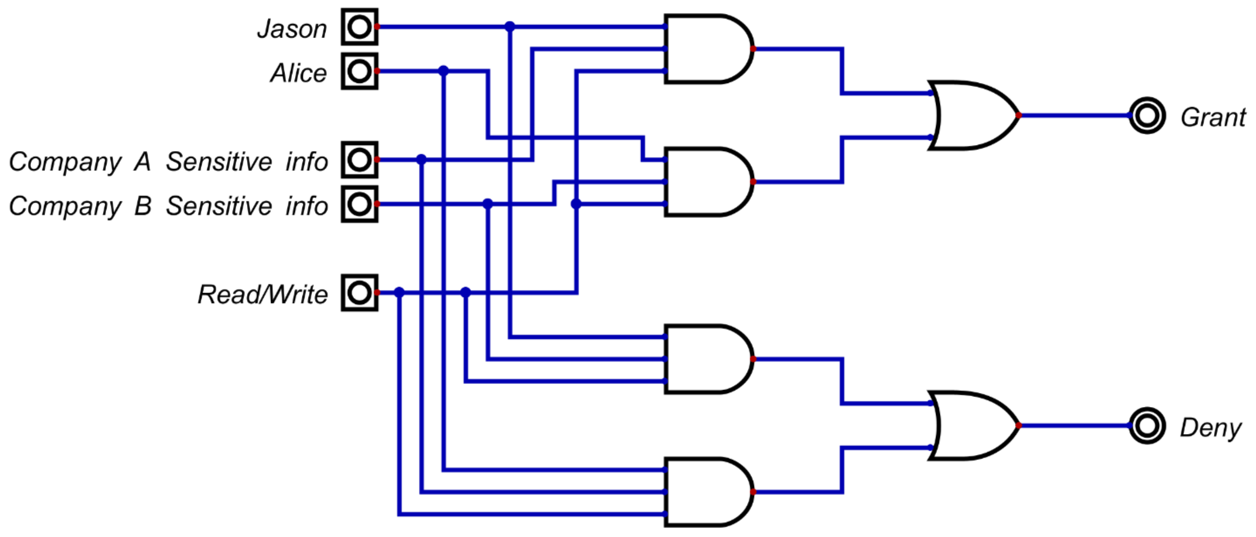


Figure 14: Schematic of a COI relationship in hardware

3.4 Policy Error Detection

After the policy is designed using the script, or by hand. The policy creator must emulate the VHDL in any computer-aided design (CAD) or electronic design automation (EDA) program to error check the policy. The program used for this work is Xilinx Vivado with the Xilinx Virtex-7 FPGA VC707 evaluation board. There are three errors studied in this research, *Conflict Fault*, *Privilege Escalation Fault*, and *Cyclic Inheritanc*e.

3.4.1 Conflict Fault

A *Conflict Fault* only exists in policies with at least one *Grant* rule and one *Deny* rule. As addressed in Section 3.2, when a *Grant* or *Deny* rule is created, they are added to the corresponding *Grant* or *Deny* LC. The outputs of rules in each LC are united with an OR gate, providing a single output for both LCs. If an LC output is high, a rule in the LC is either *Granted* or *Denied*, depending on where the signal originated. The single outputs from the LCs condense into an AND gate, and if both signals are logic ‘1’, there is a *Conflict Fault*. Figure 15 identifies the schematic for detecting a *Conflict Fault*. *Grant Logic Circuitry* comprises logic for a policy’s *Grant* rules while *Deny Logic Circuitry* contains a policy’s *Deny* rule logic. *Rule Inputs* encompass all the inputs for both rules, including *Subjects*, *Objects*, *Actions*, and *Roles*.

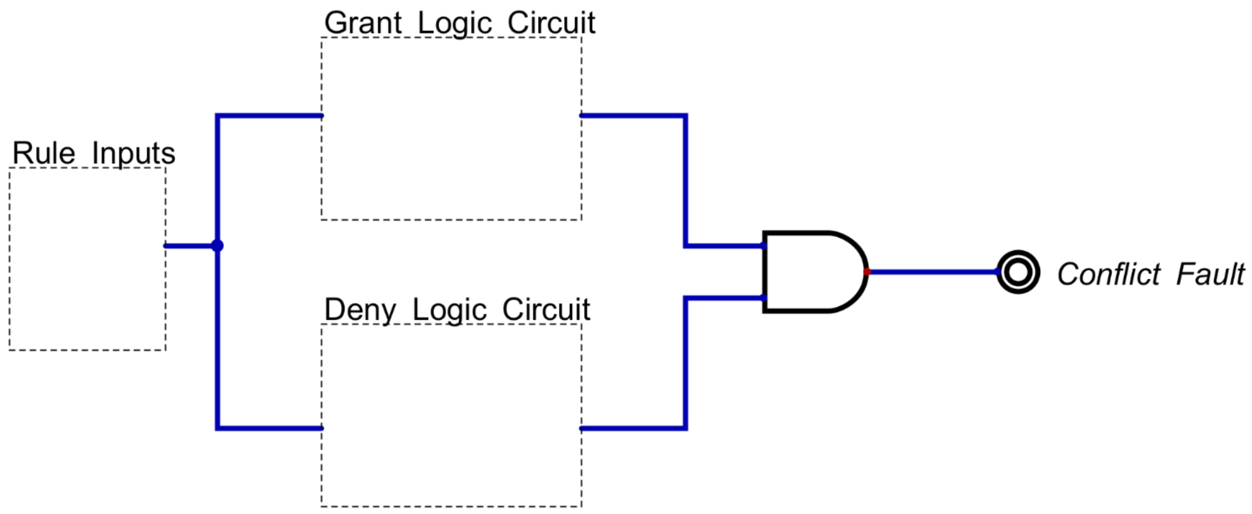


Figure 15: Example schematic for detecting *Conflicting Faults*

3.4.2 Privilege Escalation Fault

A *Privilege Escalation Fault* occurs when a *Subject* is granted access to an *Action* or *Object* for which they do not have the appropriate *Role*. In general, a *Subject* with a lower privileged *Role* should not inherit from a *Subject* with a higher privileged *Role*, but this may still transpire due to human error. For this error to develop, three conditions must be met. First, there must be an AB rule with a *Role* constraint. Second, the *Subject* with the *Role* constrained rule must be the tribute in an *Inheritance* constraint. Lastly, the beneficiary in the *Inheritance* constraint must not have a *Role* permitting access to the restricted *Object* or *Action*. For example, in an MLS policy, a *Subject* with a low-level clearance should not inherit from a *Subject* of a higher clearance. Figure 16 demonstrates a policy in which a *Subject* with no *Roles*, *Alice*, inherits from a *Subject*, *Jason*, with the *Professor Role*, leading to a *Privilege Escalation Fault*.

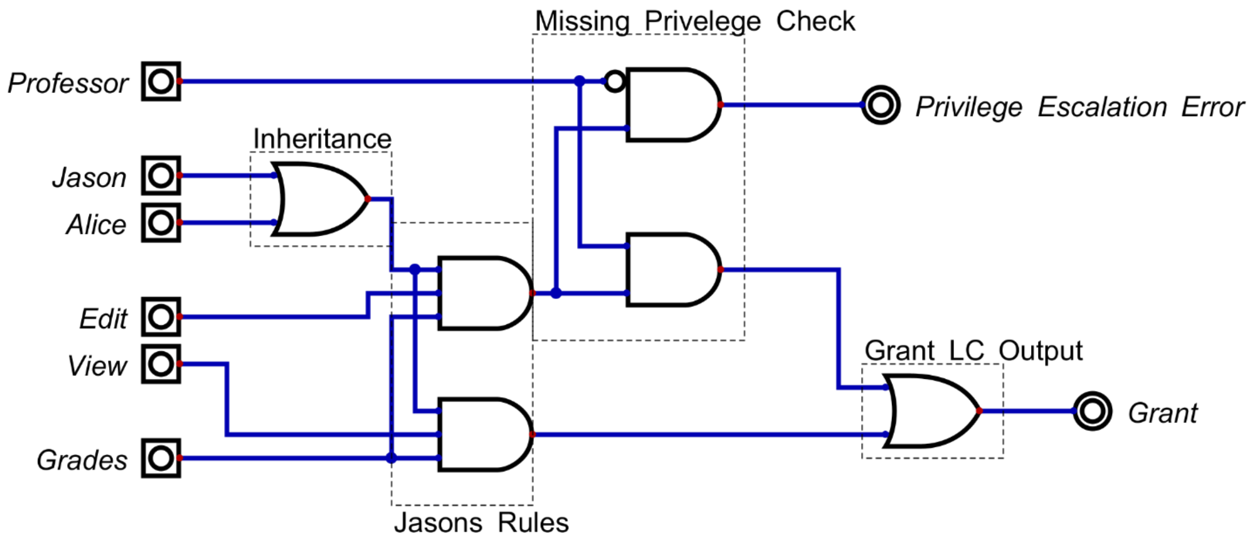


Figure 16: *Privilege Escalation* model

In the figure there are two grant rules which are as follows.

- Rule Name: rule1 | Type: simple *Attribute* rule | Permission: grant | Content -> sub: Jason | obj: Grades | *Action*: Edit | *Roles*: ['Professor']
- Rule Name: rule2 | Type: simple *Attribute* rule | Permission: grant | Content -> sub: Jason | obj: Grades | *Action*: View | *Roles*: []

There are only rules for the *Professor*, *Jason*. He is allowed to *Edit* and *View Grades*. *Alice* inherits from *Jason*, resulting in her gaining access to *Jason's* rules. To represent this, the inputs from *Alice* and *Jason* are combined into an OR gate. However, there is a problem with *Jason's* first rule. It is restricted by the *Professor Role*, meaning that *Jason* can only *Edit* the *Grades* if he contains the *Professor Role*. *Alice* has no *Roles*, but by inheriting from *Jason*, she gains the rule but not the *Role*. Before *Alice* is granted the constrained rule, a *Role* check must take place. This is executed after the initial *Grant gate* (shown in the *Missing Privilege Check* block in Figure 16). The rule will only be granted if the *Role* needed is present. If the *Role* is not present, the fault will be detected, and the *Privilege Escalation Error* output will assert as in Figure 17. Figures 17 and 18 portray these scenarios to convey how *Alice* is granted or denied *Jason's* rules.

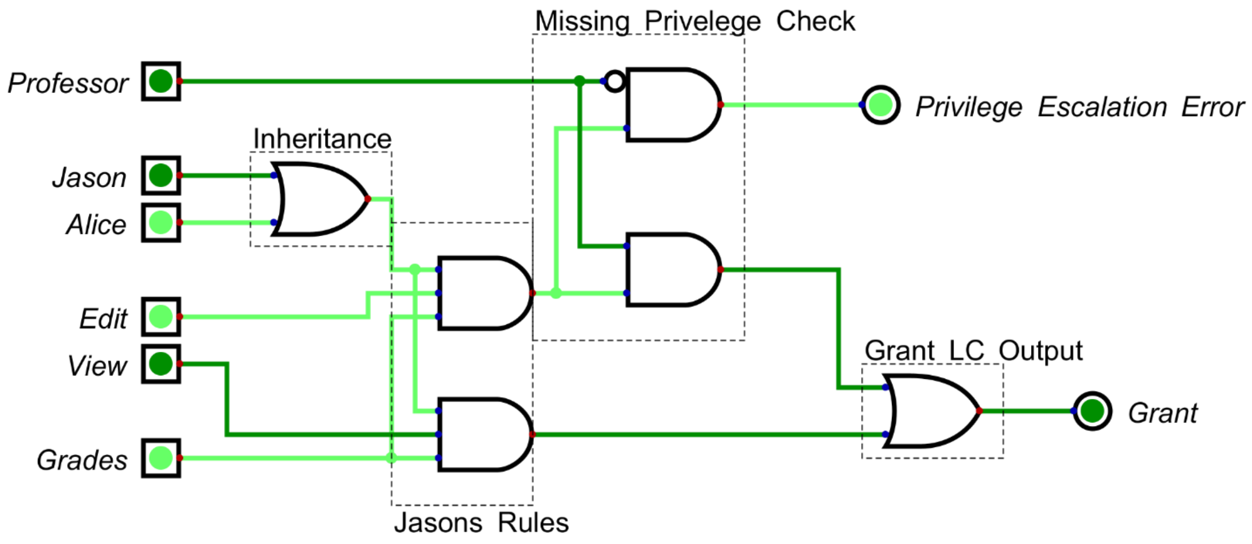


Figure 17: *Privilege Escalation* error detected from *Alice* attempting to *Edit Grades*

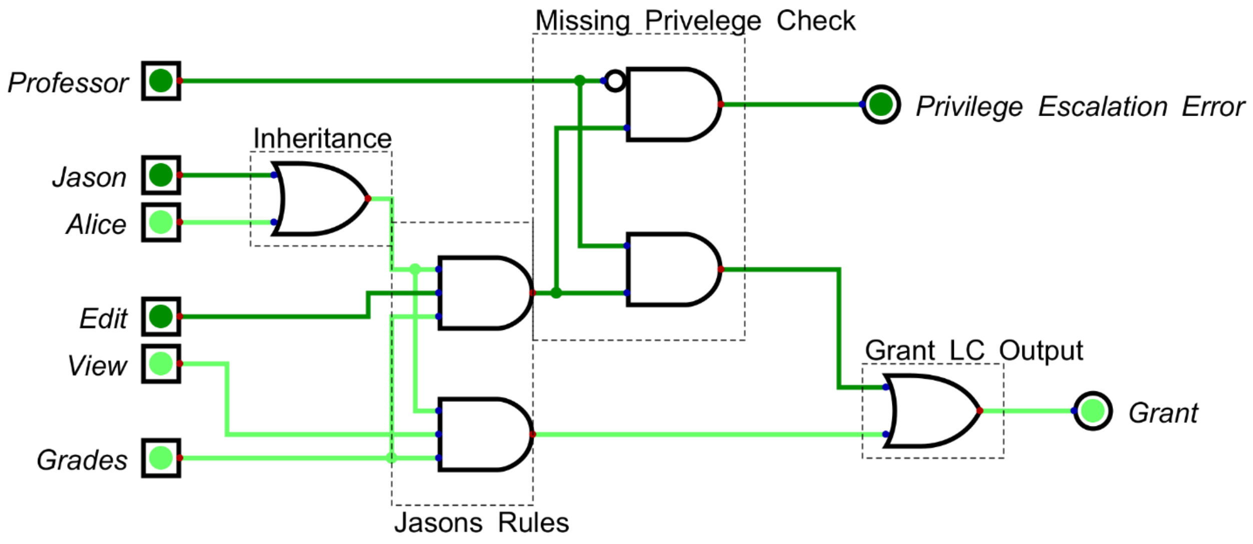


Figure 18: *Alice* granted access to *Jason's* second rule

3.4.3 Cyclic Inheritance

In essence, Cyclic Inheritance is caused by a *Subject* inheriting from itself, causing a feedback loop. Respecting the hardware model for Cyclic Inheritance detection, this causes an endless feedback loop resulting in all *Subjects* involved in the loop to be asserted. Even when the *Subjects* are de-asserted during emulation, the loop continues, suspending the *Subjects* in a constant asserted state. To detect this fault, the inputs to every *Inheritance* gate are monitored. When two or more *Subjects* are Boolean logic '1', there is a *Cyclic Inheritance Fault*. Figure 19 outlines *Cyclic Inheritance* hardware detection. Checking for *Cyclic Inheritance* only requires emulation on the *Subjects* involved in *Inheritance* constraints and is not affected by any rules. A new LC is created for *Cyclic Inheritance* detection and operates parallel to the remainder of the policy during emulation. During emulation, only one *Subject* will be set to Boolean logic '1' at a time to check for this error.

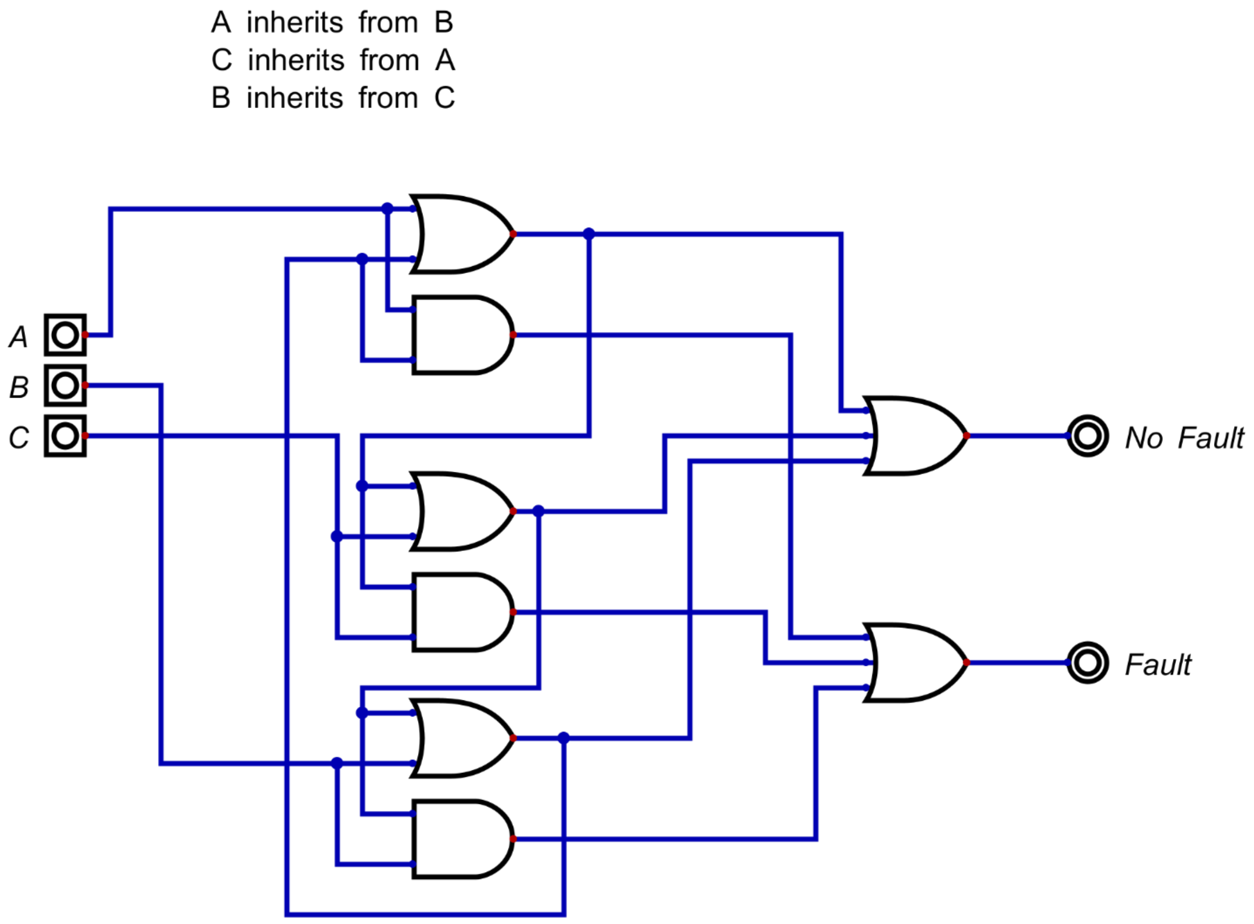


Figure 19: Example schematic of *Cyclic Inheritance* detection

3.5 Policy Testing

Policy testing is accomplished with Xilinx Vivado on the Xilinx Virtex-7 FPGA VC707 evaluation board. To begin assessing the policy, it requires VHDL conversion using the hardware models outlined in this work. When the VHDL file is complete, it must pass through three stages before it is ready to emulate using the FPGA: synthesis, implementation, and loading onto the FPGA.

Synthesis transforms abstract specifications of the VHDL design and translates it to an arrangement of logic gates. This step generates register transfer level (RTL) schematics and prepares the design for implementation. [12]

Implementation defines where the gates will be physically placed in the board and how they are routed. When this step is complete a bitstream is generated and used to program the FPGA, so emulation can ensue.

3.5.1 Emulation

A testbench is needed to emulate the design on the FPGA by manipulating the inputs of an individual rule to monitor for *Privilege Escalation* and *Conflicting Faults*. Cyclic Inheritance will also be detected during this time, but only the *Subject* needs be considered for detection.

Emulation may transpire in one of two ways: manually or automatically. The policy creator may design the testbench to allow manual rule switching. This can be accomplished by designing a testbench to change rule inputs based off a pre-determined input set by using FPGA peripherals. Alternatively, the testbench may be designed to switch rule inputs autonomously. This is achieved by using the FPGA user clock to switch between the pre-determined input set. A clock divider may be used to assure correct set-up and hold times are met to preserve data, or a specific user clock frequency may be programmed. However, when using the clock to switch between inputs, the

LEDs flash too quickly for the human eye to detect, making physical feedback impossible. Vivado offers a de-bugging tool to allow the policy creator to generate a waveform of the inputs and outputs (I/O) on the FPGA. In this case, emulations may be quickly completed, resulting in a waveform for error feedback.

4. Results

The primary objective of this work was to design a scheme for converting AC policies to hardware to allow for real-time error detection during policy creation. To evaluate the scheme, a small policy consisting of 45 RB and three AB rules was created to contain the aforementioned faults while featuring a simplistic schematic for analysis. When creating schematics for illustration, large policies become increasingly hard to trace for understanding. Thus, limited constraints were used in the example policy. It is also important to note that all previous constraints have been assessed in multiple policies throughout this research and are not critical for error detection. All constraints other than the *Role* constraint and Inheritance constraints apply only to policy functionality and are not directly responsible for any of the outlined potential errors.

The test policy is an MLS design with four users added for *Inheritances*. The Role-Based rules used in the testing policy can be found in Table 1. Attribute-Based rules and Inheritance constraints are seen in Table 2 and Table 3, respectively. Due to the amount of space necessary for listing 48 individual rules, all *Objects* and *Actions* are grouped by *Role* or *Subject*. *Object* and *Action* combinations in the tables below are an individual rule with the corresponding *Role* or *Subject* in that row. A similar approach was also used when creating the schematic in Figure 20 to minimize space and to provide a readable appearance.

Table 1: Role-Based rules in results testing policy

Role	Object	Actions	Permissions
Administrator	OS pages, OS folders, Widgets	Delete, Publish, Create, Upload, Modify, View	Grant
Designer	OS pages, OS folders, Widgets	Publish, Create, Upload, Modify, View	Grant
Editor	OS pages, OS folders	Upload, Modify, View	Grant
Editor	OS pages, OS folders, Widgets	Upload	Deny

Table 2: Attribute-Based rule in testing policy

Subject	Object	Action	Role	Permission
Gary	OS pages, OS folders, Widgets	Approve	Administrator	Grant

Table 3: Inheritances in testing policy

Tribute	Beneficiary
Gary	Alice
Alice	Frank
Frank	Derek
Derek	Gary

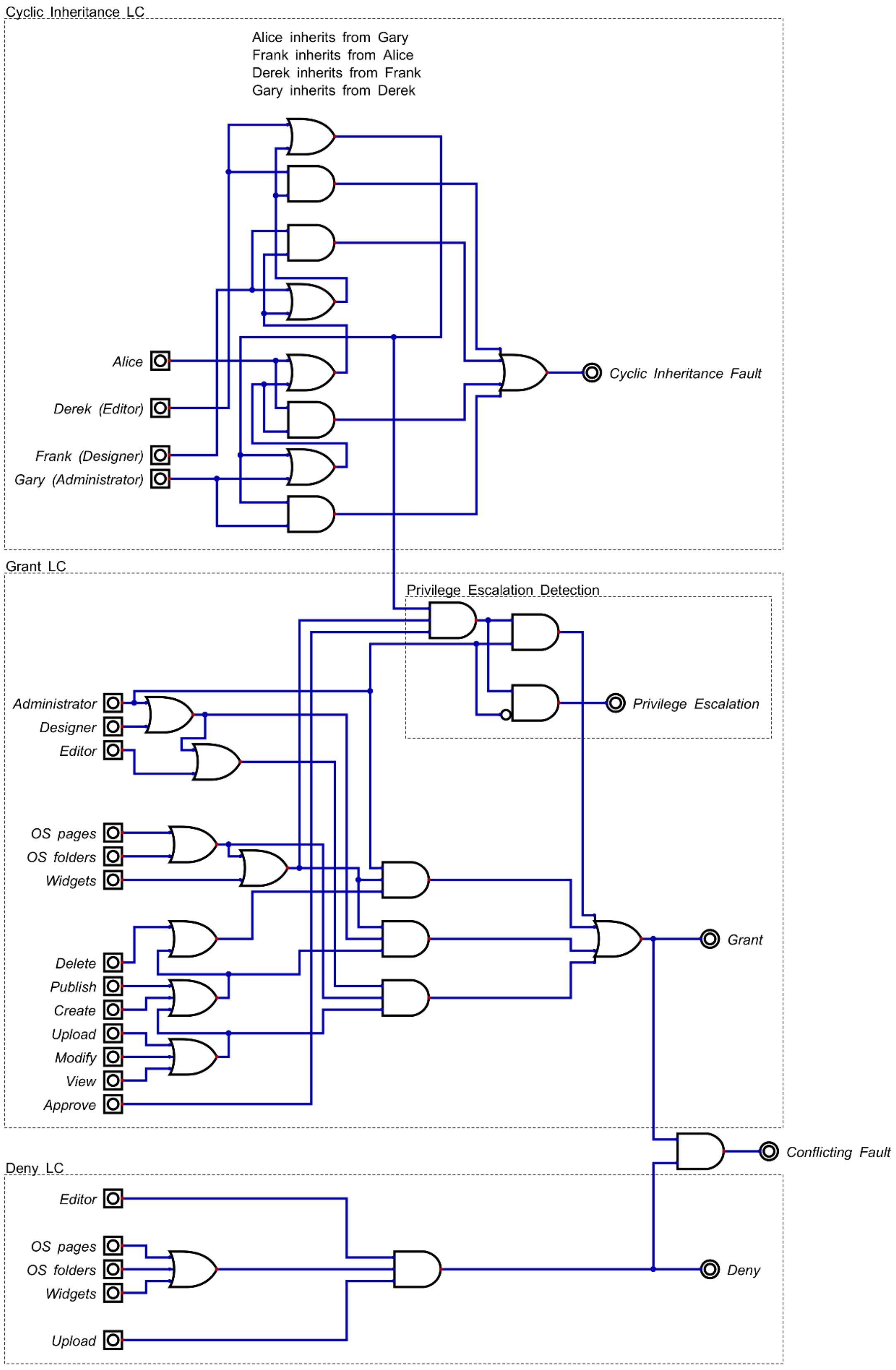
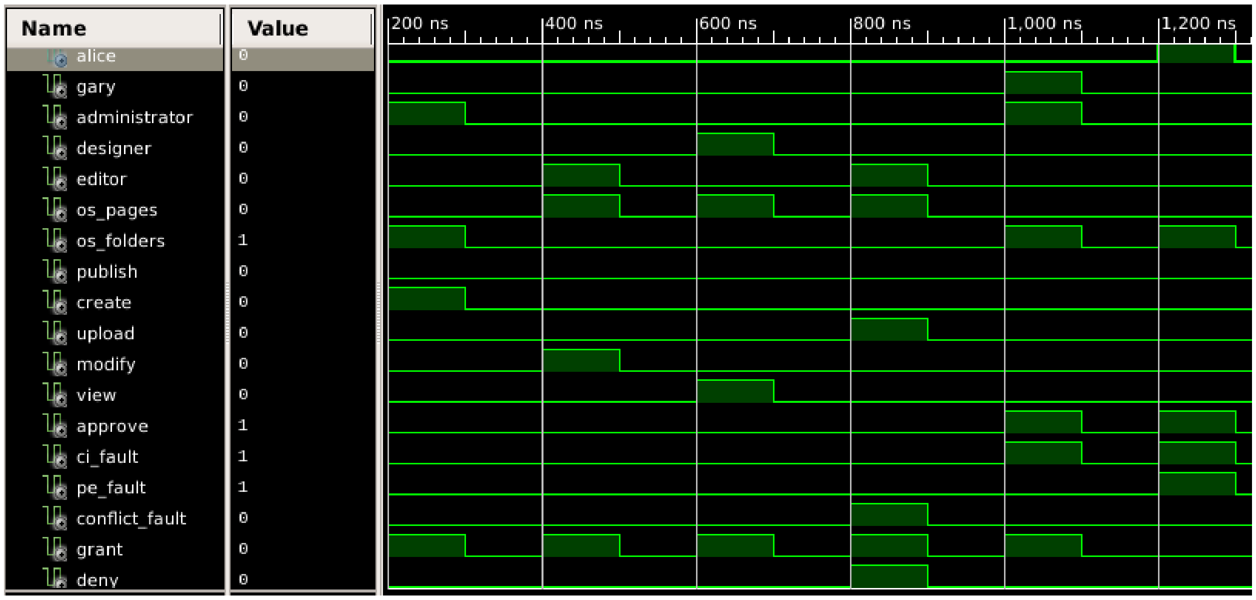


Figure 20: Testing Policy Schematic

4.1 Emulation Results

In the testing policy, *Gary* has a *Role* constrained rule and will receive a beneficiary, *Alice* who has no *Roles*, and this should cause a *Privilege Escalation Fault*. *Alice* is also a part of an *Inheritance* chain where all other *Subjects* in the policy either directly or indirectly inherit from *Alice,* including *Gary*. This *Inheritance* chain causes a loop in the system, suspending those *Subjects* in a constant asserted state. During synthesis and implementation, Vivado outputs a “logical loop” error, indicating there is a loop in the policy before it is loaded onto the FPGA. In this case, the policy creator may continue to test the design, or they may re-evaluate the *Inheritance* constraints within the policy. Figure 23 exhibits the emulation waveform for the values of I/O. Policy FPGA utilization is presented in Table 4 with the integrated Vivado debugging core, which is necessary to generate waveforms.



Name	Value
alice	0
gary	0
administrator	0
designer	0
editor	0
os_pages	0
os_folders	1
publish	0
create	0
upload	0
modify	0
view	0
approve	1
ci_fault	1
pe_fault	1
conflict_fault	0
grant	0
deny	0

Figure 21: Simulation waveform for the testing policy.

Table 4: Testing Policy hardware utilization.

Resource	Utilization	Available	Utilization Percent
Look Up Table (LUT)	1645	303600	0.54%
LUTRAM	171	130800	0.13%
Flip-Flops (FF)	2551	607200	0.42%
Block RAM (BRAM)	1	1030	0.10%
I/O	6	700	0.86%

Not all I/O are identified in Figure 23 for brevity. However, all three faults were successfully detected by analyzing the I/O in the Figure. Fault detecting outputs are found towards the bottom of the I/O list on the left side of Figure 23. Fault detecting outputs are asserted at 800ns, 1,000ns, and 1,200ns, respectively. While the waveform presents faults in a specific order, the procession of faults does not affect their detection. In this design, the first fault detected was the *Conflicting Fault*. It is at Boolean logic '1' when the *Editor Role* is attempting to *Upload OS_pages*. This fault is caused because the Editor is both allowed and denied this rule in the policy.

The next fault detected at 1,000ns is the *Cyclic Inheritance Fault*. In this test, it was asserted when *Gary* tries to *Approve OS_folder*. This rule is still granted as it should be, but due to the *Inheritance* loop, there is a resulting fault. The *ci_fault* value will only be asserted when a member of that *Inheritance* loop is also asserted. This simplifies searching for members of *Inheritance* loops. As emulation continues, the policy creator may find all *Subjects* involved in the *Inheritance* chain by analyzing active *Subjects* when *ci_fault* is '1'.

The final fault detected at 1,200ns is the *Privilege Escalation Fault*. This is apparent when *Alice* tries to *Approve OS_pages*. *Alice* inherits this *Role*-constrained rule from *Gary* who has the *Administrator Role*. Being *Role* constrained means only *Gary* may be granted this rule if and only if he has the Administrator *Role*. *Alice* does not have the *Administrator Role*; therefore, a *Privilege Escalation Fault* is detected. It is worth noting that the *ci_fault* value is high at this time as well, indicating *Alice* is also part of the *Cyclic Inheritance* loop.

5. Conclusion and Future Work

5.1 Conclusion

To practically and adequately emulate AC policies in hardware, a tool flow has been developed to allow smooth translation between AC policy rules and hardware implementation, allowing hardware conversion of any policy. The combinations of constraints and rules can create an incredible amount of diversity for a policy. In this research, there are thirteen different rules and constraints studied. Any number of constraints may be combined with one of the two core rule types, resulting in around 4,000 combinations. All combinations should work together, and despite policy diversity, the scheme will uphold the integrity and functionality of the policy at every step.

5.2 Future Work

The concept of live error detection through hardware is shown to be a success, but there is still much to be done in terms of automation. The first items planned for future work is the addition of more robust SoD support. Currently, the only SoD types supported through the script are Simple Static SoD and Simple Dynamic SoD. Dynamic, Object-Oriented, and Operation SoD types are to be implemented next.

Another phase of planned work is to add a user interface. This would facilitate analysis and error detection of large policies. The most important aspect of a future user interface is expected to be a window with a running list of added rules. Adding a user interface is not as highly prioritized as finishing the functionality of the script with full SoD support, but this will streamline rule conversion and make for a more polished error checking process.

6. References

[1] Hu, Vincent C, et al. Verification and Test Methods for Access Control Policiesmodels. June 2017, 10.6028/nist.sp.800-192.

[2] Clarke, Edmund M., et al. Model Checking. MIT Press Books, 2nd ed., Cambridge, MA, USA, MIT Press, 4 Dec. 2018, mitpress.mit.edu/books/model-checking-second-edition. Accessed 28 Apr. 2022.

[3] Trochimiuk, Maciej. "FPGA Programming - What Is It, How It Works and Where It Can Be Used." Codilime, 30 Apr. 2021, codilime.com/blog/FPGA-programming-how-it-works-and-where-it-can-be-used/. Accessed 3 Mar. 2022.

[4] Kugblenu, Francis, and Memon Asim. Separation of Duty in Role Based Access Control System: A Case Study 0 Separation of Duty in Role Based Access Control System: A Case Study. 2007.

[5] Hu, Vincent. Real-Time Access Control Rule Fault Detection Using a Simulated Logic Circuit. 3 Feb. 2014.

[6] ACRLC, https://cms.csrc.nist.gov/projects/access-control-policy-tool/access-control-rule-logic-circuit-simulation-%28acrl

[7] Li, Qinghua. Real-Time Fault Detection in Access Control Rules Using Logic Circuits. 2019

[8] Editor, CSRC Content. "Access Control - Glossary | CSRC." Csrc.nist.gov, csrc.nist.gov/glossary/term/access_control.

[9] Aiello, Samuel. "Human Error: The Nemesis of Access Control." *SSRN Electronic Journal*, 2022. *Crossref*, https://doi.org/10.2139/ssrn.3999894.

[10] Hu, Vincent, et al. Assessment of Access Control Systems. Sept. 2006.

[11] Kugblenu, Francis, and Memon Asim. Separation of Duty in Role Based Access Control System: A Case Study 0 Separation of Duty in Role Based Access Control System: A Case Study. 2007.

[12] Jiang, Jie-Hong. Logic Synthesis in a Nutshell. 2008.