Real-Time Access Control Rule Fault Detection Using a Simulated Logic Circuit

Vincent C. Hu
National Institute of Standards and Technology
Gaithersburg, MD, USA
vhu@nist.gov

Karen Scarfone
Scarfone Cybersecurity
Clifton, VA, USA
karen@scarfonecybersecurity.com

***Abstract*—Access control (AC) policies can be implemented based on different AC models, which are fundamentally composed by semantically independent AC rules in expressions of privilege assignments described by attributes of subjects/attributes, actions, objects/attributes, and environment variables of the protected systems. Incorrect implementations of AC policies result in faults that not only leak but also disable access of information, and faults in AC policies are difficult to detect without support of verification or automatic fault detection mechanisms. This research proposes an automatic method through the construction of a simulated logic circuit that simulates AC rules in AC policies or models. The simulated logic circuit allows real-time detection of policy faults including conflicts of privilege assignments, leaks of information, and conflicts of interest assignments. Such detection is traditionally done by tools that perform verification or testing after all the rules of the policy/model are completed, and it provides no information about the source of verification errors. The real-time fault detecting capability proposed by this research allows a rule fault to be detected and fixed immediately before the next rule is added to the policy/model, thus requiring no later verification and saving a significant amount of fault fixing time.**

Keywords—Access Control; Authorization; Model Verification; Testing; Verification

I. INTRODUCTION

Systems (e.g., operating systems and database management systems) often adopt access control (AC) to control which principals (such as users or processes) have access to which resources based on AC policies. AC policies can be implemented based on AC models or by semantically independent AC rules in expressions of privilege assignments. The implementations fundamentally consist of a set of rules or privilege hierarchies described by AC variables, i.e. subjects (or their attributes), actions, objects (or their attributes), and environment conditions of the protected systems. A rule assigns permission: grant/denial of specific actions on objects or object attributes to authorized subjects or subject attributes under environment variables.

Specifying correct behaviors of AC policies is a challenging task, especially when an AC policy includes a large number of rules. Identifying discrepancies between AC policies and their intended functionalities is crucial because correct policy behaviors are based on the premise that the policies are correctly specified. Incorrect AC policies result in faults that not only leak but also disable access to information, and faults are especially difficult to detect without support of formal embedded models such as Multi-Level Security (MLS) and Chinese Wall [1].

Most research on AC model or policy verification techniques is focused on one particular model, and almost all of the research is in applied methods, which require the **completed** AC policies as the input for verification or test processes to generate fault reports. Even though correct verification is achieved and counterexamples may be generated along with found faults, those methods provide no information about the source of rule faults that might allow conflicts in privilege assignment, leakage of privileges, or conflict of interest permissions. The difficulty in finding the source of fault is increased especially when the AC rules are intricately covering duplicated variables to a degree of complexity. The complexity is due to the fact that a fault might not be caused by one particular rule; for example, rule *x* grants subject/attribute *s* access to object/attribute *o*, and rule *y* denies the group subject/attribute *g*, which *s* is a member of, access to object *o*. Such conflict can only be resolved by removing either rule *x* or *y*, or the *g* membership of *s* from the policy. But removing *x* or *y* affects other rules that depend on them (e.g., a member of subject group *g k* is granted access to object *o*), and removing *s*'s membership in *g* will disable g's legitimate access to other objects/attributes through the membership. Thus, it requires manually analyzing each and every rule in the policy in order to find the correct solution for the fault.

To address the issue, we propose the AC Rule Logic Circuit Simulation (ACRLCS) technique, which enables the AC authors to detect a fault when the fault-causing AC rule is added to the policy, so the fixing can be implemented in real time (on the spot) before adding other rules that further complicate the detecting effort. In other words, instead of checking by retracing the interrelations between rules after the policy is completed, the policy author needs only check the new added rule against previous "correct" ones. In ACRLCS, AC rules are represented in a simulated logic circuit (SLC) (pronounced CELL-see). By simulation, we mean ACRLCS is not necessarily implementable by a physical electronic circuit; however, the concept can be implemented and computed through simulated software.

ACRLCS is composed of SLCs representing AC rules specified in Boolean expressions. A SLC should be able to preserve the assignments of AC variables and privilege hierarchies (through inheritance) and evaluate access permission (e.g., grant or deny) from the implemented rule. With this principle, our proposed technique includes two main processes:

- Construct a SLC based on AC variables specified by Boolean expressions, or relation hierarchies specified by relations in an AC rule. In the SLC, each AC variable is represented by an input switch, and the rule logic operator and hierarchy relation are simulated by logic gates.

- Develop an efficient algorithm to detect rule faults in the policy by triggering input switches representing AC rule variables under verification in the currently constructed SLC. Rule faults are generated as positive signal outputs from the SLC, indicating conflicts in privilege assignments caused by a new added rule.

The rest of this paper is organized as follows. Section II discusses related work. Section III describes the AC rule fault detecting algorithm and scheme. Section IV illustrates applications of ACRLCS. Section V discusses real-time performance of the ACRLCS. Section VI concludes the paper.

II. Related Work

Several verification techniques exist for applying model checking to AC policies. [2] presented a general AC model verification tool ACPT, which applied symbolic model checker NuSMV [3] for verifying security properties against standard AC models/rules and their combinations. [4] presented a model checking algorithm that evaluates if a policy can satisfy a user's access request as well as prevent intruders from reaching their malicious goals. [5] proposed the policy verification and validation framework based on model checking that exhaustively verifies a policy's validity by considering the relations between system characteristics and policies. Their approach defines the validity of policies and the information needed to verify them from the viewpoint of model checking as well as constructs the policy verification framework based on the definition. [6] presented a model checking approach to analyze the delegation and revocation functionalities of workflow-based enterprise resource management systems. The approach derived information about the workflow captured in a symbolic model verification specification together with a definition of possible delegation and revocation scenarios. [7] presented an abstraction-refinement technique for automatically finding errors in Administrative Role-Based Access Control security policies implemented on top of a bounded model checker.

In addition, a few techniques for automated verification of generic policies have been proposed in [8 - 14]. Some use verification tools as a backend. For instance, the declarative language Alloy [15] supported first order logic and relational calculus, and [3] used temporal logic properties with finite state models as well as the SPIN model checker [16]. There are cases where AC policies are defined as ordering relations, which can be further translated into Boolean satisfiability problems, and applied to SAT solvers [17]. The SAT solver is a program that takes formulas in conjunctive normal form (CNF) and returns assignments, or says none exists. These techniques serve as foundations for the verification of system specifications; a specification of a system can be defined as "*what the system is supposed to do*" [14, 18].

In summary, the above mentioned techniques applied to AC policies/models require all the rules or access constraints to be completely specified. Therefore, inevitably fixing faulty rules has to take every rule in the policy into consideration, because such after-the-effect change of any rule might implicitly impact other related rules (sharing subject/attributes, actions, object/attributes, especially privilege inheritance assignments). And it is too intricate to be thoroughly traced because, for example, in the specification of AC rules, some AC mechanisms allow later statements to overwrite previous ones without recognizing existing conflict of privilege assignments. Such difficulty can only be avoided by fixing the fault when it appears before additional rules are created. To the best of our knowledge, there is no relevant work for such real-time AC policy fault detection as proposed in ACRLCS.

III. AC Rule Fault Detecting Scheme

As many terms used in this paper are not well standardized, we use their definitions from a NIST Special Publication and an Interagency Report [19, 20] that are recommended for US government and adopted by some of industry and academia. An exception is that we treat the terms subject and attribute as the same variable denoted by "subject/attribute", because they do not make a difference in the ACRLCS scheme, and they can be separately treated without loss of generality. This merge also applies to object and object attributes, thus, we use object/attribute as well.

ACRLCS interlaces the two main tasks as described in the following subsections *A* and *B* until all the rules in the policy are completed.

A. SLC Representation for AC Rules

Intuitively, AC rules can be expressed by Boolean expressions that operate on AC rule variables, including privilege inheritance relations that denote the inheritances of access privilege (i.e., actions to object/attributes pairs) from one subject/attribute to another subject/attribute. Privilege inheritance is an efficient way for specifying privilege hierarchies [21] such as role hierarchies in Role Based Access Control (RBAC) policies. The following principles need to be followed when specifying AC rules in a SLC:

- Each SLC gate is a logic operation connecting AC variables enforced by the rules in a Boolean expression.
- An AC rule in a SLC must generate a permission output except for privilege inheritance assignments, which are specified by connecting inherited subject/attributes and beneficiary subject/attributes by an OR gate.
- A positive value (i.e., 1) from the permission output by triggering the input variables of the SLC represents a "grant" permission of the rule enforcing the triggered input variables.

- A negative value (i.e., 0) from a permission output by triggering the input variables of the SLC represents a “deny” permission of the rule enforcing the triggered input variables. Or, there is no rule associated with these variables.

For example, the SLC in Figure 1 shows these simple AC rules: subject/attribute *s1* is granted to perform action *a* on object/attribute *o1* and *o2*, and subject/attribute *s2* is granted to perform action *a* on object/attribute *o1*. The Boolean equivalences of the three rules are:

$s1 \wedge a \wedge o2 = p1$

$s1 \wedge a \wedge o1 = p2$

$s2 \wedge a \wedge o1 = p3$

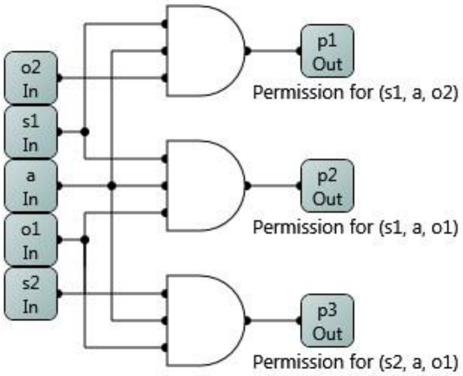
o2
In
s1
In
a
In
o1
In
s2
In
p1
Out
Permission for (s1, a, o2)
p2
Out
Permission for (s1, a, o1)
p3
Out
Permission for (s2, a, o1)

Fig. 1. AC rule examples

Trigger *a*, *s1*, *s2*, *o1*, and *o2* of the SLC will generate positive permission outputs on *p1*, *p2*, and *p3* of the three AC rules. And trigger *s2*, *a*, and *o2* will not activate a positive result from any of the three permission outputs, because there is no $s2 \wedge a \wedge o2$ rule.

Privilege inheritance relations are assigned between subjects/attributes. Usually, inheritance assignments are in compliance with the business functions of the application [21]. To implement inheritance, the inherited subject/attribute needs to create (if no rule is associated to it) or insert an OR gate directly connecting to the subject/attribute input; then the beneficial subject/attribute either directly (if it is not being inherited by other subjects/attributes) or from the output of its shared OR gate (for its beneficiaries) connects to the OR gate that was created (or inserted) by the inherited subject/attribute. Figure 2 illustrates the SLC’s implementation of inheritance assignments where subject/attribute *s3* inherits privileges (*a2*, *o2*) and (*a1*, *o1*) from subject/attribute *s2* and *s1*, and subject/attribute *s2* inherits privilege (*a1*, *o1*) from subject/attribute *s1*. *s4* does not inherit any privilege. Formally:

$s1 \wedge a1 \wedge o1 = p1$

$s2 \wedge a2 \wedge o2 = p2$

$s3 \rightarrow s2$

$s3 \rightarrow s1$

$s2 \rightarrow s1$

The symbol “→” denotes the inheritance assignment that *sx* inherits (→) *sy*’s privilege. Figure 7 is another example of privilege inheritance SLC.

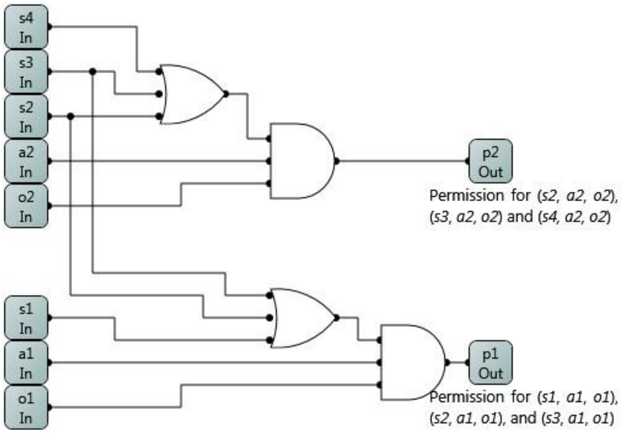
s4
In
s3
In
s2
In
a2
In
o2
In
p2
Out
Permission for (s2, a2, o2),
(s3, a2, o2) and (s4, a2, o2)
s1
In
a1
In
o1
In
p1
Out
Permission for (s1, a1, o1),
(s2, a1, o1), and (s3, a1, o1)

Fig. 2. Privilege inheritance example

Some rules may enforce more than one subject/attribute in the specification such that the included subjects/attributes need to be either presented at the same time or excluded from each other in order to grant/deny an action to objects/attributes. This type of rule is usually applied to **fine grained**, **n-person control, conflict of interest** (COI)**,** or **separation of duty** (SoD) security properties. Figure 3 illustrates SLC implementations of two such rules where permission *p1* grants action *a* to object *o* if both subject/attribute *s1* and *s2* are accessing it at the same time. Permission *p2* grants *a1* access to object/attribute *o1* to *x1* or *x2*, if they access exclusively (not at the same time) from each other.

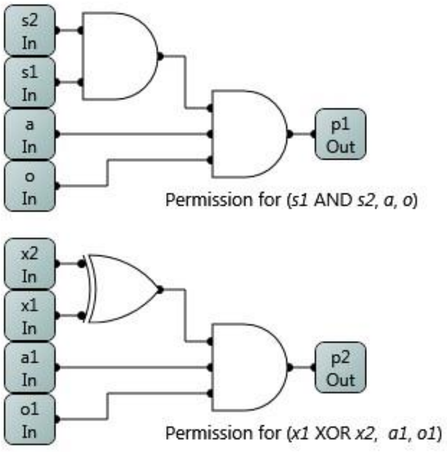
s2
In
s1
In
a
In
o
In
p1
Out
Permission for (s1 AND s2, a, o)
x2
In
x1
In
a1
In
o1
In
p2
Out
Permission for (x1 XOR x2, a1, o1)

Fig. 3. AC rules involving more than one subject/attribute

Other types of AC rules enforce the OR relation, such as the rule "*s1* OR *s2* can perform *a* to *o*" can be simply implemented either by two single rules: (*s1*, *a*, *o*) and (*s2*, *a*, *o*) or by replacing the AND gate connecting *s1* and *s2* with an OR gate in Figure 3.

Upon finishing specifying a rule in a SLC, the newly added circuit should be checked against the previous SLC for the detection of inconsistency of permission, i.e. faults of AC rules. Subsection *B* below describes the processes.

B. Rule Conflict Detection

To detect faults, ACRLCS requires two separate SLCs; one is Grant SLC (GSLC) and the other Deny SLC (DSLC). GSLC contains a SLC that implements all rules with grant permissions. DSLC contains a SLC that implements all rules with deny permissions. In other words, if an AC rule grants subject/attributes to perform actions on object/attributes, the SLC will be implemented in GSLC. And if an AC rule denies subject/attributes to perform actions on object/attributes, the SLC will be implemented in DSLC. The same actions also apply to inheritance relations (i.e., grant inheritances in GSLC, and deny inheritances in DSLC). The separation of the two SLCs allows comparing the permission output for a newly added SLC rule by trigging the subjects/attributes, actions, and objects/attributes enforced in the rule from both SLCs, to check if the new SLC rule already exists in the opposite SLC.

To implement the comparison, GSLC or DSLC, summary AND gates collect all the permissions output from the SLCs, which will then be compared through another AND gate to generate comparison (fault finding) results. As illustrated in Figure 4, if both the GSLC and DSLC contain the same rule in any of *p1 …. pn* and *p11…pnn*, the fault finding AND gate will output a positive signal, which means both grant and deny permissions are presented, i.e. a conflict fault was detected.

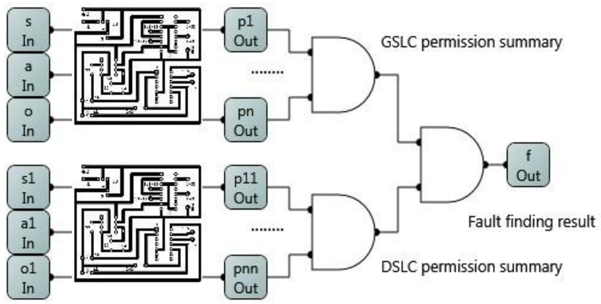

Fig. 4. Conflict resolution scheme of GSLC and DSLC

Environment conditions of rules are implemented by adding "environment" variable inputs for every SLC that the variables have influence on, as shown in Figure 5 where environment *e* affects permission *p1…pn*.

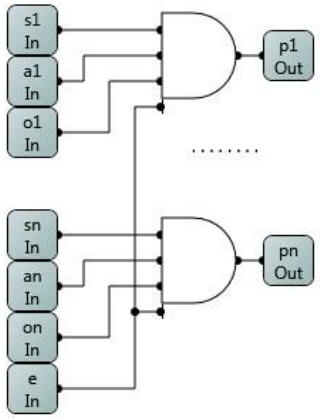

Fig. 5. Environment condition *e* affects permission *p1…pn*

The fault detecting process is described by the following algorithm:

```
If permission_of (AC_rule) = 'grant' /* identify the permission type of the AC_rule by the function permission_of
    add_grant_SLC (AC_rule)
else If permission_of (AC_rule) = 'deny'
    add_deny_SLC (AC_rule)
end
add_grant_SLC (AC_rule)
    add_SLC (GSLC, AC_rule) /* As described in Section III A
    if trigger_SLC (GSLC, subject_retrieve (AC_rule),
    action_retrieve (AC_rule), object_retrieve (AC_rule)) ==
    trigger_SLC (DSLC, subject_retrieve (AC_rule),
    action_retrieve (AC_rule), object_retrieve (AC_rule))
        remove_SLC (AC_rule, GSLC)
end
add_deny_SLC (AC_rule) /* add to DSLC means add a deny rule
    add_SLC (DSLC, AC_rule) /* As described in Section III A
    if trigger_SLC (DSLC, subject_retrieve (AC_rule),
    action_retrieve (AC_rule), object_retrieve (AC_rule)) ==
    trigger_SLC (GSLC, subject_retrieve (AC_rule),
    action_retrieve (AC_rule), object_retrieve (AC_rule))
        remove_SLC (AC_rule, DSLC)
end
```

Function *permission_of*() identifies whether the added rule is for grant or deny permission. Function *add_SLC*() applied the scheme as described in Section III *A* to build a SLC circuit according to the entered *AC_rule*. The *trigger_SLC*() function activates the GSLC and DSLC variables retrieved from the *AC_rule*. Subjects/attributes are retrieved by the *subject_retrieve*() function, actions are retrieved by the *action_retrieve*() function, and objects/attributes are retrieved by the *object_retrieve*() function. Note that for their triviality and implementation dependency, steps of each function are not included in the paper.

Creating privilege inheritance assignments is not as straightforward as the above algorithms, because each new added inheritance assignment might invoke additional grant or deny permissions to the existing ones. For example, in a GSLC the two rules have been implemented:

$(s1 \wedge s2) \wedge a \wedge o = p1_{GSLC}$ (rule 1)

$s3 \rightarrow s2$ (rule 2)

In a DSLC a rule has been implemented:

$s3 \wedge a \wedge o = p1_{DSLC}$ (rule 3)

rule 3 has no fault, because although *s3* inherits *s2*'s privilege, *s3* alone cannot perform *o* on *a* enforced by rule 1 that requires both *s1* and *s2* subject/attribute to be presented for the privilege. Now a new rule

$s3 \rightarrow s1$ (rule 4)

is entered in the GSLC, and the addition causes fault, because it allows *s3* to operate *a* on *o* through rule 2 and rule 1, and conflicts with rule 3 in the DSLC. Thus, additional steps to verify the correctness for added new inheritance assignment are described below:

```
If permission_of (inheritance_assignment) = 'grant' /*
identify the permission type of the inheritance_assignment by
the function permission_of
    add_grant_inheritance (inheritance_assignment)
else If permission_of (inheritance_assignment) = 'deny'
    add_deny_inheritance (inheritance_assignment)
end

add_grant_inheritance (inheritance_assignment)
  add_inheritance (GSLC, inheritance_assignment) /* As
  described in Section III A
    s = inheritance_subject_retrieve
    (inheritance_assignment)
    for all action a in GSLC
        for all object/attribute o in GSLC
            if trigger_SLC (GSLC, s, a, o) = trigger_SLC
            (DSLC, s, a, o)
                remove_ inheritance (GSLC,
                inheritance_assignment)
end
add_deny_inheritance (inheritance_assignment)
  add_inheritance (DSLC, inheritance_assignment) /* As
  described in Section III A
    s = inheritance_subject_retrieve
    (inheritance_assignment)
    for all action a in DSLC
        for all object/attribute o in DSLC
            if trigger_SLC (DSLC, s, a, o) = trigger_SLC
            (GSLC, s, a, o)
                remove_ inheritance (DSLC,
                inheritance_assignment)
end
```

Function *permission_of*() is overloaded for identifing whether the added inheritance assignment is for grant or deny permission. Like the *add_SLC*() function, the *add_inheritance*() function adds SLC in GSLC and DSLC. The *inheritance_subject_retrieve*() function retrieves the inherited subject/attribute of the *inheritance_assignment* (e.g., *s3* of rule 2 and rule 4 above). The *remove_inheritance*() function removes the just-added *inheritance_assignment* SLC. As shown in the algorithm, detecting inheritance faults needs to trigger all actions and objects/attributes, because when one subject/attribute inherits another subject/attribute's privilege, the beneficiary might also acquire privileges through inheritance from the inherited subject/attribute. Such interconnected privilege transferring can only be detected through triggering all the permissions related to the beneficiary subject/attribute.

Other Boolean operators for AC rules can also be used to express rule logics, for example, enforcing XOR logic of rule 5

$(s1 \otimes s2) \wedge a \wedge o = p$ (rule 5)

as illustrated in Figure 6.

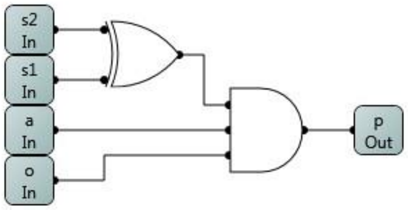

Fig. 6. SLC XOR example

Note that after rule 5 is added to a GSLC, adding inheritance assignments $s3 \rightarrow s1$ and $s3 \rightarrow s2$ will not cause fault as it did in rule 4, because the inheritance assignments do not violate rule 3 assuming it has been implemented in the DSLC.

For conciseness, in the above algorithms, we did not include a redundancy check for the case that the SLC rule has already been added in the SLC. The action for such case is simply to do nothing.

IV. Application

In addition to random rule assignments, ACRLCS can be applied to standard AC models such as Role Based Access

Control (RBAC) [22] and Multi Level Security (MLS) [19] as illustrated in Figure 7 and Figure 8 respectively:

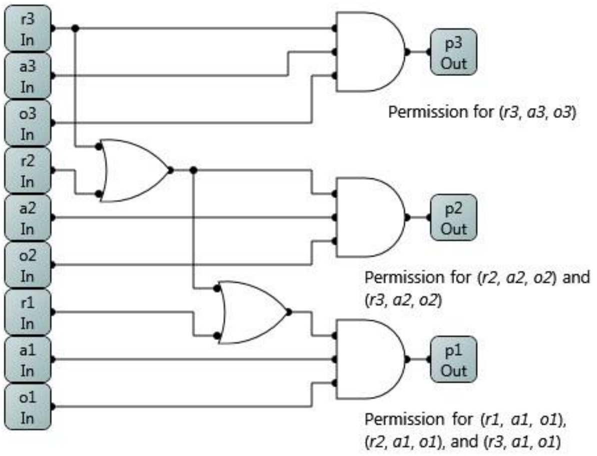
r3 In
a3 In
o3 In
r2 In
a2 In
o2 In
r1 In
a1 In
o1 In
p3 Out
p2 Out
p1 Out
Permission for (r3, a3, o3)
Permission for (r2, a2, o2) and (r3, a2, o2)
Permission for (r1, a1, o1), (r2, a1, o1), and (r3, a1, o1)

Fig. 7. ACRLCS implementation of a simple RBAC example

Figure 7 implements a simple RBAC policy model that role *r1* has privilege (*a1*, *o1*), role *r2* has privilege (*a1*, *o1*) and (*a2*, *o2*), and role r3 has privilege (*a1*, *o1*), (*a2*, *o2*), and (*a3*, *o3*). The RBAC model is equivalent to the following Boolean expressions:

$r3 \wedge a3 \wedge o3= p3$
$(r2 \wedge a2 \wedge o2) \vee (r3 \wedge a2 \wedge o2) = p2$
$(r1 \wedge a1 \wedge o1) \vee (r2 \wedge a1 \wedge o1) \vee (r3 \wedge a1 \wedge o1) = p1$

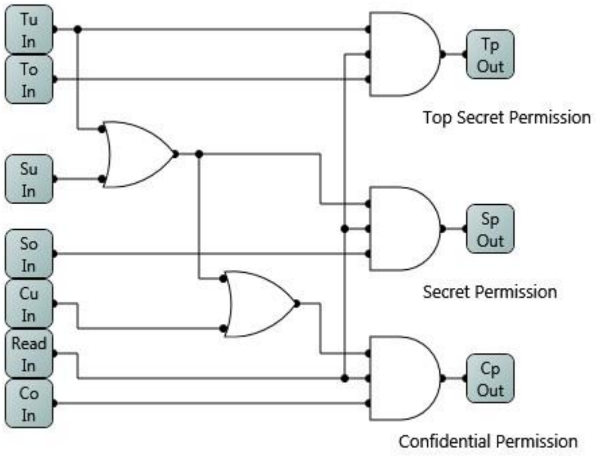

Fig. 8. ACRLCS implementation of a simple MLS example

Figure 8 implements a simple Bell-Lapadula [23] read property of MLS policy model, under which *Top_Secret* rank users can read objects in *Top_Secret*, *Secret*, and *Confidential* ranks; *Secret* rank users can read objects in *Secret* and *Confidential* ranks; and *Confidential* rank users can only read objects in the *Confidential* rank. The MLS model is equivalent to the following Boolean expressions:

$Tu \wedge read \wedge To = Tp$
$(Tu \wedge read \wedge So) \vee (Su \wedge read \wedge So) = Sp$
$(Tu \wedge read \wedge Co) \vee (Su \wedge read \wedge Co) \vee (Cu \wedge read \wedge Co) = Cp$

A standard AC model is implemented in a GSLC by representing the model in rules (e.g., aforementioned RBAC and MLS), and applications of the ACRLCS model can be adding specific constraints outside the enforcement of the model, i.e. exceptional deny access permissions in DSLC. Thus, this provides flexibility in policy specification and checking for the model.

Besides AC models, **security property** faults in AC policies can be implemented for verification by specifying the property constraints in DSLCs. An example of security property faults [14] follows:

- *Cyclic inheritance* (especially in RBAC) allows one subject/attribute to inherit privilege from another subject/attribute and vice versa in a chain of inheritance loop without rendering any access privilege. This fault property can be detected by triggering all subjects/attributes in the GSLC, and checking if any permission output is produced.

- *Privilege escalation* allows subjects/attributes to access prohibited objects/attributes through inheritance of other subjects/attributes with higher privilege. This fault property can be prevented by implementing restricted privileges that are prohibited to be escalated in DSLC.

- *Separation of Duty faults* allows Conflict-of-Interest (COI) subject/attributes to have the same privileges (actions + object/attributes). This property can be prevented by implementing XOR SLC between the COI subjects/attributes under restriction in the GSLC. So, the fault can be detected by triggering the same privilege and the different COI subject/attributes in question in both the GSLC and the DSLC.

As a general logic circuit, memory and state components such as **registers** and **flip-flops** are used for maintaining states and sequence of logic operations. ACRLCS can include logic components for specifications of Historical (or stated) based AC models, such as **Chinese Wall**, **Work Flow**, or **N-person control** models [1]. These models can be verified by adding state variables in a GSLC such that the state variables should trigger subjects/attributes, actions, and objects/attributes from the state, and then be compared to the subjects/attributes, actions, and objects/attributes activated from other states in the DSLC to detect conflicts of states in the model. Due to the limited space for depicting the complexity and details as well as not to deviate from the core scope, discussion of the fault detection techniques for these dynamic models will not be covered in this paper, and hopefully will be presented in the future.

As logical circuits are more flexible than integrating circuits through control logic, ACRLCS can be applied to multi-domain or networked AC environments. A simple application is for the **grant** or **deny overwrite** algorithms for AC policies for multi-domain AC applications such that the higher prioritized policy can overwrite the lower's permission.

Other applications include support layers of prioritized policy structure through layered AND operations as shown in Figure 9, where a verification for an access request (*s*, *a*, *o*) is instantiated in the GSLC, which contains two domains, *GSLC1* and *GSLC2*. *GSLC1 of domain 1* has higher priority than *GSLC2 of domain 2* in deciding permissions. Note that networked or hierarchical permission structures as described above can only be used for detection of specific access assignments in DSLC. To detect fault in networked or priority setting requires construction of DSLC with the image of GSLC but with reverse priority logics.

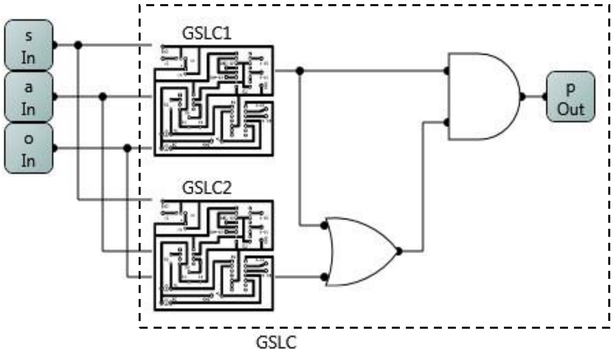

Fig. 9. Example of multiple AC domain using multiple GSLCs

V. Real time Performance

The main objective of this research is to illustrate a real-time mechanism for AC policy fault detection that is different from the other (as mentioned in section II) theorem-proof techniques. The difference can be demonstrated by analyzing computing complexities. To the best of our knowledge, there is no real-time detection technique similar to the one we proposed. Most theorem-proof verification techniques are either **black box**, which consume the whole AC policy for the input of verification process without analyzing individual rules in the policy, or **white box,** which analyze AC rules by inserting assertions in the policy under verification [2]. Both methods requires the AC policy or model (including inserted assertion statements) under verification to be completed before executing the verification process, such that the finished **box** will be translated into a formal model for testing against specified security properties. Doing so, all possible states or propositions formed by rules are examined by the logic algorithm applied. For example, the Symbolic Model Verifier NuSMV is used by many policy verification techniques [24, 10] applying Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) for verifying specified properties against AC rules or models, and generating counterexamples if a fault found. After finding faults, fault fixing is a human process that requires manually going through rules in the faulty policy.

The black/white box methods apply a formal proof technique to compute all possible sequences defined by the states or paths of rules in the AC policy. The worst-case complexity for them is in the order of $|S| \times |A| \times |O| \times |I| \times |E|$ where S is the set of subjects/attributes, A is the set of actions, O is the set of objects/attributes, I is the set of attribute pairs, and E is the set of environment variables. Thus, $|I| = C^{2}_{|S|}$ (inheritance can be assigned between every subject/attribute pair in the worst case scenario). The ACRLCS method for each added rule that triggers that rule circuit takes constant time for computing, but checking inheritance assignment validations in the worst case takes $|I| \times 2$) (2 is for both GSLC and DSLC). Thus, in total $O(|I| \times 2 \times r)$ computing complexity is required, where r is the number of rules in the policy.

For black/white box formal methods, the time required for fixing policy after a fault is found is $O(|S| \times |A| \times |O| \times |I| \times |E| \times r)$ steps assuming in the worst case a fault was found in every rule, and each rule enforces all the variables, due to the fact that each found fault requires checking all rules in the policy to determine which rule is not intended for the policy. ACRLCS under the same assumption (i.e., fault found in each rule) needs $O(|I| \times r \times 2)$ computing steps. Table 1 summarizes the complexity.

TABLE I. Complexity Summary

Complexities	**Methods**	
	Black/White box formal method	***ACRLCS***
Rule building	r	$r \times 2$[a]
Verification or fault detection	$O(|S| \times |A| \times |O| \times |I| \times |E|)$	$O(|I| \times r \times 2)$
Fault fixing	$O(|S| \times |A| \times |O| \times |I| \times |E| \times r)$	$O(|I| \times r \times 2)$

[a] For both GSLC and DSLC

As shown in Table I, the worst-case complexity is not "significantly" different for both complexities, for they all are in the exponential orders, which conforms with [25]'s theory that safety analysis is intractable. It is observed that the major cost of complexity is the number of inheritance relations $|I|$, which contributes the exponential orders. However, empirically, $|I|$, $|A|$, and $|E|$ are usually small compared to $|S|$ and $|O|$ in real-world AC policies. So, for most of the AC policies, the number $|S| \times |O|$ is critical in calculating efficiency for real-world AC applications, and $|S| \times |O|$ is usually a large number for most AC environments. One important fact is that the $O(|S| \times |A| \times |O| \times |I| \times |E| \times r)$ of fault fixing can only be performed manually by inspecting the faulty policy, as stated in Section II; rule conflicts need human judgment to resolve. Thus when comparing with most theorem-proof verification methods, ACRLCS has the advantage for saving fault fixing time by $O(|S| \times |O|)$, which is a significant amount if done manually.

VI. Conclusion

In this paper, we presented the ACRLCS technique for detecting AC rule faults in real time, performing fault detection every time a rule is added into the policy. We demonstrated simulated logic circuits, which are versatile in specifying AC rules formed by Boolean logic expressions operated on variables of AC rules. The variables include subjects/attributes, actions, objects/attributes, environment conditions as well as privilege inheritance relations between subject/attributes. We then explained the algorithm for AC rule fault detection by comparing grant (GSLC) and deny (DSLC) parts of ACRLCS. We showed that in addition to random AC permission rules, ACRLCS is capable of composing standard mandatory AC models such as RBAC and MLS as well as some fundamental security properties. Further, extended applications for multi-domain AC implemented by multiple AC policies are briefly introduced.

Like other formal AC policy verification techniques, theoretically ACRLCS is no exception and requires intractable (exponential) time complexity for the worst case scenario. However, in real world applications, the critical factor of the intractability—the number of hierarchies of privilege inheritance is limited, thus, compared to other model verification or theorem-proof methods, ACRLCS is O(|*S*| × |*O*|) more efficient for fixing policy faults, and the number is significant as most of the fault fixing is done manually.

Some rule construction of SLC such as applying XOR logic, Historical (State) Based AC models, as well as inheritance of objects/attributes was not covered in this paper due to the limited space in covering the details and not to deviate from the main scope of the topic. Exploring these extended features shall be topics for our future research.

We acknowledge Logic Circuit [26] for using their tool for the Figures in this paper.

REFERENCES

[1] V. Hu, D. Ferraiolo, and R. Kuhn, “Assessment of Access Control Systems”, NIST Interagency Report 7376, Gaithersburg, MD, USA, 2006.

[2] V. Hu, R. Kuhn, T. Xie, and J. Hwang, “Model Checking for Verification of Mandatory Access Control Models and Properties”, Int'l Journal of Software Engineering and Knowledge Engineering (IJSEKE) Vol. 21 No. 1., pages 103-127, 2011.

[3] NuSMV. http://nusmv.irst.itc.it/.

[4] N. Zhang, M. D. Ryan, and D. Guelev, “Evaluating Access Control Policies Through Model Checking”, in Proc. Information Security Conference, pages 446–460, 2005.

[5] S. Kikuchi, S. Tsuchiya, M. Adachi, and T. Katsuyama, “Policy Verification and Validation Framework Based on Model Checking Approach”, in Proc. International Conference on Autonomic Computing, pages 1–9, 2007.

[6] A. Schaad, V. Lotz, and K. Sohr, “A model-checking approach to analysing organisational controls in a loan origination process”, in Proc ACM Symposium on Access Control Models and Technologies, pages 139–149, 2006.

[7] K. Jayaraman, V. Ganesh, M. Rinard, and S. Chapin, “Automatic error finding in access-control policies”, in CCS '11 Proceedings of the 18th ACM conference on Computer and communications security, pages 163-174, 2011.

[8] F. Hansen and V. Oleshchuk, “Conformance checking of RBAC policy and its implementation:, in R. Deng, F. Bao, H. Pang, and J. Zhou, editors, Information Security Practice and Experience, volume 3439 of Lecture Notes in Computer Science, pages 144-155, Springer Berlin, Heidelberg, 2005.

[9] K. Jayaraman, V. Ganesh, M. Tripunitara, M. Rinard, and S. Chapin, “Automatic error Finding in access-control policies”, in Proceedings of the 18th ACM conference on Computer and communications security, CCS '11, pages 163-174, New York, NY, USA, 2011.

[10] V. C. Hu, D. R. Kuhn, and T. Xie, “Property verification for generic access control models”, in Proceedings of the 2008 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing - Volume 02, EUC '08, pages 243-250, Washington, DC, USA, 2008.

[11] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz, “Verification and change-impact analysis of access-control policies”, in Proceedings of the 27th international conference on Software engineering, ACM ICSE '05, pages 196-205, New York, NY, USA, 2005.

[12] H. Hu and G. Ahn, “Enabling verification and conformance testing for access control model”, in Proceedings of the 13th ACM symposium on Access control models and technologies, SACMAT '08, pages 195-204, New York, NY, USA, 2008.

[13] S. Jha, N. Li, M. Tripunitara, Q. Wang, and W. Winsborough, “Towards formal verification of role-based access control policies”, IEEE Transactions on Dependable and Secure Computing, volume 5, pages 242-255, 2008.

[14] A. Gouglidis, I. Mavridis, and V. Hu, “Verification of Secure Inter-operation Properties in Multi-domain RBAC”, IEEE Trustworthy Computing Workshop, Gaithersburg, MD, USA, 2013.

[15] Alloy, “A language and tool for relational models”, http://alloy.mit.edu/alloy/.

[16] SPIN, “The SPIN model checker”, http://spinroot.com/spin/.

[17] G. Hughes and T. Bultan, “Automated verification of access control policies using a SAT solver”, Int. J. Softw. Tools Technol. Transf., 10(6), pages 503-520, Oct. 2008.

[18] L. Lamport, “Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers”, Addison-Wesley Professional, 1st edition, 2002.

[19] V. Hu et al, “Attribute Based Access Control Definition and Consideration”, NIST Special Publication 800-162, Gaithersburg, MD, USA, 2013.

[20] V. Hu and K. Scarfone, “Guidelines for Access Control System Evaluation Metrics”, NIST Interagency Report 7874, Gaithersburg, MD, USA, 2012.

[21] V. Hu, D. Ferraiolo, and S. Gavrila, “Attribute Relations Specifications and Constraints Using Attribute Based Mechanism of Policy Machine”, International Journal of Information Assurance and Security (JIAS) Volume 6, Issue 2, 2011.

[22] R. Sandhu and P. Samarati, “Access Control: Principles and Practice”, IEEE Communications, Volume 32, Number 9, September 1994.

[23] D. Bell and La Padula, “Secure computer systems: unified exposition and MULTICS”, Report ESD-TR-75-306, The MITRE Corporation, Bedford, MA, USA, March 1976.

[24] E. Martin and T. Xie, “A fault model and mutation testing of access control policies”, in WWW '07: Proc. of the 16th ACM Intl. conference on World Wide Web, 2007.

[25] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman, “On protection in operating systems”, in SOSP '75: Proc. of the Fifth ACM symposium on Operating systems principles, 1975.

[26] http://www.logiccircuit.org/.