Evaluating the Capability and Performance of Access Control Policy Verification Tools

Ang Li†, Qinghua Li†, Vincent C. Hu§, Jia Di†
†Department of Computer Science and Computer Engineering, University of Arkansas
§Computer Security Division, National Institute of Standards and Technology
†{angli, qinghual, jdi}@uark.edu, §vincent.hu@nist.com

***Abstract*—Access control has been used in many systems such as military systems and business information systems. Access control protects sensitive information based on access control policies. Thus, assuring the correctness of policies is important. For this purpose, many access control policy verification (ACPV) tools have been proposed to check the correctness of policies. Since these tools have been designed by different mechanisms, they have different capabilities and performances. However, there lacks a set of standard approaches for evaluating them. Consequently, it is difficult for users to identify an appropriate tool for verifying their security policies. In this paper, we make an initial step towards building standard approaches for evaluating the capability and performance of ACPV tools. Specifically, we propose a set of reference metrics for analytically evaluating, as well as sets of oracles and test cases for empirically checking the run-time capability and performance of ACPV tools. To demonstrate, we apply these metrics, oracles and test cases on existing ACPV tools.**

Index Terms*—*Access control, policy, verification

I. Introduction

Access control dominates which principals (e.g., process and users) have access to which resources (e.g., memory, files and database). It has been used in many applications to protect the nation's critical information infrastructures for military systems, power grids, banking systems, etc. Access control protects sensitive information and resources based on access control policies, which typically consist of a set of rules described in terms of subjects, objects, actions and environment conditions of the protected system.

Since the protection provided by access control is based on the policy, it is vital to create correct policies that meet the protected system's requirements. However, with the increasing complexity of the protected system and large number of rules in a policy, it becomes a challenging task to write policies without faults, such as inconsistency between policy rules. Faulty policies not only leak sensitive information to unauthorized parties but also disable authorized access to information. Thus, policy verification is a crucial technique to assure the correctness of access control policies.

Many verification tools have been proposed, such as ACPT [1], Margrave [2], Mohawk [3], Mutaver [4], as well as many other theoretical approaches [5]-[8], [15], [18]-[21]. Since existing tools have been designed by different mechanisms, they have different capabilities and performances. However, there lacks a set of standard approaches for evaluating them. Thus, it is difficult for access control administrators to find an appropriate tool to verify their access control policies. For instance, which tool is suitable to verify Multi-Level Security (MLS) policies? Which tool for detecting inconsistency in policies? And which for verifying *separation of duty*? For the performance (e.g., running time) evaluation, it is also difficult to determine an appropriate tool that suits the policy with the available resources and time for verification.

Little work has been done on evaluating access control policy verification (ACPV) tools. Muhammad and Riaz [5] surveyed access control policy validation mechanisms. They classified different verification mechanisms based on the techniques used, but provided limited evaluation on functionalities. Also, they did not consider the performance of tools. Thus, their work can only provide limited guidelines for users to determine an appropriate tool based on their need.

To bridge this gap, in this paper, we make an initial step towards building standard approaches for evaluating the capability and performance of ACPV tools. Specifically, we propose a framework that comprises sets of metrics, oracles and test cases with demonstrations. Our contribution is threefold:

1) We propose a set of reference metrics to analytically evaluate the capabilities of ACPV tools, and then conduct a comprehensive analysis on existing tools based on the metrics.
2) We design a set of oracles to empirically test the run-time functions of ACPV tools (i.e., whether they can detect the faults embedded in the oracles). These oracles are designed in natural language and XACML (eXtensible Access Control Markup Language) [6], which is the de facto standard for specifying access control policies. We use the oracles to test two tools, ACPT and Margrave.
3) We design a set of test cases to empirically test the run-time performance of ACPV tools. These test cases are used not only for evaluating the performance but also for exploring the factors that affect the performance. We then apply them to ACPT and Margrave. The above oracles and test cases are available to the public.

The rest of the paper is organized as follows. Section II provides background information on access control and an overview of our evaluation framework. Section III presents the reference metrics and the analytical study on the capability of existing ACPV tools. Section IV describes our oracle and test case design. Section V presents the evaluation results of ACPT and Margrave. Section VI concludes the paper.

II. Overview

Access Control Access control provides security protection by regulating which principals have access to which resources based on access control policies. For example, in a university, a student can access the grading system to view his grade for any course, but he is not allowed to view other students' grades or change his own grades.

There are several well-known access control concepts and models [7], including Discretionary Access Control (DAC), Mandatory Access Control (MAC), and Attribute-Based Access Control (ABAC). In DAC, the owner of resources has the privilege to determine which subject can access which resource and to delegate access privilege to other users. In contrast, MAC restricts access to resources by the system (administrator), rather than the resource owner. For instance, the *level* of security of a subject determines which object he can access. ABAC is a way to restrict access based on subjects' attributes [8]. For example, using roles as attributes (i.e., RABC [8]), a student is not allowed to change any other student's grade unless the student is assigned to the role of *teaching assistant*.

There are three fundamental types of access control models according to their properties: static, dynamic, and historical [9]. Static policies regulate access rights by static system states including attributes, rules, system environments, etc. Popular access control models of this type include ABAC, and MLS. Dynamic models regulate access rights by dynamic system states, e.g., system events. A representative access control model with these properties is N-Person Control. For example, a file cannot be accessed by more than ten users at the same time. Historical models regulate access rights based on historical access states and predefined series of events. Popular access control models of this type include Workflow and Chinese Wall policies. The metrics, oracles and test cases proposed in this paper cover all these types of access control models.

ACPV is a crucial method for assuring the correctness of access control policies. In the verification process, a policy is shown correct by checking a set of high-level security properties that the system should satisfy against. For instance, "*any student should not be allowed to write the grade.*" Many ACPV techniques have been proposed, such as model checking [10], mutation testing [4], and other techniques [5]. Model checking is an important technique for ACPV, and many tools have been implemented based on it. For example, Mankai et al. [10] proposed a framework to detect inconsistency and incompleteness based on a standard logic model checking tool. These tools use a specific language such as XACML [6] for policy specifications. A more detailed survey of these techniques can be found in [5].

Figure 1 illustrates our ACPV evaluation framework. It evaluates an ACPV tool in three ways. First, the tool is statically analyzed (e.g., based on the tool's documentation) to find out if it has the capabilities described by a set of reference metrics. Second, the tool's run-time capabilities are tested by running a set of functional oracles on it. Third, the tool's run-time performance is tested by running a set of test cases. Evaluators can apply this framework to identify a proper ACPV tool that meets their verification requirements (e.g., properties to verify, scale of policy, available computing resources, etc.). If no tool satisfies the requirement (e.g., no tool can verify a particular property), it means a new tool or improvement of existing tools is needed. And, the framework will provide the directions for the design or improvement. The metrics, functional oracles, and test cases and how to use them are described in the following sections.

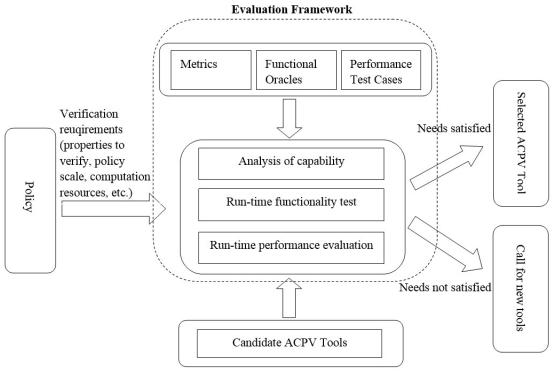


Fig. 1. Evaluation framework

III. Analyzing the Capabilities of ACPV tools

In this section, we propose metrics containing the important capabilities desired from ACPV tools, and then analyze them on available tools.

A. Metrics

The metrics are divided into five categories.

The first category contains general security properties that every ACPV tool should be able to verify.

- ***Safety*** checks if the access control policy leaks access permission to unauthorized or unintended principals.
- ***Separation of Duty (SoD)*** [11] prevents error and fraud by ensuring that no conflict-of-interest assignments are assigned to a single subject. For example, in a financial accounting system, one employee that is responsible for depositing cash should not also be responsible for reconciling bank statements.
- ***Completeness*** assures that each access request should be either accepted or denied by the access control policy.
- ***Liveness*** [9] guarantees that there is no deadlock in which the system will wait forever for system events, and there is no livelock in which access control model repeatedly executes the same operation forever.
- ***Model-specific properties*** are security properties that specifically supported by various access control models in addition to the above generic properties. We mainly focus on the properties of RBAC models for our experiment, because RBAC has been widely used in the field. RBAC's specific properties include *availability* (e.g., if a user is always a member of a certain role), *reachability* (e.g., if a user can be assigned to a certain role), *dead roles* (i.e., roles that cannot be assigned to any user), *role-role containment* (e.g., if every member of one role is also a member of another role), *information flow* (e.g., if information can flow from one object to another object), and *weakest precondition* (e.g., what is the minimal set of initial roles that a user must have to be added to roles in the goal) [12].

The second category of metrics focuses on the policy rules, which includes:

- ***Inconsistency detection*** - Due to the combination of multiple policies into one, it is possible that inconsistency between policy decisions occurs. For example, student *A* cannot change the grade in one policy, but the right is granted by other combined policy, and a conflict occurs.

- ***Real-time response*** - Users can get response in real time if the verification tool can detect fault when the fault-causing rule is added to the policy, rather than after all the rules of the policy has completed.
- ***Detection of redundant rules*** - An access control rule is redundant if removing the rule does not change the behavior of the policy.

The third category of metrics centers on the quality of security properties used to verify a policy and includes *coverage and confinement check* for the reason that if the verification properties do not thoroughly cover all rule conditions or all possible values, the policy is still not fault-proof. Coverage and confinement check makes sure that the rules in a policy are completely covered by the evaluation properties, and guarantees that no exceptional access rights are granted.

Besides assuring the quality of policy and evaluation properties, it is important to perform conformance testing. Conformance testing is applied to validate compliance of implementation of the policy models. In the test oracle, access requests are the test inputs and authorization decisions are the outputs. A test input is executed as a request is evaluated against the policy under test. Policy authors can inspect test results against the test oracles to check whether they are as expected. To facilitate this test, *oracle generation* is another desired capability of ACPV tools.

The last category of capabilities is the *support of access control models* such as the previously described static, dynamic and historical models.

B. ACPV Tools

We target currently available ACPV tools to demonstrate test metrics.

ACPT [13] is developed by NIST and North Carolina State University and enhanced by the University of Arkansas. It provides both static and dynamic verifications to ensure policy correctness. Static verification checks whether user-specified properties are satisfied by given policy models through the SMV (Symbolic Model Verification) model checker. The assurance provided by static verification relies on the quality of the specified properties. ACPT also conducts dynamic verification by automatically generating and executing conformance oracles.

Margrave [2] developed by Fisler *et al.* verifies access control policies in XACML and EPAL languages by converting them into multi-terminal binary decision diagram (MTBDDs). It specifies access constraint properties in the Scheme programming language to decide if counterexample is detected.

Other tools include **ACRLCS** [14], **ACPEG** [15], **Mutaver** [4], **Mohawk** [3], **VAC** [16], [17], and **RBAC-PAT** [12]. Due to the space limitation, details of them can be found in the references. Note that our metrics are general and they can also be used to analyze other tools.

C. Results

The analysis results are shown in Table I and Table II.

The result shows that all the tools under the test can check the *safety* property. However, none of them is able to check the *liveness* property, and only ACRLCS and Mohawk can check *SoD* and the *completeness* property. The most supported *model-specific* properties are *reachability* and *role-role containment* of RBAC and ARBAC [18] policies. Only ACPT supports MLS and Workflow policies.

For the policy rule properties, none of the tools can detect redundant rules. All tools under test deliver verification results after all the policy rules are complete, and only ACRLCS can detect inconsistency in real time. Only ACPT and Mutaver possess the *coverage and confinement* capability. In addition, all the three access control models (static, dynamic and historical) can be supported by ACPT, ACRLCS, Mutaver, and Mohawk, and only ACPT can generate oracles for *conformance testing*.

Since none of the tools supports all the capabilities of the metrics, they can be improved. Or, new tools should be developed. Thus, users should find the tool that best fits their requirements. For example, if *safety* and *SoD* properties are the main properties to be checked, ACRLCS and Mohawk may be considered.

IV. Oracle and Test Case Design

In this section, we describe the design of functional oracles and performance test cases. Functional oracles are to empirically test the run-time functionality of ACPV tools, and performance test cases are for evaluating their run-time performance and exploring the factors that affect the performance. Our oracles and test cases are general and applicable to any ACPV tool. All the oracles and test cases are available in the ACPT tool at [1].

A. Access Control Policies Used

The oracles and test cases are designed based on four representing policies in the literature: Continue [19], University [12], Hospital [12] and Bank [20]. Continue is a web-based software system for conference paper submission and review, which is used by many conferences. The University policy is an ARBAC policy including rules for student roles (e.g., graduate student) and employee roles (e.g., professor) assignments; the role hierarchy includes the relationships between various roles which simulate real-world administration conditions. Hospital policy contains RABC and ARBAC policies. It restricts resource access of a hospital, e.g., doctors' accesses to patients' records. The bank policy is a RBAC policy designed for a large bank system comprised of several departments, and the policy can be extended into an ARBAC policy [3].

In addition, we develop a multi-level policy and a workflow policy. The multi-level policy is designed according to a university file system use scenario, where we assign different ranks to students, teaching assistants, as well as some files, and then apply the Bell-LaPadula property [7] to regulate accesses to the files. The workflow policy is created for a process handling purchase orders, where different roles carry out their responsibility in three steps.

B. Functional Oracles

Here, we design policy oracles with embedded faults to test if an ACPV tool can detect the faults in a policy. In this paper, we consider three types of common faults: *access conflict*, *no object* and *undecided* due to their importance. *Access conflict* happens when a policy renders one subject to have conflicting access permissions to the same object. For example, subject *A* is in group *G1* which is disallowed to access object *o1*, but

TABLE I
THE CAPABILITIES OF ACCESS CONTROL VERIFICATION TOOLS (A)

Tools	Security Properties				
	Safety	**SoD**	**Completeness**	**Liveness**	**Model-specific Properties**
ACPT [1], [13]	Yes	No	No	No	RBAC (reachability, role-role containment), MLS, Workflow
Margrave [2]	Yes	No	No	No	RBAC (role-role containment)
ACRLCS [14]	Yes	Yes	No	No	ARBAC (reachability, role-role containment)
ACPEG [15]	Yes	No	No	No	ARBAC(reachability)
Mutaver [4]	Yes	No	No	No	-
VAC [16], [17]	Yes	No	No	No	ARBAC(reachability)
Mohawk [3]	Yes	Yes	Yes	No	ARBAC(reachability)
RBAC-PAT [12]	Yes	No	No	No	RBAC and ARBAC(reachability, role-role containment, availability, dead roles, weakest precondition, information flow)

TABLE II
THE CAPABILITIES OF ACCESS CONTROL VERIFICATION TOOLS (B)

Tools	Policy Related			Quality of Properties	Oracle Generation	Support of AC Models
	Inconsistency	**Real-time Response**	**Detecting Redundant Rules**	**Coverage and Confinement**		
ACPT [1], [13]	Yes	No	No	Yes	Yes	Static, Dynamic, Historical
Margrave [2]	Yes	No	No	No	No	Static
ACRLCS [14]	Yes	Yes	No	No	No	Static, Dynamic, Historical
ACPEG [15]	No	No	No	No	No	Static
Mutaver [4]	No	No	No	Yes	No	Static, Dynamic, Historical
VAC [16], [17]	No	No	No	No	No	Static
Mohawk [3]	No	No	No	No	No	Static, Dynamic, Historical
RBAC-PAT [12]	Yes	No	No	No	No	Static

TABLE III
CHARACTERISTICS OF ORACLES FOR DIFFERENT POLICIES

Policy	# Subjects	# Objects	# Rules	AC Model
Continue	4	6	12	static
University	5	2	15	static
Hospital	4	9	39	static
Bank	10	6	60	dynamic
Multi-level	3	2	8	static
Workflow	2	1	3	historical

group *G2*, which subject *A* is also a member of, is allowed to access object *o1*. *No object* happens if there are objects not protected by any rule in the policy. *Undecided* is the fault that a subject's access permission to an object cannot be decided by the policy.

For each of the above faults, we design an oracle by selecting rules from the original policy. Characteristics of the oracles are shown in Table III. We then embed the three common faults into the basic oracle by modifying rules. Verification against these oracles in tools demonstrates if they can detect the embedded faults.

Figure 2 shows the example oracle with access conflict fault embedded for the Continue policy. To create such fault, we add a rule into rules for *Reviewer Policy*, which allows *pcmember*_3 to read *Review*_3 even though the *ReviewStatus* is *Not_Submitted*. Obviously, this rule conflicts with the last rule in *PCmember Policy*, which prohibits any *pcmember* from reading *Review*_3 when the *ReviewStatus* is *Not_Submitted*.

In order to make the oracles easy to understand and use, we create all oracles in two formats: natural language and XACML. XACML is used because many access control policies are described in this language and many ACPV tools take XACML-format policy as input. Thus, the format can be directly used by those tools. Typically, an XACML policy consists of three levels, PolicySet, Policy and Rule. The PolicySet contains one or more Policy elements, and the Policy contains one or more Rule elements. In XACML, there are four main elements for the PolicySet, Policy and Rule which are Subject, Resource, Action and Environment. In addition, XACML provides the Target element to check the applicability according to a set of conditions. Further, the Condition element only exists in rules, which applies an array of functions to compare two or more attribute elements. Figure 3 shows the XACML format of one rule in the Figure 2 oracle.

C. Performance Test Cases

Here, we choose *University Policy* and *Hospital Policy* to develop test cases for performance evaluations. We consider three important factors that may affect the performance of ACPV tools, which are number of subjects, number of objects and number of rules, and change these factors in different test cases to explore their effects.

For each University or Hospital policy and each considered factor, we design a group of test cases by changing one factor but keeping the other two factors unchanged. For example, to test by the number of rules in the test cases of University Policy, we keep 15 subjects and 8 objects unchanged but increase the number of rules from 15 to 120. Test cases for other factors and policies are designed in similar ways.

The running time may be different for the same policy when different properties are checked. The reason is that a property is usually verified after a certain rule of the policy is checked. For convenience, we say this rule matches the property. To evaluate this effect, we consider the location of the rule that matches the verified property. We use different properties that match different rules of a policy. For example, in the first test case of University policy shown in Table V, we specify three properties which match the second rule, the tenth rule and the fourteenth rule respectively.

V. EVALUATING THE RUN-TIME FUNCTIONALITY AND PERFORMANCE OF ACPV TOOLS

In this section, we introduce the experiment methodology, and show the evaluation results on the run-time functionality and performance of the ACPV tools. We also explore which factors affect the performance.

A. Experiment Methodology

We choose ACPT and Margrave for this experiment, because these two representative ACPV tools use the two most popular formal verification methods: model checking and

Roles: Reviewer, Pcmember
Users: Reviewer (pcmember_1, pcmember_2, pcmember_3)
Action: Read, MakeReview, Update
ReviewStatus: Submitted, Not_Submitted
Inheritance: Beneficiary-> Reviewer (pcmember_1, pcmember_2, pcmember_3),
Inherited Values->PCmember
Resource: Paper (Paper_1, Paper_2, Paper_3) Review (Review_1, Review_2, Review_3)
Rules (#12):
PCmember Policy:
(pcmember, Read, Paper_1)->Permit
(pcmember, Read, Paper_2)->Permit
(pcmember, Read, Paper_3) ->Permit
(pcmember, ReviewStatus: Not_Submitted, Read, Review_1) ->Deny
(pcmember, ReviewStatus: Not_Submitted, Read, Review_2) ->Deny
(pcmember, ReviewStatus: Not_Submitted, Read, Review_3) ->Deny
Reviewer Policy:
(pcmember_1, ReviewStatus: Not_Submitted, Update, Review_1) ->Permit
(pcmember_2, ReviewStatus: Not_Submitted, Update, Review_2) ->Permit
(pcmember_3, ReviewStatus: Not_Submitted, Update, Review_3) ->Permit
(pcmember_3, ReviewStatus: Not_Submitted, Read, Review_3) ->Permit
(pcmember_1, ReviewStatus: Not_Submitted, MakeReview, Paper_1) ->Permit
(pcmember_2, ReviewStatus: Not_Submitted, MakeReview, Paper_2) ->Permit
(pcmember_3, ReviewStatus: Not_Submitted, MakeReview, Paper_3) ->Permit

Fig. 2. Continue test case in Natural Language

```
<Rule Effect="Permit" RuleId="rule_1">
<Target>
<Subjects>
<Subject>
<SubjectMatch MatchId="urn: oasis: names:tc:xacml:1.0:function:string-equal">
<AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">pcmember
</AttributeValue>
<SubjectAttributeDesignator AttributeId="PCmember"
DataType=http://www.w3.org/2001/XMLSchema#string
SubjectCategory="urn:oasis:names:tc:xacml:1.0:subject-category:access-subject"/>
</SubjectMatch>
</Subject>
</Subjects>
<Resources>
<Resource>
<ResourceMatch MatchId="urn: oasis:names:tc:xacml:1.0:function:string-equal">
<AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">paper_1
</AttributeValue>
<ResourceAttributeDesignator AttributeId="Paper"
DataType="http://www.w3.org/2001/XMLSchema#string"/>
</ResourceMatch>
</Resource>
</Resources>
<Actions>
<Action>
<ActionMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
<AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">Read
</AttributeValue>
<ActionAttributeDesignator AttributeId="Action"
DataType="http://www.w3.org/2001/XMLSchema#string"/> </ActionMatch>
</Action>
</Actions>
</Target>
</Rule>
```

Fig. 3. Continue test case in XACML

multiple binary tree respectively. We obtain the latest version of ACPT from NIST, and Margrave Version 3.0 is installed based on DrRacket Version 5.0.1. For ACPT, the test oracles are input through the graphical user interface provided by the tool. For Margrave, we manually create the XACML oracles for the input. All experiments are performed on a laptop running Windows 7 with an Intel i7-4610M CPU and 8 GB of RAM.

B. Run-time Functionality Test

We run the oracles designed based on the Continue Policy, University Policy, Hospital Policy and Bank Policy on ACPT and Margave to observe if the two tools are capable of detecting the faults in the oracles. The results are showing in Table IV. ACPT can successfully detect all three faults in these policies, but Margrave fail to detect the faults in Bank Policy and Continue Policy. The reason of the failure is that the Bank policy is a historical type of policy model, which is not supported by Margrave, and Margrave does not support XACML's <*condition*> element in Continue Policy. Such results can provide guidance for users to evaluate the capability of different tools and choose the right tool for their need.

TABLE IV
TESTED FUNCTIONALITY OF ACPT AND MARGRAVE

	ACPT	**Margrave**
Hospital Policy (Access Conflict fault)	Yes	Yes
Hospital Policy (No Object fault)	Yes	Yes
Hospital Policy (Undecided fault)	Yes	Yes
University Policy (Access Conflict fault)	Yes	Yes
University Policy (No Object fault)	Yes	Yes
University Policy (Undecided fault)	Yes	Yes
Bank Policy (Access Conflict fault)	Yes	No
Bank Policy (No Object fault)	Yes	No
Bank Policy (Undecided fault)	Yes	No
Continue Policy (Access Conflict fault)	Yes	No
Continue Policy (No Object fault)	Yes	No
Continue Policy (Undecided fault)	Yes	No
Static Model	Yes	Yes
Dynamic Model	Yes	No
Historical Model	Yes	No

C. Run-time Performance Test

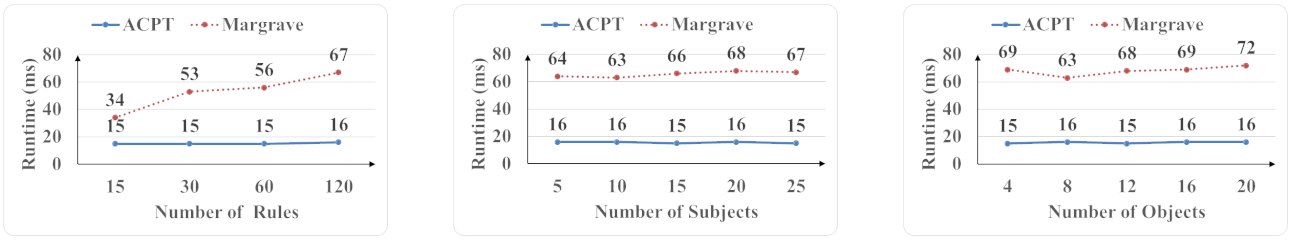
To evaluate the performance of ACPT and Margave, we measure the absolute runtime consumed to perform verification functions on designed test cases. For each test case, we specify three properties to verify, each property are verified ten times, and then the average runtime of verifying a property is calculated. The three properties match rules in different locations of the test case. For both University Policy and Hospital Policy, the three properties (#1, #2, #3) match the rules located in the first part, middle part and last part of a test case. For example, in the first University Policy test case, the three properties match the second rule, the tenth rule, and the fourteenth rule respectively.

Table V shows that ACPT and Margrave have different performance characteristics. With the increasing complexity of each test case, ACPT keeps a quite stable running time at around 16ms. In contrast, the running time of Margrave increases dramatically with growing scale and complexity of policies. For instance, Margrave needs 36ms to verify the first property in the first test case of University Policy, and the time doubles to 72ms for verifying the same property in the fourth test case with a larger scale.

D. Factors Affecting Verification Performance

This set of experiment aims to explore the effect of different factors on the performance of ACPT and Margrave based on the test cases designed in Section IV-C. The location of the rule that matches the verified property does not affect ACPT as shown in Table V. But it affects the performance of Margrave. For example, the running time for verifying the first property and the third property of the first test case of University Policy increases from 36ms to 48ms.

The results for the other three factors (number of subjects, number of objects, and number of rules) are shown in Figure 4. In spite of changing factors, ACPT's running time does not change much. For Margrave, the number of rules affects the performance significantly.



(a) Number of Rules (b) Number of Subjects (c) Number of Objects

Fig. 4. Effect of different factors on verification performance (University Policy). By default, there are 15 subjects, 8 objects, and 120 rules.

TABLE V
PERFORMANCE OF ACPT AND MARGRAVE

Policy	Test Case	Property	ACPT	Margrave
University	University Policy (15 rules, 5 subjects, 2 objects)	#1 property	15ms	36ms
		#2 property	16ms	45ms
		#3 property	16ms	48ms
	University Policy (30 rules, 10 subjects, 2 objects)	#1 property	16ms	53ms
		#2 property	15ms	62ms
		#3 property	16ms	68ms
	University Policy (60 rules, 15 subjects, 3 objects)	#1 property	15ms	55ms
		#2 property	16ms	64ms
		#3 property	16ms	78ms
	University Policy (120 rules, 30 subjects, 3 objects)	#1 property	15ms	72ms
		#2 property	16ms	84ms
		#3 property	16ms	94ms
Hospital	Hospital Policy (39 rules, 4 subjects, 9 objects)	#1 property	15ms	53ms
		#2 property	16ms	73ms
		#3 property	16ms	64ms
	Hospital Policy (78 rules, 8 subjects, 9 objects)	#1 property	15ms	75ms
		#2 property	15ms	76ms
		#3 property	16ms	92ms
	Hospital Policy (165 rules, 13 subjects, 12 objects)	#1 property	15ms	84ms
		#2 property	15ms	86ms
		#3 property	15ms	100ms
	Hospital Policy (330 rules, 26 subjects, 12 objects)	#1 property	15ms	100ms
		#2 property	16ms	96ms
		#3 property	17ms	103ms

VI. CONCLUSION AND FUTURE WORK

In this paper, we proposed metrics to analytically evaluate the capabilities of ACPV tools, and designed oracles and test cases in both natural language and XACML formats to verify the functionality and evaluate the performance in run time. Based on these metrics, oracles and test cases, we demonstrated the evaluations of capabilities and performances of existing ACPV tools, and explored the effects of different factors on the performance of verification. We found that some important capabilities are lacking from existing tools. The proposed techniques and obtained results can help access control policy administrators to select the right ACPV tool for their requirements.

This work intends to make an initial step towards developing standard approaches for evaluating ACPV tools. Future directions include enriching the set of reference metrics (e.g., through considering the properties of emerging access control models and ACPV tools), designing more robust oracles that is capable of evaluating more ACPV mechanisms, and designing large-scale test cases with many rules, subjects, and objects.

ACKNOWLEDGMENT

This work is supported in part by NIST Grant No. 60NANB14D188.

REFERENCES

[1] "ACPT," http://csrc.nist.gov/groups/SNS/acpt/acpt-beta.html.

[2] 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 (ICSE)*, 2005, pp. 196–205.

[3] 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)*, 2011, pp. 163–174.

[4] E. Martin, J. Hwang, T. Xie, and V. Hu, "Assessing quality of policy properties in verification of access control policies," in *Annual Computer Security Applications Conference (ACSAC)*, 2008, pp. 163–172.

[5] M. Aqib and R. A. Shaikh, "Analysis and comparison of access control policies validation mechanisms," *International Journal of Computer Network and Information Security (IJCNIS)*, vol. 7, no. 1, p. 54, 2014.

[6] T. Moses *et al.*, "Extensible access control markup language (xacml) version 2.0," *Oasis Standard*, vol. 200502, 2005.

[7] V. C. Hu, D. Ferraiolo, and D. R. Kuhn, *Assessment of access control systems*. NIST Interagency/Internal Report (NISTIR)-7316, 2006.

[8] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli, "Proposed nist standard for role-based access control," *ACM Transactions on Information and System Security (TISSEC)*, vol. 4, no. 3, pp. 224–274, 2001.

[9] V. C. Hu, D. R. Kuhn, T. Xie, and J. Hwang, "Model checking for verification of mandatory access control models and properties," *International Journal of Software Engineering and Knowledge Engineering*, vol. 21, no. 01, pp. 103–127, 2011.

[10] M. Mankai and L. Logrippo, "Access control policies: Modeling and validation," in *5th Nouvelles Technologies de la Répartition Conference (NOTERE)*, 2005, pp. 85–91.

[11] R. A. Botha and J. H. P. Eloff, "Separation of duties for access control enforcement in workflow environments," *IBM Systems Journal*, vol. 40, no. 3, pp. 666–682, 2001.

[12] M. I. Gofman, R. Luo, A. C. Solomon, Y. Zhang, P. Yang, and S. D. Stoller, "Rbac-pat: A policy analysis tool for role based access control," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2009, pp. 46–49.

[13] J. Hwang, T. Xie, V. C. Hu, and M. Altunay, "Acpt: A tool for modeling and verifying access control policies." in *IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY)*, 2010, pp. 40–43.

[14] V. C. Hu and K. Scarfone, "Real-time access control rule fault detection using a simulated logic circuit," in *International Conference on Social Computing (SocialCom)*, 2013, pp. 494–501.

[15] N. Zhang, M. Ryan, and D. P. Guelev, "Evaluating access control policies through model checking," in *Information Security*. Springer, 2005, pp. 446–460.

[16] A. L. Ferrara, P. Madhusudan, and G. Parlato, "Policy analysis for self-administrated role-based access control," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2013, pp. 432–447.

[17] ——, "Security analysis of access control through program verification," in *Proceedings of the 25th IEEE Computer Security Foundations Symposium (CSF)*, 2012.

[18] R. Sandhu, V. Bhamidipati, and Q. Munawer, "The arbac97 model for role-based administration of roles," *ACM Transactions on Information and System Security (TISSEC)*, vol. 2, no. 1, pp. 105–135, 1999.

[19] S. Krishnamurthi, "The continue server (or, how i administered padl 2002 and 2003)," in *Practical aspects of declarative languages*. Springer, 2003, pp. 2–16.

[20] K. Jayaraman, V. Ganesh, M. Tripunitara, M. C. Rinard, and S. J. Chapin, "Arbac policy for a large multi-national bank," *arXiv preprint arXiv:1110.2849*, 2011.