A Decentralized Solution for Combinatorial Testing of Access Control Engine

Said Daoudagh[1,2], Francesca Lonetti[1] and Eda Marchetti[1]
[1]*Istituto di Scienza e Tecnologie dell'Informazione "Alessandro Faedo", CNR, Pisa, Italy*
[2]*University of Pisa, Pisa, Italy*

Keywords: Access Control Systems, Web Service, Testing.

Abstract: In distributed environments, information security is a key factor and access control is an important means to guarantee confidentiality of sensitive and valuable data. In this paper, we introduce a new decentralized framework for testing of XACML-based access control engines. The proposed framework is composed of different web services and provides the following functionalities: i) generation of test cases based on combinatorial testing strategies; ii) decentralized oracle that associates the expected result to a given test case, i.e. an XACML request; and finally, iii) a GUI for interacting with the framework and providing some analysis about the expected results. A first validation confirms the efficiency of the proposed approach.

1 INTRODUCTION

Nowadays, the management of (personal) data becomes extremely important in many distributed environments, especially when it involves data that are subject to different regulations depending on the context in which they are stored or accessed.

A key means for guaranteeing confidentiality and security of data is represented by access control mechanisms, which grant or deny access to the resources according to subjects attributes and specific environment conditions. A standard language for specifying policies ruling these accesses is the eXtensible Access Control Markup Language (XACML) (OASIS, 2013), an XML-based standard language proposed by OASIS. It relies on the Policy Decision Point (PDP) evaluation engine in order to grant or deny the access based on the defined XACML policies. Testing of this component is very important to assure its robustness and accuracy and avoid security flaws of the access control system. Testing of PDP usually consists of probing the PDP with a set of XACML requests and checking its responses against the expected decisions.

Many approaches exist in literature using combinatorial techniques in order to generate such sets of XACML requests (Bertolino et al., 2013; Martin and Xie, 2006). These techniques have been proven to be effective in revealing faults of the PDP and improving its effectiveness in terms of mutation score or code coverage (Martin et al., 2006; Bertolino et al., 2014). As well known in literature, the main drawback of these combinatorial approaches is the high number of derived combinations that prevent their use in real contexts. Specifically, the execution of a large set of combinatorial requests represents a challenging task, mainly due to the oracle problem: determining the correct result that should be expected from the PDP for each test request.

Existing XACML-based test generation techniques leverage combinatorial methods such as AETG (Cohen et al., 1997), able to support n-way coverage of inputs, and to reduce the total number of combinations. Even though they allow to generate less test cases and are proven to be effective in terms of fault detection, the number of derived test cases could remain unmanageable when applied to XACML policies due to the generally high number of policy attributes.

In general, test suite parallelization can be adopted to speed-up tests execution. However, this has a limited usage in practice because of the effort required to deal with concurrent issues (Candido et al., 2017). The advancements on cloud and parallel computing solutions offer unlimited storage as well as virtualized resources and shared infrastructures. These can help to eliminate required computational resources as well as to reduce the execution time of large test suites in a cost-effective manner. To overcome the computational cost of combinatorial testing, recent proposals

Daoudagh, S., Lonetti, F. and Marchetti, E.
A Decentralized Solution for Combinatorial Testing of Access Control Engine.
DOI: 10.5220/0007379401260135
In *Proceedings of the 5th International Conference on Information Systems Security and Privacy (ICISSP 2019)*, pages 126-135
ISBN: 978-989-758-359-9
Copyright © 2019 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

(Tsai et al., 2015; Tsai and Qi, 2016) leverage cloud environments to: partition the testing tasks; allocate these testing tasks to different processors in the cloud platform for test execution; and then collect results.

Our proposal goes in this direction and provides an efficient decentralized and costly effective framework for the overall testing process of the access control evaluation engine [1].

The main contributions of our proposal deal with:

- A general solution able to leverage parallel computational resources in order to use all the power of combinatorial approaches for the generation of XACML requests. Specifically, our solution allows to execute all XACML policy attributes combinations without the need of applying test suites reduction techniques for cost saving purposes;
- A decentralized framework for the PDP oracle problem. Well-known approaches for automated XACML oracle derivation rely on voting mechanisms such as that of (Li et al., 2008). Specifically, the authors of (Li et al., 2008) propose to implement a PDP automated oracle through voting, i.e. collect responses of more than one PDP engine for the same request and choose as correct decision value the most frequent one. This approach has a high computation and implementation cost. Our decentralized framework is able to reduce this cost by leveraging parallel resources. In literature, an alternative solution for automated PDP testing oracle derivation is XACMET (Bertolino et al., 2018). It represents a more complex model based approach that requires a formal model of the XACML policy that could be not available in real contexts.

The proposed framework allows cost saving of the overall testing process by reducing the time needed of all the main phases of the PDP testing that are: test cases generation, test case execution and oracle derivation. Moreover, efficient testing of all possible combinations of the policy values might provide reasonably high assurance of the PDP.

It is out of scope of the paper to compare the cost reduction of a parallel solution with respect to that of a not distributed one, neither to show the effectiveness of our solution with respect to the application of test suites reduction techniques.

The rest of this paper is structured as follows. Section 2 presents an overview of XACML based access control systems. Section 3 presents the distributed architecture of our solution, whereas Section 4 shows its application to a simple access control evaluation engine. Section 5 briefly presents related work whereas Section 6 draws conclusions and gives hints for future works.

[1] In this paper, we focus on testing of XACML 3.0 based access control engines but our solution can be easily generalized to other access control specification language.

2 BACKGROUND

XACML (OASIS, 2013) is a de facto standardized specification language that defines access control policies and access control decision requests/responses in an XML format. An XACML policy defines the access control requirements of a protected system. An access request aims at accessing a protected resource and is evaluated against the policy, then the access is granted or denied. The main components of an access control systems architecture are the Policy Enforcement Point (PEP) and the Policy Decision Point (PDP). A PEP intercepts a user's request, transforms it into an XACML format and transmits it to the PDP. As showed in Figure 1, the PDP evaluates the request against the XACML policy and returns the access response (Permit/Deny/NotApplicable/ Indeterminate). The PAP (Policy Administration Point) writes policies and make them available to the PDP, whereas the PIP (Policy Information Point) acts as a source of attribute values.

Briefly, an XACML policy has a tree structure whose main elements are: PolicySet, Policy, Rule, Target and Condition. The PolicySet includes one or more policies. A Policy contains a Target and one or more rules. The Target specifies a set of constraints on attributes of a given request. The Rule specifies a Target and a Condition containing one or more boolean functions. If a request satisfies the target of the policy, then the set of rules of the policy is checked, else the policy is skipped. If the Condition evaluates to true, then the Rule's Effect (a value of Permit or Deny) is returned, otherwise a NotApplicable decision is formulated (Indeterminate is returned in case of errors). More policies in a policy set and more rules in a policy may be applicable to a given request. The PolicyCombiningAlgorithm and the RuleCombiningAlgorithm define how to combine the results from multiple policies and rules respectively in order to derive a single access result.

We show in Figure 3 an example of XACML policy.

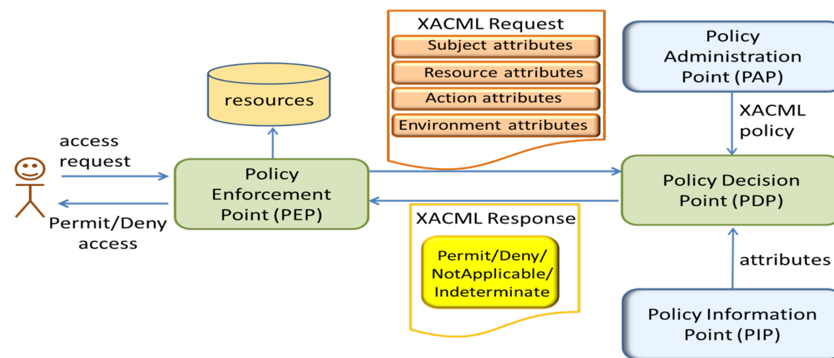

Figure 1: XACML Architecture.

3 ARCHITECTURE

Considering the problem of testing of the access control engine, here we propose a decentralized solution. Thus, we describe the distributed architecture conceived for the generation, execution, and testing results collection able to improve the quality of PDP testing and drastically reduce its computational cost. In particular, we focus on XACML 3.0 based access control engine and we target the application of massive combinatorial testing for assessing the correctness and robustness of the PDP.

The considered architecture includes the following components:

- **XACMLClient:** it is the entry point of the proposed architecture. It includes a GUI for user interaction with four distinct operations: i) loading of an XACML policy (or a set of policies) and selection of the test strategy to be used for the generation of test cases, i.e. XACML requests; ii) retrieving of already uploaded XACML policies; iii) retrieving of the XACML requests associated with a given policy for a specific test strategy; iv) execution of the set of requests, visualization of the results obtained, i.e. the authorization decisions associated with each request, and computation of final verdict (pass/fail) of each test case.

- **XACMLgenerator WS:** it is in charge of automatically generating a set of XACML 3.0 requests from an XACML policy according to a combinatorial testing strategy through a distributed approach. XACMLGenerator WS provides a set of policy testing strategies, that users can select through the XACMLClient. They have been included either by directly implementing the corresponding algorithm or by integrating, wherever possible, the associated test case generation tool. Among them, in this paper we focus on the `Multiple Combinatorial` testing strategy. Very briefly, the Multiple Combinatorial testing strategy relies on the combinatorial approaches of subject, resource, action and environment values taken from the XACML policy (Bertolino et al., 2010; Bertolino et al., 2013). It is out of the scope of this paper to focus on the definition of test strategies. The XACMLGenerator WS has been voluntarily conceived to be independent from the test strategies adopted. The only mandatory constraints are that the strategies considered are based on XACML language and there exist a tool or at least a detailed specification that lets the implementation and the integration into the XACMLGenerator WS. In case for a given XACML policy, the set of test cases for the selected test strategy was already generated, the XACMLGenerator WS is also in charge of retrieving the required data.

- **XACMLPDP WS:** it is the component in charge of executing the test cases on the PDP under test. By means of a MapReduce approach, the set of test cases id divided into sub sets, each one having a similar complexity, and distributed over different instances of PDP under test. The test results, i.e. the authorization decisions, computed by the PDP under test, are finally collected into a specific database managed by the XACMLTestingDB component.

- **XACMLOracle WS:** for each XACML request, derived by a given policy, the XACMLOracle WS derives the correct authorization decision. Specifically, through a map reduce approach, the set of test cases is divided into subsets each one having a similar complexity, and distributed to groups of different PDPs. Each group is

composed by an odd number of PDPs, different from the one under test. From a practical point of view, each PDP in each group receives a set of policies and their corresponding subset of XACML requests. It executes them, derives the authorization decisions, and sends back the collected results. The XACMLOracle WS collects for each request all the authorization decisions derived by the different PDPs in each group, and associates the correct authorization decision with the decision value most frequently received. The correct authorization decisions are finally collected into a specific database managed by the XACMLTestingDB component. If for a given XACML policy, the set of correct authorization decisions for the selected test strategy was already generated the XACMLOracle WS is also in charge of retrieving the required data.

- **XACMLComparator WS:** for each XACML request, the XACMLComparator retrieves the corresponding test result, computed by the XACMLPDP WS, and the correct authorization decision, computed by XACMLOracle WS and checks if they are equal. In this case a `pass` value verdict is associated to the request, `fail` otherwise. The final verdicts are finally collected into a specific database managed by the XACMLTestingDB component.

- **XACMLRouter Servlet:** according to the operation chosen by the user through the XACMLClient, it forwards the request to the appropriate web service.

- **XACMLTestingDB:** it is the manager of the different databases that allows for the persistence of test activity data.

Figure 2 shows the relationships between the different components of the proposed framework. As shown in the figure, the communication between XACMLClient and the services takes place through an intermediary node, proxy / router, which forwards the client's request to the appropriate service based on the content of the request. Communications with all services require secure communication able to guarantee the identity of the parties involved. Interactions with XACMLTestingDB database are not exposed to SQL Injection attacks. We used the so-called Prepared SQL statement in order to prevent SQL Injection attacks. Prepared Statements are static prevention techniques that attempt to prevent SQL Injection by allowing developers to accurately specify the structure of an SQL query, and pass the parameters values to it separately such that any unsanitary user-input is not allowed to modify the structure and the semantic of the query. The developed services take SQL Injection attacks into account and the interactions with XACMLTestingDB, that provide parameters supplied by XACMLClient, are managed via Java *Prepared-Statement*.

4 APPLICATION EXAMPLE

In this section, we provide an example of application of the proposed framework for testing a policy evaluation engine. For this example, we asked to four groups of three students of a secure software engineering course to realize a simplified version of a policy execution engine. Each group received the same small subset of functionalities to be implemented into a Java prototyped engine based on XACML 3.0 specification. At the end, one of the realized PDP has been randomly selected as SUT (System Under Test) and called SUTPDP. The other three PDPs have been used as oracle and called Oracle1, Oracle2 and Oracle3 in the remaining of the section. Even if the realized version of PDPs were very simple and limited due to the small set of functionalities implemented, the application of a combinatorial testing approach may require considerable amount of time and effort. Thus a decentralize testing framework could be a valid solution for decreasing the testing cost and increasing the final PDP quality and security.

For better quantify the cost reduction, the evaluation of the proposed approach has been performed considering four different research questions:

- **RQ1 Generation Cost**: Is the distributed framework able to reduce the time for test cases generation?
- **RQ2 Execution Cost**: Is the distributed framework able to reduce the time for test cases execution?
- **RQ3 Oracle Derivation Cost**: Is the distributed framework able to reduce the time for the correct authorization decisions derivation?
- **RQ4 Verdict Computation Cost**: Is the distributed framework able to reduce the time for verdicts computation?

In the remaining part of this section the details about the application example execution and the investigations about the above research questions are provided.

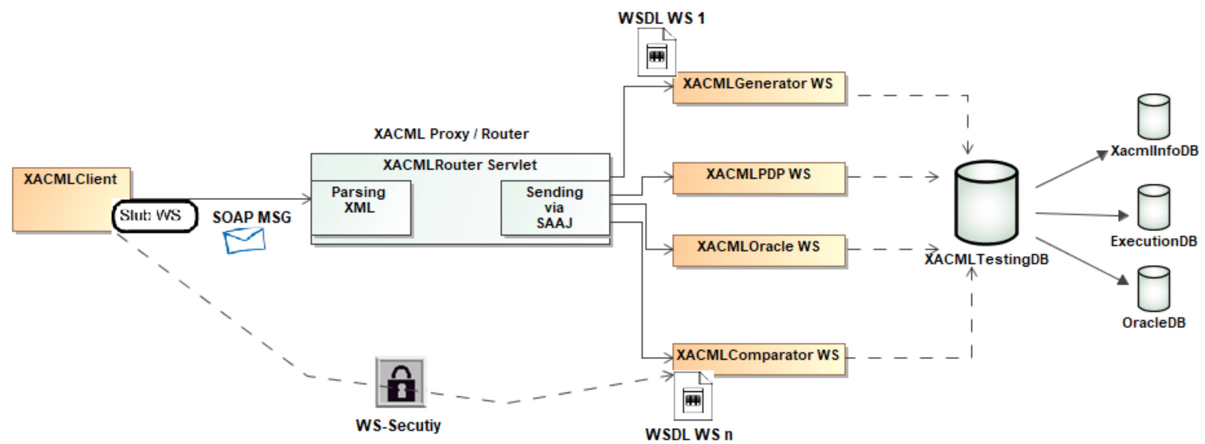

Figure 2: System Architecture.

4.1 Testing Environment Set-up

For the investigation of the target research we asked the master students to develop also a set of policies example to be used for testing the SUTPDP. A total amount of 100 access policies has been pre-included into XACMLTestingDB, each one targeting a set of different either functionalities, or subjects, resources or activities, or hierarchical structures. The simplified listing in Figure 3 represents one of the developed policies example. The policy example is composed by one XACML Policy containing a Target and three XACML rules:

- The Target specifies that the XACML policy is applicable to the `foo` resource;
- The first Rule states that the `bar1` action can be performed only by the user `bob` on resource `foo/foo1`;
- The second Rule says that the user `alice` can access to resource `foo/foo2` only in `bar2` mode;
- And finally, the third Rule denies all the accesses which are not allowed explicitly and represents a default XACML rule.

The experiment has been executed using the 20 working stations available in the university laboratory, which can be considered similar in the overall performance[2]. In particular, the execution of the combinatorial testing of the SUTPDP has been logically divided into three steps:

[2]To run the experiment we used 10 working stations having a Core i7-4790 (4.0GHz) Intel processor machine with eight virtual CPUs and 16GB of memory, running Ubuntu 14.04 (64-bit version) and 10 working stations having a Core i7-4700 (4.2GHz) Intel processor machine with eight virtual CPUs and 16GB of memory, running Ubuntu 14.04 (64-bit version).

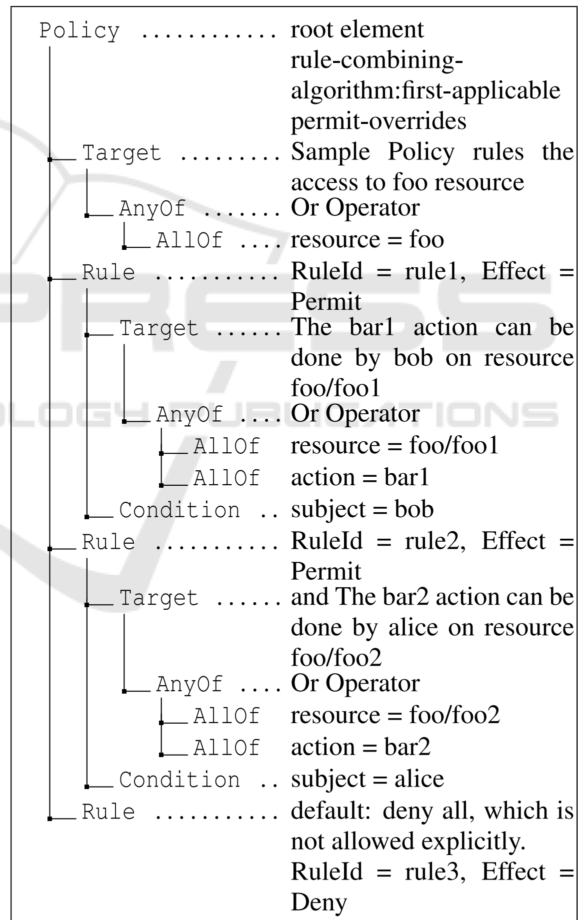

Figure 3: An XACML policy.

1. Generation of test case from the policies set;
2. Parallel execution of the test cases on:
   - The SUTPDP, and
   - The three different oracles (Oracle1, Oracle2 and Oracle3) for the correct authorization de-

cision identification;

3. Derivation of the final verdicts.

Considering the generation of the test cases, through the XACMLClient graphical interface the `Multiple Combinatorial` testing strategy has been selected and applied in parallel to each of the 100 policies generating the corresponding set of test cases. This operation generated a set of around 50,000 test requests considering all the access policies available. As an example, for the access policy of listing depicted in Figure 3 a total amount of 945 requests has been generated through the operation GetXacmlRequests, as summarized in Figure 4.

From a practical point of view, in this case the XACMLGenerator WS distributes the 100 policies over all the 20 working stations, thus each one received a group of 5 policies to be used for executing the `Multiple Combinatorial` testing strategy.

In parallel, through the XACMLClient graphical interface and specifically using the operation *ExecuteAllXacmlRequests*, each policy and the relative set of test cases:

- Have been executed on the SUTPDP through XACMLPDP WS;
- Have been used for correct authorization definition by using the XACMLOracle WS.

Figure 5 shows an extract of the derived SUTPDP decisions for the test set relative to the policy of listing in Figure 3.

For the parallel execution of the test cases on the SUTPDP and on the three different oracles, the 20 working stations has been divided as in the following:

- 5 of them have been randomly selected and assigned to the SUTPDP testing. In this case, XACMLPDP WS instantiated a version of the SUTPDP on each of the 5 machines and selected for each of them around 10,000 test cases trying uniform load balancing.
- The remaining 15 have been divided into 5 groups of 3 working stations each. Through XACMLOracle WS, Oracle1, Oracle2 and Oracle3 PDPs have been deployed on the different machines of each group. The 50,000 test cases have been successively divided into 5 sets of around 10,000 each and executed in parallel on Oracle1, Oracle2 and Oracle3 so to collect the authorization decisions and finally compute the correct ones.

Finally, through the XACMLComparator WS the final verdict for each of the 50,000 test case has been derived. Due to the simplicity of the operation a single working station has been used for this task.

Even if the SUTPDP implemented a subset of functionalities, a final amount of 127 fail verdicts have been highlighted. These were due to an incorrect implementation of a rule combining algorithm and a specific set up of the environment condition in the SUTPDP that have been successively corrected.

In the next subsections, detailed results for each of the proposed research questions are reported and discussed.

4.2 RQ1: Generation Cost

In this section, we investigate the cost of test cases generation. Thus, supposing that each test case has potentially the same impact on the overall testing effort, the generation time becomes directly connected with the number of test cases generated: i.e. the size of a test suite represents also its cost. In the experiment reported in this paper (Section 4.1) the possibility of executing the `Multiple Combinatorial` testing strategy in a distributed way on the 20 working stations, lets the test generation completion in around 5.30 minutes. By a more detailed analysis of the data collected during the experimentation, considering the different nature of the XACML policies, the application of the `Multiple Combinatorial` testing strategy requires on average from 30 to 80 seconds; therefore, the execution of a group of 5 policies requires on average from 150 to 400 seconds. Indeed the differences were manly due to the complexity of the policies analyzed, the performance of the working stations and the communications delay. With these estimations, an idea of the reduction of the time required for test generation, in case less then 20 working stations, were used can be derived.

For aim of completeness, we repeated the experiment of test case generation on a randomly selected working station and it took around 1.5 hours. In this case, the test generation time was not affected by communication delay. Comparing the distributed solution with the not distribute one a positive answer to the research question RQ1 can be collected, i.e. the distributed framework is able to reduce time for test case generation.

For aim of completeness it is important to notice that:

- Because of the didactic nature of the experiment, the policies developed by the students were quite simple in the structure and in the number of values used for subjects, resources, actions and environment attributes. Therefore, the computation time required for the test cases generation could be also tolerable for the not distributed solution

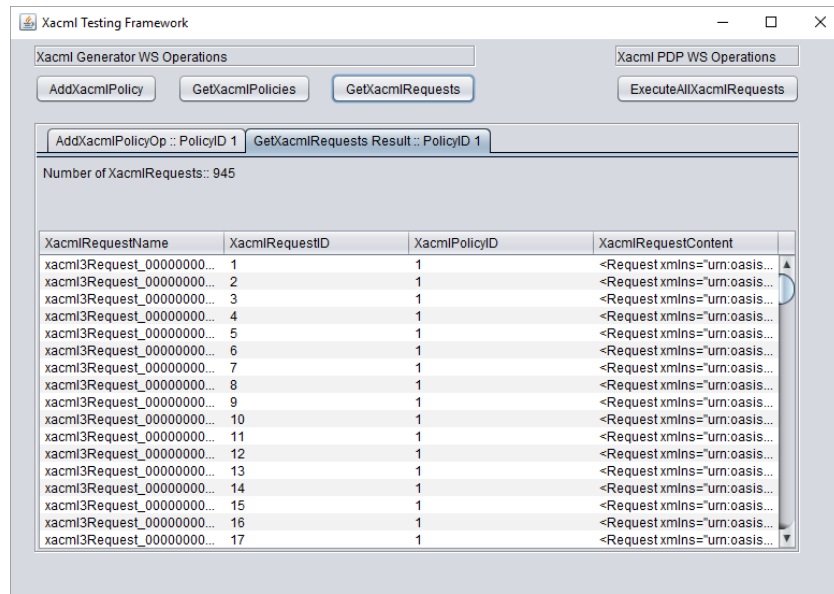

Figure 4: Result of GetXacmlRequests Operation.

even if around 16 times greater than the computation time in the distributed one.

- The test case generation can be performed one and for all for the entire policies set. Once available the test suite can be reused several times in different experimentations. Consequently, the test generation cost would be only the time for test cases retrieval.

4.3 RQ2: Execution Cost

For investigation of the execution cost as in the previous section, we suppose that each test case has potentially the same impact on the overall testing execution time. Thus the execution cost becomes directly connected with the size of a test suite. Considering the experiment set up of Section 4.1, the test cases execution on the SUTPDP has been performed in parallel on 5 working stations for a total amount of around 3.5 minutes. Of course, this estimation can vary by augmenting or diminishing the number of working stations dedicated to the SUNPDP testing. For having a clearer idea, an estimation of the number of test cases executed in each minute of testing has been derived from the data collected. In the experiment, the tests executed per second varies from 40 to 60, consequently the total amount of time necessary for executing 10,000 test cases could vary from around 4.20 to 2.8 minutes. As in the previous section, the differences were manly due to the complexity of the policies analyzed, the performance of the working stations, and the communications delay.

As for the previous question, for aim of completeness, we repeated the experiment of test case generation on a randomly selected working station and it took around 16 minutes. In this case, the test generation time was not affected by communication delay. Comparing the distributed solution with the not distribute one, a positive answer to the research question RQ2 can be collected, i.e. the distributed framework is able to reduce the time for test cases execution.

4.4 RQ3: Oracle Derivation Cost

For investigation of the evaluation cost as in the previous section, we suppose that each test case has potentially the same impact on the overall testing execution time. Thus the evaluation cost becomes directly connected with the size of a test suite.

Considering the experiment set up of Section 4.1, the test cases execution on the three versions of oracle has been performed in parallel on 5 groups of three oracles each. Because the number of tests to be executed (10,000) and the oracles performance were similar to that of the SUTPDP, the time necessary for the derivation of correct authorization decision for each test case does not evidence important differences from the execution time. Indeed, the total amount was around 3.7 minutes. In this case the differences were manly due the communications delay.

For aim of completeness we repeated the experiment on a randomly selected working station. In this case we deployed one per time the different oracle versions on the working station, we collected the au-

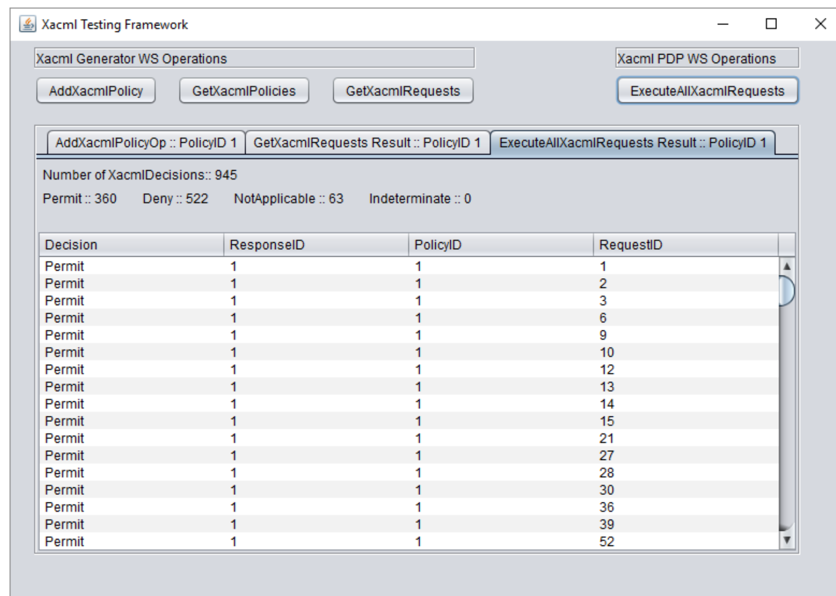

Figure 5: Result of EvaluateAllXacmlRequests Operation.

thorization decision sets and then we compared them for deriving the correct authorization decisions set. In practice, we repeated the execution of all the 50,000 test cases on an oracle version for three times. Without considering the time necessary for the different oracles instantiation, the total amount of time necessary for the correct authorization decisions set was 49 minutes.

Comparing the distributed solution with the not distribute one a positive answer to the research question RQ3 can be collected, i.e. the distributed framework is able to reduce the time for oracle derivation. For aim of completeness it is important to notice that the correct authorization decisions can be performed one and for all for the entire test suite. Once available, correct authorization decisions can be reused several times in different experimentations. Consequently the test evaluation cost would be only the time for data retrieval.

4.5 RQ4: Verdict Computation Costs

For investigation of the verdict computation cost, as in the previous section, we consider the size of the test suite. Actually this operation is just a binary comparison between two values: the authorization decision computed by the SUTPDP with the correct one. Due to the simplicity of the operation and the number of data compared, mainly the cost of this operation depends on the data retrieval for the different DBs. Comparing the distributed solution with the not distribute one, no difference can be noticed, therefore no answer can be given to research question RQ4.

5 RELATED WORK

This work spans over several research directions, including: combinatorial approaches for test case generation and testing of the PDP.

Combinatorial Approaches for Test Cases Generation. In combinatorial testing, test cases are designed to execute combinations of input parameters (Nie and Leung, 2011). Because providing all combinations is usually not feasible in practice, due to their extremely large numbers, combinatorial approaches able to generate smaller test suites for which all combinations of the features are guaranteed, are preferred. Among them, common approaches rely on t-way combinatorial criteria of input parameters (Cohen et al., 1997; Kuhn et al., 2013) and include support for constraints and variable-strength tests (NIST, 2016). Other approaches review existing solutions for the oracle problem for combinatorial testing including crash testing, embedded assertions, and model checker-based test generation (Kuhn et al., 2008).

In combinatorial testing, some tools exist (Aggarwal et al., 2016; Sabharwal and Aggarwal, 2017) that are able to identify interaction faults in the source code using for instance data flow techniques (Sabharwal and Aggarwal, 2017). They are able to generate a test set for only the failure triggering interactions with

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

the aim of reducing the test set size without much affecting the fault detection capability. Differently from these works, the advantage of our proposal is the possibility to exploit parallel computational resources to generate a large set of test cases without applying test suite reduction techniques.

In the context of XACML based access control systems, combinatorial test cases generation strategies are proposed for testing on the one side the XACML policy specification and on the other that the PDP behavior conforms to the policy specification. Among them, the Targen tool (Martin and Xie, 2006) generates test inputs using combinatorial coverage of the truth values of independent clauses of XACML policy values. A more recent tool is X-CREATE (Bertolino et al., 2013) that provides different strategies based on combinatorial approaches of the subject, resource, action and environment values taken from the XACML policy for deriving the access requests. Among the X-CREATE generation strategies, we selected in this paper *Multiple Combinatorial* strategy for deriving test suites used to validate the effectiveness of the proposed approach. It allows combinations of more than one subject, resource, action and environment values and automatically establishes the number of subjects, resources, actions and environments of each request according to the complexity of the policy structure. Other works (Xu et al., 2012; Xu et al., 2015) focus on model based testing and apply combinatorial analysis to the elements of the model (role names, permission names, context names) or pairwise combinations of tokens of a predicatetransition net to derive test cases. Finally, ACPT tool (NIST, 2018) uses combinatorial testing with model checking to produce tests for access control policies. To overcome the computational cost of combinatorial testing, recent solutions (Tsai et al., 2015; Tsai and Qi, 2016) rely on cloud environments where a large number of processors and distributed databases are adopted to perform large combinatorial tests execution in parallel. Differently from our proposal, these approaches use concurrent test algebra execution and analysis for identifying faulty interactions and reducing combinatorial tests, moreover they are not specifically tailored to access control systems.

Testing of the PDP. Test oracle automation has been for long time investigated in literature and still remains a challenging problem (Barr et al., 2015). In the context of access control systems, few proposals address PDP testing aiming to automatically check whether the test outputs are correct. The authors of (Li et al., 2008) propose to simultaneously observe the responses from different PDPs on the same test inputs, so that different responses can highlight possible issues. Although effective, the proposal is quite demanding, because it requires using different PDP implementations. Our approach is in line with this proposal, but it uses a distributed solution. A different approach proposed in (Bertolino et al., 2018) deals with PDP testing providing a completely automated model-based oracle derivation proposal for XACML-based PDP testing.

6 CONCLUSIONS

In this paper, we proposed a new decentralized framework for testing of XACML-based access control engines. The main advantages of the proposed solution are: i) reducing the computational costs of the generation and execution of large combinatorial test suites without the adoption of test suites reduction techniques; ii) providing a decentralized and cost effective automated oracle for the PDP testing.

The application of the proposed distributed solution to a simple application example showed its effectiveness in reducing the cost of all the main phases of the testing process: test cases generation, test case execution and oracle derivation. Specifically, for the policy example showed in the paper, the cost saving was of around 90% (total testing process time was of 12,5 minutes for our solution against 155 minutes of a common single working station). This time saving is mainly due to the efficient decentralized oracle derivation that represents the most challenging task of the testing process. Our solution allows to reduce testing costs also when test cases and correct authorizations decisions are already available. In this last case, the cost gain was around 78% (total testing process time was of 3,5 minutes for our solution against 16 minutes of a common single working station).

As threat to validity, we have to observe that these time measures are dependent on the computational powers of the used working stations and the scheduling of the test task on the different working stations.

In the future, we plan to investigate more about efficient scheduling approaches dealing with power constraints of different working stations as well as to address scalability issues related to the proposed framework. We would like also to study the relationship between the cost saving and the performance of the proposed approach in terms of fault detection effectiveness.

ACKNOWLEDGEMENTS

This work has been partially supported by the GAUSS national research project (MIUR, PRIN2015, Contract2015KWREMX).

REFERENCES

Aggarwal, M., Sabharwal, S., and Dudeja, S. (2016). FTCI: A Tool to Identify Failure Triggering Combinations for Interaction Testing. *Indian Journal of Science and Technology*, 9(38).

Barr, E. T., Harman, M., McMinn, P., Shahbaz, M., and Yoo, S. (2015). The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525.

Bertolino, A., Daoudagh, S., Lonetti, F., and Marchetti, E. (2018). An automated model-based test oracle for access control systems. In *Proceedings of 13th IEEE/ACM International Workshop on Automation of Software Test, Gothenburg, Sweden. May 28-29*.

Bertolino, A., Daoudagh, S., Lonetti, F., Marchetti, E., Martinelli, F., and Mori, P. (2014). Testing of polpa-based usage control systems. *Software Quality Journal*, 22(2):241–271.

Bertolino, A., Daoudagh, S., Lonetti, F., Marchetti, E., and Schilders, L. (2013). Automated testing of extensible access control markup language-based access control systems. *IET Software*, 7(4):203–212.

Bertolino, A., Lonetti, F., and Marchetti, E. (2010). Systematic XACML Request Generation for Testing Purposes. In *Proc. of 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pages 3 –11.

Candido, J., Melo, L., and d'Amorim, M. (2017). Test suite parallelization in open-source projects: a study on its usage and impact. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 838–848. IEEE Press.

Cohen, D. M., Dalal, S. R., Fredman, M. L., and Patton, G. C. (1997). The AETG system: An approach to testing based on combinatiorial design. *IEEE Trans. on Soft. Eng.*, 23(7):437–444.

Kuhn, D. R., Kacker, R. N., and Lei, Y. (2013). *Introduction to combinatorial testing*. CRC press.

Kuhn, R., Lei, Y., and Kacker, R. (2008). Practical combinatorial testing: Beyond pairwise. *It Professional*, 10(3).

Li, N., Hwang, J., and Xie, T. (2008). Multiple-implementation testing for xacml implementations. In *Proceedings of the 2008 workshop on Testing, analysis, and verification of web services and applications*, pages 27–33. ACM.

Martin, E. and Xie, T. (2006). Automated test generation for access control policies. In *Supplemental Proc. of ISSRE*.

Martin, E., Xie, T., and Yu, T. (2006). Defining and measuring policy coverage in testing access control policies. In *Proc. of ICICS*, pages 139–158.

Nie, C. and Leung, H. (2011). A survey of combinatorial testing. *ACM Computing Surveys (CSUR)*, 43(2):11.

NIST (2016). Automated Combinatorial Testing for Software. https:csrc.nist.govprojectsautomated-combinatorial-testing-for-softwaredownloadable-tools.

NIST (2018). Access Control Policy Test (ACPT). https:csrc.nist.govprojectsautomated-combinatorial-testing-for-softwaredownloadable-tools#acpt.

OASIS (2013). eXtensible Access Control Markup Language (XACML) Version 3.0. http:docs.oasis-open.orgxacml3.0xacml-3.0-core-spec-os-en.html.

Sabharwal, S. and Aggarwal, M. (2017). A novel approach for deriving interactions for combinatorial testing. *Engineering Science and Technology, an International Journal*, 20(1):59 – 71.

Tsai, W.-T. and Qi, G. (2016). Integrated fault detection and test algebra for combinatorial testing in taas (testing-as-a-service). *Simulation Modelling Practice and Theory*, 68:108–124.

Tsai, W.-T., Qi, G., and Hu, K. (2015). Autonomous decentralized combinatorial testing. In *IEEE Twelfth International Symposium on Autonomous Decentralized Systems (ISADS)*, pages 40–47. IEEE.

Xu, D., Kent, M., Thomas, L., Mouelhi, T., and Le Traon, Y. (2015). Automated model-based testing of role-based access control using predicate/transition nets. *IEEE Transactions on Computers*, 64(9):2490–2505.

Xu, D., Thomas, L., Kent, M., Mouelhi, T., and Le Traon, Y. (2012). A model-based approach to automated testing of access control policies. In *Proceedings of the 17th ACM symposium on Access Control Models and Technologies*, pages 209–218. ACM.