Introducing concurrency in policy-based access control

Maarten Decat,
Bert Lagaisse, Wouter Joosen
iMinds-DistriNet, KU Leuven
3001 Leuven, Belgium
first.last@cs.kuleuven.be

Bruno Crispo
Department of Information Engineering and
Computer Science
University of Trento
Trento, Italy
crispo@disi.unitn.it

ABSTRACT

Policy-based access control aims to decouple access control rules from the application they constrain by expressing these rules in declarative access control policies. Performance of policy-based access control is of growing importance, but concurrent and distributed policy evaluation has received little research attention and current policy evaluation engines are still single-machine and fully sequential to the best of our knowledge. We believe that concurrent policy evaluation is necessary to meet the performance and scalability requirements of next-generation internet applications and aid the maturation of policy-based access control. Therefore, this paper presents an initial exploration of concurrent policy evaluation. We illustrate the performance of current policy evaluation engines, model the performance of policy evaluation in terms of the characteristics of a policy, list opportunities for concurrency, describe the need for concurrency control and specifically show how concurrency can be used to improve throughput based on our prototype.

Categories and Subject Descriptors

C.2.4 [Computer-Communication Networks]: Distributed Systems—*Distributed Applications*; D.4.6 [Security and Protection]: Access controls

General Terms

Security, Performance

Keywords

Access control, access control policies, performance, concurrency

1. INTRODUCTION

Access control is an important part of application-level security that limits the actions which a user can perform in an application. For reasons of separation of concerns, modularity and flexibility, access control rules are decoupled from the

application they constrain and are expressed in declarative access control policies, to be evaluated by specialized middleware. Such *policy-based access control* aims to provide manageable application configuration to domain experts and can therefore become an important enabler for future internet applications such as Software-as-a-Service applications.

Access control should be enforced on every request to the application and the access control policies are therefore evaluated for every request as well. As a result, policy evaluation should be efficient, i.e., policy evaluation should provide low latency and high throughput. Recent work has therefore focused on performance of policy-based access control, amongst others by optimizing the internal policy representation (e.g., [12, 13]), focusing on attribute retrieval (e.g., [8, 6]) and introducing caching (e.g., [15]).

However, concurrent and distributed policy evaluation have received little attention in literature and current policy evaluation engines (e.g., SunXACML [4] and PicketBox XACML [3]) are still single-machine and fully sequential. While other performance techniques can push the boundaries of such policy evaluation engines, concurrency and distribution will eventually be needed to achieve the necessary performance and scalability for next-generation internet applications: interleaved policy evaluation could increase throughput of single-node policy evaluation and distribution could allow engines to scale out horizontally. Therefore, we believe that concurrency and distribution are essential techniques for the maturation of policy-based access control.

In this paper, we present an initial exploration of concurrent policy evaluation. We first illustrate the performance of current policy evaluation engines with measurements on SunXACML and show that these do not meet the needs of future internet applications. We then build a performance model for policy evaluation, i.e., a model for the latency and throughput of policy evaluation in terms of the characteristics of a policy, and describe opportunities for concurrency and the need for concurrency control. Finally, we specifically show how concurrency can improve throughput on a single machine based on this model and our prototype.

The rest of this paper is structured as follows. Section 2 gives the context and motivation of this paper. Section 3 builds a performance model for policy evaluation and describes opportunities and challenges for concurrency. Section 4 shows how concurrency can improve throughput on a single machine based on our prototype. Section 5 lists related work and Section 6 concludes this paper.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MW4NG '13 December 9-13, 2013, Beijing, China

Copyright 2013 ACM 978-1-4503-2551-6/13/12 ...\$15.00.

Request	Nb rules	Nb attr.	Total evaluation time	Attribute fetch time	Processing time	Throughput
#1	3	4	5.44 ms	2.52 ms (46%)	2.93 ms (54%)	183 req/sec
#2	4	7	7.23 ms	4.19 ms (58%)	3.04 ms (42%)	138 req/sec
#3	6	10	8.89 ms	5.68 ms (64%)	3.21 ms (36%)	112 req/sec
#4	7	13	10.95 ms	7.50 ms (68%)	3.45 ms (32%)	91 req/sec

Table 1: Detailed overview of the measurements of Section 2.3.

2. MOTIVATION

This section motivates the need for concurrency in policy-based access control. We give an overview of policy-based access control as context to this work, zoom in on its performance requirements and illustrate the performance of current policy evaluation engines based on SunXACML.

2.1 Policy-based access control

Access control is an important part of application-level security that limits the actions (e.g., read, write) which a subject (e.g., an end-user) can perform on an object in the system (e.g., a file). Originally, access control was tightly coupled with the application code, making it hard to maintain and providing low modifiability. Currently, the access control rules are separated from the mechanisms that enforce them and can be externalized from the application they constrain into modular, declarative access control policies. These policies are evaluated at run-time by a specialized middleware component referred to as the policy evaluation engine. Policy-based access control can become an important enabler for future internet applications, amongst others by providing modular and scalable access control management to domain experts.

Multiple models have been proposed for expressing access control policies. The recent Attribute-Based Access Control (ABAC, [11]) generalizes previous access control models and expresses policies in terms of general key-value properties called attributes of the subject (e.g., the subject id, username or roles), the object (e.g., the object id, location or content) and the environment (e.g., the time, physical location or usage context). Attributes provide increased expressivity with regard to previous models and offer a unit of data transport between the different components or parties involved in access control. For both reasons, this work builds upon ABAC.

2.2 Performance in policy-based access control

Current applications pose challenging performance requirements. For example, applications such as Amazon.com are highly sensitive to latency increases on their page loads [1]. In addition, paradigms such as Software-as-a-Service are aimed at being used by large numbers of customer organizations and users, thereby also requiring large throughput. Moreover, the number of requests per second is not homogeneous because of events such as Christmas shopping in the domain of e-commerce or calamities in the domain of e-health. As a result, these applications are designed to scale out for increased request rates. Towards the future, these performance requirements can only be expected to grow.

Access control policies should be enforced on every request to the application. Moreover, a single request to the

Figure 1: Policy evaluation time in terms of the number of required attributes. Lower is better.

application can internally be handled by multiple services or components which each can be constrained by their own policies and a single request to the application can thus lead to multiple policy evaluations. Therefore, evaluating the access control policies directly affects the end-user performance and as a result, access control should provide similar performance and scalability properties as the application, i.e., its latency should be low, its throughput should be high and it should be able to scale out with the application itself.

2.3 Illustration: Measurements

To illustrate the performance of current policy evaluation engines, we measured the evaluation of a realistic policy from an e-health case study. The application in the case study is a home patient monitoring system provided to hospitals as a Software-as-a-Service application. The policy (available online¹) bundles 19 rules, which for example state that physicians can only access data of patients whom they are currently treating. These 19 rules in total employ 30 different attributes, leading to a policy of modest size. For the tests, we used an optimized version of SunXACML (also available online¹) deployed on a virtual node with 2GB RAM and a dual-core CPU, simulating commodity hardware. The attributes are stored in a MySQL database deployed on another similar node. Both nodes are connected using a local, sufficiently overdimensioned virtualized network.

In the tests, we measured the evaluation time (as an approximation to latency without any queuing) and throughput of the evaluation of the policy from the case study. While the policy bundles 19 rules and employs 30 different attributes, not all rules apply to every request and not all attributes are needed for every request. More precisely, the number of rules involved in a single request ranges from 3 to 7 (with a mean of 4.79) and the number of attributes

¹<http://people.cs.kuleuven.be/~maarten.decat/mw4ng13/>

ranges from 4 to 13 (with a mean of 7.65). Therefore, we took measurements for four requests that cover these ranges. Each test is repeated 100,000 times and we report average values. Nothing is cached by the policy evaluation engine.

Table 1 shows the results of the tests. The table lists the total evaluation time, the portion of this spent on fetching attributes, the portion of this spent on processing the rules and the resulting throughput. Figure 1 illustrates the evaluation times in terms of the required number of attributes. As shown, the total evaluation time grows linearly, but the relative overhead of fetching the attributes grows for larger policies and eventually makes up the largest part of the evaluation time. The absolute numbers show that the evaluation time is modest, yet the resulting throughput is not sufficient to handle the (peak) load for current SaaS applications.

Concurrency and distribution can be employed to improve performance of policy evaluation. The goal of this paper is to provide an initial exploration of concurrent policy evaluation. In the next section, we build a performance model of policy evaluation, list opportunities for concurrency and show the need for concurrency control because of attribute updates. Afterward, we specifically show how to increase throughput on a single machine using concurrency by leveraging on the large portion of evaluation time spent on fetching attributes.

3. PERFORMANCE MODEL

In this section, we build a performance model for policy evaluation. This model expresses the performance characteristics (i.e., evaluation time and throughput) of policy evaluation in terms of the characteristics of a policy (e.g., the number of required attributes). In the end, the resulting performance model should allow us to reason about concurrency. To derive this performance model, we first clearly define how a policy is evaluated.

3.1 Policy evaluation

As said, we start by clearly defining how a policy is evaluated. For this, we build on the structure of an access control policy as well as its evaluation infrastructure, based on the core features of XACML and the XACML reference architecture for policy-based access control infrastructures [2].

Policy structure. In general, a policy states whether a certain request is permitted or not. Optionally, a policy specifies obligations that should be fulfilled in conjunction with enforcing the decision, such as writing a log, sending an e-mail or updating an attribute. The latter can amongst others be used to enforce dynamic separation of duty using object or subject histories. Similarly to XACML, we structure a policy as a tree consisting of *composed policies* as internal nodes and *atomic policies* as leaves (Listing 1 provides an example policy). Each policy in this tree states to which requests it applies by means of a target. In addition, atomic policies specify an effect (Permit or Deny), a condition for this to hold and a set of obligations to be fulfilled. Both the target and the condition are attribute-based expressions (see Listing 1). Composed policies combine the results of several other policies, either atomic policies or other composed policies, using a policy combination algorithm such as Permit-Overrides, DenyOverrides and FirstApplicable. Composed policies can specify their own obligations and also forward the obligations of the sub-policy that determined the com-

Listing 1: Example of a composite policy containing an atomic policy using a JSON-like format.

```
{ target: All,
  algorithm: FirstApplicable,
  obligations: [],
  subpolicies: [ {
    target: subject.role == 'physician',
    effect: Permit,
    condition: object.id in
               subject.treating,
    obligations: [
      subject.history.add(object.id) ]
  }, { ... } ]
}
```

bined result. Obligations are only fulfilled after the top policy has been evaluated.

Policy evaluation infrastructure. For the policy evaluation infrastructure, this work takes on the XACML reference architecture. In this reference architecture (see Fig. 2), the application requests an access control decision from the Policy Decision Point using the integrated Policy Enforcement Point (e.g., an API or a reference monitor). The Policy Decision Point then evaluates the applicable policies loaded from the Policy Administration Point, thereby fetching required attributes from one or more Policy Information Points (e.g., a database). Because we focus on the run-time behavior of policy evaluation, we assume that all policies fit in memory (which holds for all policies we have witnessed in our research projects) and have been loaded before a decision request arrives. We also assume attributes to be fetched only at the moment they are required (because the required attributes depend on the values of former attributes and it is thus impossible to determine the set of required attributes up-front). Moreover, we assume attribute values to be cached during policy evaluation for a single request (in order to avoid unnecessary attribute fetches), but not across multiple requests (in order to avoid freshness issues). After evaluating the policy, the Policy Decision Point fulfills possible obligations using the Obligation Service and returns its decision to the Policy Enforcement Point, which enforces it. Note that in the original reference architecture, the PEP fulfills the obligations. We deviate from this in order to centralize communication with Policy Information Points at the Policy Decision Point.

Resulting policy evaluation behavior. As a result of the previous, a policy evaluation behaves as follows: A policy evaluation is triggered by a decision request from a Policy

Figure 2: This work builds upon the XACML reference architecture for policy-based access control infrastructures.

Figure 3: Evaluation of a policy by a Policy Decision Point as a black box; s is subject, o is object, a is action.

Enforcement Point. The Policy Decision Point then starts evaluating the policies loaded from the Policy Administration Point. Internally, a policy is evaluated by first evaluating its target, then evaluating its condition or its sub-policies if it is applicable and finally returning the result. Attributes are fetched during evaluation. Sub-policies of a composed policy are evaluated in the given order; as a consequence, the policy tree is evaluated depth-first. In the end, the Policy Decision Point fulfills the necessary obligations and returns its decision. Figure 3 shows this behavior as a black box. As shown, we focus on attribute updates as obligations, since only these directly influence concurrency.

3.2 Resulting performance model

From the policy evaluation behavior, we can now build a performance model, i.e., a model for the latency and throughput of policy evaluation in terms of the characteristics of a policy. We here describe the model for serial policy evaluation; the next section extends it with concurrency.

We start from an atomic policy and gradually expand to general policy trees. Let us define the duration of the evaluation of an atomic policy as D_{Atom} . An atomic policy consists of a target and a condition, therefore:

$$D_{Atom}(p) = D_T(p.target) + D_C(p.condition)$$

Both the target and the condition are attribute-based expressions. Evaluating an expression requires fetching its required attributes a_i and processing the expression itself:

$$D_E(e) = \sum d_A(a_i) + d_p(e)$$

Thus, taking in account that the condition is only evaluated if the policy applies:

$$\begin{aligned} D_T(t) &= D_E(t) \\ D_C(c) &= \begin{cases} D_E(c) & \text{if applicable} \\ 0 & \text{if not applicable} \end{cases} \end{aligned}$$

As mentioned in the previous section, we assume attributes to be fetched at the time they are needed. The first time they are required, they are fetched directly from the Policy Information Point with duration $d_{A,PIP}$, otherwise they are fetched from a cache with duration $d_{A,Cache}$. It can be assumed that $d_{A,Cache} \ll d_{A,PIP}$. Thus:

$$d_A(a) = \begin{cases} d_{A,PIP} & \text{if } a \text{ not in cache} \\ d_{A,Cache} & \text{if } a \text{ in cache} \end{cases}$$

A composed policy consists of a target, a list of sub-policies p_i (atomic or composed policies) and a policy combination algorithm. The sub-policies are evaluated only if the policy applies. The duration of combining the results of the sub-policies d_{Comb} can be assumed to be very small. The evaluation time of a composed policy is thus:

$$\begin{aligned} D_{Comp}(p) &= \\ \begin{cases} D_T(p.target) + \sum D(p_i) + d_{Comb} & \text{if applicable} \\ D_T(p.target) & \text{if not applicable} \end{cases} \end{aligned}$$

Finally, the evaluation time of a certain policy p is:

$$D(p) = \begin{cases} D_{Atom}(p) & \text{if atomic policy} \\ D_{Comp}(p) & \text{if composed policy} \end{cases}$$

Since we are focusing on serial policy evaluation, the evaluation time of a single policy evaluation directly determines the throughput T of the Policy Decision Point. When only regarding the same request on the same policy:

$$T(p) = \frac{1}{D(p)}$$

As a summary, this performance model expects the evaluation time to grow linearly and the throughput to decrease inversely with the number of required attributes and the number of sub-policies involved. Table 1 and Figure 1 conform to this model.

3.3 Opportunities and challenges for concurrency and distribution

To increase the performance of policy evaluation, concurrency and distribution can be introduced. For example, throughput on a single node can be improved by interleaving policy processing for other requests in the time spent fetching attributes. Similarly, evaluation time can be improved by interleaving processing of different parts of the same policy (e.g., branches of the policy tree) for a single request. Additionally, these techniques can be used to further distribute policy evaluation over multiple nodes and allow the policy evaluation engine to scale out horizontally.

However, concurrency in policy evaluation also poses some challenges. For example, policies do not only read attributes, but also write attributes using obligations, amongst others to enforce dynamic separation of duty using object or subject histories. Another challenge for concurrency is that attribute values are cached during the policy evaluation for a single request, as mentioned in Section 3.1. Next to optimizing performance, this also makes sure that a policy sees the same value of the same attribute if requested multiple times, even in the presence of out-of-band attribute updates, e.g., by an administrator. Even in the presence of these challenges, concurrent policy evaluation should be equivalent to serial execution in order to avoid introducing security holes and policy evaluation requires a form of concurrency control. Policy evaluation does provide the opportunity to simplify this concurrency control with respect to general transactions

by leveraging on the domain-specific evaluation model specified in Section 3.1 in which all attribute updates are only executed after the last read.

4. INCREASING THROUGHPUT USING CONCURRENCY

The previous section built a performance model for serial policy evaluation and listed opportunities and challenges for introducing concurrency. In this section, we elaborate on one of these by trying to increase throughput of a single-node policy evaluation engine using concurrency. We first extend the performance model of the previous section and then show the practical impact based on our prototype.

4.1 Performance model

In order to increase throughput of a single-node policy evaluation engine, we will leverage on the time spent in attribute fetch (a large portion of the complete evaluation time, as shown in in Figure 1) to process the policy for other requests. More precisely, we will switch between policy evaluations for different requests every time an attribute is fetched from or updated in a Policy Information Point (fetching an attribute from the cache is presumed to entail very small latencies). Figure 4 shows the resulting interleaved policy evaluation for three requests on a single CPU. Because a single CPU only allows to process a single policy at each moment in time, concurrency introduces wait time in two cases: (i) when an attribute fetch or update returns, but another request is being processed ($d_{Wait,Attr}$) and (ii) when a new decision request arrives, but another request is being processed ($d_{Wait,Init}$). As a result, the evaluation time of Section 3.2 has to be adapted on two places:

$$d_A(a) = \begin{cases} d_{A,PIP} + d_{Wait,Attr} & \text{if not in cache} \\ d_{A,Cache} & \text{if in cache} \end{cases}$$

and:

$$D(p) = \begin{cases} d_{Wait,Init} + D_{Atom}(p) & \text{if atomic policy} \\ d_{Wait,Init} + D_{Comp}(p) & \text{if composed policy} \end{cases}$$

The values of $d_{Wait,Attr}$ and $d_{Wait,Init}$ are stochastic and can be modeled using principles from the domain of performance modeling and queuing theory [9]. In terms of queuing theory, this concurrent Policy Decision Point will behave as an open queue with m parallel servers (m being the number of CPUs; $m = 1$ in Figure 4) and a feedback loop (because the same policy evaluation contends for the CPU again after fetching an attribute). Without going into details here,

Figure 4: Interleaved policy evaluation for three requests on a single CPU. P is Processing, AF is Attribute Fetch, W is Wait. A time line starts when a new decision request arrives and ends when the decision is returned.

$d_{Wait,Attr}$ and $d_{Wait,Init}$ can be expected to grow with increasing system load. As a result, introducing concurrency can be expected to increase the average evaluation time. On the other hand, concurrently interleaving policy evaluation should increase the throughput. More precisely, we expect the throughput to increase with the number of threads, but with diminishing returns up to an asymptote. Again we refer to [9] for the details, we evaluate the effect on throughput in practice in the next section.

4.2 Prototype evaluation

Following the theoretical model, we evaluated the practical impact of introducing concurrency on throughput based on a prototype. The prototype extends SunXACML to the concurrency model of Section 4.1, i.e., each policy evaluation is run in a separate thread and threads are switched during attribute fetch. We tested the policies of Table 1 (i.e., requiring 4, 7, 10 and 13 attributes) with 1, 2, 4, 8, 16 and 32 threads. The tests were run on two machines, similarly to the tests of Section 2.3. The tests first queue 100,000 policy evaluation requests and measure the average evaluation time afterward. This approach does not introduce policy conflicts and does not incorporate $d_{Wait,Init}$ as described in the previous section, but does make the tests independent of a particular load model.

Figure 5 summarizes the results of the tests (the full set of measurements is available on-line¹). The measurements with 1 thread equal those of Table 1. Increasing the number of concurrent threads significantly improves throughput, but at the cost of higher evaluation times. For example, going from 1 to 4 threads increases throughput with an average factor of 2.81, but also evaluation time with an average factor of 1.42. A possible explanation is the increase in the average time to fetch a single attribute (e.g., from 0.59ms to 5.38ms when going from 1 to 32 threads). Our tests are not conclusive whether this is due to thread contention or whether the database is the bottleneck. Future work will show if more suitable scheduling or database scaling can improve latencies. The measurements fit the characteristics of a closed queue for increasing load, the queuing model that represents our test set-up. The measurements with 32 threads show the system in overload.

In summary, these results show that concurrency is an interesting track for future research to improve performance of policy-based access control.

5. RELATED WORK

This work fits in a growing collection of work on performance for policy-based access control. Specifically about concurrency and distribution, the most relevant work is that of Dhankhar et al. [7]. They describe distributed XACML evaluation by a tree of Policy Decision Points and also address the need for concurrency control because of attribute updates using two-phase locking. Our work investigates concurrency in policy evaluation more generally and tries to improve performance, while their approach leads to latencies in the seconds. Also closely related is the work of Janicke et al. [10]. They describe a theoretical model for concurrent policy evaluation on which this work builds.

Other authors have explored other performance tactics. For example, both Liu et al. [12] and Pina Ros et al. [13] focus on efficient XACML evaluation by optimizing its representation. These approaches are able to lower the process-

Figure 5: Throughput and evaluation time of concurrent evaluation of the policies of Table 1 with 1, 2, 4, 8, 16 and 32 threads. As shown, concurrency can significantly improve throughput, but at the cost of higher evaluation time.

ing time of a XACML policy (see Figure 1) and our work complements this by putting the time otherwise spent on fetching attributes to use. Other possible tactics are (i) introducing caching as explored by Wei et al. [15] and (ii) optimizing attribute fetch, an idea first posed by Brucker and Petritsch [5] and further explored by Gheorghe et al. [8] and Decat et al. [6]. Finally, an important building block for this paper was the work by Turkmen et al. [14], who presented one of the first extensive performance evaluations of XACML evaluation engines.

6. CONCLUSION

In this paper, we explored concurrent policy evaluation for improving performance of policy-based access control. We started from the observation that current policy evaluation engines do not provide the necessary performance for future internet applications, built a performance model for policy evaluations and specifically showed how concurrency can be used to improve throughput. Our prototype evaluation shows that concurrency definitely is an interesting tactic to improve the performance of policy-based access control.

We plan to refine this work, amongst others by exploring different scheduling algorithms and investigating other applications of concurrency in this domain, e.g., improving evaluation time by concurrently evaluating multiple parts of a single policy or allowing policy engines to scale out horizontally by distributing policy evaluation. While other performance techniques complement this work, we are convinced that concurrent policy evaluation is a necessary technique for the maturation of policy-based access control.

Acknowledgements This research is partially funded by the Research Fund KU Leuven, by the EU FP7 project NESSoS and by the Agency for Innovation by Science and Technology in Flanders (IWT). With the financial support from the Prevention of and Fight against Crime Programme of the European Union (B-CENTRE).

7. REFERENCES

- [1] Latency Is Everywhere And It Costs You Sales - How To Crush It. <http://highscalability.com/blog/2009/7/25/latency-is-everywhere-and-it-costs-you-sales-how-to-crush-it.html>, July 2009.
- [2] eXtensible Access Control Markup Language (XACML) Version 3.0. *OASIS Standard*, 2013.
- [3] PicketBox XACML (JBossXACML) | PicketBox | JBoss Community. <https://community.jboss.org/wiki/PicketBoxXACMLJBossXACML>, August 2013.
- [4] Sun's XACML Implementation. <http://sourceforge.net/projects/sunxacml/>, August 2013.
- [5] A. Brucker and H. Petritsch. Idea: Efficient Evaluation of Access Control Constraints. *Engineering Secure Software and Systems*, pages 157–165, 2010.
- [6] M. Decat, B. Lagaisse, and W. Joosen. Toward efficient and confidentiality-aware federation of access control policies. In *Workshop on Middleware for Next Generation Internet Computing*. ACM, 2012.
- [7] V. Dhankhar, S. Kaushik, D. Wijesekera, and A. Nerode. Evaluating distributed XACML policies. In *ACM SWS*, pages 99–110. ACM, 2007.
- [8] G. Gheorghe, B. Crispo, R. Carbone, L. Desmet, and W. Joosen. Deploy, adjust and readjust: Supporting dynamic reconfiguration of policy enforcement. *Middleware 2011*, pages 350–369, 2011.
- [9] N. J. Gunther. *Analyzing Computer System Performance with Perl: PDQ*. Springer, 2011.
- [10] H. Janicke, A. Cau, et al. Concurrent enforcement of usage control policies. In *POLICY*, pages 111–118. IEEE, 2008.
- [11] X. Jin, R. Krishnan, and R. Sandhu. A Unified Attribute-Based Access Control Model Covering DAC, MAC and RBAC. In *Data and Applications Security and Privacy XXVI*, pages 41–55. Springer Berlin Heidelberg, 2012.
- [12] A. X. Liu, F. Chen, J. Hwang, and T. Xie. XEngine: a fast and scalable XACML policy evaluation engine. In *ACM SIGMETRICS*, volume 36, pages 265–276. ACM, 2008.
- [13] S. Pina Ros, M. Lischka, and F. Gómez Mármol. Graph-based xacml evaluation. In *SACMAT 2012*, pages 83–92. ACM, 2012.
- [14] F. Turkmen and B. Crispo. Performance evaluation of xacml pdp implementations. In *Proceedings of the 2008 ACM workshop on Secure web services*, pages 37–44. ACM, 2008.
- [15] Q. Wei, J. Crampton, K. Beznosov, and M. Ripeanu. Authorization recycling in RBAC systems. In *SACMAT 2008*, pages 63–72. ACM, 2008.