From Plain English to XACML Policies: An AI-Based Pipeline Approach

Maria Teresa Paratore[a], Eda Marchetti[b] and Antonello Calabrò[c]
Institute of Information Science and Technologies "A. Faedo", National Research Council of Italy (CNR), Pisa, Italy
{mariateresa.paratore, eda.marchetti, antonello.calabro}@isti.cnr.it

Keywords: Access Control, Artificial Intelligence, Large Language Models, Validation, Cybersecurity.

Abstract: The increasing adoption of generative artificial intelligence, particularly conversational Large Language Models (LLMs), has presented new opportunities for addressing challenges in software development. This paper explores the potential of LLMs in generating eXtensible Access Control Markup Language (XACML) policies. This paper investigates current solutions and strategies for leveraging LLMs to produce verified, secure, compliant access control policies. Specifically, by discussing current methods for enhancing LLM performances in generating structured text, it introduces a pipeline approach that integrates conversational LLMs with syntactic and semantic validators. This approach ensures correctness and reliability of the generated policies. Our proposal is showcased by using real policies and compares various LLMs' performances (ChatGPT, Claude, Gemini, and LLaMA). Our findings suggest a promising direction for future developments in automated access control policy formulation, bridging the gap between human intent and machine interpretation.

1 INTRODUCTION

Data protection and safeguarding of personal, financial, and sensitive information from unauthorized access, theft, and misuse are essential in almost every application domain. Compromised data can lead to financial loss, identity theft, and damage to an organization's reputation, ultimately undermining users' confidence. Access control systems (Jin et al., 2014) are among the most effective mechanisms for safeguarding data integrity, confidentiality, and operational continuity of organizations. By regulating and limiting access to both physical and digital resources, these systems ensure, for instance, that only authorized individuals, under predefined conditions and at designated times, can access specific resources, enter restricted areas, retrieve sensitive information, or utilize critical assets. Access control is vital for protecting sensitive data, preserving privacy, and ensuring the security of valuable property.

Among the different types of access control systems, Attribute-Based Access Control (ABAC) systems (Coyne and Weil, 2013) are the most widely adopted. They regulate how to gain access to a system through specific access control policies (ACPs), typically expressed in the XML eXtensible Access Control Markup Language (XACML) [1]. Rules are based on subject attributes, resource characteristics, and environmental conditions to determine or deny access. Although XACML policies are powerful for managing access control, their complexity can make them challenging to write and error-prone. Indeed, writing XACML policies involves the management of multiple attribute and condition combinations, conflict resolution (such as overlapping rules for accessing the same resource), and maintenance and scalability assurance. Deriving a validated, compliant, and secure XACML policy requires a deep understanding of access control requirements. It presents a steep learning curve, which may prevent the use of access control systems by individuals who are not experienced and technically skilled. Getting access policies in a human-friendly or machine-readable structured form can be helpful in all situations where policy writing is defined as a one-off and does not represent the main professional activity, as in the following examples:

- **Small Companies.** They often operate with limited resources, resulting in a shortage of professionals with the specialized knowledge required for effective cybersecurity. Without skilled personnel, these organizations may struggle to develop adequate access control specifications that align with their specific needs, potentially leaving them vulnerable to security breaches.

[a] https://orcid.org/0000-0002-0416-4016
[b] https://orcid.org/0000-0003-4223-8036
[c] https://orcid.org/0000-0001-5502-303X

[1] https://www.oasis-open.org/standard/xacmlv3-0/

Paratore, M. T., Marchetti, E. and Calabrò, A.
From Plain English to XACML Policies: An AI-Based Pipeline Approach.
DOI: 10.5220/0013357200003896
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 13th International Conference on Model-Based Software and Systems Engineering (MODELSWARD 2025)*, pages 85-96
ISBN: 978-989-758-729-0; ISSN: 2184-4348
Proceedings Copyright © 2025 by SCITEPRESS – Science and Technology Publications, Lda.

- **Non-Specialized SW Developers.** They may face a steep learning curve before translating natural language requirements into formal specifications and integrating access policies into a software infrastructure. This may lead to potential oversights or misinterpretations and hinder the security of a system.
- **Stakeholders Lacking any Cybersecurity Training or Background.** Professionals such as business analysts, project managers, or even executives responsible for setting access policies may inadvertently provide vague or ambiguous requirements that could complicate the translation process, further exacerbating security risks.

The use of generative artificial intelligence, and more specifically of conversational LLMs, which are now pervasive in a wide variety of everyday tasks, could be seen as a straightforward solution to the proposed problem. Indeed, large language models (LLMs) built on the Transformer architecture, such as generative pre-trained transformers (GPTs) (Kumar et al., 2023) have become the prevailing solution for addressing the challenges faced by software developers (Buscemi, 2023; Zhong and Wang, 2023).

These LLMs are able to generate comprehensive implementation examples based on natural language descriptions of programming tasks, offer valuable code suggestions, and guide users through various problem-solving strategies. This features facilitates the coding process and improves the overall development experience by bridging the gap between human intent and machine understanding. Any content produced by pre-trained LLMs,anyway, should always be carefully evaluated before being used since they are error-prone.

In addition, not all LLMs perform equally in producing code (Siam et al., 2024). LLMs are effective in producing code snippets in widely used programming languages (such as Java, C++, and Python), but they do not perform as well in generating machine-readable structured text formats (such as SQL, JSON, XML, or XACML). In such cases, challenges and critical concerns must still be addressed (Liu et al., 2024; Brodie et al., 2006; Slankas et al., 2014). Indeed, The lack of comprehensive coverage for structured output formats prevents the integration of LLMs in software frameworks that could benefit from the automatic generation of information in such formats. A possible approach to obtain domain-specific structured text from an LLM is to perform ad-hoc training (a.k.a. fine-tuning) to broaden the models' knowledge. The most commonly used LLMs are pre-trained on vast amounts of heterogeneous, non-specialized documents to recognize and generate text patterns for general purposes; fine-tuning an LLM improves accuracy and performance on domain-specific tasks, but it requires significant expertise, computational resources, and time.

The proposal outlined in this paper is based on two research questions:

- **RQ1:** Is it possible to use LLMs to generate XACML policies?
- **RQ2:** Are there low-cost strategies to exploit the potential of LLMs for the purpose of producing xacml policies?

In replying to these RQs, we will also describe an easy-to-adopt, low-cost solution to enable LLMs to define verified, secure, and compliant access control policies. The proposal is based on a toolchain for the integrated use of conversational LLMs and syntactic and semantic validators.

The paper is structured as follows: Section 2 provides the basic background knowledge, Section 3 overviews the currently available solutions for the generation of XACML policies using LLMS, Section 4 presents a pipeline approach for XACML policies' generation. In particular, Section 4.1 discusses the strategy adopted to achieve syntactical correctness of the results, including a comparison of the performance of four popular LLMs; following, Section 4.2, describes our method to achieve also semantic correctness, and in Section 4.3 our final proposal is discussed. The concluding section suggests future developments of our approach. In the Appendix, details of the XACML policies used throughout the paper are provided.

2 BACKGROUND AND RELATED WORKS

2.1 Access Control Systems

Access control systems (Jin et al., 2014) represent one of the most effective mechanisms for ensuring security by managing and restricting access to physical and digital resources. As shown in Figure 1, a typical access control system includes the following software components:

- Policy Administration Point (PAP), which stores and manages ACPs
- Policy Decision Point (PDP), which evaluates access requests against policies and makes decisions to either grant or deny permission
- Policy Information Point (PIP), which supplies additional information needed to take decisions

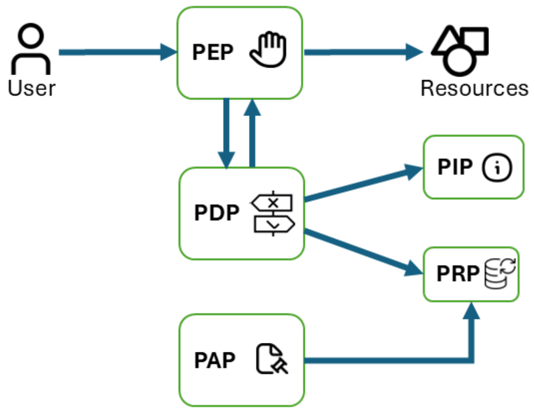


Figure 1: The access control architecture and flow.

- Policy Enforcement Point (PEP), which enforces decisions from the PDP on the incoming requests

Attribute-Based Access Control (ABAC) systems grant access based on attributes such as user credentials, user location, operation time, resource identifiers, etc., thereby enabling fine-grained and dynamic access decisions. Access control policies are expressed through ad hoc languages, in order to ensure reliability and interoperability, the Xtensible Access Control Markup Language (XACML) being the most widely used. The XACML specification [2]; defines a standard for the structure of policies, requests and responses, and a set of rule-combining algorithms, which specify how multiple rules within a policy should be evaluated to reach an authorization decision. When a user tries to execute an action on a resource, an XACML request is sent to the PDP. The request specifies attributes for the subject, the resource and the action involved; attributes for environmental variables can also be included. Requests are sent to the PDP, which evaluates them against the policies and takes authorization decisions (*Permit*, *Deny*, *NotApplicable*, or *Indeterminate*), issuing an XML response. Figures 2 and 3 schematize how XACML policies and requests are structured:

2.2 Leveraging LLMs in Cybersecurity

Research interest on the topic of using LLMs in cybersecurity has been growing constantly, yet at the time of writing we are not aware of any work on studies on automatic XACML policy generation from natural language. in (Rubio-Medrano et al., 2024) authors start from the evidence that LLMs can excel at producing code, but not at ensuring security specifications, and propose a co-operative framework in which LLMs and skilled human developers are in-

[2]https://www.oasis-open.org/standard/xacmlv3-0/

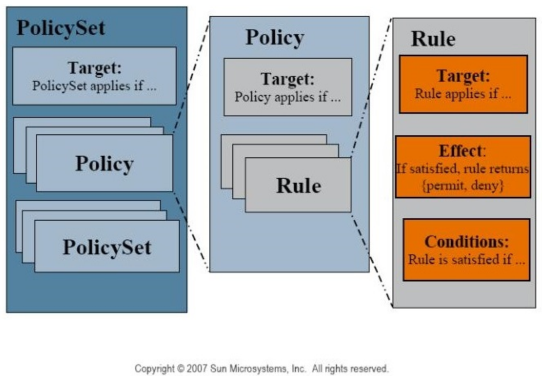


Figure 2: The structure of an XACML policy.

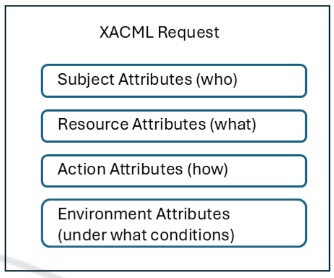

XACML Request
Subject Attributes (who)
Resource Attributes (what)
Action Attributes (how)
Environment Attributes (under what conditions)

Figure 3: Elements of an XACML request.

volved to create SW applications compliant with security principles. In (Narouei et al., 2020; Hassanin and Moustafa, 2024), the potential of using LLMs in cybersecurity is highlighted, and more specifically, it is pointed out that the analytical capabilities of LLMs with respect to large amounts of textual data make them an excellent ally for automating the detection and prediction of attacks through the processing of system logs and network traffic. Prevention of scams and phishing can be achieved through the analysis of emails, instant messages, social media posts, etc. Advantages and drawbacks of using artificial intelligence in cybersecurity are further explored in (Michael et al., 2023). In (Subramaniam and Krishnan, 2024) , authors exploit the analytical power of LLMs over natural language to automatically generate database access control primitives. LLMs can be used to automatically analyze and generate documents, assisting experts with tasks like assessing and reporting an organization's compliance with specific regulations (such as the European GDPR) and creating the necessary documentation. LLMs' generative skills can also be leveraged to make legal and administrative policies of an organization accessible and comprehensible to a wider audience (Goknil et al., 2024).

3 USING LLMS TO CREATE XACML RULES

To address RQ1 presented in the introduction (”Is it possible to use LLMs to generate XACML policies?”), it is essential to understand the generative capabilities of LLMs in dealing with XACML rules.

Getting domain-specific content from LLMs is not straightforward; LLMs learn to recognize and generate text patterns through neural networks, which are pre-trained on a huge amount of heterogeneous, non-specialized documents. These models are not able to perform tasks such as syntactic validation or code execution; therefore, in order to reply on RQ1 and to obtain correct code snippets or text that comply with domain-specific vocabulary or formats and/or refer to specific knowledge, fine-tuning or prompt engineering techniques must be considered.

3.1 Fine-Tuning

Fine-tuning offers a solution to optimize the performance of a pre-trained LLM for a specific domain. However, achieving satisfactory results is challenging, time-consuming, and resource-consuming. In fact, high-quality, accurate, bias-free, diverse data are needed to handle several situations and use cases, and computational power and ad hoc infrastructures (such as powerful GPUs or TPUs) must be employed to manage large datasets and complex algorithmic models. Skilled developers must be involved in the development process; typically, expertise is required in Python frameworks such as PyTorch[3] and TensorFlow[4], along with dedicated libraries such as Hugging Face Transformers [5].

To ensure the reliability of the results, fine-tuned models' performances must be validated through intensive human inspection of their replies, and, possibly, further, fine-tuning must be performed. A fine-tuned model on a specific domain produces accurate answers to even very specific queries, which is important in fields such as forensics or medicine.

3.2 Prompt Engineering

Prompt engineering (White et al., 2023), is the practice of designing structured and organized instructions and queries (prompts) to guide an LLM towards desired responses. This practice is extremely popular in scenarios in which generative AI is used, since well-crafted prompts yield more accurate, coherent, and relevant responses, improving the model's performance for domain-specific tasks. Many prompting patterns (or strategies) exist, which can be applied to solve a wide range of issues encountered when interacting with an LLM. In analogy to programming patterns, prompting patterns can be used in different domains to guide an LLM towards the expected outcome.

Chain of Thought (CoT) prompting is a strategy that leverages the power of language models by prompting them with explicit instructions to decompose complex tasks, thus eliciting intermediate reasoning steps, and enhancing their ability to tackle intricate problems. This strategy is particularly valuable in any domain where the extraction of structured information is needed [(Vijayan, 2023; Goknil et al., 2024)]. Additionally, step-by-step reasoning enables continuous result verification and potential prompt refinement. In the XACML specification domain, prompting techniques have been adopted in (Subramaniam and Krishnan, 2024) to obtain database access control primitives for policies, automatically synthesized from natural language specifications. Also, a dataset [6] has been gathered, containing 956 optimized XACML questions that can be used to craft ad-hoc prompts.

While programming skills are not strictly required, expertise in the application domain is needed, as generated content must be validated to assess the efficacy of the prompts. Prompt engineering allows dynamic adaptation of the model's behavior without changing the underlying model and can be proficiently used to reduce or even eliminate the need for fine-tuning, making it a perfect solution for situations in which large domain-specific datasets are not available. Since effective prompts can be crafted by non-experts, prompt engineering offers a lightweight, valuable alternative to fine-tuning—especially when skilled professionals are unavailable or when computational resources and budgets are limited.

3.3 Response to RQ1

Generating XACML access policies from natural language specifications via an LLM is certainly possible, but it is important to consider that implementing such an approach is not straightforward. Prompt engineering and fine-tuning are well-established techniques that should be considered.

[3]https://pytorch.org/

[4]https://www.tensorflow.org/

[5]https://huggingface.co/docs/transformers/v4.17.0/en/index

[6]XACML Dataset available at https://artofservice.com.au/xacml-dataset/

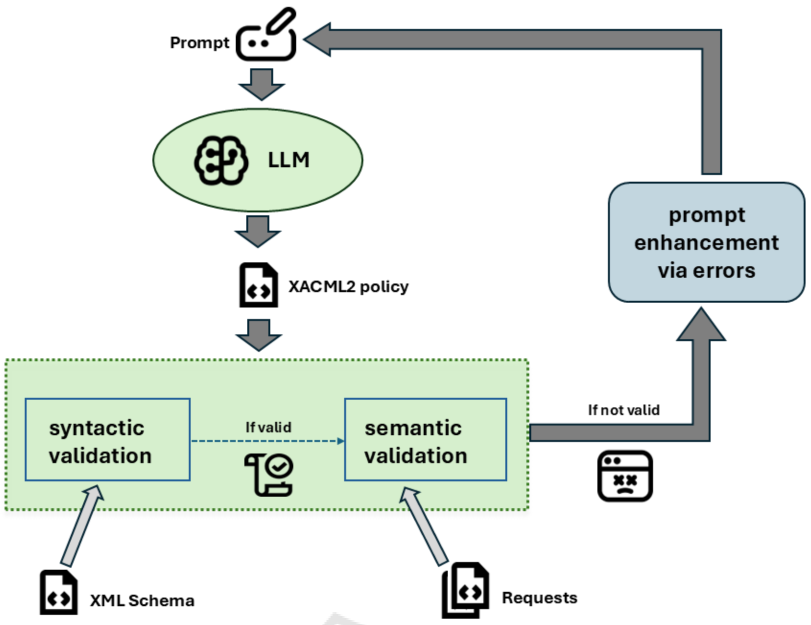


Figure 4: Prompt refinement through syntactic and semantic validation.

4 A PIPELINE FOR XACML POLICIES GENERATION

In this section we aim to address the RQ2 mentioned in the introduction, ("Are there low-cost strategies to exploit the potential of LLMs for the purpose of producing XACML policies?"). A potential low-cost solution for overcoming the limitations of LLMs in producing structured formats is to iteratively refine prompts. This can be achieved by performing semantic and syntactic validation on the generated results. We will describe an experiment in which we performed prompt refinement through cycles of semantic and syntactic validation to obtain access policies that were as accurate as possible.

As schematized in Figure 4, the following process is adopted: four popular conversational LLMs, namely ChatGPT (GPT-4o), Claude (3.5 Haiku), Gemini (1.5 Flash), and LLaMA 3 were assessed according to their capability to provide syntactically correct XACML policies. Six policy descriptions were taken from the WSO2 Identity Server official documentation [7], toghether with their XACML2 translations, associated requests and the corresponding responses. These documents were used for validation purposes according to the procedural schema shown in Figure 4. Details of each of the six policies are provided in the Appendix.

[7]https://is.docs.wso2.com/en/5.9.0/learn/writing-xacml2.0-policies-in-wso2-identity-server/

For evaluating the four LLMs, a random selection of five of the six aforementioned sample policies was considered. The policy identified as Policy 1 [8] was excluded from this experiment to be used in the unbiased final assessment of the overall process.

In the next sections, details of the experiment performed to evaluate the feasibility of the are provided.

4.1 Improving the Syntactical Correctness

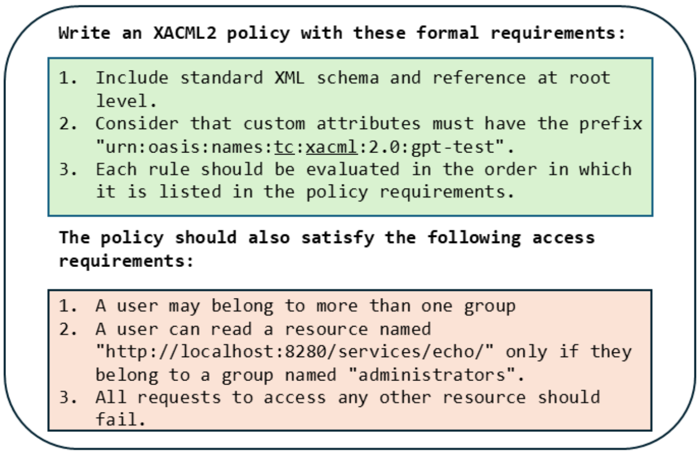
Each of the four LLMs (ChatGPT, Claude, Gemini, and LLaMA) was assessed according to its capability to produce a syntactically correct XACML policy starting from an NL description.

For the syntactic validation, Java built-in XML validation capabilities (namely, the javax.xml.validation package) were used.

To refine the schema in Figure 4, multiple efforts were made to understand the format and the necessary information for the procedural sequence of execution, while aiming to avoid bias.

Specifically, a prompt template was crafted to inject useful information to reduce the most common syntactic errors. Figure 5 shows the template structure. The text contains two distinct lists of statements: one addressing formal requirements and the other outlining access rules. Specifically, (green upper part of

[8]https://is.docs.wso2.com/en/5.9.0/learn/xacml-2.0-sample-policy-1/



```
Write an XACML2 policy with these formal requirements:

1. Include standard XML schema and reference at root
   level.
2. Consider that custom attributes must have the prefix
   "urn:oasis:names:tc:xacml:2.0:gpt-test".
3. Each rule should be evaluated in the order in which
   it is listed in the policy requirements.

The policy should also satisfy the following access
requirements:

1. A user may belong to more than one group
2. A user can read a resource named
   "http://localhost:8280/services/echo/" only if they
   belong to a group named "administrators".
3. All requests to access any other resource should
   fail.
```

Figure 5: Template for the first prompt.

Figure 5) an XML Schema statement was included, as LLMs often overlook it, resulting in the generation of documents affected by a `missing grammar` error. A prefix for custom attributes was also specified to avoid arbitrary, potentially misleading names. Finally, the evaluation order of the rules was detailed to improve the quality of translation into XACML2. In particular, the *rule-combining algorithm*, i.e., a directive for the access control engine, which defines how to take an authorization decision given a set of rules, was specified.

In the second red bottom section of Figure 5, specific access requirements for the WSO2 Identity Server are outlined. While this domain-specific information could potentially be omitted, it can help accelerate the LLM generation process and reduce the risk of misinterpretation errors.

As in Figure 4, in the first iterations, the prompt template and the NL description were provided as input to an LLM to be translated into XACML2 specification language. The obtained XACML2 policy was then syntactically assessed, and the validation results were returned to the LLM for improvement. The process was repeated iteratively for a valuable outcome. In the experiment for each of the four LLMs, after the first round of executions, several errors were collected and prompted back through the template to force LLM to rewrite the policy to fix it.

After the first iteration, ChatGPT and Gemini often responded by rewriting only the part of the policy affected by the error. To overcome the issue, the template was explicitly modified to ask for the rewrite of the entire policy ("please rewrite the whole policy"). Additionally, quite randomly, the XML schema declaration was completely or partially lost, in which cases it was necessary to prompt the model to reintroduce it ("Please add missing xsi:schemaLocation"). Figures 6 and 7 show the structure of a prompt in the nth step of the validation cycle, and how it is obtained.

Performing the experiment on the four LLMs, after three or four iterations with prompt refinement, both Claude and ChatGPT produced policies that were formally correct (i.e., without validation errors), demonstrating their capability for progressive improvement in results. In contrast, Gemini and LLaMA 3 generated outputs that did not converge to an error-free document, even after six iterations. Not only were validation errors unresolved across the entire policy, but new formal errors—such as the use of non-compliant or arbitrarily named elements—were frequently introduced. Therefore, we excluded Gemini and LLaMA 3 from our next trials. Since Claude produced syntactically correct results in the shortest time, it was adopted to continue our trials.

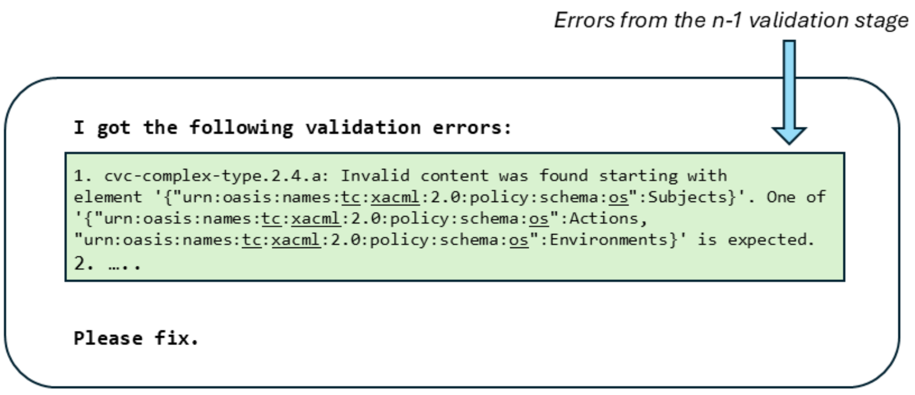
4.2 Improving the Semantic Correctness

Similar to the syntactic validation process, a strategy of successive prompt-refinement cycles based on detected errors was adopted. As in Figure 4, the input of this second experiment are: i) the result of the syntactic validation step, i.e., Claude XACML2 translation of the five NL specifications of the policies of the WSO2 Identity Server. For clarity, we label this policy as *XACML_AI* policies. ii) the available set of WSO2 Identity Server requests and corresponding responses (called *Original_req* and *Original_resp* respectively) for each of the five policies.

For the semantic validation, the tool Balana[9] has been used. This is an open-source PDP that can be easily used via prompt execution.

As in Figure 4, the XACML_AI policy is uploaded to the Balana's PAP (see Section 2 and one by one

[9]https://github.com/wso2/balana

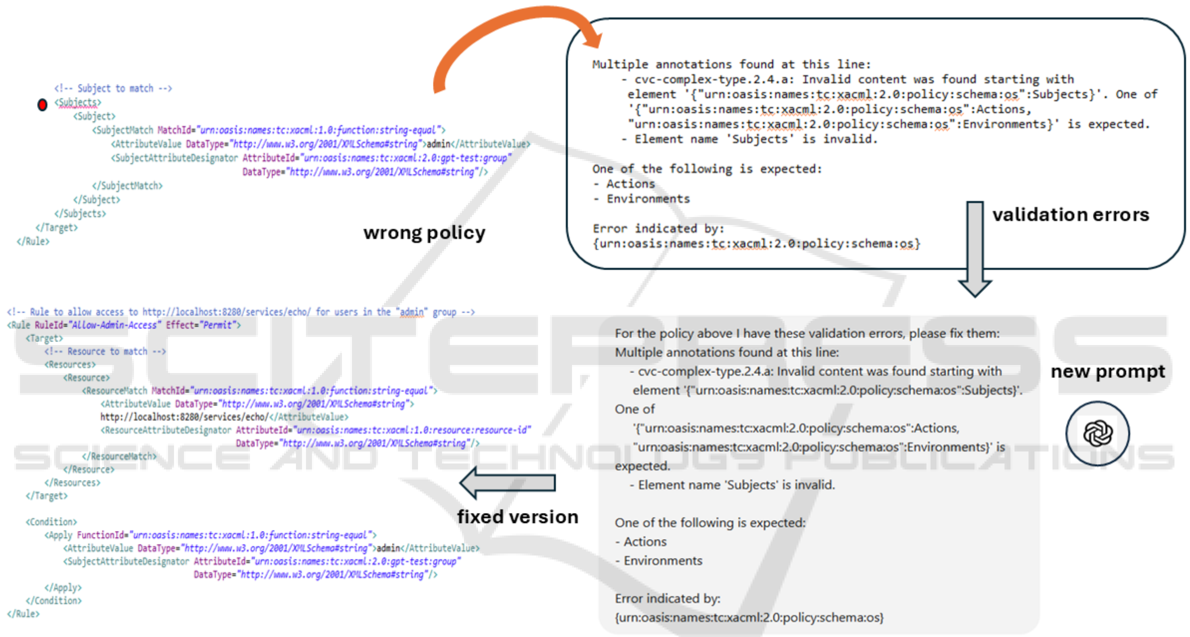

Errors from the n-1 validation stage

```
I got the following validation errors:

1. cvc-complex-type.2.4.a: Invalid content was found starting with
element '{"urn:oasis:names:tc:xacml:2.0:policy:schema:os":Subjects}'. One of
'{"urn:oasis:names:tc:xacml:2.0:policy:schema:os":Actions,
"urn:oasis:names:tc:xacml:2.0:policy:schema:os":Environments}' is expected.
2. …..

Please fix.
```

Figure 6: Template for a prompt at stage #n.



```
<!-- Subject to match -->
<Subjects>
    <Subject>
        <SubjectMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">admin</AttributeValue>
            <SubjectAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:2.0:gpt-test:group"
                                        DataType="http://www.w3.org/2001/XMLSchema#string"/>
        </SubjectMatch>
    </Subject>
</Subjects>
</Target>
</Rule>
```

```
Multiple annotations found at this line:
    - cvc-complex-type.2.4.a: Invalid content was found starting with
     element '{"urn:oasis:names:tc:xacml:2.0:policy:schema:os":Subjects}'. One of
     '{"urn:oasis:names:tc:xacml:2.0:policy:schema:os":Actions,
     "urn:oasis:names:tc:xacml:2.0:policy:schema:os":Environments}' is expected.
    - Element name 'Subjects' is invalid.

One of the following is expected:
- Actions
- Environments

Error indicated by:
{urn:oasis:names:tc:xacml:2.0:policy:schema:os}
```

```
For the policy above I have these validation errors, please fix them:
Multiple annotations found at this line:
   - cvc-complex-type.2.4.a: Invalid content was found starting with
   element '{"urn:oasis:names:tc:xacml:2.0:policy:schema:os":Subjects}'.
One of
   '{"urn:oasis:names:tc:xacml:2.0:policy:schema:os":Actions,
   "urn:oasis:names:tc:xacml:2.0:policy:schema:os":Environments}' is
expected.
   - Element name 'Subjects' is invalid.

One of the following is expected:
- Actions
- Environments

Error indicated by:
{urn:oasis:names:tc:xacml:2.0:policy:schema:os}
```

```
<!-- Rule to allow access to http://localhost:8280/services/echo/ for users in the "admin" group -->
<Rule RuleId="Allow-Admin-Access" Effect="Permit">
   <Target>
      <!-- Resource to match -->
      <Resources>
         <Resource>
            <ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
               <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
               http://localhost:8280/services/echo/</AttributeValue>
               <ResourceAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"
                                            DataType="http://www.w3.org/2001/XMLSchema#string"/>
            </ResourceMatch>
         </Resource>
      </Resources>
   </Target>

   <Condition>
      <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
         <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">admin</AttributeValue>
         <SubjectAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:2.0:gpt-test:group"
                                     DataType="http://www.w3.org/2001/XMLSchema#string"/>
      </Apply>
   </Condition>
</Rule>
```

Figure 7: Prompt refinement through cyclic validation.

the *Origial_req* provided to obtain the corresponding response (called *AI_resp*)

The *AI_resp* is then compared with the *Original_resp* to check mismatches. As in the previous experiment, in case of error, the obtained validation results are given back to Claude for improvement (*prompt enhancement via errors* in Figure 4). The process repeats iteratively for a valuable outcome.

However, to implement the correct access control concept, several rephrases of the prompt have been necessary. For instance, for the hierarchical access controls through the use of the *string-bag* and *string-subset* functions, different refinements have been performed to clearly provide the concepts of set and subset of roles. The same for transmitting the concept that a subject can belong to one subset, to more than one, or to none.

Unlike the syntactic validation phase, defining the adjustments to be provided for the next iteration was more difficult. The errors raised by Balana can be challenging to understand, and identifying their underlying causes requires thorough investigation. Indeed, when the *AI_resp* was different from the *Original_resp*, manual analysis was necessary to understand the semantic difference between the XACML_AI policy and its original NL specification.

In this stage-specific attention was devoted to solving possible ambiguities or a trivial error introduced during the various iterations and prompt activity. For instance, the “group” attribute renamed as “groups”, or the statement *“any user should access”* interpreted as *"a user named "any" should access"*.

As a final result, Claude, in a number of iterations ranging from 9 to 11, was able to derive a final set of XACML_AI policy semantic equivalent to the NL specification according to the request and response executed.

It is important to highlight that this condition is necessary but not sufficient for establishing the semantic equivalence of the XACML_AI policy to its natural language specification: A final validation from XACML experts was necessary. Nonetheless, the proposed procedural step lets Claude produce syntactically correct XACML2 policies, which can be validated by the Balana PDP.

4.3 Validation of the Proposed Strategy

The last step of the process presented in Section 4 focuses on the use of Claude, leveraged through the proposed prompt refinement, to generate an XACML policy. To avoid bias, the provided NL specification for Claude is Policy 1, which is the only one excluded from the prompt refinement process. To assess the performance of Claude, the derived XACML_AI Policy has been compared by an XACML expert with policy specification provided by the WSO2 Identity Server (called sample_policy).

In Figures 8 and 9, the two major differences have been heightened. In particular, Figure 8 highlights that the two policies use two different string comparison approaches: the *sample_policy* verifies that a string is equal to a regular expression (`string-regexp-match`) while the XACML_AI Policy checks if the string is exactly the same `string-equal`). In Figure 9 the two policies use two different comparison approaches for string sets: the *sample_policy* verifies if set A can be a subset of a set B, (`string-subset-match`) while the XACML_AI Policy checks if the set A and B have at list a comment element `string-at-least-one-member-of`).

Although there may be requests where different methodologies for managing string values yield varying responses, the level of detail outlined in the NL specification of Policy 1 suggests that the two policies can be considered semantically equivalent. Indeed, for additional confirmation, both textitsample_policy and XACML_AI Policy have been verified using the *Original_req* set available for Police 1, obtaining in both cases the same*Original_resp* response.

This final validation confirms that the outline in Figure 4 provides a response to RQ2 presented in the introduction by defining an iterative strategy to leverage LLM's ability to generate XACML policies. It also provides a baseline for prompting improvements and an opportunity to build a set of best practices for future applications.

Figure 10 summarizes the approach we followed and shows how human intervention was integrated.

4.4 Response to RQ2

A low-cost solution to exploit LLMs to generate XACML policies is possible, but a proper model must be chosen, as not all models are suitable for structured text production. A possible low-cost solution should rely on prompt engineering supported by external validation tools.

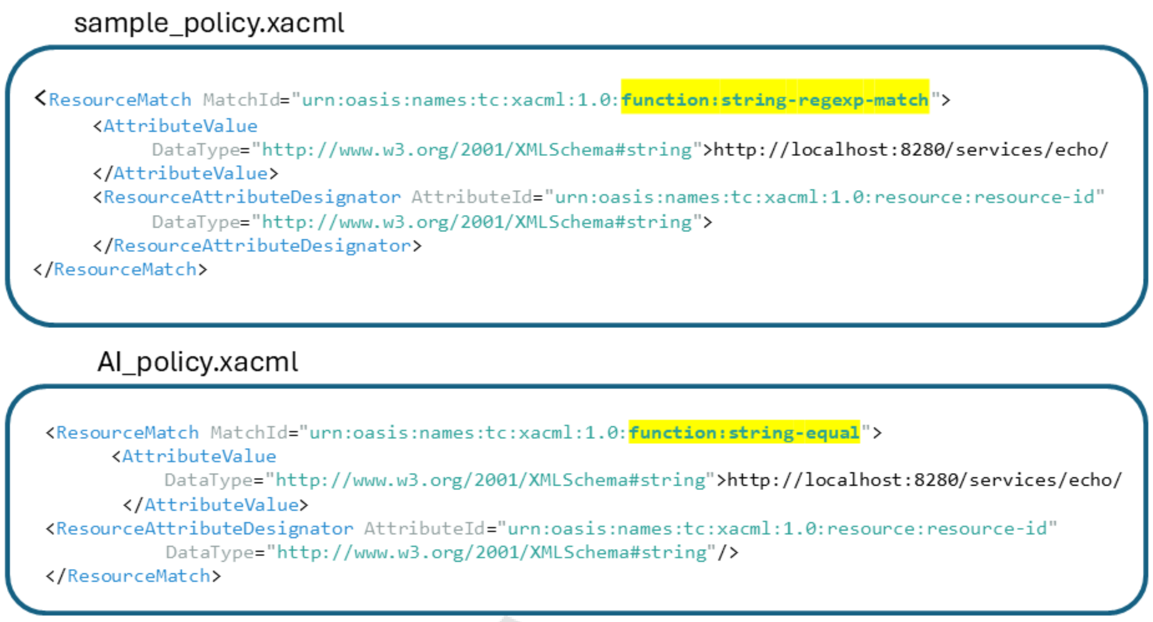
5 CONCLUSIONS AND FUTURE WORK

Starting from the premise that creating access policies can be very costly in terms of time and resources, the paper investigated the possibility of leveraging LLMs to generate XACML access policies from natural language specifications. Since LLMs have inherent limitations in producing structured text, but fine-tuning strategies are too costly and time-consuming to overcome this issue, the paper investigated the possibility of using customized prompt-engineering techniques. It took advantage of errors in policy validation and their fixes to enhance the performance of LLMs in generating syntactically and semantically correct policies.

The paper assessed the performance of four widely used LLMs: ChatGPT, Claude, Gemini, and LLaMA. This assessment was carried out using the XACML documentation related to six access policies from the WSO2 Identity Server. Specifically, five of these policy specifications were used for prompt engineering of the LLMs, while the sixth policy was used to validate the performance of the LLMs in generating policies. The experiment carried out showed that not all LLMs were equally suitable for the purpose. Claude and ChatGPT, outperformed Gemini and LLaMA in generating syntactically correct XACML2 policies through prompt refinement.

From a technical point of view, the experiment has been performed using a pipeline in which syntactic and semantic validators are used in combination, with the help of a prompt command interface. Even if requiring some manual interventions, the proposed pipeline showed the feasibility of the proposed approach and provided a suggestion for a future framework capable of fully automating the process.

Future work will focus on reducing human intervention by developing an agent which leverages the



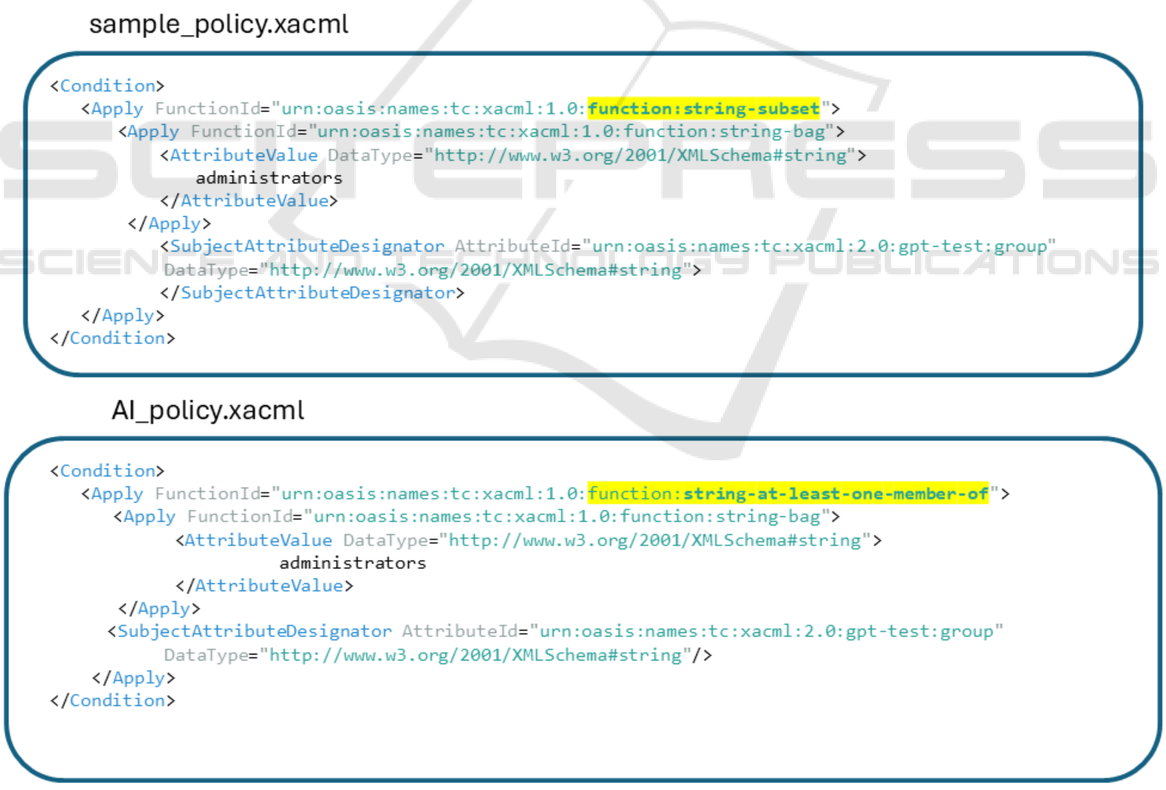
sample_policy.xacml

```
<ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-regexp-match">
      <AttributeValue
            DataType="http://www.w3.org/2001/XMLSchema#string">http://localhost:8280/services/echo/
      </AttributeValue>
      <ResourceAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"
            DataType="http://www.w3.org/2001/XMLSchema#string">
      </ResourceAttributeDesignator>
</ResourceMatch>
```

AI_policy.xacml

```
<ResourceMatch MatchId="urn:oasis:names:tc:xacml:1.0:function:string-equal">
      <AttributeValue
            DataType="http://www.w3.org/2001/XMLSchema#string">http://localhost:8280/services/echo/
       </AttributeValue>
<ResourceAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"
            DataType="http://www.w3.org/2001/XMLSchema#string"/>
</ResourceMatch>
```

Figure 8: Different string comparison methods.



sample_policy.xacml

```
<Condition>
   <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-subset">
      <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-bag">
          <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
              administrators
          </AttributeValue>
      </Apply>
          <SubjectAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:2.0:gpt-test:group"
           DataType="http://www.w3.org/2001/XMLSchema#string">
          </SubjectAttributeDesignator>
   </Apply>
</Condition>
```

AI_policy.xacml

```
<Condition>
   <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-at-least-one-member-of">
      <Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-bag">
            <AttributeValue DataType="http://www.w3.org/2001/XMLSchema#string">
                    administrators
            </AttributeValue>
      </Apply>
     <SubjectAttributeDesignator AttributeId="urn:oasis:names:tc:xacml:2.0:gpt-test:group"
           DataType="http://www.w3.org/2001/XMLSchema#string"/>
   </Apply>
</Condition>
```

Figure 9: Different comparison methods for string sets.

power of large LLMs to operate autonomously using advanced decision-making capabilities. To achieve this, we aim to automate the iterative refinement of prompts through semantic and syntactic validation of the policies generated by the LLM. An additional task that we also plan to automate using AI-based methods is the evaluation of the effective correspondence between the generated policy and the initial user requirements.

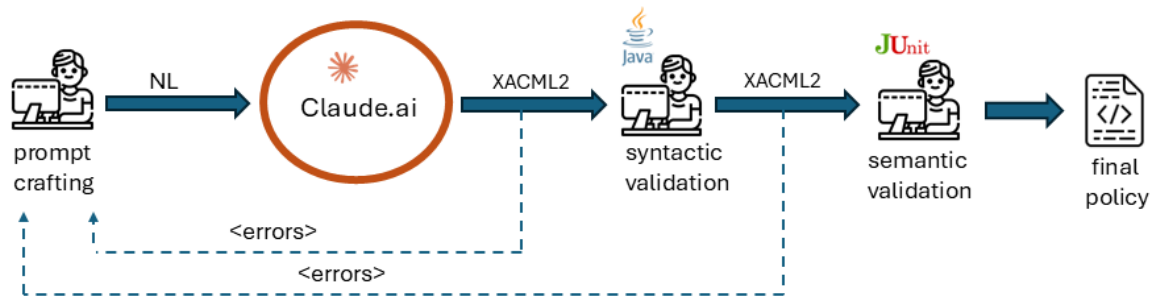


Figure 10: An outline of the final workflow.

ACKNOWLEDGEMENTS

This work was partially supported by the project RESTART (PE00000001), and the project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NextGenerationEU.

REFERENCES

Brodie, C., Karat, C.-M., and Karat, J. (2006). An empirical study of natural language parsing of privacy policy rules using the sparcle policy workbench. In *Symposium On Usable Privacy and Security*.

Buscemi, A. (2023). A comparative study of code generation using chatgpt 3.5 across 10 programming languages. *ArXiv*, abs/2308.04477.

Coyne, E. and Weil, T. R. (2013). ABAC and RBAC: scalable, flexible, and auditable access management. *IT Prof.*, 15(3):14–16.

Goknil, A., Gelderblom, F. B., Tverdal, S., Tokas, S., and Song, H. (2024). Privacy policy analysis through prompt engineering for llms. *ArXiv*, abs/2409.14879.

Hassanin, M. and Moustafa, N. (2024). A comprehensive overview of large language models (llms) for cyber defences: Opportunities and directions. *ArXiv*, abs/2405.14487.

Jin, Y., Sorley, T., O'Brien, S., and Reyes, J. (2014). Implementation of XACML role-based access control specification. *Int. J. Comput. Their Appl.*, 21(1):62–69.

Kumar, V., Srivastava, P., Dwivedi, A., Budhiraja, I., Ghosh, D., Goyal, V., and Arora, R. (2023). Large-language-models (llm)-based ai chatbots: Architecture, in-depth analysis and their performance evaluation. In *International Conference on Recent Trends in Image Processing and Pattern Recognition*.

Liu, Y., Li, D., Wang, K., Xiong, Z., Shi, F., Wang, J., Li, B., and Hang, B. (2024). Are llms good at structured outputs? a benchmark for evaluating structured output capabilities in llms. *Inf. Process. Manag.*, 61:103809.

Michael, K., Abbas, R., and Roussos, G. (2023). Ai in cybersecurity: The paradox. *IEEE Transactions on Technology and Society*.

Narouei, M., Takabi, H., and Nielsen, R. D. (2020). Automatic extraction of access control policies from natural language documents. *IEEE Transactions on Dependable and Secure Computing*, 17:506–517.

Rubio-Medrano, C. E., Kotak, A., Wang, W., and Sohr, K. (2024). Pairing human and artificial intelligence: Enforcing access control policies with llms and formal specifications. *Proceedings of the 29th ACM Symposium on Access Control Models and Technologies*.

Siam, M. K., Gu, H., and Cheng, J. Q. (2024). Programming with ai: Evaluating chatgpt, gemini, alphacode, and github copilot for programmers.

Slankas, J., Xiao, X., Williams, L. A., and Xie, T. (2014). Relation extraction for inferring access control rules from natural language artifacts. *Proceedings of the 30th Annual Computer Security Applications Conference*.

Subramaniam, P. and Krishnan, S. (2024). Intent-based access control: Using llms to intelligently manage access control. *arXiv preprint arXiv:2402.07332*.

Vijayan, A. (2023). A prompt engineering approach for structured data extraction from unstructured text using conversational llms. *Proceedings of the 2023 6th International Conference on Algorithms, Computing and Artificial Intelligence*.

White, J., Fu, Q., Hays, S., Sandborn, M., Olea, C., Gilbert, H., Elnashar, A., Spencer-Smith, J., and Schmidt, D. C. (2023). A prompt pattern catalog to enhance prompt engineering with chatgpt. *ArXiv*, abs/2302.11382.

Zhong, L. and Wang, Z. (2023). Can llm replace stack overflow? a study on robustness and reliability of large language model code generation. In *AAAI Conference on Artificial Intelligence*.

SAMPLE AUTHORIZATION REQUIREMENTS AND REQUESTS

Policy 1 (P1): *Authorization Requirements:* The resource http://localhost:8280/services/echo/ can be read only by users belonging to the administrators group. Any other operation or all requests to access any other resource should fail.

Request 1 (Permit): user which belongs only to the administrators group requires to read a the http://localhost:8280/services/echo/ resource.
Request 2 (Permit): User admin, which belongs to the admin group and the business group, attempts to read the http://localhost:8280/services/echo/ resource.
Request 3 (Deny): User admin, which belongs to the administartors group, attempts to read the http://localhost:8280/services/test/ resource.
Request 4 (Deny): User admin, which belongs to the business group, attempts to read the http://localhost:8280/services/echo/ resource.

Policy 2 (P2): *Authorization Requirements:*

1. The operation `getCustomers` in the service http://localhost:8280/services/Customers should only be accessed by users belonging to the `admin_customers` group.
2. The operation `getEmployees` in the service http://localhost:8280/services/Customers should only be accessed by users belonging to the `admin_emps` group.
3. Requests to any other service or operation should fail.

Request 1 (Permit): A subject which belongs only to the admin_customers group requires to access (read) the operation $http://localhost:8280/services/Customers/getCustomers$
Request 2 (Permit): A subject which belongs to the admin_emps group requires to access (read) the operation $http://localhost:8280/services/Customers/getEmployees$
Request 3 (Deny): A subject which belongs to the admin_emps group requires to access (read) the operation $http://localhost:8280/services/Customers/getUsers$
Request 4 (Deny): A subject which belongs only to the admin_emps group requires to access (read) the operation $http://localhost:8280/services/Customers/getCustomers$

Policy 3 (P3): *Authorization Requirements:* The operation getEmployees in the service $http://localhost:8280/services/Customers$ should only be accessed (read) by users belonging to both the $admin_emps$ and admin groups. If the user belongs to a group other than $admin_emps$ or admin, the request should fail. Requests to any other service or operation should fail.

Request 1 (Permit): A subject which belongs to both 'admin_emps' and 'admin' groups attempts to access (read) the endpoint $http://localhost:8280/services/Customers/getEmployees$
Request 2 (Permit): A subject which belongs to the admin and the $admin_emps$ groups attempts to access (read) the endpoint $http://localhost:8280/services/Customers/getEmployees$
Request 3 (Deny:) A subject belonging to the admin and $admin_emps$ groups requests to perform a write action on the on the URI $http://localhost:8280/services/Customers/getEmployees$
Request 4 (Deny): A subject belonging to the groups simpleuser and $admin_emps$ requires access (read) to the endpoint $http://localhost:8280/services/Customers/getEmployees$

Policy 4 (P4): *Authorization Requirements:*

1. The operation `getEmployees` in the service http://localhost:8280/services/Customers should only be accessed (read) by users belonging to the group(s) `admin_emps` and/or `admin` .
2. Requests to any other service or operation should fail.

Request 1 (Permit): A subject which belongs to both *admin_emps* and 'admin' groups attempts to access (read) the endpoint $http://localhost:8280/services/Customers/getEmployees$
Request 2 (Permit): A subject which belongs to the *admin_emps* group and another group attempts to access (read) the endpoint $http://localhost:8280/services/Customers/getEmployees$
Request 3 (Deny): A subject which belongs to the *admin_emps* group and another group attempts to access (read) the endpoint $http://localhost:8280/services/Customers/getUsers$
Request 4 (Deny): A subject which does not belongs neither to ht *admin_emps* nor to the 'admin' group attempts to access (read) the endpoint $http://localhost:8280/services/Customers/getEmployees$

Policy 5 (P5): *Authorization Requirements:*

1. The operation getEmployees in the service $http://localhost:8280/services/Customers$ should only be accessed (read) by users belonging to the group(s) $admin_emps$ and/or admin.
2. Requests to any other service or operation should fail, with the following exception: Users admin1 and admin2 should be able to access any resource, irrespective of their role.

Request 1 (Permit): A subject which belongs to both the admin and admin_emps groups attempts

to access (read) the endpoint $http://localhost:8280/services/Customers/getEmployees$
Request 2 (Permit): A subject whose id is admin1, which belong neither to the admin nor to the admin_emps group attempts to access (read) the endpoint $http://localhost:8280/services/Customers/getWhatever$
Request 3 (Deny): A subject whose id is admin, annd which does not belong neither to the admin nor to the admin_emps group attempts to access (read) the $http://localhost:8280/services/Customers/getEmployees$ endpoint
Request 4 (Deny): A subject belonging to the admin_emps group attemprs to access (read) the endpoint $http://localhost:8280/services/Customers/getAdmins$

Policy 6 (P6): *Authorization Requirements:*

1. The operations `getVersion1` and `getVersion2` in the service $http://localhost:8280/services/Customers$ should be accessed (read) by any user.
2. Requests to any other service or operation should only be accessed (read) by users belonging to the group(s) `admin_emps` and/or `admin`.

Request 1 (Permit): A subject which belongs to the *admin_emps* and the another_group group requires to access (read) the following: $http://localhost:8280/services/OtherService/someOperation$.
Request 2 (Permit): A subject belonging to the group_x group attempts to access (read) the URI $http://localhost:8280/services/Customers/getVersion2$.
Request 3 (Deny): A subject which does not belong to any group requires to access (read) the URI $http://localhost:8280/services/Customers/someOp$.
Request 4 (Deny): A subject which belongs to group_X attempts an access (read) to $http://localhost:8280/services/Customers/getVersion3$.