VNU Journal of Science: Com. Science & Com. Eng., Vol. 42, No.1 (2026) 1–11

VNU Journal of Science:
Computer Science and Communication Engineering

Journal homepage: http://www.jcsce.vnu.edu.vn/index.php/jcsce

Original Article

Automatic Test Case Generation for XACML Access Control Policies Using Graph-Based Modeling and Genetic Algorithms

Thanh-Binh Trinh[1], Cuong-Nguyen Van[1], Ngoc-Minh Le[2], Dac-Nhuong Le[2]

[1] *Phenikaa University, Nguyen Van Trac Street, Yen Nghia, Ha Dong, Hanoi, Vietnam*

[2] *Haiphong University, Phan Dang Luu Street, Kien An, Haiphong, Vietnam*

Received 03rd February 2026
Revised 9th March 2026; Accepted 26th March 2026

Abstract: XACML (eXtensible Access Control Markup Language) is a widely used standard for defining attribute-based access control (ABAC) policies in cloud and service-oriented systems. However, validating XACML policies is challenging due to their hierarchical structure and flexibility, which often introduce issues such as redundant rules and logical conflicts. Manual test case design is time-consuming and frequently fails to achieve sufficient coverage. To address these challenges, this paper proposes an approach for generating and optimizing test cases from XACML access control policies. We implement this approach as a tool that transforms XACML policies into an XACML Flow Graph (XFG) and systematically derives test cases through graph traversal. A genetic algorithm is then applied to optimize the generated test suite while preserving high coverage. Experimental results on standard XACML policy sets show that the proposed approach significantly reduces test suite size while maintaining effective coverage, making it suitable for practical policy testing and continuous assurance.

Keywords: XACML, Test Case Generation, Graph-Based Modeling, Genetic Algorithm, Policy Verification, ABAC

1. Introduction

XACML (eXtensible Access Control Markup Language) has emerged as a widely adopted standard for specifying attribute-based access control (ABAC) policies, allowing authorization decisions to be defined in terms of subjects, resources, actions, and environmental attributes

Testing is essential for validating access control policies prior to deployment. Nevertheless, XACML policy testing is still largely manual or ad-hoc in practice, requiring substantial effort and expertise

This paper addresses these challenges by proposing an approach for generating and optimizing test cases for XACML access control policies. The approach systematically derives test cases from a structural representation of

∗ Corresponding author.
E-mail address: binh.trinhthanh@phenikaa-uni.edu.vn

https://doi.org/10.25073/2588-1086/vnucsce.6926

policies and applies test suite optimization to reduce redundancy while preserving high coverage. The proposed approach is implemented as an automated tool and evaluated on standard XACML policy sets, demonstrating its effectiveness and practical applicability. The main contributions of this paper are summarized as follows:

- A structured graph-based representation of XACML policies to support systematic testing.
- A DFS-based test case generation method combined with a genetic algorithm (GA) for test suite optimization.
- An empirical evaluation demonstrating the applicability of the approach to realistic XACML policies.

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 presents the proposed approach. Section 4 reports the experimental results. Finally, Section 5 concludes the paper and outlines directions for future works.

2. Related Work

The complexity and security-critical of XACML access control policies have motivated extensive research on policy analysis and testing

Static analysis methods aim to detect policy anomalies such as redundancies, conflicts, and inconsistencies without executing policies. Several studies have proposed rule-based and formal techniques to analyze conflicts caused by rule combining algorithms or overlapping policy scopes. Alves *et al.*

Mutation testing has been explored to assess the effectiveness of XACML test suites

Graph-based approaches model XACML policies as control-flow or decision graphs to enable automated test case generation

To address test redundancy, metaheuristic optimization techniques, particularly genetic algorithms, have been extensively studied for test suite reduction in software testing

In contrast, this paper combines graph-based modeling with evolutionary optimization to address both systematic test generation and test suite scalability. By modeling XACML policies as an XACML Flow Graph (XFG), generating test cases via depth-first traversal, and optimizing them using a genetic algorithm, the proposed approach achieves high coverage with reduced testing effort.

3. Proposed Method

Figure 1 presents the proposed approach for automated test case generation for XACML access control policies. The approach includes four main steps: (i) XACML policies are transformed into an XACML Flow Graph (XFG) that represents rules, conditions, and decision paths; (ii) an initial test suite is systematically generated by traversing the XFG to cover feasible policy execution paths; (iii) the generated test suite is optimized using a genetic algorithm (GA) with coverage-driven fitness functions targeting rule, condition, and decision coverage; and (iv) the optimized test suite is executed against the policy decision point within a DevSecOps pipeline, where test outcomes are evaluated using a policy-based test oracle and analyzed to provide feedback for policy refinement.

The following sections describe in detail the construction of the XFG, the test case generation process, the GA-based optimization, and the integration of the approach into DevSecOps workflows.

3.1. XACML Flow Graph Construction

This section introduces the *XACML Flow Graph* (XFG) as a graph-based model for generating test cases from XACML access control policies. The *XFG* is a directed graph that

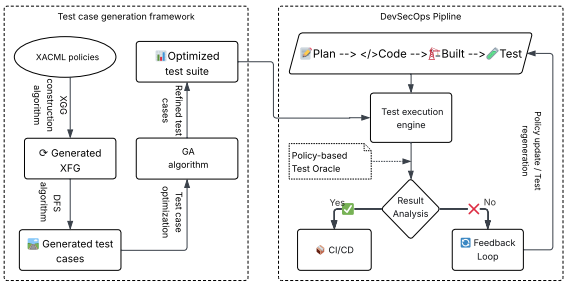

Figure 1. Architecture of the GA-based XACML test generation framework integrated into a DevSecOps pipeline.

captures both the hierarchical structure and the evaluation order of XACML policy elements.

To provide an intuitive understanding of the XFG before presenting its formal definition, Listing 1 shows a simplified XACML policy fragment from a financial system. The policy includes two rules applied to the same target (`FinancialOfficer`, `Transfer`) with overlapping conditions on the transfer amount, and is evaluated using the `deny-overrides` combining algorithm.

Listing 1: Illustrative XACML policy fragment with overlapping rule conditions

```
<PolicySet
    PolicyCombiningAlgId="deny-overrides">
  <Policy>
    <Rule RuleId="R1" Effect="Permit">
      <Target> FinancialOfficer /
          Transfer </Target>
      <Condition> amount < 10000
          </Condition>
    </Rule>
    <Rule RuleId="R2" Effect="Deny">
      <Target> FinancialOfficer /
          Transfer </Target>
      <Condition> amount > 5000
          </Condition>
    </Rule>
  </Policy>
</PolicySet>
```

In the XFG, the `PolicySet` and `Policy` elements are represented as parent nodes, while each `Rule` is mapped to a rule node connected to its `Target` and `Condition` nodes in evaluation order. Terminal decision nodes (e.g., *Permit* and *Deny*) represent authorization outcomes. Each directed path from the policy entry node to a decision node corresponds to a concrete policy evaluation scenario and forms the basis for systematic test case generation.

Based on this intuitive illustration, the following definitions formally specify the nodes and edges of the XACML Flow Graph. These definitions provide a precise foundation for systematic path enumeration and test case generation.

Definition 1 (XFG nodes)**.** *Let N be the set of nodes of an XACML policy. Each node $n \in N$ represents an evaluation element involved in XACML policy processing, including policy containers, rules, conditions, combining logic, and authorization decisions. Each node is associated with a type, formalized as*

$$type(n) \in \{\, PolicySet,\ Policy,\ Rule,\ Condition,\ CombiningAlg,\ Decision \,\} \quad (1)$$

According to Definition 1, nodes may include additional attributes (e.g., rule effects, target expressions, condition predicates, and decision outcomes) that are considered during policy evaluation and test generation.

Definition 2 (Structural and evaluation relations)**.** *For nodes $i, j \in N$:*

- *Node i is the* XFG-parent *of node j if j is hierarchically contained within i according to the XACML policy structure (e.g., a `Policy` within a `PolicySet`, or a `Rule` within a `Policy`).*
- *Nodes i and j are* siblings *if they share the same XFG-parent.*
- *Node i is a* left sibling *of j if i immediately precedes j in the evaluation order defined by the policy and its combining algorithm.*

Definition 3 (XFG edges)**.** *Let $E \subseteq N \times N$ be the set of directed edges representing possible evaluation flows in an XACML policy. For nodes $i, j \in N$, an edge $(i, j) \in E$ exists if at least one of the following conditions holds:*

1. *i is the XFG-parent of j (hierarchical descent).*
2. *i is a left sibling of j, and the combining algorithm permits evaluation to proceed from i to j.*
3. *i is the final node of a rule or condition evaluation, and j is the next node evaluated according to XACML decision-combining semantics.*

Definition 4 (XFG)**.** *An* XFG *is a directed graph defined as*

$$XFG = (N, E),$$

where N is the set of nodes (Definition 1) and E is the set of directed edges (Definition 3). The graph encodes both the structural containment and the evaluation order of XACML policy elements.

Based on the XFG, test cases are generated by systematically traversing evaluation paths using depth-first search. Path feasibility is determined by solving the conjunction of rule targets and condition predicates along each path, ensuring that only executable policy behaviors are exercised.

We propose Algorithm 1 to construct an XFG from a given XACML policy specification. The proposed algorithm realizes a structure-preserving mapping from XACML policy elements to a directed graph that explicitly captures both hierarchical containment and evaluation order defined by XACML semantics.

Given an XACML policy specification P, the construction process traverses the policy structure in a depth-first manner and incrementally builds the corresponding graph. Each policy element encountered during traversal is mapped to a unique XFG node according to Definition 1. Directed edges are added following Definition 3 to encode hierarchical relationships, sequential rule evaluation, and control flow induced by combining algorithms.

Rule targets and conditions are explicitly modeled as separate nodes and connected in their evaluation order. Finally, combining algorithm nodes aggregate rule outcomes and connect them to terminal decision nodes representing authorization results.

Algorithm 1 Construction of XACML Flow Graph (XFG)

Input: XACML policy specification P
Output: XACML Flow Graph $XFG = \langle N, E \rangle$

Initialize empty node set $N \leftarrow \emptyset$ and edge set $E \leftarrow \emptyset$
Identify the root policy element r in P (PolicySet or Policy)
Create node n_r corresponding to r and add n_r to N
for each policy element e encountered during a depth-first traversal of P **do**
    Create a corresponding XFG node n_e and add n_e to N
    if e is hierarchically contained within a parent element p **then**
        Add edge (n_p, n_e) to E
    end if
    if e is a rule **then**
        Create nodes for its target and condition expressions
        Add edges reflecting their evaluation order
    end if
    if e is governed by a combining algorithm **then**
        Create a combining node
        Connect rule outcome nodes to the combining node
    end if
end for
Create decision nodes for authorization outcomes
    `Permit`
    `Deny`
    `NotApplicable`
    `Indeterminate`
Connect combining nodes to the corresponding decision nodes
return XFG

Proposition 1 (Correctness and termination of XFG construction)**.** *Given a finite XACML policy specification P, Algorithm 1 terminates and constructs an XFG that is correct with respect to Definitions 1–3.*

Proof sketch. Correctness. During the depth-first traversal of P, each XACML policy element is visited exactly once and mapped to a unique XFG node, in accordance with Definition 1. For each structural or evaluation relation in the policy, the algorithm adds a corresponding directed edge that satisfies Definition 3. Therefore, every feasible XACML evaluation sequence is represented as a directed path in the resulting XFG.

Termination. The algorithm terminates because the input policy P is finite and the depth-first traversal processes each policy element exactly once.

Complexity. Let n be the number of elements in P. Since each element is processed in constant time, the overall time complexity is $O(n)$. □

The resulting XFG provides a precise and analyzable representation of XACML policy evaluation and serves as the foundation for path enumeration, coverage analysis, and systematic test case generation, which are described in the following subsection

3.2. Test Case Generation from XFG

Based on the XFG constructed in the previous subsection, test case generation is formulated as a systematic exploration of evaluation paths in the directed graph. Each feasible evaluation path corresponds to a distinct execution scenario of the XACML policy and serves as the basis for deriving a concrete test case.

Definition 5 (XACML evaluation path)**.** *Let $XFG = (N, E)$ be an XACML Flow Graph. An* XACML evaluation path *is a finite sequence of nodes*

$$\pi = \langle n_0, n_1, \ldots, n_k \rangle$$

such that:

- $(n_i, n_{i+1}) \in E$ *for all* $0 \leq i < k$*;*
- n_0 *is a node of type* `Policy`*, representing the entry point of policy evaluation;*
- n_k *is a node of type* `Decision`*, representing a final authorization outcome.*

Each evaluation path represents a possible policy evaluation induced by a specific access request.

Definition 6 (XACML test case)**.** *Given an evaluation path π, an* XACML test case *is defined as an access request whose attribute values satisfy all target expressions and condition predicates encountered along π. Executing the test case exercises the corresponding policy evaluation and yields the authorization decision associated with the terminal node of π.*

Evaluation path enumeration Evaluation paths are systematically enumerated using a depth-first search (DFS) traversal over the XFG, as presented in Algorithm 2, starting from nodes of type Policy. The traversal recursively follows outgoing edges until a node of type `Decision` is reached. DFS is particularly suitable for this task because it naturally explores complete evaluation paths from policy entry points to final authorization outcomes while preserving the evaluation order defined by the graph structure. Compared to breadth-first search (BFS), which focuses on level-wise traversal, DFS directly generates end-to-end evaluation paths that align with XACML evaluation semantics. In contrast, BFS may produce partial paths that do not correspond to complete policy evaluations and therefore require additional mechanisms to reconstruct full execution sequences, making DFS more suitable for subsequent feasibility checking and test case derivation.

Path feasibility analysis Not all syntactically valid evaluation paths correspond to executable policy behaviors. An evaluation path is

Algorithm 2 DFS-based evaluation path enumeration

Input: XACML Flow Graph $XFG = (N, E)$
Output: Set of evaluation paths Π

$\Pi \leftarrow \emptyset$ ▷ Set of enumerated evaluation paths
procedure DFS(n, π)
  if $type(n)$ = `Decision` **then**
    $\Pi \leftarrow \Pi \cup \{\pi\}$ ▷ A complete evaluation path is found
    return
  end if
  for all $(n, m) \in E$ **do**
    DFS($m, \pi \cdot \langle m \rangle$)
  end for
end procedure
for all $n \in N \mid type(n)$ = `Policy` **do**
  DFS($n, \langle n \rangle$) ▷ Each policy node acts as an entry point
end for
return Π

considered *feasible* if the conjunction of all target expressions and condition predicates along the path is satisfiable. To determine feasibility, constraints collected along the path are combined into a logical condition and checked for consistency, which can be implemented using automated constraint solvers or lightweight heuristic checking depending on the policy complexity. Paths whose constraints are inconsistent are discarded before test case generation to avoid producing unreachable or invalid test cases and to ensure that only executable evaluation scenarios are considered.

Test case derivation For each feasible evaluation path π, a test case is constructed by collecting the attribute constraints imposed by target and condition nodes along the path. These constraints jointly define an access request that triggers the corresponding policy evaluation. The expected authorization decision of the test case is determined by the terminal decision node of π.

Formally, a test case is represented as a tuple

$$tc = \langle req, exp \rangle,$$

where req denotes the constructed access request and $exp \in$ {`Permit`, `Deny`, `NotApplicable`, `Indeterminate`} is the expected authorization decision.

Initial test suite construction The set of all test cases derived from feasible evaluation paths constitutes the initial test suite, denoted as TS_{raw}. Although TS_{raw} achieves high structural coverage of the XFG, it may contain redundant test cases due to overlapping evaluation paths. Therefore, a test suite optimization phase based on genetic algorithms is applied in the subsequent stage to reduce redundancy while preserving coverage.

3.3. Genetic Algorithm–Based Test Suite Optimization

Given the initial test suite TS_{raw} generated from feasible evaluation paths of the XACML Flow Graph (XFG), the objective of this phase is to construct an optimized test suite $TS_{\text{opt}} \subseteq TS_{\text{raw}}$ such that the number of test cases is minimized while preserving the required coverage of XACML policy elements (e.g., rules and decisions).

This optimization problem is addressed using a genetic algorithm (GA), which is well suited for exploring large combinatorial search spaces and balancing multiple optimization objectives.

3.3.1. Encoding and Fitness Function

Each individual C in the population represents a candidate test suite and is encoded as a binary vector

$$C = [b_1, b_2, \ldots, b_M],$$

where $M = |TS_{\text{raw}}|$ and $b_i = 1$ indicates that the i-th test case in TS_{raw} is selected, while $b_i = 0$ indicates that it is excluded.

The fitness function is designed to simultaneously maximize coverage and minimize the size of the selected test suite. It is defined as:

$$F(C) = w_1 \cdot \text{Cov}(C) + w_2 \cdot \left(1 - \frac{|C|_{\text{on}}}{M}\right), \quad (2)$$

where Cov(C) denotes the coverage ratio achieved by the selected test cases (with respect to XACML rules or evaluation paths), and $|C|_{\text{on}}$ is the number of activated bits in C, i.e., the number of selected test cases. The weights w_1 and w_2 control the relative importance of coverage preservation and test suite reduction.

3.3.2. Optimization Process

The GA iteratively evolves a population of candidate solutions through selection, crossover, and mutation, as summarized in Algorithm 3. The algorithm starts by randomly initializing individuals from TS_{raw}, where each individual represents a candidate subset of feasible test cases. Instead of exhaustively enumerating the search space, this initialization generates multiple initial individuals; therefore, a moderate population size of 50 is adopted to balance search capability and the computational cost of fitness evaluation. Tournament selection is used to maintain selection pressure during evolution. A single-point crossover is applied with probability $p_c = 0.8$, which is suitable for path-based test-suite representations, while mutation is performed with a low probability $p_m = 0.05$ to introduce controlled variations while preserving high-coverage solutions.

At each generation, individuals are evaluated using the fitness function, after which fitter individuals are selected to produce offspring. The fitness function combines rule coverage preservation and test suite size minimization using weighted coefficients $(w_c, w_s) = (0.7, 0.3)$, prioritizing coverage while encouraging reduction of redundant test cases. The evolution continues for a fixed number of generations ($MaxGen = 100$), after which the best individual is decoded into the optimized test suite TS_{opt}.

Algorithm 3 Genetic algorithm for XACML test suite optimization

Input: Initial test suite TS_{raw}, population size $PopSize$, maximum generations $MaxGen$
Output: Optimized test suite TS_{opt}

```
Pop ← Init(PopSize, TS_raw)
Best ← GetBest(Pop)
for gen ← 1 to MaxGen do
    for all ind ∈ Pop do
        Fit(ind) ← CalcFitness(ind)
    end for
    NewPop ← ∅
    while |NewPop| < PopSize do
        P1, P2 ← Select(Pop)
        Child ← Crossover(P1, P2)
        Mutate(Child)
        NewPop ← NewPop ∪ {Child}
    end while
    Pop ← NewPop
    if Fit(Best_curr) > Fit(Best) then
        Best ← Best_curr
    end if
end for
return Decode(Best)
```

3.4. Integration with DevSecOps Pipeline

The proposed approach is designed to be seamlessly integrated into a DevSecOps pipeline for continuous validation of XACML access control policies. As illustrated in Fig. 1, the approach operates as an automated security testing stage within the *Test* phase of a CI/CD workflow.

In a typical pipeline, XACML policies are treated as version-controlled artifacts and evolve alongside application code. Whenever a policy is added or modified, the pipeline triggers the automatic construction of the XFG, followed by DFS-based enumeration of feasible evaluation paths and the generation of an initial test suite. The genetic algorithm then optimizes the test suite to reduce redundancy while preserving full rule coverage. The resulting optimized test cases are automatically executed against the Policy Decision Point (PDP), and the observed authorization decisions are checked using a policy-based test oracle.

The feedback produced by test execution is fed back into the pipeline to support rapid detection

of policy regressions, unintended rule overlaps, or coverage losses. If violations are detected, the pipeline can halt deployment and notify policy engineers, enabling early remediation. This tight feedback loop aligns with DevSecOps principles by shifting security validation left, reducing manual testing effort, and ensuring that access control policies remain correct and maintainable throughout their lifecycle.

By combining graph-based modeling, evolutionary optimization, and pipeline-level automation, the proposed approach supports scalable and continuous assurance of XACML policies in modern cloud-native and service-oriented systems.

4. Experimental Results

4.1. Support Tool

The proposed approach is implemented in a prototype tool, referred to as X-PTG (XACML Policy Test Generator). X-PTG supports the construction of the XACML Flow Graph (XFG), DFS-based enumeration of evaluation paths, and test suite optimization using a genetic algorithm (GA). The GA component is implemented using the *JGAP* (Java Genetic Algorithms Package) library.

All experiments were conducted on a workstation equipped with an Intel Core i5 processor at 2.4 GHz, 8 GB of RAM, running Windows 10. Figure 2 presents the prototype implementation of X-PTG, illustrating the generation of evaluation paths from XACML policies and the subsequent reduction of the initial test suite through GA-based optimization.

Using this prototype implementation, a series of experiments were conducted at the Software Engineering Laboratory (SE-Lab), Phenikaa University, to evaluate the effectiveness of the proposed approach. The following section describes the datasets and evaluation metrics used in our empirical study.

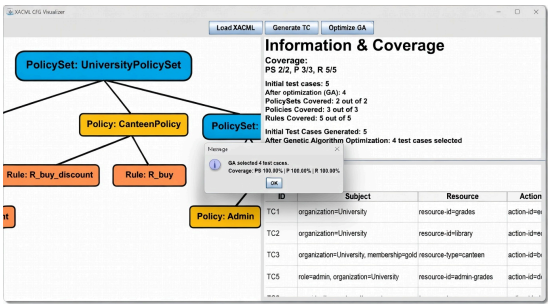

Figure 2. Prototype implementation of X-PTG illustrating evaluation path generation from XACML policies and GA-based test suite optimization.

4.2. Datasets

We conducted experiments on three representative XACML policy datasets covering different application domains and policy structures. These datasets allow us to analyze the behavior of the proposed method under varying policy sizes and organizational complexity. Table 1 summarizes their main characteristics.

Table 1: Characteristics of the XACML policy datasets

Dataset	Rules	Policies	Domain
HACS	25	5	HACS
FMS	30	3	FMS
SIS	20	4	SIS

Note: HACS – Hospital Access Control System; FMS – File Management System; SIS – Student Information System.

The *Rules* column indicates the total number of authorization rules defined across all policies in each dataset, while the *Policies* column denotes the number of individual policy containers. Together, these metrics reflect both the decision complexity and the structural depth of the policy sets, which directly influence the size of the resulting XFG and the number of evaluation paths generated.

4.3. Evaluation Metrics

The performance of the proposed method is evaluated using the following metrics:

- **Rule Coverage (%):** the percentage of XACML rules that are exercised by at least one test case in the test suite.
- **Reduction Rate (%):** the relative reduction in the number of test cases achieved by GA optimization compared to the initial test suite:

$$RR = \frac{|TS_{\text{raw}}| - |TS_{\text{opt}}|}{|TS_{\text{raw}}|} \times 100\%.$$

4.4. Results and Analysis

Table 2 summarizes the experimental results obtained from applying the proposed GA-based optimization to the initial test suites generated from XFG traversal. For each dataset, the genetic algorithm was executed ten times, and the reported values correspond to the average results in order to reduce the influence of stochastic effects.

Table 2: Results of GA-based test case optimization

Dataset	Init.	Opt.	Cov.	Red.
HACS	40	**18**	100%	55.0%
FMS	55	**22**	100%	60.0%
SIS	35	**15**	100%	57.1%
Average	43.3	18.3	**100%**	**57.4%**

Note: Init. = Initial paths; Opt. = Optimized test cases; Cov. = Coverage; Red. = Reduction rate.

Overall, the results demonstrate that the proposed GA-based optimization is effective in significantly reducing test suite size while preserving full rule coverage across all evaluated datasets.

- For the FMS dataset, the number of test cases is reduced from 55 to 22, corresponding to a reduction rate of 60%. This improvement is particularly relevant in practice, as policy evaluation at a Policy Decision Point (PDP) can incur non-negligible execution costs.
- Across all datasets, the optimized test suites achieve 100% rule coverage. This indicates that the fitness function successfully preserves test cases exercising critical and rarely triggered rules, while eliminating redundant paths that contribute limited additional coverage.
- The average time required for test case generation and GA-based optimization is approximately 3.5 seconds, which is acceptable for offline testing and integration into continuous testing pipelines.

Figure 3 visually compares the size of the initial test suites obtained via graph-based path enumeration with the optimized test suites produced by the genetic algorithm, highlighting the consistent reduction achieved across datasets.

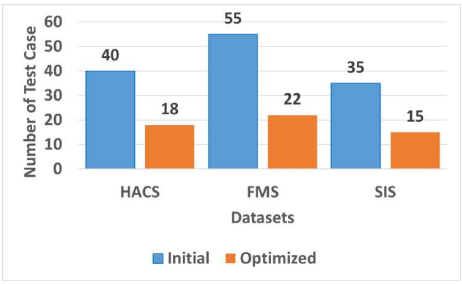

Figure 3. Comparison between initial and GA-optimized test suites in terms of test case reduction

4.5. Threats to Validity

Several threats to validity should be considered. Internal validity relates to the correctness of XFG construction, path enumeration, and genetic algorithm configuration; to mitigate this, the implementation follows the formal definitions and experiments are repeated with averaged results to reduce stochastic effects. Construct validity is associated with the chosen metrics, namely rule coverage and test suite reduction rate, which reflect testing effectiveness and efficiency but do not capture aspects such as fault detection capability. External validity is limited by the number of datasets, which, although drawn

from different domains, may not fully represent large-scale or highly dynamic access control environments. Finally, conclusion validity may be affected by the scale of the empirical study, and broader evaluations with stronger statistical analysis would further strengthen the results.

5. Conclusion

This paper presented a systematic approach for generating and optimizing test cases for XACML access control policies based on the *XACML Flow Graph* (XFG), a graph-based representation that models policy structure and evaluation semantics. By transforming policy evaluation into directed paths, test case generation becomes a process of structured path exploration with feasibility checking.

To reduce redundancy, a genetic algorithm was applied to optimize the initial test suite while preserving rule coverage. Experimental results on representative XACML datasets show that the approach achieves full coverage and reduces the test suite size by more than 50% on average, demonstrating the effectiveness of combining formal modeling, graph-based analysis, and evolutionary optimization for XACML policy testing. Future work will extend the XFG model to support advanced XACML features and enhance constraint-based test generation for regression testing scenarios.

Acknowledgment

This research is funded by Phenikaa University under grant number PU2025-3-A-04

References

[1] T.-B. Trinh, et al., Analysing Conflict of Interest Integrated in Role-Based Access Control Model Using Event-B, in: Intelligence of Things: Technologies and Applications, Springer, 2024, pp. 52–72.

[2] H. Arshad, R. Horne, C. Johansen, O. Owe, T. A. Willemse, XACML2mCRL2: Automatic Transformation of XACML Policies into mCRL2 Specifications , Science of Computer Programming 232 (2024) 103046.

[3] G. Liu, W. Pei, Y. Tian, C. Liu, S. Li, A Novel Conflict Detection Method for ABAC Security Policies, Journal of Industrial Information Integration 22 (2021) 100200.

[4] extensible access control markup language (xacml) version 3.0, OASIS Standard, approved: 23 Jan 2013. Available: http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html (January 2013).

[5] M. N. Nobi, M. Gupta, R. Krishnan, M. S. Rana, L. Praharaj, M. Abdelsalam, Machine Learning in Access Control: A Taxonomy [Systematization of Knowledge Paper], in: Proceedings of the 30th ACM Symposium on Access Control Models and Technologies, SACMAT '25, Association for Computing Machinery, New York, NY, USA, 2025, p. 145–156.

[6] Q. Li, G. Liu, Q. Zhang, L. Han, W. Chen, R. Li, J. Xiong, Efficient and Fine-Grained Access Control with Fully-Hidden Policies for Cloud-Enabled IoT, Digital Communications and Networks 11 (2) (2025) 473–481.

[7] V. C. Hu, D. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, K. Scarfone, Guide to Attribute Based Access Control (ABAC) Definition and Considerations, NIST Special Publication 800-162, National Institute of Standards and Technology, https://doi.org/10.6028/NIST.SP.800-162 (2014).

[8] M. St-Martin, A. P. Felty, A Verified Algorithm for Detecting Conflicts in XACML Access Control Rules, in: Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2016, Association for Computing Machinery, New York, NY, USA, 2016, p. 166–175, https://doi.org/10.1145/2854065.2854079.

[9] M. Yu, F. Li, N. Yu, X. Wang, Y. Guo, Detecting Conflict of Heterogeneous Access Control Policies, Digital Communications and Networks 8 (5) (2022) 664–679, https://doi.org/10.1016/j.dcan.2022.09.002.

[10] Y. L. Traon, T. Mouelhi, B. Baudry, Testing Security Policies: Going Beyond Functional Testing, in: Proceedings of the The 18th IEEE International Symposium on Software Reliability, ISSRE '07, IEEE Computer Society, USA, 2007, p. 93–102.

[11] Q. Wang, J. Chen, X. Li, Automatic Test Generation for Access Control Policies Based on XACML, Journal of Systems and Software 117 (2016) 1–14.

[12] A. Bertolino, S. Daoudagh, F. Lonetti, E. Marchetti, An Automated Model-Based Test Oracle for Access Control Systems, in: Proceedings of the 13th International Workshop on Automation of Software Test (AST'18), ACM, 2018, pp. 4:1–4:7, available: https://arxiv.org/pdf/1809.02724.pdf.

[13] G. Fraser, A. Arcuri, EvoSuite at the SBST 2016 Tool Competition, in: Proceedings of the 9th International Workshop on Search-Based Software Testing, SBST '16, Association for Computing Machinery, New York, NY, USA, 2016, p. 33–36.

[14] M. Khatibsyarbini, M. A. Isa, D. N. Jawawi, R. Tumeng, Test Case Prioritization Approaches in Regression Testing: A Systematic Literature Review, Information and Software Technology 93 (2018) 74–93.

[15] A. Pretschner, T. Mouelhi, Y. L. Traon, Model-Based Tests for Access Control Policies, in: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation, ICST '08, IEEE Computer Society, USA, 2008, p. 338–347.

[16] E. Martin, T. Xie, A Fault Model and Mutation Testing of Access Control Policies, in: Proceedings of the 16th International Conference on World Wide Web, WWW '07, Association for Computing Machinery, New York, NY, USA, 2007, p. 667–676.