IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 23 June 2025, accepted 1 August 2025, date of publication 11 August 2025, date of current version 15 August 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3597504

RESEARCH ARTICLE

Formal Verification for Preventing Misconfigured Access Policies in Kubernetes Clusters

ADITYA SISSODIYA, ERIC CHIQUITO, ULF BODIN, AND JOHAN KRISTIANSSON
Department of Computer Science, Electrical, and Space Engineering, Luleå University of Technology, 97187 Luleå, Sweden

Corresponding author: Aditya Sissodiya (aditya.sissodiya@ltu.se)

This work was supported in part by the Green Transition North (GTN) Project funded by the European Regional Development Fund, Region Norrbotten, Skellefteå Municipality, Luleå University of Technology, and Industrial Companies; in part by Digitala Stambanan IndTech under Grant Vinnova 2024-02510; and in part by the Horizon Europe Project RemaNet under Grant 101138627.

ABSTRACT Kubernetes clusters now underpin the bulk of modern production workloads, recent 2024 Cloud Native Computing Foundation surveys report >96% enterprise adoption, stretching from 5G edge nodes and AI/ML pipelines to heavily-regulated fintech and healthcare back-ends. Every action in those environments funnels through the API server, so a single access-control slip can jeopardise an entire fleet. Yet most deployments still rely on a patchwork of Role-Based Access Control (RBAC) rules and policy-as-code admission controllers such as OPA Gatekeeper or Kyverno. In practice these controls are brittle: minor syntactic oversights, wildcard privileges, or conflicting rules can silently create privilege-escalation paths that elude linters and manual review. This paper presents a framework that models both RBAC and admission policies as first-order logic and uses an SMT solver to exhaustively search for counter-examples to stated security invariants before policies reach the cluster. The approach detects policy conflicts, unreachable denies, and unintended permissions. Three real-world case studies are presented to illustrate how the framework reveals latent misconfigurations and validates the soundness of the corrected rules. These case studies include a supply-chain image bypass, an RBAC "shadow-admi" escalation, and a multi-tenant namespace breach. To aid replication and further study, we release a fully scripted GitHub testbed: a Minikube cluster, AuthzForce PDP, admission-webhook adapter, and Z3-backed CLI that recreates each scenario and verifies policies end-to-end. While the framework does not address runtime threats, it closes a critical verification gap and substantially raises the bar for attackers targeting the most widely deployed orchestration platform.

INDEX TERMS Attribute-based access control, cloud-native security, formal verification, kubernetes, policy-as-code, role-based access control, SMT solvers.

I. INTRODUCTION

Kubernetes has become the de facto standard for orchestrating containerised workloads in cloud environments [1]. A typical cluster comprises a central control plane, anchored by the API server and the etcd data store that coordinates a fleet of worker nodes and exposes a single management API [2]. Every request to that API is checked by Kubernetes' built-in RBAC engine [3], because RBAC relies on static, predefined roles, administrators often fall back on coarse-grained privileges [4], [5]. In practice, poorly scoped or overly permissive roles have led to severe incidents, most notably privilege-escalation attacks in which adversaries reached cluster-admin by exploiting unintended rights [6]. This paper strengthens that first line of defence by layering a formally verified Attribute-Based Access Control (ABAC) overlay.

Surveys reveal that an alarming share of clusters harbour such risky setups (for example, credentials that enable a full takeover from a single compromised pod) [6]. Hence, even a

The associate editor coordinating the review of this manuscript and approving it for publication was Amjad Mehmood.

minor RBAC misconfiguration can snowball into a critical exposure [3], underscoring the need for stronger access-control discipline.

To tighten security beyond static RBAC, many organisations have adopted dynamic admission controllers and policy-as-code frameworks that implement Attribute-Based Access Control (ABAC) [7], [8]. Solutions such as Open Policy Agent (OPA) Gatekeeper let operators craft admission constraints that consider request attributes and resource context [1], for instance, limiting acceptable image registries, enforcing mandatory labels, or upholding pod-security profiles. This ABAC model affords flexibility and expressiveness, but the resulting policies are easy to misconfigure. Authoring correct rules in Rego is non-trivial, subtle logic slips become more likely as policies multiply. A faulty rule can silently miss an intended block or mistakenly green-light a forbidden action [9].

These realities raise a pressing question: how can we guarantee that combined RBAC and ABAC policies are free from dangerous misconfigurations and escalation paths in Kubernetes clusters?

Manual reviews, linters, and unit tests catch only obvious faults [10], nuanced rule interactions frequently slip past. An admission policy may behave during routine tests yet harbour an edge case that grants unauthorised access, as the image-registry bypass illustrated. Likewise, an RBAC role meant to be narrow might grant broad powers through a wildcard or omitted condition with no warning from default tooling. A more rigorous assurance is needed.

Formal verification offers that path: by representing policies mathematically and exhaustively checking them against security invariants, one can surface flaws conventional methods miss [11], [12]. In short, formal methods can prove that ''forbidden'' scenarios are impossible, exposing weaknesses before attackers find them.

We address these challenges by applying formal methods to Kubernetes policies. We introduce a framework that models RBAC and ABAC rules as logical predicates and encodes critical security invariants, such as ''no unauthorised user may create privileged resources'' or ''only approved images are deployable.'' An SMT (Satisfiability Modulo Theories) solver then automatically searches for any model that violates those invariants [9], [11]. Because the verification targets the logical core of the rules, it can run incrementally during policy development or configuration changes without heavy model-checking or cluster disruption.

In summary, this paper makes the following contributions:

1) We present an SMT-backed framework that jointly models attribute-based policies and RBAC rules in Section III, detecting misconfigurations or unintended escalations and providing early warning of policy flaws.
2) We demonstrate and operationalize the framework on three real-world Kubernetes incidents (i) an image-registry policy misconfiguration that allowed unapproved images, (ii) an RBAC privilege-escalation that let a user mint cluster-admin roles, and (iii) a multi-tenant namespace isolation failure in Section IV, pinpointing the root cause policy flaws and proving the fixes.

We also provide a publicly available GitHub repository that spins up a reproducible Minikube testbed (including an AuthzForce PDP, admission-webhook adapter, Z3 encodings, verification CLI, and Kubernetes manifests). As discussed in Section V, researchers can replay each scenario end-to-end or benchmark alternative policy-analysis techniques on any Docker-enabled host.

Figure 1 gives a bird's-eye view of the Kubernetes authorization path, the misconfiguration hotspots we study, and the SMT-backed ABAC overlay that seals each gap.

To ground our contributions, Section II explores Kubernetes's access-control architecture and related work, highlighting the gaps in RBAC/ABAC that motivate our approach.

II. BACKGROUND AND RELATED WORK

A. KUBERNETES ACCESS CONTROL LANDSCAPE

Kubernetes authorization traditionally depends on RBAC and basic ABAC files [13], [14]. RBAC relies on predefined roles and resource actions, lacking dynamic contextual attributes or exceptions [3]. Kubernetes's older ABAC mode, now deprecated, only supported simplistic JSON policies [15]. To address these limitations, the community has shifted toward policy-as-code solutions like OPA Gatekeeper and policy-as-configuration frameworks such as Kyverno [13], [16], [17]. These solutions function as admission controllers by intercepting API requests before they reach the cluster, enabling administrators to define custom rules for resource validation or mutation. Gatekeeper utilizes the Rego language provided by OPA to define ConstraintTemplates and Constraints, while Kyverno applies declarative YAML policies as Custom Resources directly to targeted resources. Both solutions are Kubernetes-native and operate seamlessly through admission webhooks within the cluster, ensuring efficient performance with minimal latency.

However, despite their effectiveness, Gatekeeper and Kyverno lack certain advanced capabilities inherent in dedicated ABAC systems [18], [19]. Notably, neither platform includes automated conflict detection or built-in mechanisms for establishing rule precedence. Consequently, when two policies conflict, for example, if one allows an action and another forbids it, the result can be unclear, often dependent on the webhook invocation order or manual policy management [20]. Kyverno's documentation explicitly recognizes that multiple simultaneous policies can result in unintended behaviors due to conflicts.

Administrators must therefore carefully separate policies by namespaces or other methods to prevent overlaps. The absence of formal combining algorithms (such as an explicit ''deny overrides'' rule) represents a well-known challenge in Kubernetes policy management [21], [22]. Moreover, policy

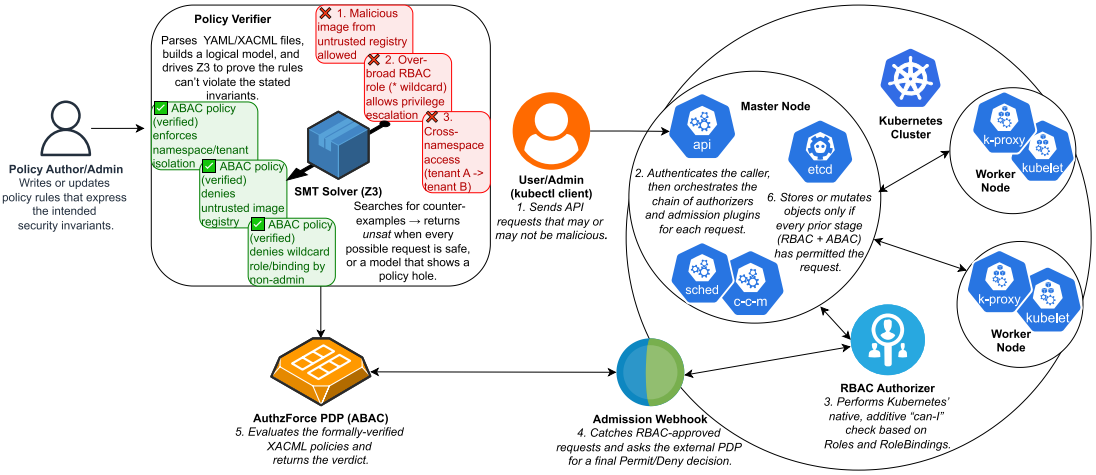

FIGURE 1. A user's API request traverses the API Server's native RBAC check and a validating admission webhook, which hands the request to an external AuthzForce PDP enforcing SMT-proved XACML policies, red call-outs mark three common misconfigurations (untrusted image registry, wildcard privilege escalation, cross-tenant access) while green call-outs show how the deny-overrides ABAC layer blocks each flaw before any change reaches etcd.

engines like Gatekeeper lack proactive validation against existing policies for logical consistency, an increasingly risky oversight as policy complexity grows. This limitation can result in misconfigurations, exemplified by recent research from Aqua Security, which revealed how a seemingly correct Gatekeeper constraint (k8sallowedrepos) was inadvertently bypassed due to a subtle error, allowing unapproved container images [23]. Specifically, the vulnerability arose from a missing trailing slash in the allowed registry entry, demonstrating how easy it is to introduce logical errors that Kubernetes's native tools cannot automatically detect [23].

B. ATTRIBUTE-BASED ACCESS CONTROL (ABAC) SYSTEMS

In contrast, dedicated ABAC engines offer a more comprehensive model for defining and evaluating policies [8].

AuthzForce, an open-source XACML Policy Decision Point (PDP) developed by OW2, is frequently referenced within cloud contexts [17], [24]. It implements the OASIS XACML 3.0 standard, providing a complete attribute-based access control solution. Policies are authored in XML or simplified syntaxes such as ALFA, supporting various attributes related to subjects, resources, actions, and environmental contexts. It operates externally from Kubernetes as a standalone service that Kubernetes' API server or an associated sidecar can query. In Kubernetes configurations, the API server's webhook authorization module can be set up to use AuthzForce as an external PDP. Each authorization query (e.g., "Can user X create a Deployment in namespace Y?") translates into an XACML request sent to AuthzForce, which evaluates relevant policies and returns a Permit or Deny decision.

This external approach separates policy decision-making from the Kubernetes cluster, facilitating standardized ABAC policies applicable across multiple systems. However, it introduces latency, as each decision requires an external call [25], and potentially creates a single point of failure if the PDP becomes unavailable [26], [27]. Kubernetes-native admission controllers, such as Gatekeeper and Kyverno, have a latency advantage here because they run within the cluster itself, maintaining functionality as long as the API server remains operational.

Prior research emphasizes the necessity of ensuring high availability and effective caching strategies for external PDPs to remain practical at scale [28]. Despite these considerations, AuthzForce and similar ABAC engines provide features not available in Kubernetes's native tools. Specifically, they inherently support complex rule logic and combining algorithms. For instance, XACML allows hierarchical policy composition with explicit Policy Combining Algorithms such as "deny-overrides," "permit-overrides," and "first-applicable" [17], [29]. These algorithms clearly determine the final authorization decision when multiple policies or rules apply simultaneously. For example, the deny-overrides algorithm ensures a Deny decision if even one rule denies access, thus guaranteeing security by design. Under the first-applicable algorithm, policies are sequentially evaluated until the first applicable rule determines the outcome, offering deterministic conflict resolution, a capability absent from Kubernetes-native admission controllers.

C. FORMAL POLICY ANALYSIS RESEARCH

To ensure Kubernetes security policies behave as intended, researchers have applied formal analysis tools that rigorously model and verify policy logic.

Margrave, for example, represents access policies in first-order logic to enable detailed scenario-based exploration. It can systematically enumerate differences in outcomes between policy versions, helping administrators foresee unintended side-effects of policy modifications [30], [31].

Similarly, the Alloy modeling language has been used to formally specify and analyze access control rules. Alloy's Analyzer automatically searches for policy inconsistencies or conflicts by exhaustively examining the formal model for scenarios that violate intended constraints, an approach shown to catch subtle errors, for instance, uncovering network policy conflicts that deviate from expected behavior [32], [33], [34], [35].

To capture dynamic behavior over time, the Event Calculus provides a temporal logic framework for reasoning about sequences of events and their outcomes; it has been used to model evolving policy state and can use logical deduction to identify policy anomalies, even detecting issues like redundant or shadowed rules that static analyses might overlook [36], [37].

At an industrial scale, cloud providers have adopted SMT-based verification; AWS's internal tool Zelkova exemplifies this by translating cloud access policies into logical constraints and using a satisfiability-modulo-theories solver to exhaustively check all possible access scenarios. Zelkova verifies that specified security properties hold (providing a sound mechanism to catch misconfigurations) and remarkably handles this complex analysis millions of times per day in AWS's production environment [12], [38].

Despite their different formalisms, all of these approaches rely on mathematical logic to rigorously analyze policy semantics. This formal foundation enables them to uncover subtle misconfigurations or conflicting rules (e.g., unreachable or overshadowed permissions) and to verify critical security invariants that would be difficult to guarantee with conventional testing alone.

Integrating such formal policy analysis methods into Kubernetes environments could mitigate the limitations of native policy enforcement by providing more comprehensive validation and automated conflict detection [39], [40], [41]. However, this integration also introduces additional complexity and potential performance overhead, so it must be balanced with practical operational constraints.

In practice, AWS's use of Zelkova demonstrates that even computationally heavy verification tasks (solving a PSPACE-complete analysis) can be engineered to run at cloud scale [38].

Ultimately, incorporating formal reasoning into Kubernetes policy management offers high assurance of correctness and consistency, but practitioners must weigh those guarantees against the implementation effort and resource costs of maintaining such rigorous verification in production.

D. EMPIRICAL SECURITY INCIDENTS IN PRODUCTION CLUSTERS

Real-world breaches illustrate how the theoretical gaps outlined above manifest in practice. The following three incidents span admission-policy errors, RBAC overreach, and multi-tenant isolation failures. Together they motivate the need for a unified, formally verified approach to Kubernetes policy assurance. These case studies are revisited in Section IV, where we formally analyse each misconfiguration and demonstrate its remediation.

```
constraint.yaml
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sAllowedRepos
metadata:
  name: repo-constraint-ecr-misconfig
spec:
  match:
    kinds:
      - apiGroups: [""]
        kinds: ["Pod"]
    namespaces:
      - "default"
  parameters:
    repos:
      - "my-ecr.azurecr.io"
```

```
> kubectl run my-pod --image=my-ecr.azurecr.io.attacker.com/malicious
pod/my-pod created
```

FIGURE 2. A misconfigured OPA Gatekeeper policy (K8sAllowedRepos) without a trailing slash in the allowed registry (top) and an attacker exploiting it by using a subdomain matching the allowed prefix (bottom). This tricks the policy into accepting a malicious image (pod creation succeeds) from an unapproved repository.

1) ADMISSION-POLICY BYPASS: MIS-SCOPED REGISTRY PREFIX (2025)

A 2025 report by Aqua Security uncovered a subtle but dangerous misconfiguration in Kubernetes admission policies that was being exploited in the wild [42]. The policy in question was an OPA Gatekeeper constraint (*k8sallowedrepos*) meant to allow only trusted container image registries. However, the policy was defined with a domain prefix that lacked a terminating slash (e.g., allowing *"myregistry.com"* instead of *"myregistry.com/"*). This caused Gatekeeper's prefix-match logic to erroneously trust any registry whose name started with *myregistry.com*. Attackers took advantage by hosting images at subdomains like *myregistry.com.attacker.com*, which Gatekeeper interpreted as a permitted domain [42]. In effect, the malicious image *myregistry.com.attacker.com/ malicious* was pulled and deployed, bypassing the intended restriction. This misconfiguration and its exploitation are show in Figure 2.

Aqua's research found multiple such misconfigurations in real clusters, demonstrating that this was not an isolated mistake. This incident underscores how a simple regex/pattern oversight can nullify an important security control.

2) RBAC OVERREACH: "RBAC BUSTER" ATTACK (2024)

A concrete example of RBAC overreach exploitation is Aqua Security's "RBAC Buster" attack observed in 2024. In that incident, a misconfigured cluster allowed an attacker initial access as an anonymous user with some privileges (the Kubernetes API server had anonymous access enabled and inadvertently privileged) [43]. The attacker then used Kubernetes APIs to establish persistence as a cluster admin. They did this by programmatically creating a *ClusterRole* with near-admin rights, creating a new *ServiceAccount*, and then binding that *ClusterRole* to the *ServiceAccount* via a *ClusterRoleBinding* [43].

```
{"apiVersion": ``rbac.authorization.k8s.io/v1'',
``kind":␣"ClusterRole'',"metadata":
{ "annotations":{},
"name": "system:controller:kube-controller"},
"rules":[{
"apiGroups":["*"],
"resources":["*"],
"verbs":["*"]},
{"nonResourceURLs":["*"],
"verbs":["*"]}]}
```

Above is an excerpt of a Kubernetes ClusterRole created by an attacker (from Aqua Security's "RBAC Buster" incident) giving near-admin privileges. The ClusterRole (named system:controller:kube-controller) grants wildcard access to all API groups, resources, and verbs. The attacker bound this role to a new service account to stealthily gain full cluster control. This effectively granted the attacker a new admin-equivalent identity in the cluster, all through standard Kubernetes APIs, a clever abuse of RBAC mechanics. After setting up this backdoor, even if the initial misconfiguration (anonymous access) was corrected, the attacker's malicious Role/Binding would remain, keeping them in power. They proceeded to deploy a *DaemonSet* to run cryptominers on all nodes (leveraging their new privileges) [43].

This attack showcases both an RBAC bypass (gaining higher privileges than should be allowed, via a misconfig) and the challenge of exceptions, the cluster's configuration lacked a way to say "nobody (including anonymous) should have admin, except maybe a proper user," and once the attacker was in, there was no global policy to stop the creation of an all-powerful role. Other real-world examples include administrators accidentally binding the *system:unauthenticated* user group to privileges (effectively making the cluster open to the world) [44], a disastrous misconfiguration that has indeed been found in some clusters. These illustrate how overly permissive or mistaken RBAC configurations can be exploited by attackers to bypass intended security boundaries.

3) MULTI-TENANT NAMESPACE BREACH ON AKS (2023–2024)

A notable incident was reported (in forums and later formalized by security researchers) involving an Azure Kubernetes Service (AKS) multi-tenant setup with Apache Airflow (an orchestration tool) deployed. In that case, a misconfigured RBAC setup in the Airflow namespace granted far more access than intended, effectively a *RoleBinding* that conferred cluster-admin capabilities cluster-wide to the Airflow service account. Unit 42's analysis of Azure Data Factory's Airflow integration identified this as a serious weakness [45]: an attacker who could inject code into Airflow (for example by uploading a malicious DAG workflow) would execute in a pod that unexpectedly had cluster-admin permissions, thereby compromising the entire cluster. The researchers demonstrated that exploiting this misconfiguration could allow persistent admin access over all namespaces and resources in the cluster, essentially breaking multi-tenancy isolation.

Microsoft considered the risk low severity (perhaps because it required prior access to the Airflow DAG), but it exemplifies how a small RBAC oversight in a multi-tenant context (Airflow's namespace meant for one team's tools vs. other teams' namespaces) opens the door to cluster takeover. Another real example occurred on a cloud cluster where an internal team's CI/CD deployment inadvertently had a wildcard *RoleBinding*. Reports on security forums noted an AKS incident (circa 2023) where a team's deployment was meant to be confined to their namespace, but a mistake granted it privileges across all namespaces [16], leading to an internal "breach" where one team could modify others' resources. These scenarios are more configuration failures than novel exploits, but adversaries can and do look for such mistakes. In a multi-tenant Kubernetes environment, any component that isn't properly restricted to its namespace can become a Trojan horse to attack the whole cluster or other tenants.

E. SUMMARY OF GAPS

Kubernetes-native policies are simple and fast (but leave conflict-prevention to the admin), whereas ABAC engines rigorously enforce correctness (but add deployment latency and complexity) [8], [21]. This trade-off has left a gap between Kubernetes's agility and the stronger guarantees of formal policy engines. Table 1 conceptually contrasts these approaches.

We identify three fundamental weaknesses in Kubernetes's default policy enforcement stack:

1) FRAGILE IMAGE PROVENANCE ENFORCEMENT

Kubernetes RBAC on its own cannot restrict which container images users deploy; it governs only who may deploy them [3]. As a result, clusters rely on admission controllers (Gatekeeper, Kyverno) to enforce image provenance policies (e.g., only allow images from approved registries) [46]. These admission rules, unfortunately, are brittle and easy to misconfigure. A subtle pattern mistake for example, forgetting a trailing slash in an allowed registry prefix can trick a policy into accepting a malicious container image by interpreting a hostile domain as if it were an approved subdomain [42].

2) INFLEXIBLE, PERMISSIVENESS-PRONE RBAC MODEL

Kubernetes RBAC is built on static role definitions that are purely additive (permissions can be granted but not explicitly denied) [47]. There is no native concept of a denial or exception within a role, meaning administrators cannot easily express policies like "allow X except Y" [21]. This lack of exception semantics often forces clumsy workarounds and makes it difficult to enforce true least-privilege [4]. If an overly broad permission is granted e.g., a wildcard privilege in a *ClusterRole* Kubernetes provides no second-layer check

TABLE 1. Comparison of Kubernetes-native Policy Tools and External ABAC Systems.

Feature	Kubernetes-native (RBAC, Gatekeeper, Kyverno)	External ABAC (e.g. AuthzForce, AWS Cedar)
Policy definition model	Role-based, limited attributes; policies as K8s resources	Attribute-based, rich attributes; policies defined and stored externally
Policy validation	None built-in; relies on manual review	Native consistency checks via SMT/formal methods
Conflict resolution	No automated conflict detection; webhook order is implementation-defined	Explicit combining algorithms (deny-overrides, permit-overrides, first-applicable)
Attribute support	Static resource fields only	Dynamic subject, resource, action, and environmental attributes
Performance and latency	In-cluster, low latency	External calls introduce additional latency
Deployment complexity	Simple, integrated into Kubernetes	Requires separate PDP infrastructure and integration
Scalability	Scales with cluster; no extra components	Must scale PDP separately (e.g. clustering, caching)
Availability	Tied to API server availability	Potential single point of failure; needs HA for PDP

to catch or override that mistake. Any such loophole will persist until a human finds and fixes it, in contrast to ABAC systems that could have explicit deny rules or precedence to block unauthorized actions [8].

Attackers can and do exploit this rigidity. A single role misconfiguration can silently escalate privileges, since Kubernetes has no way to automatically block an unintended action that technically falls within a granted role. For instance, a known Kubernetes bug (CVE-2018-1002105) demonstrated this gap by allowing specially crafted API requests to bypass RBAC checks entirely [48]. There is no built-in notion of ''deny override'' in RBAC to stop an action that was unintentionally permitted. Adversaries are well aware of this weakness: MITRE ATT&CK technique T1098.006 (Additional Container Cluster Roles) specifically notes that attackers create or bind new roles to themselves to maintain stealthy persistence in a cluster [49]. In essence, once an attacker finds any crack in the RBAC armor, whether through a software flaw or an operational mistake, they can escalate their privileges and persist with ''shadow admin'' capabilities that are extremely hard to detect [43]. Kubernetes's native model offers no automated way to say ''nobody (not even an authenticated user) should have admin rights except X,'' and once an attacker is in, nothing in RBAC prevents them from creating an all-powerful role for themselves. These inherent limitations make manual privilege-scoping and out-of-band checks fragile; a single overlooked permission can undermine the entire cluster's security.

3) WEAK MULTI-TENANCY ISOLATION

By default, Kubernetes is not a strongly multi-tenant platform [50]. It provides namespaces and RBAC policies as building blocks for isolation, but there are no hard default safeguards to prevent one tenant from impacting another if the cluster is misconfigured [51]. The responsibility is on cluster administrators to compose fine-grained controls (network policies, strict RoleBindings, etc.) to wall off each tenant environment [15]. A simple human error, such as binding an overly broad role to a namespace, or failing to define restrictive network policies can collapse those isolation walls. In practice, seemingly minor mistakes like an overly permissive *RoleBinding* or an absent default network policy have led to tenants escaping their namespaces [16]. An attacker who compromises a workload in one namespace can leverage such a misconfiguration to pivot into other namespaces or even gain cluster-wide control, violating intended tenancy boundaries [52].

In summary, Kubernetes' native multi-tenancy controls lack both default safety nets and the granular composition needed to absolutely contain each tenant. A single misstep in configuration can let an attacker traverse what should be inviolable namespace boundaries, endangering every tenant in the cluster [45].

These structural gaps in Kubernetes's native security controls motivate our approach of bridging the cluster with an external ABAC engine to combine the best of both worlds. By leveraging ABAC's expressive, verifiable policies on top of Kubernetes, we aim to retain Kubernetes's performance and compatibility while addressing the above weaknesses, for example, enforcing strict image provenance checks, injecting deny-exceptions into policy logic, and tightening multi-tenant boundaries with global constraints [53].

Accordingly, the next section introduces our formal verification approach, formalizing a unified RBAC–ABAC model that bridges these gaps with mathematical rigor.

III. FORMAL ANALYSIS AND POLICY VERIFICATION

The empirical incidents detailed in the previous section expose a common theme: each breach originated not from an exotic zero-day, but from an imprecise or outright missing policy guarantee [16], [42], [43]. ''Best-practice'' checklists proved inadequate because they cannot exhaustively reason about the combinatorial space of Kubernetes roles, bindings, and admission rules [10], [23]. To advance beyond post-mortem patchwork, we now formalise Kubernetes authorisation semantics and security invariants in logic [8], [11], [12].

This section introduces a verification framework that proves a cluster configuration upholds its stated security properties, turning the informal insights of case studies into mathematically defensible guarantees. To support such verification, Kubernetes' multi-layered authorisation model must be unified into a formal representation. Kubernetes enforces access control through a layered pipeline: RBAC decisions at the API server, admission controllers (e.g., OPA/Gatekeeper), and external policy engines. Since each layer is specified and evaluated independently, reasoning about their combined effect is non-trivial. To analyse the cluster's actual security posture, we model the entire pipeline as a single first-order predicate, denoted π_{unified}, and verify it against explicit security invariants φ using an SMT solver. The verification process involves three steps:

Step 1: Define the unified policy π_{unified}. Each request $q = (u, r, a)$ is mapped to `Permit` or `Deny` by conjoining all RBAC and admission decisions:

$$\pi_{\text{unified}}(q) = \pi_{\text{RBAC}}(q) \wedge \pi_{\text{Admission}}(q).$$

Step 2: State the security invariant φ. Invariants codify "business–critical" rules such as *"only users with* `admin=true` *may create ClusterRoles."* Formally, for user u, resource r, and action a,

$$\varphi(u, r, a) \; : \; (r = \text{ClusterRole} \; \wedge \; a = \text{create}) \\ \Rightarrow \; (u.admin = \text{true}).$$

Step 3: Prove soundness (or find a counter–example). Soundness means every request the cluster permits satisfies the invariant:

$$\forall q : \; \pi_{\text{unified}}(q) = \text{Permit} \implies \varphi(q).$$

We ask an SMT solver to search for a violating request:

$$\exists q : \; \pi_{\text{unified}}(q) = \text{Permit} \; \wedge \; \neg\varphi(q).$$

If the formula is unsat the policy set is sound, if it is sat the model returned by the solver is a concrete misconfiguration. For example, let $\varphi(u, r, a) \equiv (r = \text{ClusterRole} \wedge a = \text{create}) \Rightarrow (u.\text{admin} = \text{true})$. This invariant states if the resource is *ClusterRole* and the action is *create*, then the user must have *admin=true*. Suppose the solver returns

$$q = (\text{alice}, \; \text{ClusterRole}, \text{create}), \text{alice.admin} = \text{false}.$$

Because $\pi_{\text{unified}}(q) = \text{Permit}$ yet $\neg\varphi(q)$ holds, the framework flags an unsound policy: *alice*, a non-admin, can create ClusterRoles, contradicting the stated security requirement. Note that in this verification-only approach, we leave the cluster's RBAC and admission policies unchanged (no additional rules are introduced for φ). We simply check whether there exists any request q such that $\pi_{\text{combined}}(q) = \textit{Permit}$ while $\neg\varphi(q)$ holds. If such a q exists, it constitutes a counterexample indicating a policy violation.

As shown in Figure 3, our framework combines the RBAC and ABAC models to compute and compare effective permission sets, with formal invariants incorporated into the ABAC layer for verification. Table 2 maps each formal symbol in our framework to its corresponding Kubernetes concept and provides an intuitive explanation for non-expert readers. With this approach in mind, the following sections formalise each step above.

A. PART I: FORMAL MODEL OF RBAC

To formally reason about access control, we first define policies as logical functions over user, resource, and action triples. This sets the foundation for verification by enabling policies to be composed, analyzed, and checked systematically.

1) RBAC AUTHORIZATION SEMANTICS

- Let U be the set of *users*, R the set of *roles*, and P the set of *permissions* (action-resource pairs).
- Let $B \subseteq U \times R$ be the *user-role binding relation*.
- Let $A \subseteq R \times P$ be the *role-permission relation*.

Define the effective permission for a user $u \in U$:

$$\text{Perms}_{\text{RBAC}}(u) \; = \bigcup_{\substack{r \in R \\ (u,r) \in B}} \{p \in P \mid (r, p) \in A\}$$

2) RBAC EVALUATION FUNCTION

The RBAC authorization decision is a predicate:

$$\text{RBACAllow}(u, p) \iff p \in \text{Perms}_{\text{RBAC}}(u)$$

This model has no concept of deny or exceptions, if p is reachable via any role bound to u, access is granted.

B. PART II: FORMALIZING RBAC MISCONFIGURATIONS

Sound policy enforcement depends on domain-specific security invariants that must always hold. We express these invariants φ, against which candidate policies will be verified.

A misconfiguration occurs if:

- An *unintended permission* p becomes reachable by u (e.g., via a wildcard ClusterRole).
- A *security invariant* is violated. Let $\varphi(u, p)$ be such an invariant (e.g., "user X must never create ClusterRoles").

Then, a policy is *unsound* with respect to φ if:

$$\exists u \in U, \; \exists p \in P : \text{RBACAllow}(u, p) \; \wedge \; \neg\varphi(u, p)$$

This defines a *counterexample* that proves a violation.

C. PART III: ABAC SEMANTICS

We formalize RBAC to later compare and transform it within the more expressive ABAC framework.

We let $\varphi = \{\varphi_1, \varphi_2, \ldots\}$ denote the set of security invariants in our model (each φ_i is a predicate encoding a required security property, such as tenant isolation or no privileged pods).

We denote the set of policy rules as $\pi = \{\pi_1, \pi_2, \ldots\}$, where each π_j is an access control rule (either an RBAC permission or an ABAC rule with conditions).

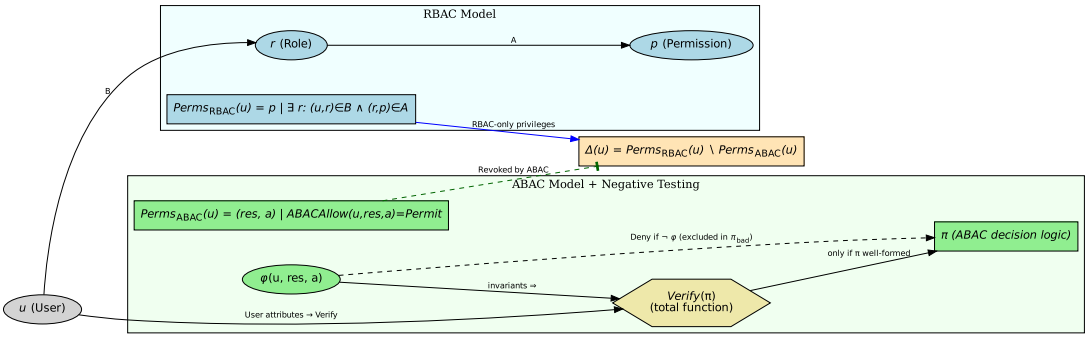

FIGURE 3. The top "RBAC Model" cluster (blue background) shows how roles r and permissions p determine $\text{Perms}_{\text{RBAC}}(u)$. The bottom "ABAC Model + Negative Testing" cluster (green background) introduces Verify(π) as a total function: well-formed policies proceed to π (ABAC decision logic) and invariants $\varphi(u, \text{res}, a)$ to compute $\text{Perms}_{\text{ABAC}}(u)$. Finally, $\Delta(u) = \text{Perms}_{\text{RBAC}}(u) \setminus \text{Perms}_{\text{ABAC}}(u)$ (orange) highlights privileges revoked by ABAC.

TABLE 2. Mapping formal notation to Kubernetes concepts.

Symbol / relation	Kubernetes artefact(s)	Plain-English interpretation
U	*User* or *ServiceAccount*	Identity that makes API requests.
R	*Role* / *ClusterRole*	Named collection of permissions.
P	$\langle$action, resource$\rangle$ pairs	Concrete action, e.g. *get pods*.
$B \subseteq U \times R$	*RoleBinding* / *ClusterRoleBinding*	Links a user or service account to a role.
$A \subseteq R \times P$	*rules* field inside a Role	Which verbs/resources each role grants.
$\text{Perms}_{\text{RBAC}}(u)$	Output of *kubectl auth can-i*	All actions user u can perform under current RBAC.
$\text{RBACAllow}(u, p)$	RBAC admission step in API server	"Does RBAC let u execute action p?"
π	Entire RBAC/ABAC rule set (YAML, Rego, XACML)	Code that decides *Permit* or *Deny*.
φ	Stated security invariant	Property that must always hold (e.g. "only admins edit roles").
$\Delta(u)$	Permissions revoked by ABAC layer	Actions RBAC would allow but ABAC explicitly blocks.

We use Δ to represent a derived set of interest – typically, the set of attribute combinations or access requests that violate at least one invariant under the given policy. In other words, Δ corresponds to the counterexample scenarios identified by the solver. Each time Δ appears, we define its specific meaning in context for clarity.

Let:

- $A_U(u)$ be the set of attributes for user u.
- $A_R(r)$ the set of attributes for resource r.
- $\mathcal{P}$ the set of ABAC *policies* as logical formulas.
- A policy $\pi \in \mathcal{P}$ is a predicate:

$$\pi(u, r, a) : A_U(u) \times A_R(r) \times \text{a} \longrightarrow \{Permit, Deny\}.$$

The ABAC decision function with *deny-overrides* semantics becomes:

$$\text{ABACAllow}(u, r, a) = \begin{cases} Deny & \text{if any } \pi \in \mathcal{P} \text{ returns } Deny, \\ Permit & \text{if any } \pi \in \mathcal{P} \text{ returns } Permit, \\ Deny & \text{if no policy applies.} \end{cases}$$

1) COMBINED DECISION

In a real cluster the final decision is

$$\pi_{\text{combined}}(q) = \text{Permit iff } (\pi_{\text{RBAC}}(q) = \text{Permit}) \wedge (\pi_{\text{ABAC}}(q) = \text{Permit}).$$

Accordingly, a misconfiguration is modelled as any request where $\pi_{\text{combined}}(q) = \text{Permit}$ yet $\neg\varphi(q)$ holds. We treat π_{bad} as shorthand for this combined but flawed policy in the case studies.

D. PART IV: FORMAL TRANSLATION FROM RBAC TO ABAC

We encode ABAC policy logic into a unified formal language to enable automated reasoning over its decisions.

We define a translation function $\tau : R \times P \to \pi$ that transforms RBAC entries into ABAC policies.

For example, a role r granting permission p to user u becomes:

$$\tau(r, p) = \pi_{r,p}(u, r', a) \equiv \big(u.roles \ni r\big) \wedge \big(r'.kind = p.resource\big) \wedge \big(a = p.action\big)$$

This maintains existing semantics.

To enforce constraints (e.g., global denial), we inject negative policies π^- with higher priority:

$$\pi^-(u, r, a) = Deny \quad \text{if} \quad \neg\varphi(u, r, a)$$

These act as *safety constraints* overriding any positive permissions. However, in our verification-only mode, we do not introduce such phantom rules. Instead, the admission control policies are modeled exactly as they exist in the cluster, and φ remains purely an invariant to be checked against the combined policy.

E. PART V: POLICY SOUNDNESS CHECK

To *verify soundness*, define:

- Intended security invariant $\varphi(u, r, a)$ (e.g., "only users with admin=true can create ClusterRoles").
- Effective permission set under ABAC:

$$\text{Perms}_{\text{ABAC}}(u) = \{(r, a) \mid \text{ABACAllow}(u, r, a) = \textit{Permit}\}.$$

The ABAC policy set $\mathcal{P}$ is *sound* with respect to φ iff:

$$\forall u, r, a : \text{ABACAllow}(u, r, a) = \textit{Permit} \Rightarrow \varphi(u, r, a).$$

To test this, use SMT or model-checking to search for a *counterexample*:

$$\exists u, r, a : \text{ABACAllow}(u, r, a) = \textit{Permit} \wedge \neg\varphi(u, r, a).$$

If such a tuple exists, the policy set violates the intended security property.In that case, the framework flags the policy as unsound. Rather than modifying the policy on the fly, we report this violation along with a recommended fix (e.g., adding an OPA admission rule that denies the disallowed action in production).

F. PART VI: DELTA ANALYSIS OF ACCESS (RBAC VS. ABAC)

$$\Delta(u) = \text{Perms}_{\text{RBAC}}(u) \setminus \text{Perms}_{\text{ABAC}}(u).$$

If $\Delta(u) \neq \emptyset$, then ABAC has successfully *reduced the privilege surface* for user u. If any $p \in \Delta(u)$ violates a security property, then the ABAC policy has prevented a vulnerability that RBAC permitted.

G. PART VII: NEGATIVE TESTING AND FAILURE SEMANTICS

The previous parts formalised *policy soundness*: if a specification authorises any request, that request must satisfy the declared security invariants.

We now make sure that the *verification procedure itself* remains sound when presented with malformed, incomplete, or adversarial inputs.

Let Π be the set of candidate policy terms in the specification language. Define two partial functions

$$\text{Parse} : \Pi \rightharpoonup \mathsf{Syntax}, \text{Model} : \mathsf{Syntax} \rightharpoonup \mathsf{Formula},$$

where Parse maps a textual policy to an abstract syntax object, and Model maps that object to a first-order formula. A policy π is *well-formed* when both stages succeed:

$$\text{Parse}(\pi)\downarrow \wedge \text{Model}\big(\text{Parse}(\pi)\big)\downarrow .$$

Otherwise π is *malformed*:

$$\pi \in \text{Malformed} \iff \neg\exists F \,.\, \text{Model}\big(\text{Parse}(\pi)\big) = F.$$

where F denotes an element of the set of all well-formed logical formulas, which means "there is no formula F such that parsing π succeeds and then building its model yields F."

We lift the verifier to a total function

$$\text{Verify} : \Pi \longrightarrow \{\text{Valid}, \text{Invalid}, \text{Rejected}, \text{Error}\},$$

where

$$\begin{aligned} \text{Valid} &\iff \pi \text{ well-formed} \wedge \text{model is unsatisfiable}, \\ \text{Invalid} &\iff \pi \text{ well-formed} \wedge \text{model is satisfiable}, \\ \text{Rejected} &\iff \pi \in \text{Malformed}, \\ \text{Error} &\iff \text{unexpected failure}. \end{aligned}$$

Thus every input receives an explicit, unambiguous classification.

The global safety theorem is refined to

$$\forall \pi \in \Pi \,.\, \text{Verify}(\pi) \in \{\text{Valid}, \text{Invalid}, \text{Rejected}\} \Longrightarrow \neg\exists (u, r, a) \,.\, \pi(u, r, a) = \text{Permit} \wedge \neg\varphi(u, r, a).$$

Hence a policy is either formally sound, accompanied by a counter-example, or conservatively rejected.

The totality of Verify establishes a logical trust boundary around the verifier itself: every candidate policy yields a deterministic verdict, and malformed inputs are always rejected. This maintains end-to-end soundness of the overall framework.

We now have a *mathematically defined framework* to:

- Model RBAC and ABAC in logical form.
- Express security properties as formulas $\varphi(u, r, a)$.
- Compare policies via delta analysis $\Delta(u)$.
- Formally verify soundness using SMT counterexample search.
- Systematically transform RBAC role bindings into ABAC rules while injecting verified safeguards.
- Handle malformed policies, preventing any undefined or unsound outcome.

We applied this unified verification framework retroactively to the three case studies, validating that it reproduces each known failure and its fix in a comparable manner. In each case, the framework's counterexample query finds the same underlying issue that was previously identified by the ad-hoc analyses, but now with a single standardized procedure.

IV. CASE STUDY OF REAL WORLD KUBERNETES SECURITY INCIDENTS

As previewed in Section II-D, this section revisits three documented Kubernetes security failures and examines them through the lens of our formal verification framework. These cases, each previously outlined at a high level, are now analyzed in detail to demonstrate how seemingly minor oversights (such as a mis-scoped registry prefix, an overly permissive RoleBinding, or a lack of namespace isolation) lead to exploitable gaps in cluster security.

Each scenario exemplifies a broader systemic weakness, supply-chain compromise via admission-controller misconfiguration, privilege escalation through RBAC overreach, and

isolation failure in multi-tenant setups. By formalizing the policies and invariants involved, and applying SMT-based validation, we show how the proposed framework detects these flaws and proves the correctness of their corresponding fixes, providing a structured and reproducible path from real-world incidents to verified policy enforcement.

A summary of the formal verification results for these scenarios, including the violated invariants, discovered counterexamples, and corresponding ABAC fixes is presented in Table 3.

A. CASE 1: IMAGE-POLICY BYPASS (ADMISSION-RULE MISCONFIGURATION)

1) THREAT DESCRIPTION

An attacker exploits a subtle admission-controller oversight to deploy a backdoored container image even though image provenance checks are enabled. Kubernetes RBAC controls *who* may deploy workloads but says nothing about *which* images they may run, so clusters rely on validating webhooks (e.g. OPA Gatekeeper or Kyverno) to restrict images to approved registries [46]. If that admission logic is misscoped here, a missing trailing slash in an allowed prefix an adversary can masquerade *myregistry.com.attacker.com/...* as belonging to *myregistry.com*, bypassing the intended safeguard [42]. The result is a classic supply-chain breach: once admitted, the malicious image executes with the same privileges as any sanctioned workload. Because the decisive check lives entirely in Gatekeeper, this case is *purely* an admission-controller misconfiguration; RBAC alone has no concept of image provenance.

2) FORMAL RBAC MODEL

Let U be users, R roles, and P permissions (action–resource pairs). User–role bindings $B \subseteq U \times R$ and role–permission assignments $A \subseteq R \times P$ yield each user's effective permission set

$$\text{Perms}_{\text{RBAC}}(u) = \bigcup_{(u,r)\in B} \{p \mid (r,p) \in A\},$$

and $\text{RBACAllow}(u,p)$ holds iff $p \in \text{Perms}_{\text{RBAC}}(u)$.

A policy is *unsound* w.r.t. an invariant $\varphi(u,p)$ (e.g. "image must come from an approved registry") when

$$\exists u \in U,\ p \in P:\ \text{RBACAllow}(u,p) \wedge \neg\varphi(u,p).$$

3) MISCONFIGURATION ENCODED

The intended invariant and the flawed Gatekeeper rule are

$$\varphi(\textit{registry}) := \text{"myregistry.com"},$$
$$\pi_{\text{bad}}(\textit{registry}) :=$$

prefix-match("myregistry.com",registry),

so any registry merely *prefixed* with the trusted domain is wrongly permitted.

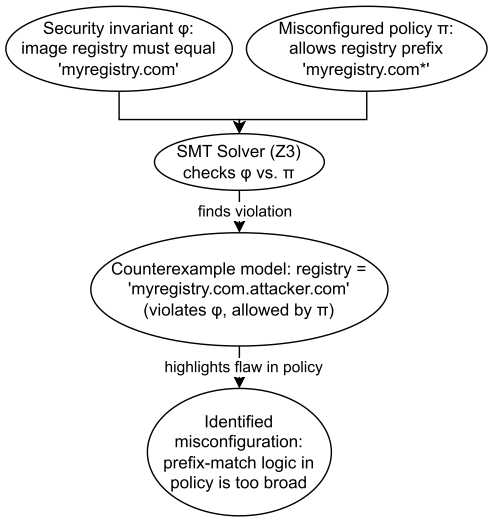

FIGURE 4. Flow of counterexample discovery for the image-policy bypass. The security invariant φ requires the registry to be exactly "myregistry.com,"but the misconfigured policy π permits any prefix "myregistry.com*."The SMT solver (Z3) checks φ against π, finds a counterexample where (registry = "myregistry.com.attacker.com"), and thus highlights that the prefix-match logic in π is too broad.

4) SMT VERIFICATION

To model the missing constraint (''no privileged pods unless allowed'') we assert *(assert (not (Privileged pod_i)))* for each pod *i*.

As shown in Figure 4, the SMT solver exposes a model that violates φ yet is permitted by π, illustrating the registry prefix flaw.

The SMT-LIB excerpt below checks policy soundness:

```
(set-logic QF_S)
(declare-fun registry () String)
(define-fun allowedRegistry () String ''myregistry
    .com'')

(assert (str.prefixof allowedRegistry registry))
(assert (not (= registry allowedRegistry)))

(check-sat)
(get-model)
```

Z3 returns

```
sat
(model
(define-fun registry () String
''myregistry.com.attacker.com''))
```

confirming the unsoundness: the malicious registry is permitted because only a prefix match was enforced.

5) ABAC REPAIR

Introduce an ABAC rule with deny-overrides semantics:

$$\pi_{\text{ABAC}}(\textit{registry}) := \begin{cases} \textit{Permit}, & \textit{registry} \in \textit{ApprovedSet}, \\ \textit{Deny}, & \text{otherwise}. \end{cases}$$

Formally,

$$\forall registry: \ \text{ABACAllow}(registry) = Permit \Rightarrow \varphi(registry).$$

Re-running the solver:

```
(assert (not (= registry allowedRegistry)))
(assert (str.prefixof allowedRegistry registry))
(check-sat)
```

yields *unsat*, proving the exact-match policy eliminates the exploit.

6) PRIVILEGE DELTA

$$\Delta(u) = \text{Perms}_{\text{RBAC}}(u) \setminus \text{Perms}_{\text{ABAC}}(u),$$

and indeed $\Delta(u) \neq \emptyset$: RBAC's unsound privilege (*myregistry.com.attacker.com*) is revoked by ABAC. Thus the formal analysis shows how an invariant and SMT check exposes the prefix-matching flaw and how an exact-match ABAC policy provably closes the gap, hardening container image security in Kubernetes.

B. CASE 2: PRIVILEGED ACCESS EXCEPTIONS (RBAC BYPASS/OVERREACH)

1) THREAT DESCRIPTION

Kubernetes RBAC is additive: roles grant permissions but provide no built-in notion of denial or exception [21], [47]. Attackers exploit this rigidity by (i) obtaining an overly privileged role through a configuration slip (RBAC *overreach*) or (ii) bypassing intended restrictions altogether (RBAC *bypass*). Once an initial foothold is gained, an adversary can escalate from limited rights to full *cluster-admin* by creating wildcard *ClusterRole* objects or abusing mis-scoped bindings, behaviour made possible because Kubernetes cannot natively express "allow X except Y" [4], [8]. Even without a CVE, a single role or binding mistake (or a compromised admin account) grants the attacker free rein in the cluster [43].

2) FORMAL RBAC MODEL

Let users U, roles R, permissions P; bindings $B \subseteq U \times R$; assignments $A \subseteq R \times P$.

$$\text{Perms}_{\text{RBAC}}(u) = \bigcup_{(u,r)\in B} \{p \mid (r,p) \in A\},$$
$$\text{RBACAllow}(u,p) \iff p \in \text{Perms}_{\text{RBAC}}(u).$$

3) MISCONFIGURATION AND INVARIANT

Let $A = \{\text{ClusterRole}, \text{ClusterRoleBinding}\}$. The security property is

$$\varphi(u,r,a) := \big(r \in A \wedge a = \text{"create"}\big) \Rightarrow u.admin = true.$$

A policy is unsound if $\exists u, r, a: \ \pi_{\text{bad}}(u,r,a) = Permit \wedge \neg\varphi(u,r,a)$.

4) SMT COUNTER-EXAMPLE

```
(set-logic QF_S)
(declare-fun admin () Bool)
(declare-fun resource () String)
(declare-fun action () String)
(assert (= action ``create''))
(assert (or (= resource ``ClusterRole'')
(= resource ``ClusterRoleBinding'')))
(assert (not admin))
(check-sat) (get-model)
```

sat $\longrightarrow$ a non-admin may create a *ClusterRole*, violating φ.

5) ABAC REPAIR (DENY-OVERRIDES)

$$\pi_{\text{ABAC}}(u,r,a) = \begin{cases} Permit, & u.admin = true, \\ Deny, & r \in A \wedge a = \text{"create"}, \\ Permit, & \text{otherwise.} \end{cases}$$

With this rule, any *Deny* overrides an *Allow*; admins are explicitly exempt so legitimate workflows remain intact.

6) SOUNDNESS PROOF

$$\forall u, r, a: \ \text{ABACAllow}(u,r,a) = Permit \Rightarrow \varphi(u,r,a).$$

Re-running Z3 (*unsat*) proves no counter-example remains.

7) PRIVILEGE DELTA

$$\Delta(u) = \text{Perms}_{\text{RBAC}}(u) \setminus \text{Perms}_{\text{ABAC}}(u),$$

and $\Delta(u) \neq \varnothing$ shows ABAC revokes the illicit "create ClusterRole" privilege for non-admins.

8) POLICY ORDERING

Because *deny-overrides* is active, the admin-permit rule is placed before the generic deny so that verified administrators retain legitimate powers while non-admins are blocked from lateral movement.

Note. AuthzForce's engine enforces *deny-overrides*. We therefore exclude administrators from the cross-tenant deny (or equivalently use first-applicable ordering) to avoid false denials.

9) OUTCOME

Formal modeling plus SMT analysis exposes RBAC's inability to enforce attribute-driven constraints and shows how an ABAC overlay provably eliminates the privilege-escalation path, closing a significant real-world gap.

C. CASE 3: MULTI-TENANT NAMESPACE ISOLATION FAILURES

1) THREAT DESCRIPTION

A Kubernetes cluster often hosts multiple tenants teams, business units, or external customers segregated only by namespaces. Isolation collapses when a misconfiguration (e.g. an overly broad *RoleBinding* or missing *NetworkPolicy*)

lets one tenant access another tenant's resources [15]. The attacker, after compromising a workload in *namespace-A*, can pivot laterally into *namespace-B* or escalate to cluster-wide privileges, violating intended tenancy boundaries, a classic lateral-movement scenario [52]. Because Kubernetes is not multi-tenant by default, every boundary depends on flawless configuration; one mis-scoped binding or an add-on with excess rights (e.g. Azure AKS Airflow's "shadow admin") breaks containment [45]. Threat actors actively abuse this gap: MITRE notes attackers creating additional cluster roles to dominate other namespaces [49], and real incidents show that a single namespace with a wildcard role can yield full cluster compromise [16].

2) FORMAL RBAC MODEL AND INVARIANT

Define users (or service accounts) U, namespaces N, and tenants T. Each user and namespace has a tenant label: $tenant(u)$ and $tenant(n)$. Permissions derive purely from (Cluster)Role,RoleBindings, none are tenant-aware.

The intended invariant is strict tenant isolation:

$$\varphi(u, n) : \ tenant(u) = tenant(n).$$

An unsound policy therefore admits

$$\exists u, n : \ \pi_{\text{bad}}(u, n) = Permit \ \wedge \ \neg\varphi(u, n).$$

3) SMT COUNTER-EXAMPLE

```
(set-logic QF_S)
(declare-fun userTenant () String)
(declare-fun namespaceTenant () String)
(declare-fun allowed () Bool)

(assert allowed); RBAC permits the request
(assert (not (= userTenant namespaceTenant)));
    tenant mismatch
(check-sat) (get-model)
```

Z3 returns

```
sat
(model
(define-fun userTenant () String ''teamA'')
(define-fun namespaceTenant () String ''teamB'')
(define-fun allowed () Bool true))
```

so RBAC wrongly lets *teamA* act inside *teamB*.

4) TENANT-AWARE ABAC REPAIR

$$\pi_{\text{ABAC}}(u, n) = \begin{cases} Permit, & \text{if } tenant(u) = tenant(n), \\ Deny, & \text{otherwise.} \end{cases}$$

With deny-overrides semantics, any tenant mismatch is immediately rejected.

Soundness check:

$$\forall u, n : \ \text{ABACAllow}(u, n) = Permit \ \Rightarrow \ \varphi(u, n).$$

```
(assert allowed)
(assert (not (= userTenant namespaceTenant)))
(assert (= allowed false)); ABAC forces Deny
(check-sat)
```

The solver now reports *unsat*, proving no cross-tenant access is possible.

5) PRIVILEGE DELTA

$$\Delta(u) = \text{Perms}_{\text{RBAC}}(u) \setminus \text{Perms}_{\text{ABAC}}(u),$$

and $\Delta(u) \neq \varnothing$ shows ABAC revokes RBAC's illegitimate cross-tenant privileges.

6) OUTCOME

Formal analysis reveals RBAC's lack of tenant context and demonstrates that an ABAC rule enforcing $tenant(u) = tenant(n)$ provably restores isolation, eliminating an entire class of multi-tenant escalation paths.

V. EXPERIMENTAL VALIDATION

To assess the practicality of our framework, we developed a prototype implementation and conducted targeted experiments [54]. The goal was not a large-scale performance evaluation, but a concrete demonstration that the logical model can be executed and enforced in a real Kubernetes environment, therefore a cluster-scale performance benchmark is intentionally out of scope for this work.

The prototype consists of two main components:

1) An offline verification tool that ingests Kubernetes policy definitions and checks them with an SMT solver.
2) An enforcement mechanism that integrates the formally verified policies into a live cluster.

A. PROTOTYPE IMPLEMENTATION

We built a Python CLI that ingests synthetic Kubernetes RBAC and admission-policy YAML, converts them into SMT-LIB constraints (per Section IV), and checks them with Z3 v4.12. Given any policy set and security invariant (e.g., "only users with attribute 'admin' may create sensitive resources"), Z3 looks for a model that violates the invariant while still satisfying the policies. A model flags a misconfiguration; no model proves the invariant holds.

In our architecture, as defined by our model (see Figure 5), the Kubernetes API server's admission control pipeline was extended with a custom validating webhook (written in Python) that functions as the Policy Enforcement Point. This webhook intercepts every incoming AdmissionReview request (which encapsulates an API operation under review) and translates it into an equivalent XACML request in JSON form. The request includes all relevant attributes (such as the user's identity/role, the target resource's attributes like namespace labels or image registry, the action being attempted, etc.) as defined by our model. In particular, the SMT solver (highlighted in Figure 5) was used to check each policy against its security invariant before runtime enforcement.

To illustrate how the verifier handles malformed or adversarial inputs, Figure 6 shows the internal steps of the SMT Solver. Each candidate policy is first processed by Parse

TABLE 3. Formal-verification results for the three misconfiguration scenarios. For each case the SMT solver finds a counterexample that violates the stated invariant φ under the native policy set; the rightmost column shows the abstract ABAC rule that eliminates the flaw and makes $\forall u, r, a : Allow(u, r, a) \Rightarrow \varphi(u, r, a)$ provable.

Case study	Security invariant (φ)	Counterexample discovered	ABAC fix (abstract logic)
Image-provenance bypass (unapproved image)	Pod image must originate from an approved registry: φ : "if a pod creation is allowed, then *image.repo* $\in$ *ApprovedList*."	Gatekeeper accepted myregistry.com.attacker.com/* because it matched the misscoped prefix myregistry.com; pod creation was erroneously permitted.	Require exact registry match: *Deny* pod creation if *image.repo* $\notin$ *ApprovedList*. Rule is formally validated so any unanchored prefix is rejected.
Privileged access exception (ClusterRole creation)	Only administrators may create cluster-scoped RBAC objects: $\varphi(u, r, a)$: $(r \in \{\text{ClusterRole}, \text{ClusterRoleBinding}\} \wedge a = \text{create}) \Rightarrow u.\text{admin} = \text{true}$.	Solver finds a non-admin service account permitted to create a ClusterRole (e.g. action=create, resource=ClusterRole, admin=false); violates φ.	Tenant-/attribute-based rule: *Deny* if $r \in \{\text{ClusterRole}, \text{ClusterRoleBinding}\} \wedge a = \text{create} \wedge u.\text{admin} \neq \text{true}$; otherwise *Permit*. Deny-overrides prevents privilege escalation.
Tenant-isolation failure (cross-namespace access)	Non-admins may only touch resources in their own tenant: $\varphi(u, n)$: $\text{tenant}(u) = \text{tenant}(n)$.	Service account airflowSA (tenant A) accessed objects in namespace of tenant B after a permissive binding; solver confirms $Allow$ while $\text{tenant}(u) \neq \text{tenant}(n)$.	Global tenant-match rule: *Permit* if $\text{tenant}(u) = \text{tenant}(n)$, else *Deny* (admins may carry an override attribute if needed). Makes the $\exists$ query unsatisfiable.

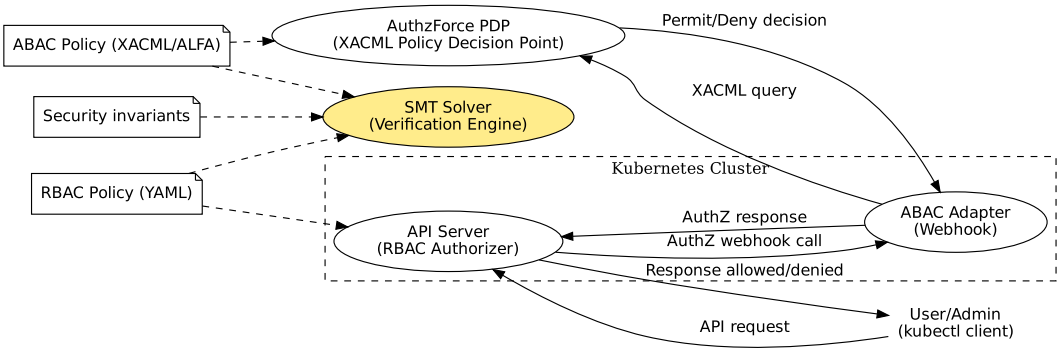

FIGURE 5. Integrated RBAC-ABAC verification and enforcement architecture. The Kubernetes API server (with built-in RBAC authorizer) delegates admission decisions to a custom ABAC adapter webhook, which translates requests into XACML for an external AuthzForce PDP. Separately, RBAC/ABAC policies and security invariants are fed to an SMT solver for offline verification. Solid arrows show the runtime request flow; dashed arrows indicate policy distribution and verification inputs.

to produce an abstract syntax tree and then by Model to yield a first-order formula. Finally, SATCheck produces one of four outcomes Valid, Invalid, Rejected, or Error ensuring that every malformed specification is explicitly rejected and no unsafe policy can slip through.

B. ENFORCEMENT IN A KUBERNETES CLUSTER

Having verified the policies, we next deployed them to a live Kubernetes environment to validate end-to-end enforcement. We set up a local Kubernetes cluster (using Minikube) and installed an external XACML Policy Decision Point (AuthzForce) to act on the verified ABAC rules.

The webhook then queries the AuthzForce PDP with this XACML-JSON payload and receives an authorization decision ('Permit' or 'Deny'). This setup mirrors Kubernetes's external authorization hook model, the API server pauses the request until the PDP's decision is returned, enforcing the outcome. We configured AuthzForce with the XACML policies that were proven sound, so that it would enforce exactly those rules in realtime.

Notably, this approach cleanly separates the policy decision logic from Kubernetes internals: the cluster delegates complex attribute-based decisions to the formally verified PDP, which in turn ensures that only requests satisfying the proven invariants are allowed. The use of AuthzForce (an off-the-shelf XACML engine) also demonstrates that our framework's output is compatible with standard ABAC technology, not just a custom solver environment.

In our experiments, we verified that:

- All well-formed but insecure ("bad") policies triggered the Invalid outcome in Figure 6, producing concrete counterexamples.
- All well-formed and secure ("fixed") policies were classified Valid, confirming their soundness.
- Any intentionally malformed input was classified Rejected, as shown by the dotted edges leading to the Rejected node in Figure 6.

While limited in scope (three focused case studies on a local cluster), this exercise proves the key point: the approach works in practice. The formal policy specifications and proofs

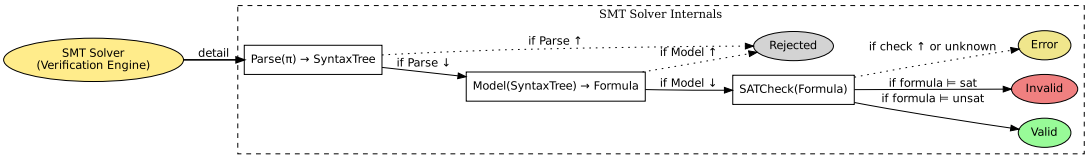

FIGURE 6. Expanded view of the SMT solver ("Verification Engine") showing its three internal stages i.e. Parse, Model, and SATCheck along with the four possible outcomes. A policy π is first parsed into a syntax tree, then modeled as a logical formula. The satisfiability check (SATCheck) yields Valid (unsat) or Invalid (sat). If parsing or model construction fails, the policy is Rejected, and if the solver itself fails or returns unknown, the result is Error.

from Section III can be realized as a working security mechanism that intercepts and prevents misconfigurations.

VI. DISCUSSION

Our findings highlight a clear issue: Kubernetes' built-in controls RBAC and admission controllers like Gatekeeper or Kyverno offer speed and flexibility, but fall short on assurance. There's no built-in mechanism to prove that a ruleset will behave as intended in all cases [55]. Contradictory or overlapping rules can quietly coexist, creating unseen risks. RBAC is tied to static roles, and even more advanced tools like Gatekeeper and Kyverno generally operate only on the object being created, not on richer context like user department, access time, or risk classification. Multiple studies applying formal methods to real-world clusters routinely uncover hidden policy conflicts, confirming this problem.

Attribute-based access control (ABAC) systems like XACML, particularly when paired with a PDP like AuthzForce fill many of these gaps. They let you express decisions based on users, resources, actions, and even environmental factors. Their semantics are precisely defined, and policies can be translated into logic for formal verification.

But ABAC also comes with tradeoffs. Its XML-based syntax is verbose and not user-friendly, writing policies can be fragile, and external PDPs may introduce latency. Kubernetes itself dropped its early ABAC implementation for exactly these reasons. In short, ABAC brings more power and precision but often at the cost of added complexity and operational friction.

Our framework tries to balance this tradeoff. It brings ABAC's expressiveness into Kubernetes, but adds SMT-based formal verification behind the scenes. Policies are written as logical constraints over user, resource, and environmental attributes. The system compiles the full policy set into SMT formulas and uses a solver to check for consistency and safety violations. If something's wrong like a misconfiguration or shadowed rule we find a concrete counterexample.

This gives administrators baked-in conflict detection and rich attribute reasoning, something Kubernetes lacks natively. And because this happens at design time, before deployment, it prevents flawed policies from ever being enforced.

The same pattern applies to other domains, too. Major cloud providers already use SMT-backed tools at scale (like AWS Zelkova or Tiros). CI/CD pipelines often include policy-like rules "two reviewers before deploy," for example, that could benefit from formal safety checks. Network policy frameworks, data governance systems, and policy-as-code tools more broadly could all see value when verification becomes as routine as writing a unit test.

Our SMT-based approach shares some DNA with prior formal policy analyzers, but diverges in important ways. Take Margrave, for example, a well-known tool for XACML analysis using decision diagrams. Later work built on Margrave with tools like Alloy to reason about role hierarchies and policy redundancy using first-order logic.

However, these tools operate entirely offline, analyzing static policy files. They weren't designed for direct integration into live systems. In contrast, AWS Zelkova provides scalable, SMT-backed policy analysis for cloud environments. It can prove that policies meet properties like "no public S3 access," but it too functions as an external check, not as an inline decision engine.

Our work aims squarely at Kubernetes, blending verification with enforcement. We don't just report that a policy has a flaw, we ensure that only verified policies reach the PDP that Kubernetes consults at runtime. To get there, we tailored the logic to Kubernetes' specific semantics and integrated the system carefully to avoid hurting performance.

Still, there are real hurdles to broader adoption:

- *Tooling*: The underlying solver is powerful, but to be useful, it needs accessible tools. Admins should be able to feed in familiar YAML policies and get clear diagnostics back. Without good interfaces, formal methods will remain niche.
- *Expertise*: Many DevOps teams don't have formal methods backgrounds. Right now, using this framework means understanding logical invariants and solver outputs. To make it more approachable, we need higher-level tools, smarter editors, and easy-to-understand counterexample explanations. Even simple, reusable templates for common security needs could help.
- *Complexity & Integration*: Specifying the right security invariants is itself a challenge. The model only checks what you ask it to, so missing or miswriting an invariant leaves gaps. That takes domain knowledge and careful threat modeling. On top of that, deploying an external PDP introduces some operational complexity. Making this modular and lightweight with containerized PDPs will be key.

Also, since this framework works at design time, it won't catch runtime threats or misuses that arise after deployment. Attacks involving stolen credentials or runtime exploits

remain out of scope. So while we gain rigor, we trade off some responsiveness.

Still, this work shows that it's possible to raise the bar significantly without altering Kubernetes itself. Attribute-rich, formally verified policies aren't just theoretical, we've shown them working in practice on real misconfigurations. The next step is to refine the system so that formal verification becomes a routine part of the policy development lifecycle.

Ultimately, if we can make this balance work rigorous checks with usable tooling we can close critical gaps in Kubernetes security and help prevent the kinds of silent policy flaws that attackers rely on.

VII. CONCLUSION

We introduced a formally verified ABAC layer for Kubernetes that converts RBAC and admission policies into first-order logic, checks them with an SMT solver, and enforces only those rules that satisfy stated security invariants. The framework closes three long-standing gaps, conflict detection, attribute expressiveness and proof of policy correctness, without modifying the Kubernetes core, and it demonstrably prevents misconfigurations that would otherwise yield image-supply-chain, privilege-escalation, and multi-tenant breaches.

Because verification is performed offline, the current prototype does not yet cover runtime misconfigurations or PDP availability, nor does it integrate seamlessly into CI/CD pipelines. Future work will automate invariant authoring, cache solver results for incremental checks, and pair the offline proofs with lightweight runtime monitors. These extensions aim to make mathematically grounded policy assurance a routine part of cluster operations rather than a specialised add-on.

In short, the paper shows that rigorous, solver-backed analysis can be brought to everyday Kubernetes deployments, turning fragile policy files into mathematical guarantees. This lowers the bar for adopting SMT based verification for access control policy management for Kubernetes clusters.

REFERENCES

[1] E. Russell and K. Dev, "Centralized defense: Logging and mitigation of kubernetes misconfigurations with open source tools," 2024, *arXiv:2408.03714*.

[2] Kubernetes. (2024). *Cluster Architecture–Kubernetes Documentation*. Accessed: Jun. 12, 2025. [Online]. Available: https://kubernetes.io/docs/concepts/architecture/

[3] G. Rostami, "Role-based access control (RBAC) authorization in kubernetes," *J. ICT Standardization*, pp. 237–260, Sep. 2023.

[4] J. Currey, R. McKinstry, A. Dadgar, and M. Gritter, "Informed privilege-complexity trade-offs in RBAC configuration," *Proc. 25th ACM Symp. Access Control Models Technol.*, vol. 6, 2020, pp. 119–130.

[5] Z. Lichen, "Research progress on attribute-based access control," Tech. Rep., 2010.

[6] B. Gajbhiye, O. Goel, and P. K. Gopalakrishna Pandian, "Managing vulnerabilities in containerized and kubernetes environments," *J. Quantum Sci. Technol.*, vol. 1, no. 2, pp. 59–71, Jun. 2024.

[7] A. M. Tall and C. C. Zou, "A framework for attribute-based access control in processing big data with multiple sensitivities," *Appl. Sci.*, vol. 13, no. 2, p. 1183, Jan. 2023.

[8] V. C. Hu, D. F. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, and K. Scarfone, "Guide to attribute based access control (ABAC) definition and considerations," Tech. Rep., 2014.

[9] E. Malul, Y. Meidan, D. Mimran, Y. Elovici, and A. Shabtai, "GenKubeSec: LLM-based kubernetes misconfiguration detection, localization, reasoning, and remediation," 2024, *arXiv:2405.19954*.

[10] B. Kim and S. Lee, "KubeAegis: A unified security policy management framework for containerized environments," *IEEE Access*, vol. 12, pp. 160636–160652, 2024.

[11] D. Pahuja, A. Tang, and K. Tsoutsman, "Automated SELinux RBAC policy verification using SMT," 2023, *arXiv:2312.04586*.

[12] K. Arkoudas, R. Chadha, and J. Chiang, "Sophisticated access control via SMT and logical frameworks," *ACM Trans. Inf. Syst. Secur.*, vol. 16, no. 4, pp. 1–31, Apr. 2014.

[13] A. Fatima, Y. Ghazi, M. A. Shibli, and A. G. Abassi, "Towards attribute-centric access control: An ABAC versus RBAC argument," *Secur. Commun. Netw.*, vol. 9, no. 16, pp. 3152–3166, Nov. 2016.

[14] K. Riad, Y. Zhu, H. Hu, and G. Ahn, "AR-ABAC: A new attribute based access control model supporting attribute-rules for cloud computing," in *Proc. IEEE Conf. Collaboration Internet Comput. (CIC)*, 2015, pp. 28–35.

[15] C. Shankar Kummarapurugu, "Role-based access control in cloud-native applications: Evaluating best practices for secure multi-tenant kubernetes environments," *World J. Adv. Res. Rev.*, vol. 1, no. 2, pp. 45–53, Mar. 2019.

[16] Md. S. Islam Shamim, F. Ahamed Bhuiyan, and A. Rahman, "XI commandments of kubernetes security: A systematization of knowledge related to kubernetes security practices," in *Proc. IEEE Secure Develop. (SecDev)*, Sep. 2020, pp. 58–64.

[17] S. Das, B. Mitra, V. Atluri, J. Vaidya, and S. Sural, "Policy engineering in RBAC and ABAC," Tech. Rep., 2018, pp. 24–54.

[18] K. Soni and S. Kumar, "Comparison of RBAC and ABAC security models for private cloud," in *Proc. Int. Conf. Mach. Learn., Big Data, Cloud Parallel Comput. (COMITCon)*, Feb. 2019, pp. 584–587.

[19] S. Long and L. Yan, "RACAC: An approach toward RBAC and ABAC combining access control," in *Proc. IEEE 5th Int. Conf. Comput. Commun. (ICCC)*, Dec. 2019, pp. 1609–1616.

[20] G. Zheng and Y. Xiao, "A research on conflicts detection in ABAC policy," in *Proc. IEEE 7th Int. Conf. Comput. Sci. Netw. Technol. (ICCSNT)*, Oct. 2019, pp. 408–412.

[21] G. Batra, V. Atluri, J. Vaidya, and S. Sural, "Enabling the deployment of ABAC policies in RBAC systems," in *Proc. 32nd Annu. IFIP WG 11.3 Conf. Data Appl. Secur. Privacy XXXII*, Jul. 2018, pp. 51–68.

[22] R. Kuhlisch, "Modeling and recognizing policy conflicts with resource access requests on protected health information," *Complex Syst. Informat. Model. Quart.*, no. 11, pp. 1–19, Jul. 2017.

[23] E. Zahoor, M. Chaudhary, S. Akhtar, and O. Perrin, "A formal approach for the identification of redundant authorization policies in kubernetes," *Comput. Secur.*, vol. 135, Dec. 2023, Art. no. 103473.

[24] A. Meshram, S. Das, S. Sural, J. Vaidya, and V. Atluri, "Abacaas: Attribute-based access control as a service," *Proc. 9th ACM Conf. Data Appl. Secur. Privacy*, 2019.

[25] S. Jha, S. Sural, V. Atluri, and J. Vaidya, "An administrative model for collaborative management of ABAC systems and its security analysis," in *Proc. IEEE 2nd Int. Conf. Collaboration Internet Comput. (CIC)*, 2016, pp. 64–73.

[26] S. Jadhav and N. Pise, "Secure and transparent blockchain donations: An attribute-based access control (ABAC) framework for enhanced donor control," in *Proc. IEEE Int. Conf. Blockchain Distrib. Syst. Secur. (ICBDS)*, Oct. 2024, pp. 1–6.

[27] A. Rizzardi, S. Sicari, and A. Coen-Porisini, "Attribute-based policies through microservices in a smart home scenario," *Comput. Commun.*, vol. 231, Feb. 2025, Art. no. 108039.

[28] M. Burmester, E. Magkos, and V. Chrissikopoulos, "T-ABAC: An attribute-based access control model for real-time availability in highly dynamic systems," in *Proc. IEEE Symp. Comput. Commun. (ISCC)*, Jul. 2013, pp. 143–148.

[29] B. Stepien, A. Felty, and S. Matwin, "Challenges of composing XACML policies," in *Proc. 9th Int. Conf. Availability, Rel. Secur.*, Sep. 2014, pp. 234–241.

[30] T. Nelson, "Margrave: An improved analyzer for access-control and configuration policies," Tech. Rep., 2010.

[31] G. Hughes and T. Bultan, "Automated verification of access control policies," Tech. Rep., 2004.

[32] B. Abdelkrim, "Secure ehr access in the cloud: An alloy-based formalization of abac in collaborative and non-collaborative models," in *Studies in Engineering and Exact Sciences*, 2024.

[33] N. Huynh, M. Frappier, A. Mammar, and R. Laleau, "Verification of SGAC access control policies using alloy and ProB," in *Proc. IEEE 18th Int. Symp. High Assurance Syst. Eng. (HASE)*, Jan. 2017, pp. 120–123.

[34] G. Liu, W. Pei, Y. Tian, C. Liu, and S. Li, "A novel conflict detection method for ABAC security policies," *J. Ind. Inf. Integr.*, vol. 22, Jun. 2021, Art. no. 100200.

[35] L. Krautsevich, A. Lazouski, F. Martinelli, and A. Yautsiukhin, "Towards policy engineering for attribute-based access control," Tech. Rep., 2013, pp. 85–102.

[36] A. K. Bandara, E. C. Lupu, and A. Russo, "Using event calculus to formalise policy specification and analysis," in *Proc. POLICY IEEE 4th Int. Workshop Policies Distrib. Syst. Netw.*, 2003, pp. 26–39.

[37] M. Charalambides, P. Flegkas, G. Pavlou, J. Rubio-Loyola, A. K. Bandara, E. C. Lupu, A. Russo, N. Dulay, and M. Sloman, "Policy conflict analysis for diffserv quality of service management," *IEEE Trans. Netw. Service Manage.*, vol. 6, no. 1, pp. 15–30, Mar. 2009.

[38] E. Zahoor, Z. Asma, and O. Perrin, "A formal approach for the verification of aws iam access control policies," Tech. Rep., 2017, pp. 59–74.

[39] M. Ait El Hadj, A. Khoumsi, Y. Benkaouz, and M. Erradi, "Formal approach to detect and resolve anomalies while clustering ABAC policies," *ICST Trans. Secur. Saf.*, vol. 5, no. 16, Dec. 2018, Art. no. 156003.

[40] Y. Li, X. Hu, C. Jia, K. Wang, and J. Li, "Kano: Efficient cloud native network policy verification," *IEEE Trans. Netw. Service Manage.*, vol. 20, no. 3, pp. 3747–3764, Sep. 2023.

[41] M. Fernández and B. Thuraisingham, "A category-based model for ABAC," in *Proc. 3rd ACM Workshop Attribute-Based Access Control*, Mar. 2018, pp. 32–34.

[42] A. M. Y. Kadkoda. (2025). *Opa Gatekeeper Bypass Reveals Risks in Kubernetes Policy Engines*. [Online]. Available: https://www.aquasec.com/blog/risks-misconfigured-kubernetes-policy-engines-opa-gatekeeper/#:~:text=A%20,This%20bypasses%20the%20policy%20entirely

[43] A. M. M. Katchinskiy. (2023). *First-ever Attack Leveraging Kubernetes RBAC to Backdoor Clusters*. [Online]. Available: https://www.aquasec.com/blog/leveraging-kubernetes-rbac-to-backdoor-clusters/#:~:text=using%20DaemonSets%20to%20run%20Monero,cryptominers

[44] A. M. M. Katchinskiy. (2023). *Kubernetes Exposed: One Yaml Away From Disaster*. [Online]. Available: https://www.aquasec.com/blog/kubernetes-exposed-one-yaml-away-from-disaster/#:~:text=By%20default%2C%20the%20anonymous%20user,a%20severe%20misconfiguration%20is%20created

[45] S. Buchanan, J. Rangama, and N. Bellavance, "Deploying Azure kubernetes service," Tech. Rep., 2019, pp. 63–77.

[46] N. K. Amrutham, "Optimizing kubernetes environments: Best practices for configuring and managing admission webhooks," *Int. J. Res. Appl. Sci. Eng. Technol.*, vol. 12, no. 10, pp. 1044–1051, Oct. 2024.

[47] P. Priya Patharlagadda, "Enhance the application security using kubernetes role based access control for applications," *Int. J. Sci. Res. (IJSR)*, vol. 8, no. 8, pp. 2339–2342, Aug. 2019.

[48] SentinelOne. (2024). *Top 10 Kubernetes Security Issues*. [Online]. Available: https://www.sentinelone.com/cybersecurity-101/cloud-security/kubernetes-security-issues/#:~:text=%2A%20CVE,in%20LoadBalancer%20and%20ExternalIPs%20services

[49] MitreATTACK. (2023). *Account Manipulation: Additional Container Cluster Role*. [Online]. Available: https://attack.mitre.org/versions/v16/techniques/T1098/006/

[50] F. Pizzato, D. Bringhenti, R. Sisto, and F. Valenza, "An intent-based solution for network isolation in kubernetes," in *Proc. IEEE 10th Int. Conf. Netw. Softwarization (NetSoft)*, Jun. 2024, pp. 381–386.

[51] P. S. Patchamatla, "Optimizing kubernetes-based multi-tenant container environments in OpenStack for scalable AI workflows," *Int. J. Adv. Res. Educ. Technol.*, vol. 5, no. 3, Mar. 2018.

[52] R. Molleti, "Highly scalable and secure kubernetes multi tenancy architecture for fintech," *J. Eng. Appl. Sci. Technol.*, pp. 1–5, Jun. 2022.

[53] V. C. Hu, D. R. Kuhn, D. F. Ferraiolo, and J. Voas, "Attribute-based access control," *Computer*, vol. 48, no. 2, pp. 85–88, Feb. 2015.

[54] A. Sissodiya. (2025). *K8s-ABAC-Verification-Demo: Scripts and Code for the Paper Closing RBAC Gaps*. version v1.0, commit a1b2c3d. Accessed: May 30, 2025. [Online]. Available: https://github.com/adityasissodiya/k8s-abac-verification-demo

[55] D. Servos and S. L. Osborn, "Hgabac: Towards a formal model of hierarchical attribute-based access control," Tech. Rep., 2014, pp. 187–204.

ADITYA SISSODIYA received the B.Sc. (Hons.) degree in computer science from Doon University and the M.Sc. degree in computer science from Christ University, India. He is currently pursuing the Ph.D. degree in cyber-physical systems with Luleå University of Technology, Sweden. Before beginning his Ph.D. studies, he was a Software Engineer for four years in industry, contributing to supply chain optimization and healthcare technologies. His research interests include access control and secure collaboration mechanisms, with a recent focus on technologies supporting the circular economy.

ERIC CHIQUITO received the B.S. degree in nanotechnology engineering from Instituto Tecnológico y de Estudios Superiores de Occidente (ITESO), Mexico, and the Ph.D. degree in cyber-physical systems from Luleå University of Technology (LTU), Sweden, in 2024. He is currently a Postdoctoral Researcher with LTU. Prior to this, he was a Data Scientist with the Largest Hotel Group, Grupo Posadas, Mexico. His research interests include decentralized systems, data sharing, circular economy, and matchmaking mechanisms.

ULF BODIN received the Ph.D. degree in computer networking from Luleå University of Technology. He is currently a Professor with Luleå University of Technology, where he is conducting research on the industrial IoT, distributed system of systems, computer communications, distributed ledgers, and applied machine learning. He has more than 15 years of experience in academia and the software industry, including standardization in ETSI and other organizations.

JOHAN KRISTIANSSON received the Ph.D. degree in multimedia technology from Luleå University of Technology, where he is currently a Senior Lecturer in cyber-physical systems. He has extensive industry experience, including more than a decade at Ericsson, where he was a Principal Researcher specializing in container orchestration. As an early adopter of Docker, he explored microservices architectures and container security, and developed an in-house orchestration platform at Ericsson, in 2014. He later served as the CTO of RockSigma AB, where he built a Kubernetes-based seismic processing engine. He is also the Lead Developer of ColonyOS, a meta-OS designed to create computing continuums. He has authored more than 25 scientific publications and holds 23 patents in distributed computing, cloud computing, and container orchestration.