Toward a Query-Driven Approach for Formal Verification of Cloud Security Configuration

Toward a Query-Driven Approach for Formal Verification of Cloud Security Configuration

Francesco Pizzato, Daniele Bringhenti, Riccardo Sisto, Fulvio Valenza
Dip. Automatica e Informatica, Politecnico di Torino, Torino, Italy, Emails: {first.last}@polito.it

***Abstract*—Managing security configuration in cloud environments is increasingly challenging due to their dynamism, and traditional manual inspection and testing are inadequate to ensure correctness and prevent misconfiguration. To address these issues, this paper introduces a query-driven formal verification approach that enables administrators to validate cloud security configurations using a high-level, user-friendly query language. Correctness is ensured by modeling the cloud environment and verification queries as a Satisfiability Modulo Theories problem, automatically solved by a state-of-the-art solver. Moreover, the approach covers diverse security aspects and detects complex multi-level attacks by integrating multiple security domains.**

***Index Terms*—cloud security, formal verification, Kubernetes**

I. Introduction

As cloud environments often host critical services and applications, their security configuration has become a significant management task for cloud administrators. In the case of Kubernetes clusters, two configuration aspects that should be correctly set up are Role-Based Access Control (RBAC) policies, to ensure authorization for all operations accessing the cluster's resources, and Network Policies, to control how entities, such as pods, can communicate with each other or with external elements. Unfortunately, the scale and dynamism of cloud environments increased the complexity of ensuring the correctness of their security configurations. Consequently, human administrators often create discordances with respect to the intended security policies [1] and open the door to attacks and data breaches [2].

Manual inspection and testing are no longer sufficient for validating if a cloud configuration correctly enforces some security policies, as those approaches are not compatible with the complexity and dynamism of cloud environments. Therefore, in the literature, formal verification methods started to be adopted as a starting point for the creation of human-aiding tools [3]. The reason is that, by providing mathematically sound assurance, formal methods can automatically detect subtle misconfigurations, identify policy conflicts, and help align the deployed setup with the intended security objectives. However, despite their potential, existing approaches commonly focus on a limited subset of configuration aspects or lack usability features such as user-friendly query languages.

In order to fill this gap in the literature, this paper proposes a formal approach for cloud security configuration verification, so that it can help administrators in verifying the result of their security policy enforcement, specifically in Kubernetes clusters. On the one hand, this approach is based on a high-level query language, which abstracts from the low-level technicalities of the managed cloud environment, so that administrators can efficiently use it to express specific security properties or best practices they would like to check. On the other hand, this methodology formalizes the configuration verification problem as a Satisfiability Modulo Theories (SMT) problem. In this way, the problem can be automatically solved by state-of-the-art solvers, providing formal assurance about the correctness of the result. Moreover, the proposed approach models a larger number of security features than alternative solutions, so that more attacks can be found, including lateral movements and multi-level attacks [4].

The remainder of this paper is structured as follows. Section II analyzes the related work. Section III presents the approach. Section IV validates the methodology. Section V draws conclusions and outlines future work.

II. Related Work

Several cloud configuration verification solutions have been proposed in literature in recent years. However, unlike the solution proposed in this paper, they focus on a single configuration aspect (i.e., access control or Network Policies), and they lack a query language allowing a user-friendly interaction.

Concerning access control, early work on anomaly detection in firewalling and authorization systems laid the foundations for subsequent cloud-oriented approaches [5]–[8]. More recent studies specifically target cloud platforms and Kubernetes environments, addressing misconfiguration detection in AWS IAM policies [9] or formal reasoning over Kubernetes authorization policies using logic-based representations, such as formal Event-Calculus [10], [11] or first-order logic models combined with SMT solvers [12].

Concerning network reachability, some studies [13]–[15] investigated Kubernetes Network Policy verification techniques, so as to overcome the limitations of the solutions originally proposed for traditional computer networks [16]–[19]. In particular, Kano [14] introduces a matrix-based verification algorithm for container Network Policies, improving scalability compared to prior traditional approaches. Verikube [13] adopts a graph-based representation of policies combined with an SMT solver, enabling more fine-grained verification up to L4/L7 policies, though its applicability remains limited to Cilium. NPV [15] is based on a symbolic execution method for Kubernetes Network Policy verification, achieving faster and accurate detection of conflicts and anomalies at scale.

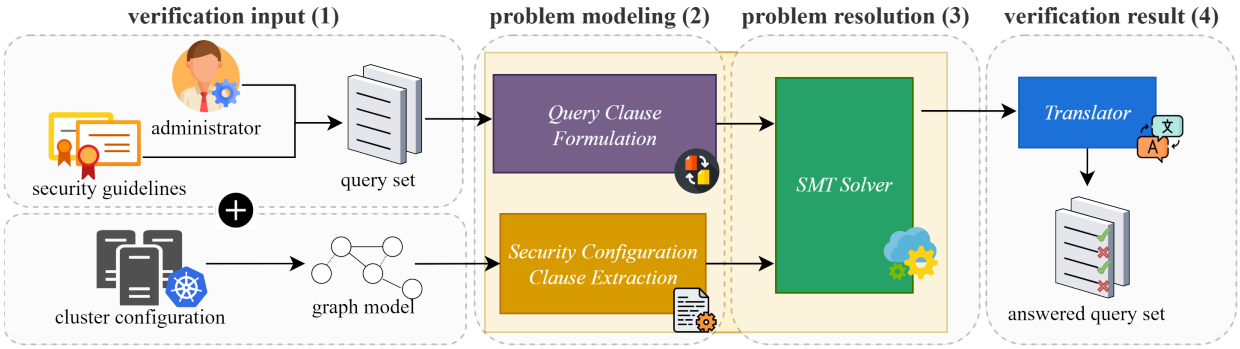


Fig. 1: Architecture of the proposed verification approach

All the approaches discussed so far are not only limited to a single cloud configuration scope, but do not provide either broad usability features, such as high-level query languages or policy explanations for administrators. These features are instead offered by other state-of-the-art solutions [20]–[23].

On the one hand, a semantic model is used in [20] to detect security weaknesses such as unintended data flows or misconfigured storage across heterogeneous cloud platforms. However, its applicability is limited by the need to access and analyze source code, often unavailable in real-world deployments. Instead, [21] proposes a stress-test methodology for cloud configurations, combining permissions, capabilities, and known vulnerabilities into knowledge graphs to support "what-if" analyses. However, these two approaches do not provide formal verification guarantees, unlike SMT-based approaches.

On the other hand, SMT-based approaches provide formal guarantees for specific domains, such as access control [22] or network reachability [23] in AWS deployments. Nonetheless, these approaches are tailored to particular cloud platforms, remaining limited in generality.

III. The Proposed Approach

This section presents the proposed approach for verifying cloud security configuration, helping the administrator assess compliance with a set of queries to detect errors or discrepancies. The approach, shown in Fig. 1, comprises four main phases: the verification inputs, the SMT problem modeling, the problem resolution, and the verification output.

A. Verification Inputs

The required verification inputs are two: 1) the information related to the configuration of the administered cluster, and 2) the queries expressing the properties to be verified.

1) Cluster Configuration: The cluster configuration information provided as input must exhaustively describe all deployed resources (e.g., Pods, Namespaces, Services) and their security properties. In particular, at the moment, the proposed approach supports only RBAC and Network Policies, which are the most important security features in a cluster. However, the approach is flexible enough to allow further extensions, so as to include other security aspects with minor updates.

$\langle \mathbb{Q}_{predicate} \rangle ::= \langle root \rangle\ \langle predicate \rangle | \langle \mathbb{Q}_{predicate} \rangle\ (and|or) \langle \mathbb{Q}_{predicate} \rangle$
$\langle \mathbb{Q}_{function} \rangle ::= \langle root \rangle\ \langle function \rangle | \langle \mathbb{Q}_{function} \rangle\ (in|notin)\ \langle \mathbb{Q}_{function} \rangle$
$\langle root \rangle := \langle role \rangle | \langle subject \rangle | \langle endpoint \rangle$
$\langle role \rangle ::= Role(name)$
$\langle subject \rangle ::= User(name) | Group(name) | ServiceAccount(name)$
$\langle endpoint \rangle ::= Pod@namespace[\#(name|label)] | IP_{CIDR}$
$\langle predicate \rangle ::= [not]\ has\ \langle parameter\_type \rangle : parameter\_value | \langle predicate \rangle (and|or) \langle predicate \rangle$
$\langle function \rangle ::= list\ \langle parameter\_type \rangle\ [if\ \langle predicate \rangle] | \langle function \rangle (in|notin) \langle function \rangle$
$\langle parameter\_type \rangle ::= action|role|subject|resource|connection|permission$
$\langle resource \rangle ::= type@namespace\#name$
$\langle connection \rangle ::= \langle endpoint \rangle, dstPort, transportProto, \langle mode \rangle$
$\langle mode \rangle ::= ingress|egress|bidirectional$
$\langle permission \rangle ::= (\langle resource \rangle, \langle action \rangle)$
$\langle action \rangle ::= get|update|list|delete|patch|...$

TABLE I: Query Grammar.

The cluster configuration can be retrieved automatically through Kubernetes APIs, but it must be processed into an intermediate representation before moving to the next phase. Specifically, the configuration is modeled as a graph, where vertices represent cluster resources and edges encode their relationships. For RBAC, non-labeled edges represent user–role assignments, while labeled edges capture role permissions with allowed actions as labels. For Network Policies, labeled edges represent endpoint connectivity, including transport protocol and destination port. Non-labeled edges are also used to represent hierarchical relationships among resources.

2) Queries: The properties to be verified are expressed through a custom-defined query language. This language is characterized by a high level of abstraction, so that also non-expert users can use it to verify complex security configuration aspects. It has been designed to support a wide range of queries and with a focus on extensibility, i.e., all of them are currently used for RBAC and Network Policies, but could be extended with the modeling of other security features. Currently, the language supports queries for correctness and consistency verification, role auditing and optimization, hypothetical (what-if) analysis, and holistic checks combining RBAC and Network Policies. Typical use cases include access authorization checks, detection of conflicting permissions (e.g., separation of duties), analysis of role assignments, impact evaluation of configuration changes, and verification of isolation and data-exfiltration constraints. The grammar of the language is presented in TABLE I. Each query is in the form of a predicate

or a function, depending on the return value. Predicate-based queries can be combined using logical operators, whereas function-based queries can be combined with set operators. A predicate example is *has*, whereas a function example is *list*. Moreover, to increase the language's expressiveness, special characters are supported for query parameters. In particular, the wildcard symbol "*" represents all elements of the set, i.e., *Role(*)* represents all roles, and, if the parameter is a set, the symbol "+" indicates one or more elements in the set, i.e., *+User(*)* indicates any combination of one or more users.

B. Problem Modeling

Once both inputs are defined, they are processed by dedicated modules, in charge of creating the clauses of an SMT problem: 1) the *Security Configuration Clause Extraction* module derives constraints related to the security configuration from the input graph, while 2) the *Query Clause Formulation* module encodes the user queries as logical clauses.

The constraints formulated by these two modules rely on a formal model that abstracts the cluster composition and configuration. This model is characterized by a set of entities representing the main elements of cloud environments, such as users, roles, resources, and network endpoints, and by a collection of predicates and functions capturing their relationships and operational semantics. Specifically, authorization is modeled through role assignments, permissions, and some additional derived predicates. For instance, an effective authorization is captured by the predicate $\texttt{CanAccess}(u, a, o)$, which holds if a user u is assigned a role granting action a on resource o. Network connectivity predicates represent allowed communication paths between endpoints, accounting for protocol, port ranges, and directionality. Finally, additional predicates encode other semantic aspects, such as hierarchical relations among resources and action applicability constraints.

These model elements are used by the two SMT clause formulation modules as follows.

1) Security Configuration Clause Extraction: The first module works on the graph model representing the cluster composition and configuration. As first operation, all entities are mapped to uninterpreted sorts in the SMT domain, enabling symbolic reasoning without relying on concrete identifiers. Second, the module generates a set of clauses that constrain the interpretation of authorization and connectivity predicates, as well as hierarchical and other predicates, based on the observed configuration. These clauses bind the abstract model to the concrete behavior of the deployed system.

2) Query Clause Formulation: The second module processes the input queries to create another set of clauses. For their definition, the same entity models, functions and predicates previously introduced are employed. However, here each clause contains open variables, to which the output of functions and predicates is mapped, so that their value assignment is later computed by the SMT solver as output.

If a query implies just an affirmative or negative answer, a single open Boolean variable is enough. For instance, a permission check can be encoded as $q = \texttt{CanAccess}(u, a, o)$, with q evaluated by the solver. Instead, if a query requires an explicit value as an answer, e.g., to retrieve multiple entities satisfying a certain condition, its formalization as a clause is more complex, because it may require either multiple open Boolean variables, one for each candidate object to retrieve, or a variable of a datatype created specifically for this purpose and representing a list of entities.

C. Problem Resolution

The resulting clause sets of the problem modeling phase correspond to the inputs of the problem resolution phase. There, they are combined and fed to a state-of-the-art SMT solver. Modern SMT solvers implement efficient resolution strategies, enabling quick computation of a valid solution to the problem. Moreover, since the formal correctness of the algorithms implemented in such solvers has long been proven, the solution is formally correct as long as all model elements faithfully represent the real cluster security configuration.

The solver's output is the assignment for the open variables introduced in the clauses modeling the query. However, it may not be immediately comprehensible for users inexperienced in constraint programming.

D. Verification Result

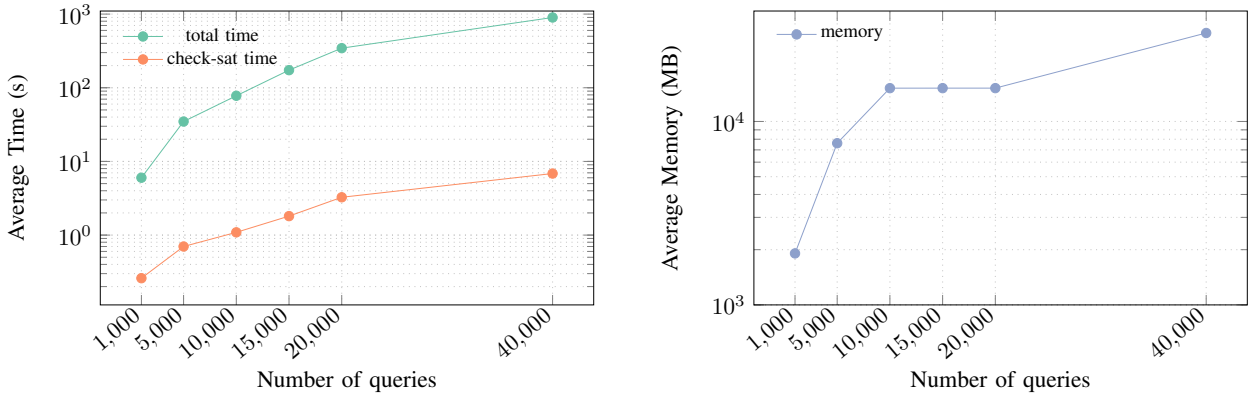
To provide the end user with an intelligible result, the solution to the SMT problem must be translated and mapped back to the original set of queries. Therefore, the assertion results are parsed via a dedicated module, i.e., the translator module, to map each one of them to its originating query.

In particular, the answer to a query of type $\mathbb{Q}_{predicate}$ does not need any complex translation operation, since its clause formulation involved a single open Boolean variable.

Instead, the translation operations required for queries of type $\mathbb{Q}_{function}$ are slightly more complex. In this case, the clause formulation of a single query may involves multiple open Boolean variables as the goal is to find a set of objects, e.g., resources, roles, etc., answering the query. If a variable is mapped to true by the solver, it means that the corresponding object must be considered in the answer, while if it is mapped to false, it must be excluded from the answer. In the end, the answer assigned to the query is the set of objects whose corresponding variables were assigned the true value by the solver. Instead, if the clause involves an open variable of a specifically crafted datatype rather than multiple Boolean ones, the solver assigns to it a symbolic value encoding the collection of entities satisfying the query condition. This result is then decoded by the translator module.

IV. Implementation and Validation

The approach has been implemented using Python for all the designed modules and the Z3 solver [24] for the resolution of the SMT problem, exploiting its Python API. The implementation has been validated through a quantitative evaluation that focuses on the SMT problem resolution with different scenarios designed to stress this component, being the most computationally intensive phase of the approach.



(a) Computation time vs number of queries

(b) Memory usage vs number of queries

Fig. 2: Scalability of the problem resolution with respect to varying number of queries

To quantitatively evaluate the scalability of the problem resolution phase, a generator was created to produce synthetic SMT-LIB files of varying problem complexity. In particular, the tested scenarios are characterized by an increasing number of queries and a fixed cluster size. These tests were conducted using v4.15.3 for Z3, on a laptop equipped with an Intel Core Ultra 7 155H and 32 GB of memory. The results are shown in Fig. 2. More specifically, the scalability test evaluates how the computation time and memory usage increase with respect to the number of queries. For the results in Fig. 2a and Fig. 2b, the size of the cluster is fixed (1,000 users, 1,000 roles, 1,000 IP addresses, and 1,200 resources), while the number of queries ranges from 1,000 to 40,000. The set of queries is uniformly distributed among the different types to ensure uniform problem complexity, ranging from simple correctness verification queries (e.g., "Is u_1 allowed to do a_{get} on o_3?") up to cross-domain interrogations using multiple existential quantifiers (e.g., "Is there a user u_i able to do a_{create} on any $o_{pod/exec}$ which can communicate with $IP_{external}$?"). Ten different problem instances were solved for each different number of query, so that average results could be computed. On the one hand, Fig. 2a reports the average "total time" and "check-sat time" estimated for problem resolution. The "total time" includes the times of all operations executed by Z3 from the moment it is invoked, including parsing the input file and setting up the internal problem state. Meanwhile, the "check-sat time" is the fraction of the total time during which the engine actively searches for a valid interpretation. As shown, the highest tested scenario with 40,000 queries is solved in less than 15 minutes. This is orders of magnitude smaller than what could be expected from manual configuration. Besides, it must be noted that the verification of each query is independent and thus highly parallelizable. On the other hand, Fig. 2b plots the memory cost. As expected, memory represents a more significant scalability challenge. However, even the largest scenario requires just above 30GB, which was manageable with the 32GB of the test machine.

Overall, the scalability results show that memory consumption, rather than computation time, becomes the main bottleneck, mainly due to the number of symbolic entities and Boolean variables required to encode complex queries involving existential quantifiers. It is important to note, however, that the presented validation considers deliberately pessimistic synthetic scenarios, which are not necessarily representative of typical production Kubernetes deployments. Empirical data from large-scale industry surveys show that real-world clusters are generally much smaller and more fragmented, since operators tend to partition the workloads (e.g., namespaces, teams, or multiple clusters). For instance, recent measurements across thousands of production clusters report a median cluster size of approximately a few hundred pods, which is a modest footprint when compared to the tested scenarios [25]. Furthermore, several mitigation strategies can be applied to improve scalability, including decomposition of the verification per namespace or administrative domain and incremental solving to reuse solver state of the cloud configuration across multiple queries (e.g., reusing solver state across queries via the push-pop mechanisms offered by Z3).

V. Conclusion and Future Work

This paper presented a formal approach for the verification of cloud security configuration. Combining a high-level query language with an SMT-based formalization, the approach allows administrators to express and automatically verify a broad range of security properties. The validation confirmed the expressiveness of the proposed query language, the feasibility of the methodology, and its scalability in solving problems of large size and high complexity.

Future work aims to extend the scope of the approach in multiple directions. First, it will be extended to support more complex cloud deployments, such as multi-cluster and federated environments. Second, it will be enriched with additional configuration aspects, such as storage volumes and claims, which would allow a more comprehensive coverage of security-related properties.

REFERENCES

[1] A. Continella, M. Polino, M. Pogliani, and S. Zanero, "There's a hole in that bucket!: A large-scale analysis of misconfigured S3 buckets," in *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. ACM, 2018, pp. 702–711.

[2] S. Mehra, S. Sinha, and V. Chaudhary, "Data breach in cloud computing due to misconfigurations," in *Integration of Cloud Computing with Emerging Technologies*. CRC Press, 2023, pp. 55–64.

[3] A. Souri, N. J. Navimipour, and A. M. Rahmani, "Formal verification approaches and standards in the cloud computing: A comprehensive and systematic review," *Comput. Stand. Interfaces*, vol. 58, pp. 1–22, 2018.

[4] S. Khan, I. Kabanov, Y. Hua, and S. E. Madnick, "A systematic analysis of the capital one data breach: Critical lessons learned," *ACM Trans. Priv. Secur.*, vol. 26, no. 1, pp. 3:1–3:29, 2023.

[5] E. Al-Shaer, H. H. Hamed, R. Boutaba, and M. Hasan, "Conflict classification and analysis of distributed firewall policies," *IEEE J. Sel. Areas Commun.*, vol. 23, no. 10, pp. 2069–2084, 2005.

[6] H. Hu, G. Ahn, and K. Kulkarni, "Detecting and resolving firewall policy anomalies," *IEEE Trans. Dependable Secur. Comput.*, vol. 9, no. 3, pp. 318–331, 2012.

[7] M. Cheminod, L. Durante, L. Seno, F. Valenza, and A. Valenzano, "A comprehensive approach to the automatic refinement and verification of access control policies," *Comput. Secur.*, vol. 80, pp. 186–199, 2019.

[8] D. Bringhenti, S. Bussa, R. Sisto, and F. Valenza, "Atomizing firewall policies for anomaly analysis and resolution," *IEEE Trans. Dependable Secur. Comput.*, vol. 22, no. 3, pp. 2308–2325, 2025.

[9] T. van Ede, N. Khasuntsev, B. Steen, and A. Continella, "Detecting anomalous misconfigurations in AWS identity and access management policies," in *Proceedings of the 2022 on Cloud Computing Security Workshop, CCSW 2022, Los Angeles, CA, USA, 7 November 2022*. ACM, 2022, pp. 63–74.

[10] E. Zahoor, A. Ikram, S. Akhtar, and O. Perrin, "A formal approach for the identification of authorization policy conflicts within multi-cloud environments," *J. Grid Comput.*, vol. 20, no. 2, p. 18, 2022.

[11] 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, p. 103473, 2023.

[12] A. Sissodiya, E. Chiquito, U. Bodin, and J. Kristiansson, "Formal verification for preventing misconfigured access policies in kubernetes clusters," *IEEE Access*, vol. 13, pp. 141 798 – 141 813, 2025.

[13] H. Kang and S. Shin, "*Verikube*: Automatic and efficient verification for container network policies," *IEICE Trans. Inf. Syst.*, vol. 105-D, no. 12, pp. 2131–2134, 2022.

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

[15] S. Dong, Y. Xie, J. Zhao, and K. Qiu, "NPV: fast network policy verification for cloud-native networking," in *44th IEEE International Conference on Distributed Computing Systems, ICDCS 2024, Jersey City, NJ, USA, July 23-26, 2024*. IEEE, 2024, pp. 461–472.

[16] H. Yang and S. S. Lam, "Real-time verification of network properties using atomic predicates," *IEEE/ACM Trans. Netw.*, vol. 24, no. 2, pp. 887–900, 2016.

[17] A. Panda, O. Lahav, K. J. Argyraki, M. Sagiv, and S. Shenker, "Verifying reachability in networks with mutable datapaths," in *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*. USENIX Association, 2017, pp. 699–718.

[18] P. Zhang, X. Liu, H. Yang, N. Kang, Z. Gu, and H. Li, "Apkeep: Realtime verification for real networks," in *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*. USENIX Association, 2020, pp. 241–255.

[19] D. Bringhenti, S. Bussa, R. Sisto, and F. Valenza, "A two-fold traffic flow model for network security management," *IEEE Trans. Netw. Serv. Manag.*, vol. 21, no. 4, pp. 3740–3758, 2024.

[20] C. Banse, I. Kunz, A. Schneider, and K. Weiss, "Cloud property graph: Connecting cloud security assessments with static code analysis," in *14th IEEE International Conference on Cloud Computing, CLOUD 2021, Chicago, IL, USA, September 5-10, 2021*. IEEE, 2021, pp. 13–19.

[21] F. Minna, F. Massacci, and K. Tuma, "Towards a security stress-test for cloud configurations," in *IEEE 15th Int. Conf. on Cloud Computing, Barcelona, Spain, July 10-16, 2022*, 2022, pp. 191–196.

[22] J. Backes, P. Bolignano, B. Cook, C. Dodge, A. Gacek, K. S. Luckow, N. Rungta, O. Tkachuk, and C. Varming, "Semantic-based automated reasoning for AWS access policies using SMT," in *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, 2018, pp. 1–9.

[23] J. Backes, S. Bayless, B. Cook, C. Dodge, A. Gacek, A. J. Hu, T. Kahsai, B. Kocik, E. Kotelnikov, J. Kukovec, S. McLaughlin, J. Reed, N. Rungta, J. Sizemore, M. A. Stalzer, P. Srinivasan, P. Subotic, C. Varming, and B. Whaley, "Reachability analysis for aws-based networks," in *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II*, ser. Lecture Notes in Computer Science, vol. 11562, 2019, pp. 231–241.

[24] L. M. de Moura and N. S. Bjørner, "Z3: an efficient SMT solver," in *Proc. of the Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008*, ser. Lecture Notes in Computer Science, vol. 4963, 2008, pp. 337–340.

[25] Dynatrace, "Kubernetes in the wild - 2025 report," Available: https://www.dynatrace.com/resources/ebooks/kubernetes-in-the-wild/, (Visited: 2026-01-29).