CloudSec: An Extensible Automated Reasoning Framework for Cloud Security Policies

Joe Stubbs(✉), Smruti Padhy, Richard Cardone, and Steve Black

Texas Advanced Computing Center, University of Texas at Austin, Texas, USA
{jstubbs,spadhy,rcardone,scblack}@tacc.utexas.edu

Abstract. Users increasingly create, manage and share digital resources, including sensitive data, via cloud platforms and APIs. Platforms encode the rules governing access to these resources, referred to as *security policies*, using different systems and semantics. As the number of resources and rules grows, the challenge of reasoning about them collectively increases. Formal methods tools, such as Satisfiability Modulo Theories (SMT) libraries, can be used to automate the analysis of security policies, but several challenges, including the highly specialized, technical nature of the libraries as well as their variable performance, prevent their broad adoption in cloud systems. In this paper, we present CloudSec, an extensible framework for reasoning about cloud security policies using SMT. CloudSec provides a high-level API which can be used to encode different types of cloud security policies without knowledge of SMT. Further, it is trivial for applications written with CloudSec to utilize and switch between different SMT libraries. We use CloudSec to analyze security policies in Tapis, a cloud-based API for distributed computational research used by tens of thousands of researchers, and we present a performance case study of using CloudSec with Z3 and cvc5, two popular SMT solvers.

Keywords: Cloud Security Policy · SMT · Extensible Framework · Tapis · Role Based Access Control

1 Introduction

Through the use of cloud-based applications and services, users create valuable digital assets that must be secured. Each cloud platform makes use of a system for managing and enforcing the rules regarding which users have access to which digital assets, and there is little standardization across the vast number of such systems. For example, each of the major cloud computing providers have their own, independent systems for access management: Amazon Web Services makes use of AWS IAM [4], Google Cloud Platform uses Google IAM [14], and Microsoft Azure provides Azure Role Based Access Control (RBAC) [5]. Kubernetes, the popular container orchestration system, has its own RBAC policy system [19].

There are also many popular open source projects providing these capabilities, including Casbin [10], KeyCloak [17], Open Policy Agent [22], etc.

The Tapis [24] project represents another such platform. Tapis is an NSF-funded platform providing APIs that enable automated, secure, collaborative research computing to thousands of academic researchers. Using Tapis, researchers store, manage, and share datasets, executable code, execution outputs, project metadata and other resources with individual colleague as well as entire communities of researchers. Tapis stores authorization data in the form of policies describing which users have access to which resources. Tapis policies conform to certain rules governing their format; Table 1 shows an example Tapis policy granting the `jdoe` user `read`, `execute` and `modify` access to all files in her home directory on the Frontera HPC cluster, a world-class supercomputer hosted at the Texas Advanced Computing Center.

Table 1. An Example Tapis Policy

```
username=jdoe,
system=frontera.tacc.utexas.edu,
path=/home/jdoe/*,
action=[READ, EXECUTE, MODIFY],
decision=allow
```

Projects manage thousands of such Tapis policies on behalf of their users, and as these systems evolve, the policies must be updated accordingly. Over time, ensuring that policies are correctly written and match what is being enforced becomes increasingly difficult.

Software based on formal methods such as Satisfiability Modulo Theories (SMT) [9,18] provide techniques for automatically reasoning about entire collections of policies, and to prove or find counter-examples to mathematically precise statements regarding sets of policies. As an example, in the case of Tapis, a project may want to know that all users have modify access to their home directory on all systems but only specific administrative users have access to the root directories. Such statements form the *policy specification* for an application, and the goal is to determine if the actual policies are no more permissive than the policy specification. In general, describing the policy specification is a difficult problem.

While SMT libraries can be used to prove or disprove such statements, the broad adoption of SMT in cloud systems faces the following challenges: 1) the use of these tools requires a sophisticated understanding of the underlying SMT; 2) a significant effort must be made to encode a particular security policy's semantics into an SMT library; and 3) performance on different security policy sets varies across different SMT libraries and even different versions of the same library.

To address these challenges, we built CloudSec [1], an extensible automated reasoning framework for cloud security policies that does not require any SMT knowledge to use or extend to new cloud policy systems. CloudSec provides a core set of data types for defining the semantics of a security policy language, and these data types are linked to SMT solvers through CloudSec's library of `backends`, which implement encodings of the data types as well as proof methods based on the functionality provided by the solver. The initial release of CloudSec includes support for the Z3 [3,21] and CVC5 [2,7] libraries as `backends`. Using CloudSec's building blocks, the policy types for a new system can be defined in just a few lines of Python code without any understanding of SMT. CloudSec's `connectors` abstraction allows policy types to be instantiated with real policies from source systems.

To establish the extensibility of CloudSec, we created basic implementations of policy types for multiple systems, including Tapis and a generic REST API service. For the Tapis platform, we also implement connectors, yielding a tool capable of retrieving and analyzing policies from Tapis's multi-tenant permissions system, which includes hierarchical roles and permission "types" with schemas for many individual services. We automatically establish or find counter-examples to policy requirements across entire sets of Tapis permissions. Finally, we study the scalability of our tool and establish that it performs well when analyzing Tapis policy sets consisting of thousands of policies.

In summary, the main contributions of this paper are:

1. Design and implementation of CloudSec, an extensible Python framework for leveraging SMT for security policy analysis with a user-friendly interface.
2. Description of CloudSec usage in Tapis, a real-world cloud platform used by thousands of researchers.
3. Performance study showing CloudSec scales to thousands of policies.

The rest of the paper is organized as follows: In Sect. 2, we provide background material on Tapis and SMT; in Sect. 3, we describe the CloudSec design, its use in Tapis, and give examples of policy types and encodings; in Sect. 4, we describe our performance study and Sect. 5 compares related work; we conclude in Sect. 6 and outline some areas for future work.

2 Background

In this section, we provide background information on topics used throughout the rest of this paper.

2.1 Tapis

Tapis [24] is a web-friendly, application programming interface (API) for research computing, allowing users to automate their interactions with advanced storage and computing resources in cloud and HPC datacenters. Primary Tapis features include a full-featured data management service, with synchronous endpoints for data ingest and retrieval as well as a reliable asynchronous data transfer facility; workload scheduling and code execution; a highly-scalable document store and metadata API; and support for streaming IoT/sensor data. Tapis supports reproducibility by recording a detailed data provenance and computation history of actions taken in the platform. Additionally, Tapis enables collaboration via a fine-grained permissions model, allowing data, metadata and computations to be kept private, shared with specific individuals or disseminated to entire research communities. Tapis has been used by thousands of researchers across projects funded by a number of government agencies, including CDC, DARPA, NASA, NIH, and NSF.

2.2 Tapis Security Policies

Tapis is organized as a set of 14 independent HTTP web services (sometimes called *microservices*) that coordinate together to accomplish larger tasks. The Tapis Security Kernel (SK) manages all authorization data – information specifying which users have access to which Tapis objects and at what access level. SK stores this authorization data as *permission* objects and provides HTTP endpoints for creating, retrieving, and modifying permissions.

2.3 SMT Solver

CloudSec encodes security access policies into logical formulas represented using Satisfiability Modulo Theories (SMT). Then the satisfiability problem is solved using an SMT solver. An SMT solver is a software that uses decision procedures to report whether a formula is satisfiable in some finite amount of computation and find some counter-example [9,18]. We use two efficient SMT solvers in our backends- Z3 [3] and CVC5 [7].

3 Related Work

There have been several works on cloud-access control policies analysis using SMT [6,11,20,23]. Our work is closely related to the approach used in Zelkova [6], [23]. Zelkova is an AWS policy analysis tool that verifies AWS policies by reasoning if a policy is less or equally permissive than the other. It encodes AWS policies to SMT formulas and uses SMT solvers, Z3, CVC, and Z3AUTOMATA, to verify and prove the properties. Zelkova is not open-source and cannot be extended to other policy languages. CloudSec's approach is similar to Zelkova and other prior works in defining a policy language and translating policies to SMT formulas for reasoning using SMT solvers. The main differences are that 1) CloudSec is an open-source, extensible automated reasoning framework where users or developers can define their policy types not restricted to cloud policies depending on their application, and 2) developers can plug different SMT solvers into CloudSec's extensible backend. Additionally, by defining a policy converter, CloudSec can be easily integrated into existing cloud-hosted services, as demonstrated in Sect. 4.5. In another related work [13], the authors built pySMT, a SMT solver agnostic open-source python library that allows defining, manipulating, and solving SMT formulae. However, to use the pySMT API, one still needs the knowledge of SMT. We envision to support pySMT as one of the backends to Cloudsec.

Eiers et al. [11] proposed a framework to quantify the permissiveness of access policies using model counting constraint solvers and relative permissiveness between the policies. They also built an open-source tool, QUACKY, that analyzes AWS and Azure policies. The CloudSec framework focuses on easy extensibility, defining different policy types, and making the reasoning tool more accessible to users. We envision that QUACKY could potentially be a component in the CloudSec framework to compute relative permissiveness.

Several works have studied the verification of network access control, connectivity, and configuration policies. Jayaraman et al. [15,16] proposed and built a tool called SECGURU that automatically validates network connectivity policies using SMT bit vector theory and the Z3 solver. SECGURU is a closed source tool used in Azure. Fogel et al. [12] proposed and developed an open-source tool, Batfish, that analyzes network configuration and detects errors. Campion [25] is another open-source tool for debugging router configuration and has been implemented as an extension to Batfish. It localizes crucial errors to relevant configuration lines. Beckett et al. [8] proposed a general approach for network configuration that translates both control and data plane behaviors into a logical formula and use SMT solver, Z3, to verify the properties. They have implemented this approach in the tool called Minesweeper. CloudSec could be extended to support network access policies such as Firewall and router policies.

4 Approach

As mentioned in the Introduction, CloudSec provides a toolkit in the form of a Python library for utilizing SMT solvers to analyze security policies in real-world systems such as Tapis. The primary goal of CloudSec is to reduce the expertise needed to apply SMT technology to the study of security policies (Fig. 1).

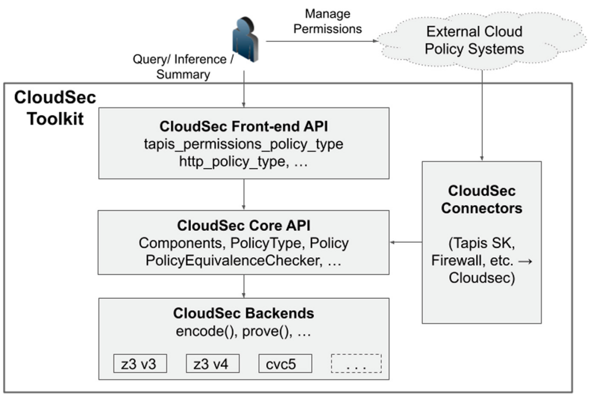

Manage Permissions
External Cloud Policy Systems
Query/ Inference / Summary
CloudSec Toolkit
CloudSec Front-end API
tapis_permissions_policy_type
http_policy_type, ...
CloudSec Connectors
(Tapis SK, Firewall, etc. → Cloudsec)
CloudSec Core API
Components, PolicyType, Policy
PolicyEquivalenceChecker, ...
CloudSec Backends
encode(), prove(), ...
z3 v3
z3 v4
cvc5
...

Fig. 1. CloudSec Overview and Usage Model

4.1 CloudSec - Extensible Framework Design

The primary components of the CloudSec framework include (i) the `core` module, with basic data types that can be used to build encodings of real-world security policy systems; (ii) the `backends` library, which provides implementations of encodings of the types provided in the `core` module as well as proof methods for analyzing sets of policies via various solvers, such as Z3 and cvc5; (iii) the `cloud` module, which utilizes the types provided in `core` to define ready-to-use policy types for real systems; and (iv) the `connectors` library, which provides functions for converting security rules in source systems to CloudSec policies.

Central to the design of CloudSec are the concepts of `Component`, `PolicyType`, and `Policy`, provided by the `core` module. Each `Component` type contains a data type, such as a `String`, `Enumeration`, `IP Address` or `Tuple`, the set of allowable values for the data type, and a matching strategy, which defines a "match" relation on two values from the set of allowable values for the data type. Examples of currently supported matching strategies include exact matching and wildcard matching. `PolicyTypes` build on the notion of `Components`, with each `PolicyType` defining a list of `Components` that it is comprised of. Finally, a `Policy` object represents a specific value for a given `PolicyType`.

Using these primary notions, one can create `PolicyTypes` comprised of `Components` for real-world systems in just a few lines of Python without requiring any knowledge of SMT. The `cloud` module provides examples of `PolicyType` objects, including a `tapis_files_policy_type`, used to represent policies related to file objects in Tapis, and an `http_api_policy_type`, which can be used to model policies in an arbitrary HTTP API. Moreover, by decoupling the definitions of `Components` and `PolicyTypes` from their implementations in different backend solvers, CloudSec provides a highly-extensible system in which support for additional solvers can be added independently of defining policy types for new systems.

4.2 An Example Policy Type and Policy Definition

As a first example, we describe the `http_api_policy_type`, available in the CloudSec toolkit. We created this policy type to illustrate what can be achieved with CloudSec. The policy type represents policies governing access to resources defined in a multi-tenant, microservice API platform. The policy type is comprised of three components: *principal*, *resource*, and *action*. A principal is a `Tuple` component type with two fields, `tenant` and `username`, representing a unique user identity in the platform to whom access to some resource is being allowed or denied by the policy. A tenant is a `StringEnum` component with enumerated values representing all possible tenants. We chose a `StringEnum` to model the tenant under the assumption that the number of tenants would be relatively small. On the other hand, the username field is modeled with a `String` component type defined over an alphanumeric set and maximum length.

Similarly, a resource is modeled using a `Tuple` component type with three fields, *tenant, service*, and *path*, that collectively distinguish a specific HTTP endpoint (i.e., HTTP resource) within a service to which access is being allowed or denied in some tenant. A service is a `StringEnum` component with enumerated values representing the names of the APIs in the platform. The path is a String component defined over a path character set with a maximum length. The path represents the URL path corresponding to the resource. Finally, the action is a `StringEnum` that denotes the HTTP method being authorized or not authorized, such as {"GET","POST","PUT","DELETE"}. An example policy definition of `http_api_policy_type` is shown below (Table 2):

Table 2. An Example HTTP Policy

```
p = Policy(policy_type=http_api_policy_type,
    principal=("a2cps","jdoe"),
    resource=("a2cps","files","/ls6/home/jdoe"),
    action="GET",
    decision="allow")
```

Note that all policy types have a special `decision` field which is a `StringEnum` with values "allow" and "deny".

4.3 Tapis Policy Types

The CloudSec toolkit includes policy types representing permissions objects in the Tapis API platform. For example, we provide the `tapis_files _policy_type` for dealing with Tapis permissions related to file objects. The `tapis_file_perm` component is a `Tuple` type representing the Tapis files "perm spec" (see [26]) and includes fields for the tenant, system id, permission level, and file path. Because of the simple and flexible CloudSec core API, the entirety of the Tapis policy type implementations constitutes less than 20 lines of Python code.

4.4 Translating a Policy Set to SMT Formula

A policy set is defined as a set of policies, i.e., $PS = \{P_1, P_2, \cdots, P_i, \cdots, P_{n-1}, P_n\}$ where $1 \leq i \leq n$. Each policy, $P_i = (c_1, c_2, \cdots, c_j, \cdots, c_{m-1}, c_m, Decision)$ where $1 \leq j \leq m$. c_j denotes the value for `Component` C_j of Policy P_i. Note that a policy type determines the number and type of `Components` in a policy. *Decision* is a value from the set {"*allow*", "*deny*"} to denote if a policy allows or denies access. A policy is translated to an SMT formula as:

$$\mathcal{P} = \bigwedge_{j=1}^{m} \left(\bigvee_{c \in C_j(P)} C_j = c \right)$$

$C_j(P)$ denotes set of values defined for `Component` C_j in a policy P. If $C_j(P)$ is a `Tuple` component type with k components , let say, $(t_1, t_2, .., t_k)$, then it is further encoded as $\bigwedge_{l=1}^{k} (t_k = v)$ where component t_k takes one of the component allowed values, v.

A policy set can be SMT encoded as:

$$\mathcal{PS} = \left(\bigvee_{AllowSet} \mathcal{P} \right) \bigwedge \neg \left(\bigvee_{DenySet} \mathcal{P} \right)$$

where $AllowSet = \{\forall P \in PS : P.Decision = \text{``}allow\text{''}\}$ and $DenySet = \{\forall P \in PS : P.Decision = \text{``}deny\text{''}\}$.

4.5 Connectors and Converting SK Policies to Cloudsec

Using CloudSec to analyze security rules from a real-world system requires one to generate CloudSec policy objects (of the appropriate type) from authorization data residing in the external system. A CloudSec `connector` can be written for the external system to simplify this effort. The CloudSec toolkit currently includes a connector for Tapis files permissions, allowing CloudSec `tapis_files_policy_type` policies to be generated from Tapis permissions data with a single function call.

Using CloudSec, we developed a program that generates Tapis files policy objects from all Tapis files permissions in the SK for a configurable set of users. The program then uses CloudSec solver backends to prove that the source policies conform to certain rules or find counter examples. For instance, using our program, we analyzed policies for users within a certain project built with Tapis, generating over 3,000 CloudSec policies. We then were able to prove that for this project, no users except for the users in the admin-scientist role had read/write access to files in a protected `data` directory on a specific Tapis system. Similarly, we prove that public access to generated result files is restricted to files with a `.png` extension.

Establishing these kinds of results harnesses the full power of CloudSec – while the Tapis SK is highly efficient at answering the question, "does a given user have access to a specific resource?", it cannot reason about an entire set of permissions at once. Furthermore, our program must use both Z3 and cvc5 to find proofs like the ones above, as some proofs could not be found by one solver or the other.

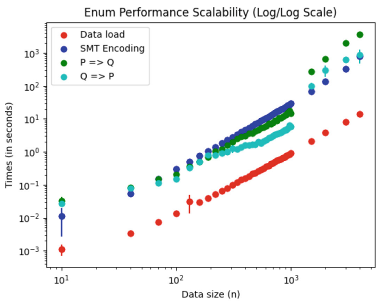
5 Performance

We did a performance evaluation of the CloudSec toolkit 1) to establish that the CloudSec software was a viable approach (i.e., can be done on commodity hardware in a reasonable amount of time) to analyze policies of the sizes that show up in real systems such as Tapis (i.e., on the order of a few thousand policies) (Sect. 4.5); and 2) to show that CloudSec could be used as a framework for comparing different SMT backends for specific analyses [1]. We also observed that there are scenarios for which either Z3 or CVC5 or both, along with their different versions exhibit performance cliffs. We provide the details of the performance evaluation in subsequent subsections.

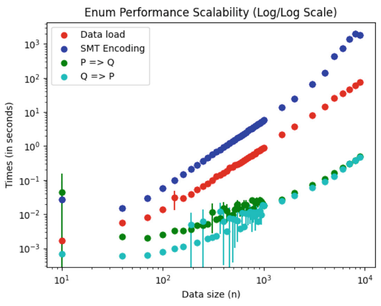
We measured the performance to check trivial implications for two policy sets as the number of policies grows. We defined different components (`String` and `StringEnum`) and policy types using those components to create policy sets. Each policy set consists of policies of the same policy type. We repeated the test for both Z3 and cvc5 backends. We ran the tests on a machine with the following configuration: 32 CPUs, Intel(R) Xeon(R) CPU E5-2660 0 @ 2.20 GHz, 128 GB RAM.

5.1 StringEnum Scalability

We defined a policy type containing a `StringEnum` with a variable number, N, allowable values and with wildcard matching. For example, with $N = 5$, the policy type's `StringEnum` will have allowable values { '0', '1','2','3','4'}. We created two policy sets, P and Q. P contained a policy P_i for every allowable value, i.e., $P = \cup_{i=0}^{N-1}\{(P_i, \text{“}allow\text{”})\}$ where $P_i = i$. Q contained a single wildcard policy, $Q = \{(*, \text{“}allow\text{”})\}$. We varied the number of unique elements, N, in the range $10 \leq N \leq 4000$. We measured the time to perform: 1) data load, 2) SMT encoding, 3) $P \implies Q$ and 4) $Q \implies P$ for Z3 as well as for cvc5. In Fig. 2 a, we observed that when using the Z3 backend, the SMT encoding, $P \implies Q$ and $Q \implies P$ took similar amounts of time, while the data load took significantly less time. When we used cvc5, SMT encoding took the most of the computation time while the data load time was more than the implication prove time. For this performance test, cvc5 was roughly 90% faster than Z3 in total time.



(a) Using Z3 backend



(b) Using cvc5 backend

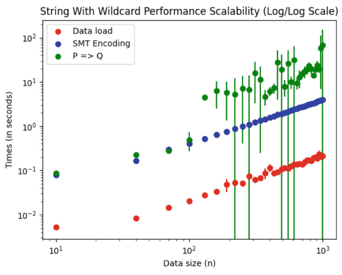
Fig. 2. Dynamic StringEnum Policy type Scalability

5.2 String with Wildcard Scalability

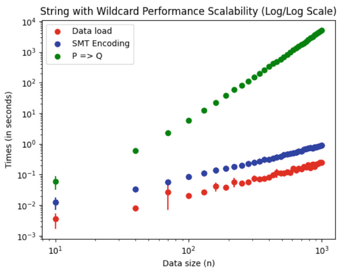
We defined a policy type with a single `String` component with wildcard matching. The string component was defined over a character set that included alphanumeric characters, '/', and the wildcard character '*'. The maximum length of the string was 100. We created two policy sets, P and Q, for each $10 \leq$

$N \leq 1000$. The policy set $P = \cup_{i=0}^{N-1}\{(P_i, \text{“}allow\text{”})\}$ where each P_i was defined to be the static string “$a1b2c3d4e5/i$”. The policy set $Q = \cup_{i=0}^{N-1}\{(Q_i, \text{“}allow\text{”})\}$ where each Q_i is defined as the string “$a1b2c3d4e5/i*$” ending in the wildcard character. For example, if $N = 2$, $P = \{$“$a1b2c3d4e5/1$”, “$a1b2c3d4e5/2$”$\}$ and $Q = \{$“$a1b2c3d4e5/1*$”, “$a1b2c3d4e5/2*$”$\}$. In this case, there are 2 policies in P and two policies in Q.

We varied N and measured the data load, SMT encoding, and $P \implies Q$ times for Z3 as well as for cvc5. Note $Q \implies P$ is not valid. In Fig. 3, we observe that Z3 was faster than cvc5 in total time and, in particular, in proving $P \implies Q$, Z3 was often an order of magnitude or more faster. Both backends follow the same pattern where data load and SMT encoding times are less than implication prove time. Z3 shows good performance even for $N = 1000$, which is 1000 policies in P and in Q.



(a) Using Z3 backend



(b) Using cvc5 backend

Fig. 3. String with Wildcard Policy type Scalability

5.3 Performance Cliffs for SMT Solvers

We observed some performance cliffs while using Z3 and cvc5. First, consider two simple policy sets in which the policy type has just one component, *path*, of string type. Let $P = (\text{“}/sys1*\text{”}, \text{“}allow\text{”})$ and $Q = (\text{“}/*\text{”}, \text{“}allow\text{”})$. While trying to prove P is less permissive than Q, the Z3 backend hangs but cvc5 is able to prove it.

Second, consider the policy sets:

$$\begin{aligned} P = \{(&\text{“}jstubbs\text{”}, \text{“}s2/home/jstubbs/*\text{”}, \text{“}*\text{”}, \text{“}allow\text{”}),\\ &(\text{“}jstubbs\text{”}, \text{“}s2/*\text{”}, \text{“}PUT\text{”}, \text{“}deny\text{”}),\\ &(\text{“}jstubbs\text{”}, \text{“}s2/*\text{”}, \text{“}POST\text{”}, \text{“}deny\text{”})\} \end{aligned}$$

and

$$\begin{aligned} Q = \{(&\text{“}jstubbs\text{”}, \text{“}s2/home/jstubbs/a.out\text{”}, \text{“}GET\text{”},\\ &\text{“}allow\text{”}),\\ (&\text{“}jstubbs\text{”}, \text{“}s2/home/jstubbs/b.out\text{”}, \text{“}GET\text{”}, \text{“}allow\text{”})\} \end{aligned}$$

Z3 was able to to prove $Q \implies P$ but cvc5 hangs.

6 Conclusion and Future Work

In this paper, we presented a description of the CloudSec library, which simplifies the use of SMT in analyzing security policies in real-world systems. We applied CloudSec to the Tapis API platform to analyze thousands of permissions records at once. In the future, we plan to incorporate CloudSec into a public API within Tapis, allowing any user to easily submit security analysis jobs using HTTP requests. Further, we will explore adding support for additional backends to the project and applying CloudSec to additional real-world systems, such as AWS, and Kubernetes.

Acknowledgement. This material is based upon work supported by the NSF OAC Award #1931439.

References

1. CloudSec (December 2023). https://github.com/applyfmsec/cloudsec
2. CVC5: An efficient open-source automatic theorem prover for satisfiability modulo theories (SMT) problems (December 2023). https://cvc5.github.io/
3. Z3 (December 2023). https://github.com/Z3Prover/z3
4. AWS IAM (December 2023). https://docs.aws.amazon.com/IAM/latest/UserGuide/access.html
5. AzureRBAC (December 2023). https://learn.microsoft.com/en-us/azure/role-based-access-control/overview
6. Backes, J., et al.: Semantic-based automated reasoning for AWS access policies using SMT. In: 2018 Formal Methods in Computer Aided Design (FMCAD), pp. 1–9 (2018). https://doi.org/10.23919/FMCAD.2018.8602994
7. Barbosa, H., et al.: cvc5: a versatile and industrial-strength SMT solver. In: TACAS 2022. LNCS, vol. 13243, pp. 415–442. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_24
8. Beckett, R., Gupta, A., Mahajan, R., Walker, D.: A general approach to network configuration verification. In: Proceedings of the Conference of the ACM Special Interest Group on Data Communication, pp. 155–168. SIGCOMM 2017 (2017)
9. Bradley, A.R., Manna, Z.: The Calculus of Computation: Decision Procedures with Applications to Verification. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74113-8
10. Casbin (Dec 2022). https://casbin.org/
11. Eiers, W., Sankaran, G., Li, A., O'Mahony, E., Prince, B., Bultan, T.: Quantifying permissiveness of access control policies. In: Proceedings of the 44th International Conference on Software Engineering, pp. 1805–1817. ICSE 2022 (2022)
12. Fogel, A., et al.: A general approach to network configuration analysis. In: Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation, p. 469–483. NSDI 2015, USENIX Association, USA (2015)
13. Gario, M., Micheli, A.: PySMT: a solver-agnostic library for fast prototyping of SMT-based algorithms. In: SMT Workshop 2015 (2015)

14. Google IAM (December 2023). https://cloud.google.com/iam/
15. Jayaraman, K., et al.: Validating datacenters at scale. In: Proceedings of the ACM Special Interest Group on Data Communication, pp. 200–213. SIGCOMM 2019 (2019)
16. Jayaraman, K., Bjørner, N., Outhred, G., Kaufman, C.: Automated analysis and debugging of network connectivity policies. Tech. Rep. MSR-TR-2014-102, Microsoft (2014)
17. KeyCloak (December 2023). https://www.keycloak.org/
18. Kroening, D., Strichman, O.: Decision Procedures: An Algorithmic Point of View. In: Juraj, H., Mogens, N. (eds.) Texts in Theoretical Computer Science. An EATCS Series, Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-50497-0
19. Kubernetes (December 2023). https://kubernetes.io/docs/reference/access-authn-authz/rbac/
20. Liu, A., Du, X., Wang, N., Wang, X., Wu, X., Zhou, J.: Implement security analysis of access control policy based on constraint by SMT. In: 2022 IEEE 5th International Conference on Electronics Technology (ICET), pp. 1043–1049 (2022). https://doi.org/10.1109/ICET55676.2022.9824517
21. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
22. Open Policy Agent (December 2022). https://www.openpolicyagent.org/
23. Rungta, N.: A billion SMT queries aăday (invited paper). In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification, pp. 3–18. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-031-13185-1_1
24. Stubbs, J., et al.: Tapis: an API platform for reproducible, distributed computational research. In: Arai, K. (ed.) FICC 2021. AISC, vol. 1363, pp. 878–900. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-73100-7_61
25. Tang, A., et al.: Campion: debugging router configuration differences. In: Proceedings of the 2021 ACM SIGCOMM 2021 Conference, pp. 748-761. SIGCOMM 2021 (2021)
26. Tapis: Tapis permissions (2022) (December 2023). https://tapis.readthedocs.io/en/latest/technical/security.html#id3