Pre-Deployment Security Assessment for Cloud Services through Semantic Reasoning

Claudia Cauli[1], Meng Li[2], Nir Piterman[1], and Oksana Tkachuk[2]

[1] University of Gothenburg
[2] Amazon Web Services

Abstract. Over the past ten years, the adoption of cloud services has grown rapidly, leading to the introduction of automated deployment tools to address the scale and complexity of the infrastructure companies and users deploy. Without the aid of automation, ensuring the security of an ever-increasing number of deployments becomes more and more challenging. To the best of our knowledge, no formal automated technique currently exists to verify cloud deployments during the design phase. In this case study, we show that Description Logic modeling and inference capabilities can be used to improve the safety of cloud configurations. We focus on the Amazon Web Services proprietary declarative language, CloudFormation, and develop a tool to encode template files into logic. We query the resulting models with properties related to security posture and report on our findings. By extending the models with dataflow-specific knowledge we use more comprehensive semantic reasoning to further support security reviews. When applying the developed toolchain to publicly available deployment files we find numerous violations of widely-recognized security best practices, which suggests that streamlining the methodologies developed for this case study would be beneficial.

1 Introduction

The term *Infrastructure as Code* (*IaC*) refers to the practice of configuring, provisioning, and updating systems resources from source code files, which are compiled into atomic instructions and then executed to deploy the desired architecture [29]. The advantage of handling code, instead of manually provisioning resources, lies in the capability to use version control systems, orchestration frameworks, and automated testing tools as part of the deployment process. In addition to instructions relevant for resource creation, dependencies, and updates, IaC configuration files contain information about settings, dataflow, and access control. In a time when cloud companies provide customers with simple-to-launch, albeit extremely powerful infrastructure, it is crucial to automatically and provably verify the security of such systems.

In this study, we investigate IaC deployments and how these are formally modeled and reasoned upon. We explore the usage of description logics (DLs) as a conceptual-modeling formalism that is expressive, decidable, and equipped with mature tooling. We argue that formal reasoning techniques applied to deployment templates are an immensely valuable tool for developers and security

engineers; substantially aiding the automation of time-consuming security reviews; helping them to detect complex logical errors at earlier stages, and containing the costs that finding, and fixing, security issues at later stages would cause. As the prevalence of usage of cloud infrastructure increases, besides experts, automated reasoning tools could benefit inexperienced users as well.

System Studied. We focus on the Amazon Web Services proprietary IaC tool, CloudFormation, the first to be introduced at a large scale, over ten years ago. AWS, cloud provider within Amazon, serves millions of customers worldwide. These include private businesses as well as government, education, nonprofit, and healthcare organizations. While the cloud provider is responsible for the faithful deployment of the customers' desired configurations, it is the customer's duty to make sure that these comply with the security requirements of their business context. Few management tools of this scale exist. Notable mentions are Terraform [36], Microsoft Azure's *Resource Manager* [28], Google Cloud's *Deployment Manager* [19], and the recently introduced OASIS standard TOSCA [6].

Goal of Study. Our goal is to improve the quality of the security analyses that are performed over IaC configurations pre-deployment; and doing so, their overall security. With this study, we investigate the application of description logics to the formalization and reasoning over IaC deployments. In particular, we are interested in three aspects: *i*) whether proposed cloud configurations comply with security best practices, *ii*) how to aid customers in building more secure infrastructure *before deploying it*, and *iii*) to what extent formal automated techniques can support manual pre-deployment security reviews.

Challenges. Little research has been done so far on the possibility to formalize IaC languages, and no research has been done to devise a logic that is well-suited to reason about cloud infrastructure. By nature, cloud infrastructure interacts with an open environment that is, at best, only partially known. In particular, external-facing APIs and users participate in these interactions. By design, cloud services allow for the composition of smaller components into large infrastructure, the complexity of which creates a challenge with respect to security. Our models should capture the connectivity of resources, the flow of information that spans across multiple paths, and the rich security-related data available in IaC configuration files. This is further complicated by the need for a query language for verification and falsification, able to express that mitigations must be present (vs. may be absent), and security issues must be absent (vs. may be present). Importantly, we need practical tools that support the implementation of all these parts and that can scale to real-world IaC configurations.

Our Contribution. We provide a framework to encode IaC into description logic, and investigate its effectiveness in answering configuration queries and reasoning about dataflow, trust boundaries, and potential issues within the system. Specifically, we test DLs reasoning capabilities to infer new facts about underspecified resources (such as those not included in a given deployment but used by it) and leverage DLs *open-world assumption* to perform verification and refutation, depending on the property being checked. We formalize additional security knowledge that allows for checking system-level semantic properties; i.e.,

properties that consider the nature of the cloud environment and more complex reachability over an *inferred* graph representation of the infrastructure.

Throughout the study, we make four novel contributions: (*i*) the formalization and logical encoding of AWS CloudFormation (Section 3); (*ii*) a technique to express security properties (Section 4); (*iii*) an experimental evaluation of encoding and query times, accounting for the most common security issues that we found over publicly available IaC templates (Section 5); and (*iv*) an extension that enables semantic dataflow reasoning (Section 6). Our tool is implemented in Scala and available online [14]. We include preliminaries in Section 2; discuss related work in Section 7; and conclude in Section 8.

2 Preliminaries

Description Logics DLs are a family of logics well suited to model *relationships between entities*. They provide the logical foundation of the well-known *Web Ontology Language* [23,20,31], for which extensive tool support exists (e.g., the Protégé editor and off-the-shelf reasoners such as FaCT, HermiT, and Pellet [30,37,18,35]). We introduce the description logic $\mathcal{ALC}$ [33,1,24], *Attributive Logic with Complement*, and two additional features that are relevant for our study. $\mathcal{ALC}$ formulae are built from symbols from the alphabets N_C, of atomic concept names; N_R, of role names; and N_I, of individual names. These are the DL equivalents of FOL unary predicates, binary predicates, and constants, respectively. $\mathcal{ALC}$ concept expressions are built according to the grammar:

$$C, D ::= \bot \mid \top \mid \mathsf{A} \mid \neg C \mid C \sqcup D \mid C \sqcap D \mid \exists \mathsf{r}.C \mid \forall \mathsf{r}.C$$

where A is an atomic concept from the set N_C; C, D are possibly complex concepts; and r is a role from the alphabet N_R. Terminological knowledge is represented via general concept inclusion axioms $C \sqsubseteq D$. As an example, in the remainder of this paper we will refer to two standard axioms that enforce the domain and range of binary relations: $\mathsf{dom}(r, C) \equiv \exists r.\top \sqsubseteq C$ and $\mathsf{ran}(r, C) \equiv \exists r^{-}.\top \sqsubseteq C$. Assertional knowledge is represented via concept assertions $\mathsf{C}(a)$ and role assertions $\mathsf{r}(a, b)$. In this paper, we will use two additional operators: *functionality constraints* and *complex role inclusions*. The former enforces binary relationships to be functional. The latter expression, written $r \circ s \sqsubseteq t$, establishes that the chaining of the two relationships r and s implies the relationship t, and can be used to implement transitivity (when $r = s = t$). A model of a DL knowledge base is an interpretation $\mathcal{I}$, over a domain Δ, that satisfies all the axioms and assertions contained and implied by the knowledge base. For the purpose of our application, we leverage two classical inference problems: *satisfiability* and *instance retrieval*, whose full definitions are found in standard textbooks [2,3].

AWS CloudFormation AWS CloudFormation, cfn, provides users with a declarative programming language and a framework to provision and manage over 500 *resources* spread across 70 *services* [15].[3] Services are products such as

[3] As of August 2020, exact number is region-dependent.

storage, databases, and processors, and their interface is implemented through resources, which are the actual modules that users declare and deploy. Their declaration is done by writing one or more so-called *CloudFormation Templates* (JSON-formatted configuration files). Within a template, users configure settings and communication of the desired resource instances. As an example, let us consider one of the most widely known storage products within AWS: the Simple Storage Service S3 (also illustrated in Listings 1.1 and 1.2). The CloudFormation interface for S3 consists of two resources: S3::Bucket and S3::BucketPolicy. A Bucket is a single unit of storage whose properties include encryption, replication, and logging settings, which can be viewed as the bucket's own configuration parameters. They could also be *references* to other resources that are connected to the current one, e.g., the unique ID of another bucket where logs are stored. A BucketPolicy is a resource that links an access control policy to a bucket. All the properties that can be instantiated and the structure of resource-types such as S3::Bucket and S3::BucketPolicy are given in the *CloudFormation Resource Specification*. The *resource specification* is a collection of files that prescribe resource properties and their allowed values. Provided that a *configuration file* is valid w.r.t. the specifications, an IaC deployment environment compiles it into instructions that are then executed to provision the requested resources in the correct dependency order and with the desired settings.

3 Formalization and Encoding of IaC Deployments

While setting up this case study, we found it convenient to come up with a formalization, of both IaC resource specifications and IaC configuration files, to use as an intermediate representation during the encoding process.[4] This was also needed since we could not find suitable research in the area (although some preliminary research on IaC formalization does exist: e.g. the PhD thesis in [12]). As mentioned in Section 2, users consult the *resource specifications* to find out what fields and values are allowed when declaring a resource. Intuitively, these provide a sort of type-system, or JSON schema, against which *configuration files* must validate. Configuration files contain the resource declarations of the instances that the user wishes to deploy. Let us illustrate this with some examples. Listing 1.1 shows a snippet of the S3::Bucket resource-type specification. In addition to the main resource type, the specification includes definitions for its subproperties, their types, and whether these are required. Although the example only shows string properties, in general, allowed properties values range over objects, arrays, and primitive types

```
"ResourceType":
"S3::Bucket": {
  "Properties":{
     "BucketName" : "String",
     "LoggingConfiguration":  {
       "Type": "LoggingConfiguration",
       "Required": false } ... }},

"PropertyTypes": ...,
"S3::Bucket.LoggingConfiguration":{
  "Properties": {
     "DestinationBucketName":{
       "Type": "String",
       "Required": false },
     "LogFilePrefix":{
       "Type": "String",
       "Required": false }}}
```

Listing 1.1. S3::Bucket specification

[4] The full formalization and encoding can be found in Appendices A and B

such as integers, doubles, longs, strings, and booleans. Listing 1.2, on the other hand, shows a common usage scenario of the S3 storage service, where a bucket with basic configuration is used to store the desired data. The instance has logical ID `ConfigS3Bucket`, is of type S3::Bucket, and specifies two top-level properties, `BucketName` and `LoggingConfiguration`. It is easy to see that this instance declaration validates against the resource specification of Listing 1.1. This snippet is taken from one of the benchmark deployments evaluated in Section 5 (StackSet 15) and, incidentally, it violates a security best practice: *"No bucket should store its own logs"* (See also Appendix F). Such formalization has been instrumental to capture infrastructure configurations, resources settings and interconnections, and to precisely and automatically encode it into DL.

```
"ConfigS3Bucket": {
  "Type": "AWS::S3::Bucket",
  "Properties":
    "BucketName": "ConfigStore",
    "LoggingConfiguration": {
      "DestinationBucketName": "ConfigStore",
      "LogFilePrefix": "config-bucket-logs/"}}
```

Listing 1.2. S3::Bucket instance declaration

Encoding We translate IaC specifications into DL terminological knowledge, and IaC configurations into assertional knowledge. The conceptual modeling features needed to model the former include axioms to define *domain* and *range* of properties, *requiredness*, and *functionality*. These give us enough expressivity to infer qualities of nodes that are underspecified, such as those that are referenced by a template but *not declared in it* (e.g., already deployed and running elsewhere), whose configuration is unknown. To give readers an intuition of the encoding procedure, let us look at the equation below, which contains some of the axioms and assertions generated by the translation of the code in Listings 1.1 and 1.2 (the full encoding can be found in Appendix C).

$$\begin{aligned} \mathit{Spec}_{\mathsf{S3::Bucket}} = \{\ & \mathsf{dom}(\mathsf{bucketName}, \mathsf{BUCKET}),\ \mathsf{ran}(\mathsf{bucketName}, \mathit{String}), \\ & (\mathsf{Funct\ bucketName}),\ \ldots,\ \mathsf{dom}(\mathsf{destinationBucket}, \mathsf{LOGCONFIG}), \\ & \mathsf{ran}(\mathsf{destinationBucket}, \mathsf{BUCKET}),\ \ldots\ \} \end{aligned}$$

$$\begin{aligned} \mathit{Config} = \{\ & \mathsf{BUCKET}(\mathit{ConfigS3Bucket}), \mathsf{bucketName}(\mathit{ConfigS3Bucket}, \text{“}\mathit{ConfigStore}\text{”}), \\ & \mathsf{loggingConfig}(\mathit{ConfigS3Bucket}, x),\ \mathsf{destinationBucket}(x, \mathit{ConfigS3Bucket}), \\ & \mathsf{logFilePrefix}(x, \text{“}\mathit{config\text{-}bucket\text{-}logs}\text{”})\ \} \end{aligned}$$

4 Security Properties Specification

We group properties into three categories that reflect their high-level meaning: *security issues*, *mitigations*, and *global protections* to security concerns. We view these in analogy to *must* and *may* specifications, which one would use to express that an issue may be present (vs. must be absent) or that a protection must be in place (vs. may be missing). Each property type is matched to a corresponding query structure, which aids the translation of security requirements into formal specifications and implements different fail/pass logics. Queries are written as

description logic expressions whose outcome can be one of UNSAT, SAT with no instance found (SAT/0), and SAT with instances (SAT/+). These are achieved by running a satisfiability check, possibly followed by an instance retrieval call.

Mitigations are configurations of single resources that reduce the likelihood of a security event. In order to pass, these checks must be verified. Examples are:

M1 *"All buckets must keep logs"*,
M2 *"Only buckets that host websites can have a public preset ACL"*, and
M3 *"Data stores must have backup or versioning enabled"*.

Security Issues are configurations that potentially increase exposure to security concerns. In order to pass, these checks must be falsified. Examples are:

I1 *"There may be a bucket that is not encrypted"*,
I2 *"Encrypted bucket that sends events to a not-encrypted queue"*, and
I3 *"There may be a networking component that opens all ports to all"*.

Global Protections are more general mitigations, applied on single resources or as configuration patterns, whose presence and proper configuration ensures protection over the system as a whole. Examples are:

P1 *"There is an alarm configured to perform an action when triggered"*, and
P2 *"There is a configuration recorder logging changes to the infrastructure"*.

We report additional examples in Appendix E and refer the reader to the tool repository [14] for the properties specification files.[5]

5 Application to Existing Infrastructure

We now discuss the application of our approach to real-world IaC deployments. We analyze AWS CloudFormation specification and configuration files, showing that the approach is practical, scalable, and identifies potential security issues.

Operation of the Tool We develop a tool that performs three main tasks. First, the encoding of the cfn resource specifications into formal models (*Resource Terminologies*).[6] Second, the encoding of the actual cfn configuration files, also called *StackSet*, into formal models (*Infrastructure Model*). Third, inference and query answering for a set of predefined queries. We use the OWLApi [22] for the encoding phase, and JFact [37] as the inference engine.

Experimental Setup We run our tool on 15 CloudFormation StackSets openly available on GitHub. Regarding metrics, we define the infrastructure size as the numbers of both declared resources (N) and their types (N_{RT}). The latter determines which *resource terminologies* are imported into the final encoded model and thus influences its size, measured in number of logical axioms (N_α). The smallest StackSet has 6 resources and 6 resource types, the largest has 508

[5] https://tiny.cc/PropertiesSpecifications
[6] Available here: https://tiny.cc/ResourceTerminologies

Table 1. Evaluation results (mean times in millisec).

ID	N	N_{RT}	ENC	N_α	INF	USAT	SAT0	SAT+
05	6	6	44.53	814	30.64	0.67	–	2.46
11	8	8	79.22	917	37.09	0.72	–	2.86
03	10	7	59.94	886	35.65	0.64	2.23	1.56
09	10	9	76.33	940	38.66	0.68	5.03	2.96
02	11	8	76.73	1194	49.99	0.85	2.66	2.02
01	16	7	94.95	1007	43.38	0.66	3.96	1.83
08	19	8	87.66	1051	50.93	0.78	5.40	3.23
10	30	9	89.07	1177	71.23	0.86	2.62	2.08
06	30	12	102.00	1666	108.30	1.05	–	4.91
12	31	21	185.06	2798	301.61	4.99	24.93	36.43
13	51	32	241.17	3835	608.09	7.16	38.56	47.93
14	73	31	264.56	4143	847.36	2.83	51.36	19.20
15	79	21	313.40	4596	901.18	2.86	–	17.55
04	132	33	363.58	4834	2100.85	2.94	162.95	23.21
07	508	21	1005.46	10161	15834.14	7.34	40.86	13.52

resources and 21 resource types. We implement 50 properties from the ScoutSuite collection [34] that are applicable at design time and, thus, over IaC deployment files. Of the 50 properties, 29 are *mitigations*, 18 are *security issues*, and 3 are *global protections*. We conduct our evaluation on an Intel Core i5 with 16GB RAM and perform warmup runs and clear the heap before each measurement. This tuning helps to minimize the impact of just-in-time compilation and to reduce the likelihood of garbage collection during the measured benchmark runs.

Results Evaluation The average compilation time of the entire cfn resource specifications (542 files) was 940ms. Table 1 reports the results of our experimental evaluation. StackSets are sorted by number of resources. For each, we measure the time taken by the stackset encoding (ENC), inference (INF), and query answering task (grouped by outcome: UNSAT, SAT with no instances, and SAT with instances). As we can see from the table, the encoding time increases with the infrastructure's size, producing larger models that require longer inference times. Average query answering times increase accordingly. UNSAT queries have shorter average answering times than those evaluating to SAT/0 or SAT/+ (UNSAT proofs are found before a SAT outcome can be deduced). In addition, once a query is proved SAT, we invoke a procedure for *instances retrieval* to determine whether satisfying instances are present or not. The specific infrastructure configuration and its size are the main influencing factors of query answering times. Considering that the average template has about 50-100 resources, and templates having 100-500 resources are rare, the results suggest that our approach scales to real-world IaC templates. For example, StackSet **04** has 132 resources, is encoded in 363ms, classified in 2.1s, and has a max average per-query time of 162ms. Assuming a pool of 100 checks to be run, the automated modeling and verification of such an infrastructure would take, in the worst-case, around 18s.

Found Security Issues. Across all 15 deployments, we run 15x50 = 750 checks: 608 pass and 142 fail. Of the 142 failing checks, 73 do not return any instance and 69 return one or more instances (i.e., they fail with a SAT/+ outcome). Such a difference is due to the nature of the single check and its definition of failure. A *global protection* check fails when no instance implementing the protection is found; a *security issue* check fails whenever is possible (SAT/0 or SAT/+); and a *mitigation* check fails when no instance is found. We consider SAT/+ findings particularly important, as not only they witness a potential security issue but also an actual misconfiguration. In particular, the 69 SAT/+-failing checks fail on 239 resource instances, with the most found issues being:

Missing or misconfigured encryption	131
Missing or misconfigured logging	46
Missing or misconfigured versioning/backup/replication	44
Missing User password reset requirement	12
Misconfigured authorization	3
Misconfigured networking configuration	3

The 73 findings returning no instances fall into two groups: the absence of any monitoring or alarming system is very frequent, so it is the dependency on external resources whose security posture cannot be assessed. As an illustration of this, we refer the reader to Appendix F for the security report of StackSet **15**.

Absent global monitoring/alarming/logging protection	41
Usage of external resources with unknown configuration	32

6 Semantic Reasoning about Dataflows

To conclude our study, we manually craft two proof-of-concept models of terms related to cloud security (ontologies). We use these to extend the formalization of the CloudFormation IaC specification that was automatically generated by our tool. Such domain-specific ontologies formalize several common cloud terms, such as account, deployment, authenticated and unauthenticated users; generic dataflow terms, such as storage, process, nodes, and flows of different kind; and service-specific dataflow terms. By adding these on top of the underlying IaC formal specification, we can reason about the higher-level business logic and reachability of the infrastructure, and we can abstract it and visualize it in a more convenient way. This is where the full inference power of description logics comes into play. Such an inference power would be hard to achieve with an alternative encoding (e.g. using a modal logic). Let us illustrate how this technique is applied to system-level analyses of interest for a security review: *dataflow* and *trust boundary* analyses. A trust boundary is a portion of a system whose components trust each other and where data can securely flow. Multiple trust boundaries may exist within one system. Dataflows that travel across boundaries may introduce security issues and should be carefully reviewed. In Fig. 1, we see an example of such a situation, where the infrastructure is deployed across two accounts,

```
"CustomerData": {
  "Type": "AWS::S3::Bucket",
  "Properties": {
    "LoggingConfig": {
      "DestinationBucket": "
          AccessLog" }}},
"TopicSubscription":{
  "Type": "AWS::SNS::Subscription",
  "Properties": {
    "Endpoint": "devs@mail",
    "Protocol": "email",
    "TopicArn": "AccessTopic" }}
```

```
"TestData": {
  "Type": "AWS::S3::Bucket",
  "Properties": {
    "LoggingConfig": {
      "DestinationBucket": "
          AccessLog" }}},
"AccessLog": {
  "Type": "AWS::S3::Bucket",
  "Properties": {
    "NotificationConfig" : {
      "TopicConfig" : {
        "Topic":"AccessTopic" }}}},
"AccessTopic": {
  "Type": "AWS::SNS::Topic" ... }
```

Fig. 1. Sample template: accounts *prod* (left) and *test* (right). Encoding in Appendix D

prod and *test*, sharing resources `AccessLog` and `AccessTopic`. In our encoding, we use the so-called DLs *inclusion* axioms to rewrite properties that (when chained) imply the existence of a more general relation and to infer additional characteristics of nodes. For example, in the following list axioms 2-7 formalize the relationships of *"logging to"* and *"sending notifications to"* a resource, which imply the existence of a *transitive dataflow* between nodes; and axioms 8-9 allow to infer that the node *devs@mail* is an external node.

$$\mathsf{LoggingConfig} \circ \mathsf{DestinationBucket} \sqsubseteq \mathsf{logsTo} \tag{1}$$

$$\mathsf{TopicArn}^{-} \circ \mathsf{Endpoint} \sqsubseteq \mathsf{sendsNotifications} \tag{2}$$

$$\mathsf{NotificationConfig} \circ \mathsf{TopicConfig} \circ \mathsf{Topic} \sqsubseteq \mathsf{sendsNotifications} \tag{3}$$

$$\mathsf{logsTo} \sqsubseteq \mathsf{dataflow} \tag{4}$$

$$\mathsf{sendsNotifications} \sqsubseteq \mathsf{dataflow} \tag{5}$$

$$\mathsf{dataflow} \circ \mathsf{dataflow} \sqsubseteq \mathsf{dataflow} \tag{6}$$

$$\exists \mathsf{Protocol}.\{\text{“}\texttt{email}\text{”}\} \sqsubseteq \forall \mathsf{Endpoint}.\mathsf{EmailAddress} \tag{7}$$

$$\mathsf{EmailAddress} \sqsubseteq \mathsf{ExternalNode} \tag{8}$$

This encoding enables us to compute a succinct dataflow diagram from the reasoned IaC configuration (see Fig. 2), and to formally verify properties that usually require a manual analysis of the infrastructure and its underlying graph representation. E.g., the question *"Can data flow from the customer-data bucket to the outside?"* can now be formalized as a DL formula and, using a reasoning engine, the existence of a dataflow that starts on the *customer-data* bucket and reaches the *devs@mail* node can now be inferred. We note that, due to the structure of the `TopicSubscription` resource, this dataflow could not have been detected with simple reachability analysis on a graph built without the aid of semantic reasoning. Moreover, the dataflow diagram highlights another potential source of information leakage: testers being

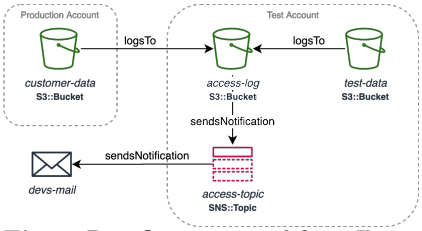

Fig. 2. Dataflow extracted from Fig. 1

exposed to customer access information. This needs to be mitigated by enforcing the proper trust boundaries, in particular, by adding a dedicated access log storage for *customer-data* bucket in the *prod* account.

7 Related Work

To the best of our knowledge, the problem of formally verifying the design of a cloud infrastructure in its entirety has not been addressed before. Formal reasoning techniques have been successfully applied to different aspects of the cloud, e.g. networks and access policies [16,7,4,5]. Non-formal tools exist that recommend and run checks against *already deployed* resources [34,13], or scan IaC templates [11,10] for syntactical patterns violating security best practices. These checks overlap considerably and can be expressed in our framework too. The disadvantages of these tools are that checks are local to single components, can be performed only post-deployment, need complex configurations, access permissions, or even manual interaction. The CFn-Linter [10] has a rule-based component that users can extend with custom syntax checks, but none of the rules currently available focus on security. The CFn-nag linting tool [11] checks compliance to best practices only locally to the single resources; e.g., it cannot detect issues such as *"There is an events queue, receiving from a bucket with critical functionality, that may not be encrypted"* or *"There might be a user that is shared by multiple policies"* (which would go against the *least privilege* principle); as well as including in its analysis external resources that are referenced by the template being linted. Regarding our choice of logic, large-scale configuration problems have been tackled with description logic before [26,27]. We took advantage of DL's open-world assumption to implement, in our properties encoding, verification and falsification. One potential alternative to DLs as a modeling language would be to use 3-valued models with labels on states and transitions and apply model checking [8,9]. However, expressive branching-time logics [25,32] have not been studied in the context of 3-valued models. We are also not aware of tool support at the level available for DLs (cf. [17,21]).

8 Conclusion and Future Work

Throughout this case study, we investigated the usage of description logics-based semantic reasoning to checking the security of cloud infrastructure pre-deployment. We formalized and encoded IaC specifications and configurations into DL models and verified the presence and absence of potential security issues. We showed how this approach enables deeper system-level analyses. All results can be generalized to other existing IaC tools. To conclude, this experience helped us to identify common misconfigurations and to understand where to direct our future research. We plan to continue researching for an even better-fitting description logic formalism, query language, three-valued semantics, and decision procedures for verification and falsification of properties relevant to security analyses, such as dataflows, trust boundaries, and threat modeling.

References

1. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press (2003)
2. Baader, F., Horrocks, I., Lutz, C., Sattler, U.: An Introduction to Description Logic. Cambridge University Press (2017)
3. Baader, F., Horrocks, I., Sattler, U.: Description logics. In: Handbook of Knowledge Representation, Foundations of Artificial Intelligence, vol. 3, pp. 135–179. Elsevier (2008)
4. Backes, J., Bayless, S., Cook, B., Dodge, C., Gacek, A., Hu, A.J., Kahsai, T., Kocik, B., Kotelnikov, E., Kukovec, J., McLaughlin, S., Reed, J., Rungta, N., Sizemore, J., Stalzer, M.A., Srinivasan, P., Subotic, P., Varming, C., Whaley, B.: Reachability analysis for aws-based networks. In: CAV (2). Lecture Notes in Computer Science, vol. 11562, pp. 231–241. Springer (2019)
5. Backes, J., Bolignano, P., Cook, B., Dodge, C., Gacek, A., Luckow, K.S., Rungta, N., Tkachuk, O., Varming, C.: Semantic-based automated reasoning for AWS access policies using SMT. In: FMCAD. pp. 1–9. IEEE (2018)
6. Binz, T., Breitenbücher, U., Kopp, O., Leymann, F.: TOSCA: Portable Automated Deployment and Management of Cloud Applications, pp. 527–549. Springer New York, New York, NY (2014)
7. Bouchenak, S., Chockler, G.V., Chockler, H., Gheorghe, G., Santos, N., Shraer, A.: Verifying cloud services: present and future. Operating Systems Review **47**(2), 6–19 (2013)
8. Bruns, G., Godefroid, P.: Model checking partial state spaces with 3-valued temporal logics. In: CAV. Lecture Notes in Computer Science, vol. 1633, pp. 274–287. Springer (1999)
9. Bruns, G., Godefroid, P.: Model checking with multi-valued logics. In: ICALP. Lecture Notes in Computer Science, vol. 3142, pp. 281–293. Springer (2004)
10. The AWS CloudFormation Linter (2020), `https://github.com/aws-cloudformation/cfn-python-lint`, Last accessed on 2020-10-15
11. The CFnNag Linting Tool (2020), `https://github.com/stelligent/cfn_nag`, Last accessed on 2020-10-15
12. Challita, S.: Inferring Models from Cloud APIs and Reasoning over Them: A Tooled and Formal Approach. (Inférer des modèles à partir d'APIs cloud et raisonner dessus: une approche outillée et formelle). Ph.D. thesis, Lille University of Science and Technology, France (2018)
13. Infrastructure Security, Compliance, and Governance (2020), `http://www.cloudconformity.com/`, Last accessed on 2020-08-04
14. CloudFORMAL: Prototype Implementation, `http://github.com/claudiacauli/CloudFORMAL`, Last accessed on 2020-10-15
15. Resource Specification (2020), `https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cfn-resource-specification.html/`, Last accessed on 2020-08-13
16. Cook, B.: Formal reasoning about the security of amazon web services. In: CAV (1). Lecture Notes in Computer Science, vol. 10981, pp. 38–47. Springer (2018)
17. D'Ippolito, N., Fischbein, D., Chechik, M., Uchitel, S.: MTSA: the modal transition system analyser. In: ASE. pp. 475–476. IEEE Computer Society (2008)
18. Glimm, B., Horrocks, I., Motik, B., Stoilos, G., Wang, Z.: Hermit: An OWL 2 reasoner. J. Autom. Reasoning **53**(3), 245–269 (2014)

19. Google Deployment Manager, `https://cloud.google.com/deployment-manager`, Last accessed on 2021-01-28
20. Grau, B.C., Horrocks, I., Motik, B., Parsia, B., Patel-Schneider, P.F., Sattler, U.: OWL 2: The next step for OWL. J. Web Semant. **6**(4), 309–322 (2008)
21. Gurfinkel, A., Wei, O., Chechik, M.: Yasm: A software model-checker for verification and refutation. In: CAV. Lecture Notes in Computer Science, vol. 4144, pp. 170–174. Springer (2006)
22. Horridge, M., Bechhofer, S.: The OWL API: A java API for OWL ontologies. Semantic Web **2**(1), 11–21 (2011)
23. Horrocks, I., Patel-Schneider, P.F., van Harmelen, F.: From SHIQ and RDF to OWL: the making of a web ontology language. J. Web Semant. **1**(1), 7–26 (2003)
24. Krötzsch, M., Simancik, F., Horrocks, I.: A description logic primer. CoRR **abs/1201.4089** (2012)
25. Kupferman, O., Grumberg, O.: Buy one, get one free!!! J. Log. Comput. **6**(4), 523–539 (1996)
26. McGuinness, D.L., Resnick, L.A., Jr., C.L.I.: Description logic in practice: A CLASSIC application. In: IJCAI. pp. 2045–2046. Morgan Kaufmann (1995)
27. McGuinness, D.L., Wright, J.R.: Conceptual modelling for configuration: A description logic-based approach. AI EDAM **12**(4), 333–344 (1998)
28. Microsoft Azure Resource Manager (2020), `https://azure.microsoft.com/en-us/features/resource-manager/`, Last accessed on 2021-01-28
29. Morris, K.: Infrastructure as code: managing servers in the cloud. " O'Reilly Media, Inc." (2016)
30. Musen, M.A.: The protégé project: a look back and a look forward. AI Matters **1**(4), 4–12 (2015)
31. Patel-Schneider, P., Grau, B.C., Motik, B.: OWL 2 web ontology language direct semantics (second edition). W3C recommendation, W3C (December 2012), http://www.w3.org/TR/2012/REC-owl2-direct-semantics-20121211/
32. Sattler, U., Vardi, M.Y.: The hybrid μ-calculus. In: IJCAR. Lecture Notes in Computer Science, vol. 2083, pp. 76–91. Springer (2001)
33. Schmidt-Schauß, M., Smolka, G.: Attributive concept descriptions with complements. Artif. Intell. **48**(1), 1–26 (1991)
34. Multi-Cloud Security Auditing Tool (2020), `http://github.com/nccgroup/ScoutSuite`, Last accessed on 2020-08-04
35. Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. J. Web Semant. **5**(2), 51–53 (2007)
36. Terraform, `https://www.terraform.io/`, Last accessed on 2021-01-28
37. Tsarkov, D., Horrocks, I.: Fact++ description logic reasoner: System description. In: IJCAR. Lecture Notes in Computer Science, vol. 4130, pp. 292–297. Springer (2006)

A AWS CloudFormation Formalization

As a preliminary step in the process of representing CloudFormation templates as formal models, we identify the key components that are responsible for both the resources' settings and their interactions, and map them into a precise mathematical notation. This notation will serve as an intermediate representation, aiding the translation of cfn templates into description logic knowledge bases. The formalization, and the successive translation, are straightforward; however, we report it here in appendix to give a more rigorous definition of the contribution presented in the paper.

As introduced in the paper, the two main components of IaC tools, and therefore of CloudFormation, are *resource specifications* and configuration (or *template*) files.

Resource Specifications For each "resource type" declarable in a template, the specification defines its properties and their allowed values. We call such a structure the *Resource-Type*. Similarly to a JSON Schema, a specification file is a JSON document itself that supports validation,[7] meaning that every *instance* of the given type must validate against it. More precisely, each resource specification provides the schema of the main resource-type and of possibly many (sub)property-types. This is done by using *keywords* to shape and restrict the set of valid documents.

Template Files Configuration files contain the resource declarations of the instances that the user wishes to deploy. This is done by writing JSON-formatted text files containing a list of resource instances to be deployed together with their settings (conform to the definitions given in the resource's type specification). Under each resource instance's name and type, users specify properties as a list of attribute-value pairs. In line with the general definition of JSON documents, in cfn templates attributes have values that range over objects, arrays, or primitive types. When referring to a property's value type, we say that the values allowed for that property range over subproperties, arrays, strings, integers, longs, doubles, booleans, and the constant null. Overall, the values that are allowed in cfn templates coincide with the standard JSON values: integers, longs, and doubles are JSON numbers; subproperties are JSON objects.

Formalizing Specifications Syntax. Both resource- and property-types have a name and a collection of property definitions. Each property definition is given by a name, a type, and a flag indicating whether the property is required or not. Let $\texttt{RT}_i$ denote the name of the principal resource-type within the i-th resource specification, and the labels $\texttt{PT}_{i.1} \dots \texttt{PT}_{i.\ell^i}$ denote the names of all its (sub)property-types (collectively referred to as the set $\texttt{PT}_i$). We assume the existence of a renaming function that unambiguously maps specification names to unique identifiers and, when talking about properties and types, we mean their unique renaming. We call the properties appearing under the specification of a

[7] Strictly speaking, according to the technical standard of *"JSON Schema Language"* in IETF Internet-Draft, this specification is not the JSON Schema of a template.

resource- or property-type its *defining properties*, or simply *its* properties. The type of a property from the i-th specification ($\mathtt{T_i}$) is one of:

$$\begin{aligned} \mathtt{T}_i &::= T \mid \mathit{List}(T) \mid \mathtt{PT}_i \mid \mathit{List}(\mathtt{PT}_i) \\ T &::= \mathit{String} \mid \mathit{Integer} \mid \mathit{Long} \mid \mathit{Double} \mid \mathit{Boolean} \end{aligned} \tag{9}$$

Properties within a given RT_i always range over types of the i-th specification, never referencing a set $PT_{i'}$ s.t. $i' \neq i$ or RT_i itself. In other words, resources do not *explicitly* point to each other. However, as explained in the *Inter-Resource References* paragraph below, there can be *implicit* references.

Formalizing Templates Syntax. Every resource declared in a cfn template can be modeled as a tree-shaped structure whose root represents the resource itself and the branching follows from its properties. Accordingly, the whole template is a forest of resource trees. Formally, each tree t is the tuple (N, E, A, v), where N is the set of nodes; E and A are sets of edges; and v is a value function. The set N is the tree domain, which contains the root and a node for each object, array, and value appearing within a resource declaration's `Properties` block. It is, therefore, partitioned into the disjoint sets $\{r\}$, *Obj*, *Arr*, and *Val*. The root node r carries extra information: the pair $\langle id, type \rangle$ of the resource's logical ID and type. The edges of the tree derive from two nesting relations, found in properties and arrays, respectively. The first is the *property-value* relation E, connecting a root or object node to the nodes encoding their properties values. E-edges are labeled by property names from the set $\mathsf{N_R}$ and are contained in the set $(\{r\} \cup Obj) \times \mathsf{N_R} \times (N \setminus \{r\})$. The second relation is called *array* relation A, and links every array node to its elements by introducing indexed edges. It is contained in $Arr \times \mathbb{N} \times (Obj \cup Val)$. Nodes in N do not carry information about the actual content of a property value. This is particularly relevant for nodes in the set *Val* that are leaf nodes and need to be assigned a concrete element from their respective domains. An exception is made only for *Val* nodes corresponding to the JSON null, which do not have an assignment. Let *String*, *Int*, *Long*, *Double*, and *Bool* be the sets of all concrete elements that can be associated with the content of a value node. The partial function $v : Val \to (String \cup Int \cup Long \cup Double \cup Bool)$ maps the leaf nodes to concrete elements and it is undefined for null nodes.

Inter-Resource References. In Listing 1.1, the type of property `DestinationBucketName` is declared as *String*. However, further documentation specifies that the string must contain a reference to a resource of type S3::Bucket. Usually, such reference takes the form of an *Amazon Resource Name* (aka ARN, the instance's unique address) or, if the resource is in the same template, of its name. The snippet in Listing 1.2 is a case in point – `"ConfigStorage"` is the *name* of the referenced resource of type S3::Bucket. To cover for such cases, we extend our formalization to include the definition and the resolution of these references, which are needed when encoding the cfn resource specifications and the cfn templates, respectively. Let $\mathsf{N_C}$ be the alphabet of distinct type names, $\mathsf{N_R}$ the alphabet of property names, and $\mathsf{N_I}$ a set of individual names. To *define* the references as part of the specification, we introduce a function $\mathsf{range} : \mathsf{N_R} \to \mathsf{N_C}$ whose domain

is the set of *all* properties and range matches the grammar of Equation (9) extended with resource-types. If a property $p \in \mathsf{N_R}$ is of type string and intended to reference a resource, $\mathsf{range}(p)$ is set to the correct resource-type RT_j (for some j given in the documentation); otherwise, $\mathsf{range}(p)$ is set to the original type prescribed in the specification file. To *resolve* the references in actual templates, we introduce a $\mathsf{resolve}$ procedure that given a node x returns the corresponding DL individual i_x; i.e., $\mathsf{resolve} : N \to \mathsf{N_I}$. The function is undefined over nodes not referencing any resource; returns a pre-existing individual over those nodes that point to a resource declared in the same template; returns a fresh individual in $\mathsf{N_I}$ for those nodes referencing a resource that is external to the template. From now on, we assume range to capture the *real* type of a given property and $\mathsf{resolve}$ to return the correct individual. We rely on these for the encoding of the next section.

B AWS CloudFormation Description Logic Encoding

We now describe the translation implemented in our tool from the mathematical notation formalized in the previous Appendix to DL. The tool creates a *"Terminological Box"* (also called *TBox* in description logic) from the resource specifications and an *"Assertional Box"* (also called *ABox* in description logic) from the cfn templates.

Resource Specifications Translated to Terminological Boxes. We construct a global cfn DL specification $\mathcal{T}^{\mathsf{cfn}}$ as the union of TBoxes $\mathcal{T}_{RT_i}$ that we create for each individual resource specification RT_i. We introduce concept predicates matching resource-, property-, and primitive types, and role predicates matching property names. The structure of the cfn resource specifications is preserved by introducing axioms that simulate the characteristics of the properties declared therein: domain, range, requiredness, and functionality. Let X be a property- or resource-type in RT, and p one of its defining properties. The collection of TBox axioms enforce (1) the connectivity of a resource to its properties, (2) the type application of the properties, (3) required properties, and (4) connection to individuals rather than lists. The first and the second are enforced using dom and ran axioms (see Sec. 2). For the third we use axioms of the form $\mathsf{req}(p, X) \equiv X \sqsubseteq \exists p.\top$, which enforce X-nodes to have at least one p-successor. For the fourth we use axioms of the form $\mathsf{fun}(p) \equiv {\geq_2} p.\top \sqsubseteq \bot$, which enforce a single p successor when p is not a list. Overall, the tool builds $\mathcal{T}_{RT}$ as:

$$\mathcal{T}_{RT} = \bigcup_{(X,p) \in RT} \{\ \mathsf{dom}(p, X), \mathsf{ran}(p, \mathsf{range}(p)), \mathsf{req}(p, X), \mathsf{fun}(p)\ \}$$

We note that we have not created any individuals or assertions. We created axioms that constrain the valid interpretations and enable inference on the characteristics of underspecified nodes, such as those that are *pointed by* a cfn template but not declared in it.

Templates Translated to Assertional Boxes. For a template t, valid against the cfn specifications, our tool creates ABox assertions $\mathcal{A}^{\mathsf{t}}$. The construction ensures that $\mathcal{A}^{\mathsf{t}}$ is consistent w.r.t. the DL encoding of the specifications, $\mathcal{T}^{\mathsf{cfn}}$. For resource declarations we add individuals, and model their configuration by means of assertions. Because DLs are interpreted under an *open-world assumption*, which treats missing information as unknown, and since we assume a template to be complete, we *"close the world"* around the given properties' assertions by using additional concept assertions. Let us recall, from the formalization, that a template is a forest of resource trees, each having its own root, nodes, edges, and value function; and that trees may be implicitly connected via inter-resource references. In the following, we give an overview of how: *i*) our tool creates individuals for nodes, and *ii*) creates concept and role assertions from node types and properties.

Create Individuals for Nodes We construct the alphabet of individual names $\mathbb{I}$ and a map ind from nodes into individuals, which is used for the construction of role assertions. For x a root node with identifier id, our tool adds the symbol id to $\mathbb{I}$ and sets $\mathsf{ind}(x) = \{id\}$. For x a node in *Obj*, our tool adds individual i_x to $\mathbb{I}$ and sets $\mathsf{ind}(x) = \{i_x\}$. For x a node in *Val* s.t. $\mathsf{resolve}(x)$ is defined, our tool does not add any new individual names (the $\mathsf{resolve}$ invocation takes care of that, see Sec. 3) and sets $\mathsf{ind}(x) = \{\mathsf{resolve}(x)\}$. For x a node in *Val* s.t. $\mathsf{resolve}(x)$ is undefined and $v(x) = y$, our tool adds the symbol y to $\mathbb{I}$ and returns $\mathsf{ind}(x) = \{y\}$. For x a node in *Arr* s.t. $A(x) = \{(1, v_1), \ldots, (k, v_k)\}$, our tool does not add any fresh names to $\mathbb{I}$ but sets $\mathsf{ind}(x) = \bigcup_{i \in [k]} \mathsf{ind}(v_i)$. The result is that we do not preserve the encoding of the array parent node x but rather relate the parent of x to each encoded element of the array. We note that ind is set to a singleton set in all cases except the latter. When clear from context that such set is a singleton, we denote by i_x the individual encoding node x, omitting the statement $i_x \in \mathsf{ind}(x)$.

Create Assertions from Configurations The newly introduced individuals are related through assertions, which encode the structure of the resource trees and their settings. The encoding of root and object nodes is straightforward. However, since DLs are interpreted under an *open-world assumption* that treats missing information as unknown, we seek a special treatment for the explicit encoding of properties that are *not* given in the template and for the encoding of bounded arrays. The assertions created from nodes and properties are as follows:

$$
\begin{aligned}
\mathsf{root}(j) &= \{\ \mathsf{RT}(id), \mathsf{p}(id,i) \mid r_j = \langle id, RT \rangle,\ (r_j,p,c) \in E^j,\ i \in \mathsf{ind}(c)\ \} \\
\mathsf{obj}(j,x) &= \{\ \mathsf{p}(i_x,i) \mid (x,p,c) \in E^j,\ i \in \mathsf{ind}(c)\ \} \\
\mathsf{abs}(j,x) &= \{\ \neg \exists \mathsf{p}.\top(i_x) \mid (x,p,c) \notin E^j,\ p \text{ is a property of } x\text{'s type}\ \} \\
\mathsf{arr}(j,x) &= \{\ {\leq_k}\mathsf{p}.\top(i_x) \mid (x,p,a) \in E^j,\ a \in Arr^j,\ |A^j(a)| = k\ \}
\end{aligned}
$$

For every root node r_j, with identifier id and type RT, our tool adds the concept assertion $\mathsf{RT}(id)$ (see root). We add the role assertion $\mathsf{p}(i_x,i)$ to encode the properties of either a root or object node x for every triplet (x,p,c) in the set E and every i arising from the mapping of node c (see root, obj). For a node x

and property p, which could be set (according to the specification) under a node of x's type but is absent in the template, our tool adds the axioms $\mathsf{abs}(\cdot)$. These axioms enforce individuals i_x to have no p-successors. For a node x linking to an array node a, our tool adds the axioms $\mathsf{arr}(\cdot)$, which enforce the exact array cardinality. In addition, we remark that we do not need any further assertions to encode value and array nodes, as their mapping into individuals is enough. Lastly, no axioms are added for the individuals introduced by the resolve function (and external to the template), since their information is incomplete. The tool, thus creates the following ABox $\mathcal{A}^{\mathsf{t}}$:

$$\mathcal{A}^{\mathsf{t}} = \bigcup_{t_j \in \mathsf{t}} \Big(\mathsf{root}(j) \ \cup \bigcup_{x \in Obj^j} \mathsf{obj}(j,x) \ \cup \bigcup_{x \in \{r_j\} \cup Obj^j} \big(\mathsf{abs}(j,x) \ \cup \ \mathsf{arr}(j,x) \big) \Big)$$

C Formal Encoding of the Infrastructure in Listing 1.2

By following the encoding procedure presented, the S3::Bucket resource specification (partly reported in Listing 1.1) and the deployment template snippet of Listing 1.2 are encoded into the knowledge base $\mathcal{K} = (\mathcal{T}_{\mathsf{S3::Bucket}}, \mathcal{A})$ that is defined as follows:

$$\begin{aligned}
\mathcal{T}_{\mathsf{S3::Bucket}} = \{\ & \exists \mathsf{bucketName}.\top \sqsubseteq \mathsf{BUCKET},\ \exists \mathsf{bucketName}^{-}.\top \sqsubseteq \mathsf{STRING},\\
& {}_{\geq 2}\mathsf{bucketName}.\top \sqsubseteq \bot,\ \exists \mathsf{loggingConfig}.\top \sqsubseteq \mathsf{BUCKET},\\
& \exists \mathsf{loggingConfig}^{-}.\top \sqsubseteq \mathsf{LOGGINGCONFIG},\ {}_{\geq 2}\mathsf{loggingConfig}.\top \sqsubseteq \bot,\\
& \exists \mathsf{destinationBucket}.\top \sqsubseteq \mathsf{LOGGINGCONFIG},\\
& \exists \mathsf{destinationBucket}^{-}.\top \sqsubseteq \mathsf{BUCKET},\ {}_{\geq 2}\mathsf{destinationBucket}.\top \sqsubseteq \bot,\\
& \exists \mathsf{logFilePrefix}.\top \sqsubseteq \mathsf{LOGGINGCONFIG},\ \exists \mathsf{logFilePrefix}^{-}.\top \sqsubseteq \mathsf{STRING},\\
& {}_{\geq 2}\mathsf{logFilePrefix}.\top \sqsubseteq \bot,\ \exists \mathsf{notifConfig}.\top \sqsubseteq \mathsf{BUCKET},\\
& \exists \mathsf{notifConfig}^{-}.\top \sqsubseteq \mathsf{NOTIFCONFIG},\ {}_{\geq 2}\mathsf{notifConfig}.\top \sqsubseteq \bot,\\
& \exists \mathsf{topicConfigs}.\top \sqsubseteq \mathsf{NOTIFCONFIG},\ \exists \mathsf{topicConfigs}^{-}.\top \sqsubseteq \mathsf{TOPICCONFIG},\\
& \exists \mathsf{topic}.\top \equiv \mathsf{TOPICCONFIG},\ \exists .\mathsf{topic}^{-}.\top \sqsubseteq \mathsf{TOPIC}\ \}\\
\mathcal{A} = \{\ & \mathsf{BUCKET}(\mathit{ConfigS3Bucket}),\\
& \mathsf{bucketName}(\mathit{ConfigS3Bucket}, \mathit{"ConfigStore"}),\\
& \mathsf{loggingConfig}(\mathit{ConfigS3Bucket}, x),\ \mathsf{destinationBucket}(x, \mathit{ConfigS3Bucket}),\\
& \mathsf{logFilePrefix}(x, \mathit{"config\text{-}bucket\text{-}logs"})\ \}
\end{aligned}$$

The assertions introduced in the ABox $\mathcal{A}$ only specify the nature of the root node *ConfigS3Bucket*. No concept assertion is introduced for the nested nodes x, *ConfigStore*, and *config-bucket-logs*. However, by combining the axioms in $\mathcal{T}_{\mathsf{S3::Bucket}}$ with the knowledge asserted in $\mathcal{A}$, the reasoning engine will infer that $\mathsf{LOGGINGCONFIG}(x)$, $\mathsf{STRING}(\mathit{ConfigStore})$, and $\mathsf{STRING}(\mathit{config\text{-}bucket\text{-}logs})$.

D Formal Encoding of the Infrastructure in Fig. 1

The encoding procedure applied to the template of Fig. 1 and to the resource specifications of the S3 and SNS resource types (for the sake of brevity not reported here) results in the knowledge base $\mathcal{K} = (\ \mathcal{T}_{\mathsf{S3::Bucket}} \cup \mathcal{T}_{\mathsf{SNS::Topic}} \cup \mathcal{T}_{\mathsf{SNS::Subscription}}, \mathcal{A}\ )$, where $\mathcal{T}_{\mathsf{S3::Bucket}}$ is the same as in the previous appendix, and the remaining components are as follows:

$$\begin{aligned}
\mathcal{T}_{\mathsf{SNS::Topic}} = \{\ & \exists \mathsf{subscription}.\top \sqsubseteq \mathsf{TOPIC},\ \exists \mathsf{subscription}^{-}.\top \sqsubseteq \mathsf{TOPICSUBSCR},\\
& \exists \mathsf{endpoint}.\top \equiv \mathsf{TOPICSUBSCR},\ \exists \mathsf{endpoint}^{-}.\top,\ \ {\geq}_2\mathsf{endpoint}.\top \sqsubseteq \bot,\\
& \exists \mathsf{protocol}.\top \equiv \mathsf{TOPICSUBSCR},\ \exists \mathsf{protocol}^{-}.\mathsf{STRING},\\
& {\geq}_2\mathsf{protocol}.\top \sqsubseteq \bot, \dots \}\\
\mathcal{T}_{\mathsf{SNS::Subscription}} = \{\ & \exists \mathsf{endpoint1}.\top \sqsubseteq \mathsf{SUBSCRIPTION},\ \exists \mathsf{endpoint1}^{-}.\top,\\
& {\geq}_2\mathsf{endpoint1}.\top \sqsubseteq \bot,\ \ \exists \mathsf{protocol1}.\top \equiv \mathsf{SUBSCRIPTION},\\
& \exists \mathsf{protocol1}^{-}.\mathsf{STRING},\ \ {\geq}_2\mathsf{protocol1}.\top \sqsubseteq \bot,\\
& \exists \mathsf{topicArn}.\top \equiv \mathsf{SUBSCRIPTION},\ \exists \mathsf{topicArn}^{-}.\mathsf{TOPIC},\\
& {\geq}_2\mathsf{topicArn}.\top \sqsubseteq \bot, \dots \}\\
\mathcal{A} = \{\ & \mathsf{BUCKET}(\mathit{CustomerData}),\ \mathsf{loggingConfig}(\mathit{CustomerData}, x),\\
& \mathsf{destinationBucket}(x, \mathit{AccessLog}),\ \mathsf{BUCKET}(\mathit{AccessLog}),\\
& \mathsf{notifConfig}(\mathit{AccessLog}, x'),\ \mathsf{topicConfig}(x', x''),\\
& \mathsf{topic}(x'', \mathit{AccessTopic}),\ \ \mathsf{SUBSCRIPTION}(\mathit{TopicSubscription}),\\
& \mathsf{endpoint1}(\mathit{TopicSubscription}, \mathit{devs@mail}),\\
& \mathsf{protocol1}(\mathit{TopicSubscription}, \text{“}\mathit{email}\text{”}),\\
& \mathsf{topicArn}(\mathit{TopicSubscription}, \mathit{AccessTopic}),\ \mathsf{TOPIC}(\mathit{AccessTopic}),\\
& \mathsf{BUCKET}(\mathit{TestData}),\ \mathsf{loggingConfig}(\mathit{TestData}, y),\\
& \mathsf{destinationBucket}(y, \mathit{AccessLog})\}
\end{aligned}$$

E Sample of Security Properties

E.1 Mitigations

Alsoo called *All-known-must* queries, meaning that a property φ *certainly* holds on all the *known* objects of interest, formulated as $(\sqcup_{x \in inst(\mathsf{X})}\ \{x\}\ ) \sqcap \neg\varphi$ and interpreted as *tt-ff-ff* over the three outcomes UNSAT, SAT/0, and SAT/+.

These properties are used to express that *all known instances of a given concept* X *certainly satisfy a property* φ_{X}. They are suitable for expressing *mitigations* to security threats that must be in place in order for the property to be satisfied and for the check to pass. We adopt an epistemic approach, where by *known instances* we mean all the individuals that satisfy the concept X in all models. We simulate an epistemic operator (not yet implemented in any description logic reasoner) by building a composite query that, first, fetches the

set of all individuals certainly satisfying X (which we denote as $inst(\mathsf{X})$) and, then, builds the corresponding nominal concept set to be included in the DL query. The DL queries have the following structure: $\sqcup_{x \in inst(\mathsf{X})} \{x\} \sqcap \neg\varphi_{\mathsf{X}}$ and their interpretation over the three outcomes (UNSAT, SAT/0, and SAT/+) is of True, False, and False, respectively.

Example 1. "Only S3::Buckets that host a website can be public"

$$\sqcup_{x \in inst(\mathsf{Bucket})} \{x\} \sqcap \exists \mathsf{accessControl}.\{Public\} \sqcap \neg \exists \mathsf{websiteConfig}.\top$$

Example 2. "All S3::Buckets with a critical functionality must be encrypted and log to an encrypted bucket."

$\sqcup_{x \in inst(\mathsf{CriticalBucket})} \{x\} \sqcap \neg \mathsf{EncrLoggBucket}$, where

$$\begin{aligned}
\mathsf{CriticalBucket} \equiv\ & \exists \mathsf{s3bucketName}^{-}.\mathsf{DeliveryChannel} \sqcup \\
& \exists \mathsf{s3bucket}^{-}.(\exists \mathsf{code}^{-}.\mathsf{Function}) \sqcup \\
& \exists \mathsf{s3bucket}^{-}.\mathsf{AccessLoggingPolicy} \sqcup \\
& \exists \mathsf{bucket}^{-}.(\exists \mathsf{bodyS3location}^{-}.\mathsf{RestApi}) \\
\mathsf{EncrBucket} \equiv\ & \exists \mathsf{encryption}.(\ \exists \mathsf{sseConfiguration}.(\ \exists \mathsf{kmsMasterKeyId}.\mathsf{Key}\ )) \\
\mathsf{EncrLoggBucket} \equiv\ & \mathsf{EncrBucket} \sqcap \\
& \exists\, \mathsf{loggingConfig}.(\ \exists \mathsf{destinationBucket}.\mathsf{EncrBucket}\ ))
\end{aligned}$$

Example 3. "All known KMS::Keys must be enabled"

$\sqcup_{x \in inst(\mathsf{Key})} \{x\} \sqcap \neg \exists \mathsf{enabled}.\{tt\}$

Example 4. "No S3::Bucket should store its own logs"

$\sqcup_{x \in inst(\mathsf{LogBucket})} (\ \{x\} \sqcap \exists \mathsf{loggingConfig}.\exists \mathsf{destinationBucket}.\{x\}\ )$, where

$$\mathsf{LogBucket} \equiv \exists \mathsf{destinationBucket}^{-}.\exists \mathsf{loggingConfig}^{-}.\mathsf{Bucket}$$

E.2 Security Issues

Also called *Exists-known-may* queries, meaning that a property φ *may* hold on some *known* instances, formulated as $(\sqcup_{x \in inst(\mathsf{X})} \{x\}) \sqcap \varphi$ with an interpretation of *ff-tt-tt* over the three outcomes UNSAT, SAT/0, and SAT/+.

These properties are used to express that *there exist a known instance of a concept* X *that may satisfy a property* φ_{X}. They are suitable for expressing *security issues* that might be present and could cause the check to fail. Similarly to the previous type of property, we adopt an epistemic approach and simulate the epistemic operator by building a composite query that, first, fetches the set of all individuals certainly satisfying X (which we denote as $inst(\mathsf{X})$) and, then, builds the corresponding nominal concept set to be included in the DL query. The DL queries have the following structure: $\sqcup_{x \in inst(\mathsf{X})} \{x\} \sqcap \varphi_{\mathsf{X}}$ and their interpretation over the three outcomes (UNSAT, SAT/0, and SAT/+) is of False, True, and True, respectively.

Example 5. "There is a SQS::Queue known to receive from a bucket with critical functionality that might not be encrypted"
$\sqcup_{x \in inst(\mathsf{CriticalQueue})} \{x\} \sqcap \neg \exists \mathsf{kmsMasterKeyId.Key}$, where

$$\mathsf{CriticalQueue} \equiv \mathsf{Queue} \sqcap \exists \mathsf{queue}^{-}.\exists \mathsf{queueConfig}^{-}.\exists \mathsf{notificationConfig}^{-}.\mathsf{Bucket}$$

Example 6. "There might be an IAM::User that is shared by two or more policies"
$\sqcup_{x \in inst(\mathsf{User})} \{x\} \sqcap \geq^{2} \mathsf{users}^{-}.\mathsf{Policy}$

Example 7. "There might be an EC2::SecurityGroup that opens all ports to all"

$$\sqcup_{x \in inst(\mathsf{SecurityGroup})} \{x\} \sqcap \exists \mathsf{ingress}.(\ \exists \mathsf{cidrIp}.\{0.0.0.0/0\} \sqcap \exists \mathsf{fromPort}.\{0\} \sqcap \exists \mathsf{toPort}.\{65535\}\ )$$

E.3 Global Protections

Also called *Exists-must* queries, more general than the previous, formulated as $\mathsf{X} \sqcap \varphi_{\mathsf{X}}$ with an interpretation of *ff-ff-tt* over the three outcomes UNSAT, SAT/0, and SAT/+ (corresponding to the natural semantics of DL simple concepts). These properties coincide with simple DL concept queries and are used to express that *there exists an instance of a concept* X *that certainly satisfies a property* φ_{X}. They are suitable for expressing global mitigations and configuration queries of simple facts not involving potentially underspecified resources. The DL queries are simply written as $\mathsf{X} \sqcap \varphi_{\mathsf{X}}$ and their interpretation over the three outcomes (UNSAT, SAT/0, SAT/+) is of False, False, and True, respectively.

Example 8. "There is a CloudWatch::Alarm that must perform an action when triggered" $\mathsf{Alarm} \sqcap \exists \mathsf{alarmActions}.\top$

Example 9. "There is a CloudTrail::Trail that logs global service events"
$\mathsf{Trail} \sqcap \exists \mathsf{isLogging}.\{tt\} \sqcap \exists \mathsf{includeGlobalEvents}.\{tt\}$

F StackSet 15 (Snippet of Listing 1.2) - Full Automated Reasoning Checks Report

P/F	Property ID	T/F	Outc	Outcome Description	Instances
PASS	01_AKM_BUCKETS_SHOULD_LOG	T	USAT	Either there are no S3::Buckets or there are some and they certainly keep logs	N/A
FAIL	02_AKM_NO_BUCKETS_STORING_OWN_LOGS	F	SAT+	There are S3::Buckets declared in this template that store their own logs	(configs3bucket)
PASS	03_AKM_BUCKET_STORING_LOGS_NOT_PUBLIC	T	USAT	No bucket that store logs and is public can be found	N/A
FAIL	04_AKM_BUCKETS_ENCRYPTED	F	SAT+	There are S3::Buckets - declared in this template - that are not encrypted	(configs3bucket - datas3bucket)
PASS	05_AKM_BUCKET_NOT_PUBL_UNL_WEB_CORS	T	USAT	No bucket that does not host a website or allow CORS and is public can be found	N/A
PASS	06_AKM_CRIT_LAMBDA_BUCKETS_ENCR_ROTAT	T	USAT	All S3::Bucket declared in the template are encrypted and keep logs whenever they store lambda code	N/A
FAIL	07_AKM_LOGS_STORED_ON_ENCR_BUCKETS	F	SAT+	There are S3::Buckets in this template that store logs and are not encrypted	(configs3bucket)
PASS	08_EM_ALARM_ACTION	T	SAT+	There are CloudWatch::Alarms that certainly perform an action when triggered	(cpureservationtoohighalarm memoryreservationtoohighalarm tracebackindartlogalarm subscriptionqueuedepthhigh subscriptionqueuedepthlow cpualarmhigh cpualarmlow)
FAIL	09_EM_CONFIG_RECORDER	F	USAT	There is no Config::ConfigurationRecorder that is recording changes of all resource types	N/A
FAIL	10_EM_GLOBAL_SERVICES_TRAIL	F	USAT	There is no CloudTrail::Trail that logs events from global services	N/A
PASS	11_EKM_SECURITYGROUP_ALL_PORTS_TO_ALL	F	USAT	Either there are no EC2::SecurityGroups or there are some and they certainly do not open all ports to all	N/A
PASS	12_EKM_SHARED_BY_MULTIPLE_POLICIES	F	USAT	Either there are no IAM::Users or there are some and they are certainly not shared by two or more policies	N/A
PASS	13_EKM_QUEUE_CRITICAL_NOT_ENCRYPTED	F	USAT	If there are SQS::Queues receiving notifications from a bucket - they certainly are encrypted	N/A
PASS	14_AKM_LAMBDA_FAILED_INPUT_TO_ENCRYPTED	T	USAT	Either there are no Lambda::Functions or they all send failed input to an encrypted resource	N/A
PASS	15_AKM_IAM_POLICIES_MUST_LINKED_TO_SMTH	T	USAT	Either there are no IAM::Policys or they are all attached to something	N/A
PASS	16_AKM_VPC_NO_DEFAULT_SECURITYGROUP	T	USAT	Either there are not VPC::EC2 or there are some and they are all associated with a security group	N/A
PASS	17_AKM_KEYS_ENABLED	T	USAT	All KMS::Keys are enabled	N/A
PASS	18_AKM_REPLICAS_ENCRYPTED	T	USAT	Either there are no bucket replicas or they are all encrypted	N/A
PASS	19_AKM_LAMBDAS_ENCRYPTED	T	USAT	Either there are no Lambda::Functions or they are all encrypted	N/A
PASS	20_EKM_SECURITYGROUP_OPENTOALL	F	USAT	Either there are no EC2::SecurityGroups or there are some and they are certainly not open to all IPs	N/A
PASS	21_EKM_TRAIL_CARRYING_NOTHING	F	USAT	Either there are no CloudTrail::Trails or there are some and they all certainly carry logs about data events	N/A
PASS	22_EKM_TRAIL_NO_LOGFILE_VALIDATION	F	USAT	Either there are no CloudTrail::Trails or there are some and they all certainly have log file validation enabled	N/A
PASS	23_AKM_TRAIL_LOG	T	USAT	Either there are no CloudTrail::Trails or there are some and they all certainly keep logs	N/A
PASS	24_EKM_TRAIL_NOT_ALL_REGIONS	F	USAT	Either there are no CloudTrail::Trails or there are some and they all certainly log events from all regions	N/A
PASS	25_EKM_VOLUME_NOT_ENCRYPTED	F	USAT	Either there are no EC2::Volumes or there are some and they are all certainly encrypted	N/A
PASS	26_EKM_INSTANCE_PUBLIC	F	USAT	Either there are not EC2::Instances or there are some and they certainly do not allow public IPs	N/A
PASS	27_EKM_INSTANCE_USERDATA	F	USAT	Either there are no EC2::Instances or they don't have user data on template	N/A
PASS	28_EKM_SECURITY_GROUP_OPEN_PORTS_SELF	F	USAT	Either there are no EC2::SecurityGroups or there are some and they certainly do not open all ports to self	N/A
PASS	29_EKM_SECURITYGROUP_UNUSED	F	USAT	Either there are no EC2::SecurityGroups or there are some and they are all linked to a VPC	N/A
FAIL	30_AKM_LOADBALANCER_ACCESSLOGGINGPOLICY	F	SAT+	There is at least one ElasticLoadBalancing::LoadBalancer that certainly has Access Logging Policy disabled	(elasticloadbalancer - elasticloadbalancer)
PASS	31_AKM_LOADBALANCER_S3ACCESSLOGS	T	USAT	Either there is no ElasticLoadBalancingv2::LoadBalancer or they all have S3 Access Logs enabled	N/A
PASS	32_AKM_LOADBALANCER_DELETION_PROTECTED	T	USAT	All ElasticLoadBalancingv2::LoadBalancers - if any - certainly have deletion protection enabled	N/A
PASS	33_AKM_LISTENER_NO_OLD_POLICIES	T	USAT	All LoadBalancingV2::Listeners - if any - certainly do not use old SSL policies	N/A
PASS	34_AKM_POLICIES_ATTACHED_TO_GROUPS_ONLY	T	USAT	Either there are no IAM::Policies or they are all attached to groups	N/A
PASS	35_AKM_GROUP_NO_USERS	T	USAT	All IAM::Groups - if any - are linked to some user	N/A
PASS	36_AKM_USERS_PWDRESET	T	USAT	All known IAM::Users have password reset required	N/A
PASS	37_EKM_USER_WITH_LOGINPROFILE	F	USAT	Either there are no IAM::Users or they all have password in template	N/A
PASS	38_AKM_ACCESSKEY_ROTATING	T	USAT	Either there are no IAM::Users or they are all linked to a key that is rotating	N/A
PASS	39_AKM_USER_ONE_ACCESSKEY	T	USAT	All IAM::Users have max one access key	N/A
FAIL	40_AKM_RDS_BACKUP	F	SAT+	There exists at least one declared RDS::DBInstance that has backup retention set to 0	(dbinstance)
PASS	41_AKM_RDS_MINORUPGRADE_DISABLED	T	USAT	Either there are no RDS::DBInstances or they all have auto minor upgrade disabled	N/A
PASS	42_AKM_RDS_SHORT_BACKUP	T	USAT	All RDS::DBInstances - if any - certainly do not have short backup retention	N/A
FAIL	43_AKM_DECLARED_RDS_ONEAZ	F	SAT+	There is at least one RDS::DBInstance that is certainly not replicated in different Availability Zones	(dbinstance)
FAIL	44_AKM_RDS_ENCRYPTED	F	SAT+	There exists at least one declared RDS::DBInstance that is not encrypted	(dbinstance)
PASS	45_EKM_RDS_SECURITYGROUP_OPEN	F	USAT	If there are RDS::DBinstances - they are certainly not linked to an open security group	N/A
FAIL	46_AKM_BUCKETS_VERSIONING	F	SAT+	There are S3::Buckets - declared in this template - that have versioning disabled	(configs3bucket - datas3bucket)
PASS	47_EKM_VPC_OPEN_NETWORK_ACL	F	USAT	If there are EC2::VPCs - they certainly do not have an open-to-all Network Acl	N/A
PASS	48_EKM_NETWORKACL_NOT_USED	F	USAT	Either there are not EC2::NetworkAcl or there are some and they are all linked to a VPC	N/A
PASS	49_EKM_SUBNET_BAD_ACL	F	USAT	Either there are no EC2::Subnets or there are some and they all have good ACLs	N/A
PASS	50_EKM_SUBNET_NO_FLOWLOG	F	USAT	If there are EC::Subnets - they all have FlowLog	N/A