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, Gothenburg, Sweden
[2] Amazon Web Services, Seattle, U.S.A.

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 (AWS) 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 deployment frameworks and how these are formally modeled and reasoned upon. We explore the usage of description

© The Author(s) 2021
A. Silva and K. R. M. Leino (Eds.) CAV 2021, LNCS 12759, pp. 767–780, 2021.
https://doi.org/10.1007/978-3-030-81685-8_36

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 by 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 cloud infrastructure increases, in addition to 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 [37], 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 by 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 (Sect. 3); (*ii*) a technique to express security properties (Sect. 4); (*iii*) the experimental evaluation of encoding and query times, accounting for the most common security issues that we found over publicly available IaC templates (Sect. 5); and (*iv*) an extension that enables semantic dataflow reasoning (Sect. 6). Our tool is implemented in Scala and available online [14]. We include preliminaries in Sect. 2; discuss related work in Sect. 7; and conclude in Sect. 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* [20,23,32], for which extensive tool support exists (e.g., the Protégé editor and off-the-shelf reasoners such as FaCT, HermiT, and Pellet [18,30,36,39]). We introduce the description logic $\mathcal{ALC}$ [1,24,34], *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 three additional operators: *inverse roles*, *functionality constraints*, and *complex role inclusions*. The first, denoted r^-, encodes the converse of the binary relationship r. The second enforces binary relationships to be functional. The third, 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].[1] Services are products such as 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* [15]. The *resource specification* is a collection of files that prescribe resource properties and their allowed values. Provided that a *configuration file* is valid with respect to 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. 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 Sect. 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

```
"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

[1] As of August 2020, exact number is Region-dependent.

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 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 Sect. 5 (StackSet 15) and, incidentally, it violates a security best practice: "no bucket should store its own logs." Such formalization has been instrumental to capture infrastructure configurations, resources settings and inter-connections, and to precisely and automatically encode it into DL.

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.

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

Listing 1.2. S3::Bucket instance declaration

$$Spec_{\mathsf{S3::Bucket}} = \{\ \mathsf{dom}(\mathsf{bucketName}, \mathsf{BUCKET}),\ \mathsf{ran}(\mathsf{bucketName}, \mathit{String}),\ (\mathsf{Funct\ bucketName}),\ \ldots,\ \mathsf{dom}(\mathsf{destinationBucket}, \mathsf{LOGCONFIG}),\ \mathsf{ran}(\mathsf{destinationBucket}, \mathsf{BUCKET}),\ \ldots\ \}$$

$$\mathit{Config} = \{\ \mathsf{BUCKET}(\mathit{ConfigS3Bucket}), \mathsf{bucketName}(\mathit{ConfigS3Bucket}, \mathit{"ConfigStore"}),\ \mathsf{loggingConfig}(\mathit{ConfigS3Bucket}, x),\ \mathsf{destinationBucket}(x, \mathit{ConfigS3Bucket}),\ \mathsf{logFilePrefix}(x, \mathit{"config\text{-}bucket\text{-}logs"})\ \}$$

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 refer the reader to the repository in [14] for the properties specification files.[2]

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*).[3] 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 [39] as the inference engine.

[2] https://tiny.cc/PropertiesSpecifications.
[3] 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

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 resources and 21 resource types. We implement 50 properties from the ScoutSuite collection [35] 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 16 GB 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 940 ms. 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 363 ms, classified in 2.1 s, and has a max average per-query time of 162 ms. 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 18 s.

5.1 Found Security Issues

Across all 15 deployments, we run $15 \times 50 = 750$ 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 they do not only 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, as is the dependency on external resources whose security posture cannot be assessed.

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,

```
"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).

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, *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 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.

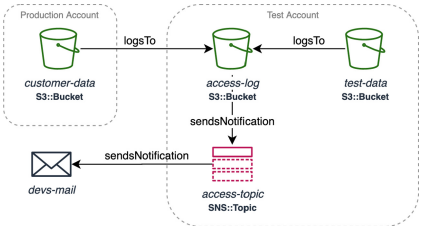

Fig. 2. Dataflow extracted from Fig. 1

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 [4,5,7,16]. Non-formal tools exist that recommend and run checks against *already deployed* resources [13,35], or scan IaC templates [10,11,38] for syntactical patterns violating security best practices. These checks overlap considerably and can be expressed in our framework as well. The disadvantages of such 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]. Simpler first-order logic formulas with operators to represent object-oriented interface relationships could be used to model IaC specifications. However, such an encoding would only partially

solve our problem, which is more complex because our overall goal is to do formal semantic analyses (e.g., dataflow and threat modeling). Semantic-based approaches, even DL-based, are being used to do conceptual modeling of security engineers' expertise with the provable and explainable inference capabilities of logics. As an example, we refer the reader to the OWASP *"Ontology-driven Threat Modeling"* project [31] that aims at the formalization of security-related knowledge in the context of different types of computer systems by means of description logic ontologies. In contrast to logic programming languages, such as Datalog, DLs inherently support functionality axioms and the existence of anonymous individuals within a domain that is assumed to be open. These are supported out-of-the-box without the need for an additional, more complex, axiomatization or encoding. In particular, we took advantage of DL's open-world assumption to implement, in our properties encoding, verification and falsification. Another 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 [25,33] have not been studied in the context of 3-valued models and 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 evaluate the security of cloud infrastructure pre-deployment. We encoded Amazon Web Services' Infrastructure as Code specifications and configurations into description logic models and verified the presence and absence of potential security issues. We showed how this approach enables deeper system-level analyses such as dataflow analysis. All results can be generalized to other existing IaC tools. While working on this project, we interacted with developers on two occasions. First, for the benchmark templates used in our experimental evaluation, we contacted the owners, told them about the misconfigurations, and discussed potential security implications. Second, within AWS, security engineers use a technique based on this paper for security reviews of AWS products before they are launched, helping developers fix real issues pre-deployment. In the process, we received valuable feedback that we used for improving precision and reducing the number of false-positive results. 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.

Acknowledgements. This research is supported by the ERC consolidator grant D-SynMA under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 772459) and by Amazon Web Services.

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., et al.: Reachability analysis for AWS-based networks. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11562, pp. 231–241. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25543-5_14
5. Backes, J., et al.: 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. In: Bouguettaya, A., Sheng, Q., Daniel, F. (eds.) Advanced Web Services, pp. 527–549. Springer, New York (2014). https://doi.org/10.1007/978-1-4614-7535-4_22
7. Bouchenak, S., Chockler, G.V., Chockler, H., Gheorghe, G., Santos, N., Shraer, A.: Verifying cloud services: present and future. Operating Syst. Rev. **47**(2), 6–19 (2013)
8. Bruns, G., Godefroid, P.: Model checking partial state spaces with 3-valued temporal logics. In: Halbwachs, N., Peled, D. (eds.) CAV 1999. LNCS, vol. 1633, pp. 274–287. Springer, Heidelberg (1999). https://doi.org/10.1007/3-540-48683-6_25
9. Bruns, G., Godefroid, P.: Model checking with multi-valued logics. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 281–293. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27836-8_26
10. The AWS CloudFormation Linter (2020). https://github.com/aws-cloudformation/cfn-python-lint. Accessed 15 Oct 2020
11. The CFnNag Linting Tool (2020). https://github.com/stelligent/cfn_nag. Accessed 15 Oct 2020
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/. Accessed 04 Aug 2020
14. CloudFORMAL: Prototype Implementation. http://github.com/claudiacauli/CloudFORMAL. Accessed 15 Oct 2020
15. Resource Specification (2020). https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cfn-resource-specification.html. Accessed 13 Aug 2020
16. Cook, B.: Formal reasoning about the security of Amazon web services. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 38–47. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_3
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. Reason. **53**(3), 245–269 (2014)
19. Google Deployment Manager. https://cloud.google.com/deployment-manager. Accessed 28 Jan 2021
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: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 170–174. Springer, Heidelberg (2006). https://doi.org/10.1007/11817963_18
22. Horridge, M., Bechhofer, S.: The OWL API: a Java API for OWL ontologies. Semant. 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., Isbell, C.L., Jr.: 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/. Accessed 28 Jan 2021
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. OWASP Ontology-driven Threat Modeling. https://github.com/OWASP/OdTM. Accessed 14 May 2021
32. 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/
33. Sattler, U., Vardi, M.Y.: The hybrid μ-calculus. In: Goré, R., Leitsch, A., Nipkow, T. (eds.) IJCAR 2001. LNCS, vol. 2083, pp. 76–91. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45744-5_7
34. Schmidt-Schauß, M., Smolka, G.: Attributive concept descriptions with complements. Artif. Intell. **48**(1), 1–26 (1991)
35. Multi-cloud Security Auditing Tool (2020). http://github.com/nccgroup/ScoutSuite. Accessed 4 Aug 2020
36. 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)
37. Terraform. https://www.terraform.io/. Accessed 28 Jan 2021
38. Static Analysis Security Scanner for Terraform (2020). https://tfsec.dev/. Accessed 10 May 2021
39. Tsarkov, D., Horrocks, I.: FaCT++ description logic reasoner: system description. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 292–297. Springer, Heidelberg (2006). https://doi.org/10.1007/11814771_26

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.