# Static Analysis of Serverless Applications with a Semantic Code Search Engine

**Giuseppe Raffa**

Supervisor: Dr. Jorge Blasco Alis

CDT in Cyber Security for the Everyday
Royal Holloway, University of London

Summer Project

August 2021

# Abstract

Over the last few years, the serverless computing paradigm has become increasingly popular. Thanks to its cost-effectiveness and the possibility of relying on a cloud provider to manage the underlying infrastructure, including its scalability, companies adopting this novel approach can fully focus on developing the business logic of their software products. However, with applications making use of configurable authentication schemes and sharing information in cloud-based repositories, among other pieces of functionality, it is crucial to understand what the security implications of embracing the serverless paradigm are.

The shared responsibility model clarifies that software developers have still to perform some security-related tasks. While this seems to be now a consolidated concept, researchers and industry practitioners have pointed out that the tools available to secure serverless applications should be further improved. Static analysis, in particular, can be potentially very useful, but, as highlighted in previous academic studies, it is generally not comprehensive, as it cannot include the provider-managed platform services.

For this reason, this work has explored a different approach to static analysis, which consists of employing a general-purpose semantic code search engine (CodeQL). This can be used to inspect at coding time both the implementation of a serverless application, i.e., the application code, and the definition of the resources needed to support its functionality, commonly referred to as infrastructure code. To achieve this, since CodeQL features a code-as-data approach, a library of 30 serverless-specific queries, 23 of which focused on infrastructure code, has been developed at the beginning of this project after identifying a set of relevant security vulnerabilities.

To automatically analyse a repository with the implemented queries, a new Python command-line tool, called Serverless Inspector (SI), has also been developed. The tool relies on the public API of LGTM, a CodeQL-compatible code analysis platform, and it is available on GitHub for future research work. In addition to validating the queries, the SI was used to scan a test set of 77 infrastructure code files and 38 application code files downloaded from the open-source Serverless Framework examples repository. Despite the limitations of the developed queries, which are detailed in the report, and the reduced size of the test set, the results provide interesting indications. The infrastructure code queries raised a warning in

75.3% of the samples, with potential vulnerabilities associated with functions triggered via HTTP events detected in 51 cases. In addition, functions returning not sanitized data were identified in five out of 38 samples, and were the cause of a total of 11 warnings. Both cases should therefore be considered in future research work.

Finally, as part of the assessment of the proposed approach, the query execution times were measured as well. Static analysis tools, in fact, aim at supporting the developer at coding time, and are, for this reason, expected to produce results within a short period of time. The conducted tests indicate that, in most cases, the average execution time of a query is less than one second per sample file. However, it is important to emphasize that the SI tool relies on a remote code analysis engine, which implies that the query execution times will also inevitably be affected by its load conditions and the quality of the network connection.

# Table of contents

# List of figures

# List of tables

# Nomenclature

**Acronyms / Abbreviations**

| | |
|---|---|
| API | Application Programming Interface |
| ARN | Amazon Resource Name |
| AWS | Amazon Web Services |
| CD | Continuous Delivery |
| CI | Continuous Integration |
| DAST | Dynamic Application Security Testing |
| DB | Database |
| FaaS | Function as a Service |
| GUI | Graphical User Interface |
| HTTP | Hypertext Transfer Protocol |
| IaC | Infrastructure as Code |
| IAM | Identity and Access Management |
| IDE | Integrated Development Environment |
| IT | Information Technology |
| JSON | JavaScript Object Notation |
| LGTM | Looks Good To Me |
| MFA | Multi-Factor Authentication |

| | |
|---|---|
| OS | Operating System |
| S3 | Simple Storage Service |
| SAST | Static Application Security Testing |
| SI | Serverless Inspector |
| SQL | Structured Query Language |
| URL | Uniform Resource Locator |
| YAML | YAML Ain’t Markup Language |

# Chapter 1

# Introduction

This chapter describes the motivation behind this project (Section 1.1) as well as its objectives (Section 1.2) and the adopted methodology (Section 1.3). Furthermore, an outline of all chapters is provided in Section 1.4.

## 1.1 Motivation

The serverless computing paradigm aims at reducing the overhead associated with maintenance and monitoring of traditional servers by adopting a stateless and event-driven model, which relies on platform services offered by a cloud provider [31]. By abstracting away almost all operational concerns, including infrastructure scalability and IT hardware maintenance, enterprises can cut their costs and focus on developing their products.

However, it is important to emphasize that relying on serverless environments, such as AWS [6], Azure [84] and Google Cloud Platform [60], does not mean that companies can outsource all their security-related tasks. Recent data breaches caused by inadequately secured cloud-based data repositories [56] and SQL injection attacks conducted with a voice-enabled device [104, 127] confirm that it is necessary to adopt a different model to secure applications and services implemented with the technology being discussed[1].

In addition to common cloud vulnerabilities, such as insecure APIs [101], it is crucial to consider that serverless applications routinely receive their inputs from a variety of sources and that code execution can be triggered by different kinds of events, such as a database update or a file upload [31]. Therefore, several recent academic studies [5, 52, 123, 109] have been focused on the implementation of new frameworks for information and data flow analysis. Despite their generally low overhead, all of them require the deployment and

[1]Additional examples of disclosed AWS-related vulnerabilities can be found at [105].

maintenance of additional code and, in one case [5], the use of an instrumented version of the original application.

While the above-mentioned frameworks are very helpful to improve the level of security of code running within a serverless environment, static analysis, which by definition allows testing software without executing it [62, Ch. 3], is routinely used by developers to assess an application at coding time. This kind of analysis is particularly challenging when serverless technology is used, because there is a high number of events and platform services to be considered and, in many cases, obtaining detailed information about them is not possible. However, the work by Obetz *et al.*, who first extended the concept of call graphs [91] and then formalized the notion of event [90], shows that static analysis is undoubtedly useful to audit serverless applications, despite the inevitable limitations.

Taking all this into consideration, this project explores an alternative way of statically analysing both code to be executed within a serverless environment and its configuration, which is referred to as *infrastructure code*. Differently from [91] though, the idea pursued in this work is to use the semantic code search engine CodeQL [55, 7, 54] rather than a fully-customized framework, which would be time-consuming to implement and, more importantly, not easily extensible[2]. It should be noted though that CodeQL, which relies on SQL-like queries, is a generic tool, so the most challenging part of this research was to identify ways of leveraging its features in the specific context of serverless applications.

Finally, another important motivation for this project is that a static analysis tool centred around a well-known code search engine can be more easily integrated into widely used Integrated Development Environments (IDEs). To further understand this aspect, it should be observed that modern Static Application Security Testing (SAST) tools are cloud-based and typically provide their findings via a web interface. Luo *et al.* [82][3], though, have ascertained that software engineers scan their code bases with a SAST tool up to three times more often if the results are made available within their IDE. While replacing a cloud-based tool with one that is executed locally would have many disadvantages, especially in terms of configuration, results' tracking and computing resources, the implementation of IDE plug-ins, such as the one presented in [82], has become an important trend, which SAST tool vendors are considering as well [50].

[2]The authors of [91] were contacted in May 2021 for a possible evaluation activity and additional questions. However, no further information was received prior to the end of this research activity.

[3]The paper [82] is due to be published at the end of August 2021. At the time of this writing, a copy can be downloaded from Amazon Science.

## 1.2 Objectives

The main objective of this project was to assess the suitability of a well-known semantic code search engine (CodeQL) for static analysis of serverless applications. The work by Obetz *et al.* [91] shows that this kind of analysis is valuable, but, differently from their approach, this project has explored the possibility of using a generic tool rather than a fully-customized framework. By doing so, both *infrastructure* and *application code* can be scanned at coding time, and the supported types of analyses can be extended more easily.

Considering that AWS is the most widely used environment for serverless applications and that these are very frequently deployed via the open-source Serverless Framework [121][4], the aforementioned objective was achieved as follows:

- In order to identify security issues present in infrastructure code for AWS applications deployed via the Serverless Framework, a set of dedicated CodeQL queries was developed.
- An additional set of CodeQL queries focused on application code was also implemented. Their scope was limited to AWS applications written in Python.
- To automatically test a given code base directly from the developer IDE, a Python command-line tool called *Serverless Inspector* was implemented. The tool relies on the public API of the LGTM tool [79, 68], which fully supports the chosen code search engine[5], and the CodeQL queries developed as part of this project.

## 1.3 Methodology

The adopted methodology can be summarized as follows:

- Review of the available literature on serverless computing with particular attention on the approaches considered by previous studies to statically and dynamically analyse serverless applications. Furthermore, the information provided by Amazon [6] and other cloud vendors [84, 60] played a key role in understanding how to conduct this research.

[4]It should be noted that *Serverless* will be used to refer to the framework introduced in Section 3.2, whereas the word *serverless* indicates the computing paradigm.

[5]Both CodeQL and LGTM were developed by the company Semmle [118]. Further information on these tools is provided in Section 3.3 and Section 3.4, respectively.

- Identification of the most common security vulnerabilities of infrastructure code [107]. Taking also into account the specific syntax used for Serverless Framework configuration files as well as some AWS-specific aspects, a total of 23 infrastructure code CodeQL queries were developed. The queries were manually tested with the LGTM [79] online interface against code samples prepared by the author.

- Identification of the most common security vulnerabilities of application code [107]. Due to the time constraints of this project, only AWS applications developed in Python were considered. Similarly to the infrastructure code queries, those focused on application code were manually tested with LGTM against application code samples collected by the author.

- Analysis of the LGTM public API [68] to implement the Serverless Inspector command-line tool, which enables the automated test of a code base with the CodeQL queries developed for this project. The tool was also used to scan a set of applications obtained from the Serverless Framework examples repository [120], and, thanks to its configuration options and open structure, can be employed to execute any CodeQL query.

## 1.4 Outline

This project report is organized as follows:

- Chapter 2: This chapter starts with the necessary theoretical background and ends with a summary of relevant academic work. The former, in particular, includes an overview of the serverless computing paradigm, which illustrates advantages, disadvantages, and security challenges, as well as an introduction to the wider concept of Infrastructure as Code.

- Chapter 3: After illustrating the challenges of testing infrastructure code and introducing the Serverless Framework, CodeQL, and the code analysis platform LGTM, the aim of this chapter is to explain how the selected code search engine was used to implement a collection of queries for infrastructure code.

- Chapter 4: This chapter is focused on the CodeQL queries specifically developed for static analysis of application code. After describing the adopted strategy, the chosen implementation, and the limitations of the implemented queries, the chapter ends with a summary that explains how to access the CodeQL code and its documentation.

- Chapter 5: The first objective of this chapter is to introduce the Serverless Inspector tool, which was implemented to automatically test a given code base with the developed CodeQL queries. Furthermore, the results obtained both with the tool self-test mode and with a repository based on the Serverless Framework examples in [120] are presented.

- Chapter 6: This chapter contains a summary of the achieved results, some recommendations, and suggestions for future research work.

# Chapter 2

# Background

The aim of this chapter is to provide the reader with all the required theoretical background. A summary of related academic work is included as well (Section 2.4).

## 2.1 Serverless computing paradigm

After a general overview (Section 2.1.1), the advantages and disadvantages of the serverless computing paradigm are introduced and explained in Section 2.1.2.

### 2.1.1 Overview

Over the last few years, many enterprises have adopted the *serverless* computing model, also referred to as *FaaS* (Function as a Service), which, as shown in Fig. 2.1, allows them to run application code by leveraging pieces of functionality made available by a cloud service provider. Therefore, as explained by Katzer [67][1], serverless is a misnomer, as the execution of the deployed applications still relies on servers, but their provisioning, maintenance, updates, scaling and capacity planning are delegated to another organization [36]. As further clarified in Section 2.1.2, this implies making use of a *pay-for-usage* billing model, which enables modern enterprises to focus on implementing business logic and shipping new features, rather than managing the supporting infrastructure.

Serverless computing technology is still in its early days and is constantly evolving. Interestingly though, the first experiments with it began in 2008 with the release of the Google App Engine [67]. Despite the modest success of that platform, over the last decade software engineers have recognized that to successfully develop and maintain complex applications, these would have to be implemented as distributed systems with loosely coupled

[1]The citations of [67] included in Section 2.1 refer to *Introduction to Serverless* (Pages: xxi-xxxiv).

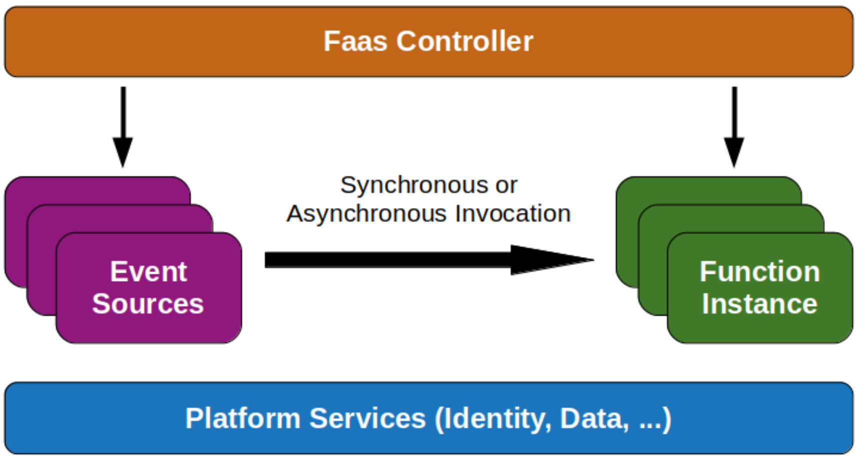

Faas Controller
Synchronous or Asynchronous Invocation
Event Sources
Function Instance
Platform Services (Identity, Data, ...)

Fig. 2.1 Key elements of a FaaS solution [36]

components, thus abandoning more traditional monolithic architectures. Consequently, as pointed out in [91], more attention was devoted to microservices-oriented design approaches, which prompted Amazon Web Services (AWS) [6] to release the first version of the Lambda platform in 2014 [20]. The latter, which has the largest service catalogue on the market, is widely considered the most popular serverless platform, though both Google with its Cloud Platform system [60] and Microsoft with Azure [84] have competitive offerings. While the serverless computing paradigm is more enterprise-oriented [91], open source solutions such as OpenWhisk [95], OpenFaaS [93], OpenLambda [94], and Knative [35] are available as well.

### 2.1.2 Advantages and disadvantages

To further understand the reasons behind the growing popularity of serverless platforms, it is important to emphasize that they allow for increased scalability and reliability [67]. By adopting this model, companies can prioritize implementation and delivery of application logic without planning for future capacity, though the limits imposed by the cloud provider as well as the interactions with non-serverless components of their systems will always have to be taken into account. In addition, it is crucial to observe that delegating scalability and maintenance-related tasks to another organization does not imply that security is outside the scope of enterprises relying on serverless platforms. As illustrated in Fig. 2.2, in fact, the *shared responsibility model* [67] establishes that while there are tasks, such as installation of OS security updates, only the cloud provider is responsible for, enterprises still have to test their code to identify vulnerabilities. Therefore, as further illustrated in Section 2.2, developing applications with this novel paradigm implies considering new security-related challenges.

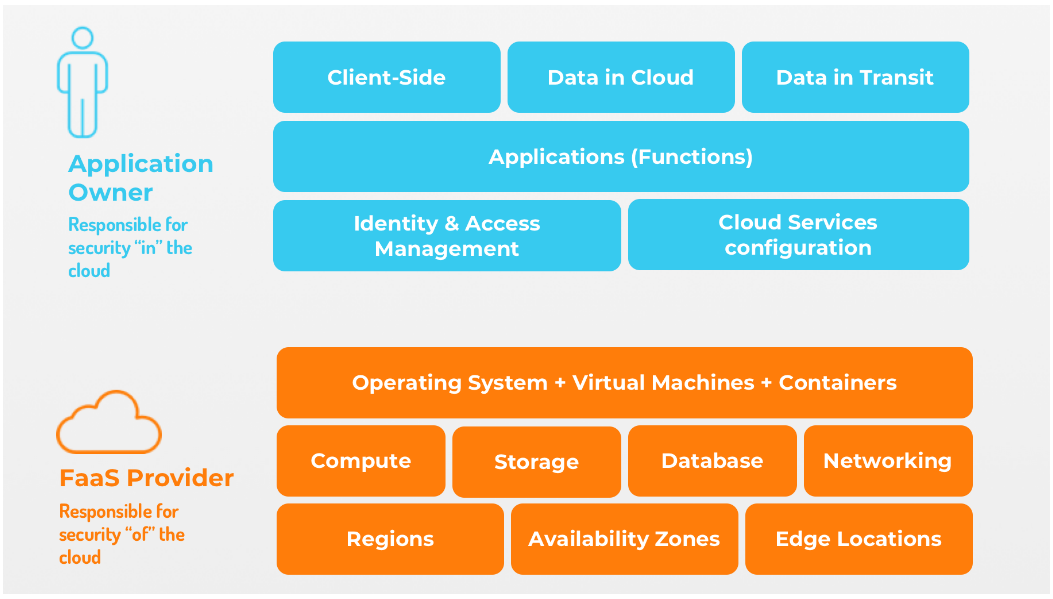

Application Owner
Responsible for security "in" the cloud
Client-Side
Data in Cloud
Data in Transit
Applications (Functions)
Identity & Access Management
Cloud Services configuration
FaaS Provider
Responsible for security "of" the cloud
Operating System + Virtual Machines + Containers
Compute
Storage
Database
Networking
Regions
Availability Zones
Edge Locations

Fig. 2.2 Shared responsibility model [107]

Despite the possibility of Denial of Wallet attacks, which are further discussed in Section 2.2.2, another attractive feature of serverless platforms is the billing model [67]. As the cost for running applications depends on how often they are executed, enterprises no longer have to pay for idle time, which in the past was avoided by simply switching off underutilized platforms, for instance during holiday seasons or at weekends. Thanks to serverless architectures, availability can be guaranteed in a cost-effective manner, because, if an unexpected request for a Internet-facing application arrives, for example, the cloud environment will spin up a resource to execute the corresponding function. The opposite case, i.e., a sudden increase in workload, can also be efficiently managed, as the service provider is in charge of scalability, and software-focused companies no longer have to bear the costs of a permanent platform team, who is normally responsible for capacity planning and related tasks.

Serverless platforms also allow increasing developer productivity [67]. Implementing applications suitable for this computing paradigm implies adopting a finely grained model of small functions triggered by events, such as an API request or the update of a database. While these architectural features can be an obstacle when migrating legacy software, developing applications by chaining together custom code and platform services with events simplifies the entire process and decreases the development time. A practical example is provided by the AWS Cognito service [13], which allows handling user accounts and can be used to automatically send a welcome email to a newly-registered user without having to include or re-implement the code necessary for it.

To provide a comprehensive overview of the serverless computing paradigm, it is also important to highlight its weaknesses. The first problem is known as *cold start* [67], which occurs when an application invokes a function, but the latter cannot be executed because there are no readily available resources. This inevitably leads to unresponsive applications and explains why several of them simply keep their functions *warm*, i.e. ready to be executed in the cloud, despite the inevitable increase in cost. As pointed out in [67, 137], solutions based on Content Delivery Networks (CDN) are being considered along with the *snapshotting*, whereby faster invocation is achieved by employing an image of a fully-booted function stored on disk. A study recently conducted by Ustiugov *et al.* [137] with their open-source framework vHive [138], shows that an improved methodology based on optimized memory management can reduce cold-start delays nearly four times on average.

Obetz *et al.* [91] also emphasize that serverless platforms impose strict limits on function execution times, which is one more reason for developers to adopt a microservice-oriented architecture. According to Katzer [67] though, it is possible the cloud providers will relax these constraints in the near future. AWS, in fact, increased the limits on Lambda from five to fifteen minutes in 2018. As a result, implementing complex applications by linking together several small functions in event-driven architectures seems the only viable option, though this means handling numerous software components, which can, in turn, cause problems during debugging, e.g., while tracing an user action throughout the entire system [67]. However, the availability of increasingly sophisticated monitoring services, such as AWS CloudWatch [12], can be greatly facilitate these tasks.

In summary, the advantages offered by serverless platforms in terms of scalability and cost reduction make this novel computing paradigm an attractive choice for applications comprising tasks that can be accomplished through small and independent steps, especially if their usage pattern is infrequent or unpredictable [67]. When responsiveness is an issue, the undesirable effects of a cold start can always be mitigated by employing a serverless platform only for background tasks. However, it should be observed that the computing paradigm considered in this work is not particularly suitable if computationally-intensive or long-running tasks that cannot be divided into smaller ones are required, and when the necessary functionality is not supported by cloud providers, being highly specialized [67].

## 2.2 Securing serverless applications

Securing serverless applications is undoubtedly a challenging task. Adopting the shared responsibility model (Section 2.1.2), in fact, does not imply that only the cloud provider is responsible for security. Development teams that intend to leverage serverless platforms still have to take into account generic principles, such as *least privilege* [59, Ch. 10], test their code to identify vulnerabilities and scrutinize all external software dependencies, as suggested in [67, Ch. 9]. Moreover, it is important to consider more serverless-specific aspects, which include, amongst others, cloud storage services management and the absence of traditional countermeasures, e.g., intrusion prevention and detection systems, as their detection logic has yet to be translated to support serverless environments [107]. Consequently, as emphasized by the OWASP Serverless Top 10 project [98, 97], migrating to a serverless architecture does not mean eliminating security risks, but rather considering different ones.

As explained in the analysis published in 2019 by the company PureSec [107], which is now part of Palo Alto Networks [99], serverless applications are, in fact, complex systems characterized by an increased attack surface, as their functions consume data from a wide range of event sources, such as HTTP APIs, cloud storage facilities, and IoT devices. While infrastructure providers offer services dedicated to authentication, authorization, and cryptographic key management [110, Ch. 6] that developers should systematically use, the general advice, as highlighted in the OWASP report [98] and by Katzer [67, Ch. 9], is that almost each serverless function should have a carefully-crafted set of permissions. This approach is time-consuming though, and requires not-as-yet available expertise, given that the computing paradigm being discussed is relatively recent.

From a security testing point of view, which is particularly relevant to this project, the PuresSec report [107] also explains that current automated scanning tools are not suitable for serverless applications. In addition, *dynamic application security testing* (DAST) tools are still being improved to stimulate the software under test with a variety of suitable inputs (i.e., including non-HTTP sources), and to properly monitor the interactions with back-end cloud services. It should also be observed that *static application security testing* (SAST) tools, which typically rely on data and control flow analysis, are also available, but the presence of numerous event triggers and cloud services make them inaccurate, as rules to identify sources and sinks in the inspected code are difficult to formulate [107].

To further understand the security challenges involved in the development of serverless applications, the identity and access management service, which is supported by every infrastructure provider, is further discussed in Section 2.2.1, whilst a summary of the most important serverless-related risks is presented in Section 2.2.2.

### 2.2.1 Identity and access management

All serverless providers offer an Identity and Access Management (IAM) service to manage user authentication and authorization. As explained in [18], this is a crucial security-related component, which is used by administrators to grant granular permissions, also known as *credential scoping* [110, Ch. 6], deploy multi-factor authentication (MFA) and manage federated identities, so that users can access cloud-based resources with their corporate network credentials or through external identity providers, e.g., Google.

Amazon, in particular, provides a highly-integrated IAM service, which can be accessed in several ways, for instance via the AWS command line tools or HTTPS APIs, the latter being the preferred method when issuing HTTPS requests directly to the service is necessary [18]. In essence, the AWS IAM can be described as the infrastructure that allows the management of *identities*, which include *users*, *groups*, and *roles*, through the application of one or more sets of rules known as *policies*. This means that every time an *entity*, i.e., an authenticated identity, makes a request to access or modify a resource, e.g., a database, the IAM double-checks the relevant policies prior to authorizing the request. It is important to emphasize that in this context an user does not have to represent an actual person, as it can be associated to an application, and that policies have to be written to explicitly authorize a request, otherwise the latter will be denied by default [18].

To further clarify the concept of policy, an AWS-specific example is provided in Fig. 2.3 [18]. In most cases, AWS policies are stored as JSON documents, which explains the syntax used in the example being discussed. As regards the permissions that the policy specifies, the user is allowed to perform all DynamoDB [15] actions (`dynamodb:*`) on the `Books` table in the `123456789012` account within the `us-east-2` region. It also worth noting that this is an example of *identity-based* policy, because it controls what actions the identity it is attached to can perform, in particular on which resources and under what conditions [18]. This type of policy can be further categorized in *managed*, which are stand-alone and can be associated to multiple users and roles, and *inline*, which are directly specified as part of an identity and, therefore, more difficult to reuse. Moreover, AWS supports *resource-based* policies, which control what actions a specified entity can perform on a given resource and under what conditions. Differently from identity-based policies, resource-based policies are always inline [18].

Finally, it is important to note that the above-described policy mechanism well supports the traditional role-based access control (RBAC) model [59, Ch. 9]. An IAM administrator, in fact, can create role-specific policies, where a role typically corresponds to a person's job function, and list all the resources that can be accessed as well as the permitted actions. However, managing policies developed to support RBAC can be time-consuming, as they

```
{
 "Version": "2012-10-17",
 "Statement": {
  "Effect": "Allow",
  "Action": "dynamodb:*",
  "Resource": "arn:aws:dynamodb:us-east-2:123456789012:table/Books"
 }
}
```

Fig. 2.3 AWS-specific example of IAM policy [18]

will have to be updated every time a new resource, e.g., an additional database, is added to the set that a given role should be able to access. Therefore, it is worth mentioning that AWS also supports an alternative authorization strategy, known as *attribute-based access control* (ABAC), which defines permissions based on attributes [18]. ABAC is implemented in AWS by specifying attributes as *tags*, which are added to IAM identities and resources. When an entity sends a request, the latter is authorized only if the ABAC tags match. The advantage of this access control scheme is that, if new resources are created with the necessary tags, then no policy updates are needed, which is particularly advantageous in large, multi-project enterprise environments [18].

### 2.2.2 Critical risks of serverless applications

Having introduced the most important factors that affect the security of serverless applications, the purpose of this section is to illustrate the associated risks. As part of this project, the author has compared the list presented in the 2018 OWASP Serverless Top 10 report [98] with the one published by PureSec in 2019 [107]. Despite the partial overlap, while the first is undoubtedly an excellent starting point, the second appears to be more focused on the peculiarities of the current serverless platforms. Therefore, the PureSec classification, which is arranged in order of criticality in Table 2.1, will be the main reference point for this project.

It is interesting to observe that, despite the availability of dedicated platform services, broken authentication is considered a major risk both by PureSec [107] and in the OWASP report [98]. Serverless applications are typically comprised of a large number of functions and cloud services that require authentication, which often has to be implemented by using different schemes due to the specific requirements of the integrated components. While adopting dedicated platform services, e.g., AWS Cognito [13], rather than deploying customized authentication schemes, allows mitigating this risk, testing serverless authentication is a challenging tasks, as brute force tools available for web applications are unsuitable [107].

Table 2.1 Critical risks for serverless applications according to PureSec [107]

| **Identifier** | **Critical Risk** |
|---|---|
| SAS-1 | Function Event Data Injection |
| SAS-2 | Broken Authentication |
| SAS-3 | Insecure Serverless Deployment Configuration |
| SAS-4 | Over-Privileged Function Permissions & Roles |
| SAS-5 | Inadequate Function Monitoring and Logging |
| SAS-6 | Insecure 3rd Party Dependencies |
| SAS-7 | Insecure Application Secrets Storage |
| SAS-8 | Denial of Service & Financial Resource Exhaustion |
| SAS-9 | Serverless Function Execution Flow Manipulation |
| SAS-10 | Improper Exception Handling and Verbose Error Messages |

Another significant risk is posed by insecure serverless deployment configuration [107], which is a generalization of the sensitive data exposure issue highlighted in [98]. The security posture of serverless applications is affected by customizations and configuration settings, especially those that have an impact on cloud-based storage. To facilitate scalability, in fact, the latter is an ubiquitous feature of the applications being discussed, as their functions are *stateless* and need to store information permanently. While vendor-specific best practices exist, the absence of widely understood industry standards and tools for deployment of secure applications are important contributing factors [107].

Unsurprisingly, over-privileged function permissions and roles [107], which are reported in the OWASP report under the broken access control category [98], are a major risk as well. As previously mentioned, microservices-oriented design approaches inevitably lead to a high number of functions, which far too often are managed by development teams by using one or very few security roles. Permission policies should instead be implemented on a per-function basis, i.e, in a granular manner, and by adopting the IAM-related best practices provided by each vendor. However, as illustrated in Section 2.2.1, IAM is a complex service [107], which implies that inexperienced developers and administrators are very likely to make mistakes.

As regards more serverless-specific risks, the first examples are denial of service and financial resource exhaustion [107], with the latter also known as *Denial of Wallet* (DoW) [98]. Since serverless providers define default limits that affect, for instance, per-function and per-account concurrent execution, if an attacker succeeds in triggering resources by exploiting a code vulnerability, then, when one of the limits is reached, the application will no longer be available for other users. Similarly, malicious actors can also attempt to execute functions for a prolonged period of time, thus inflicting a financial loss on the target organization [107].

Finally, it should be observed that functions execution flow manipulation attacks [107], which aim at altering the business logic of an application [98], can have a considerable impact. Even though such manipulations are not specific to serverless applications, these can be severely affected because they consist of microservices chained together by events. Consequently, if an attacker replicates an event, e.g. by sending an email to a platform service, they can trigger the execution of functions, thus bypassing access control or causing a DoW attack. Designing the software without making any assumptions about the invocation flow, which implies using fine-grained permissions and access control, is considered the best approach to mitigate this risk [107].

## 2.3 Infrastructure as code

To further illustrate the foundations of the serverless computing paradigm, it is necessary to introduce the wider concept of *Infrastructure as Code* (IaC), which, as pointed out by Morris [86], has very recently become a cornerstone of modern software engineering. The advent of cloud-based technologies and infrastructure has created a different and very challenging environment for IT architects, system administrators and software developers. Old approaches to managing deliverables, which nowadays include infrastructure as well, are unsuitable for the *Cloud Age* [86, Ch. 1], as they have been developed when static pre-cloud systems were mainstream. By contrast, modern IT architectures are highly dynamic and require constant changes to integrate new applications and features, provide better scalability and, even more importantly in the context of this work, guarantee adequate levels of security.

The concept of IaC was conceived to meet these challenges and can be defined as an approach to infrastructure automation based on practices from software development [86, Ch. 1]. From a more practical perspective, this definition has three important implications. The first is that everything should be defined as code, thus achieving reusability, consistency and transparency. Using automation to continuously test and deploy new components is another core practice of IaC, because it crucially supports the implementation of systems that are reliable and cost-effective. Furthermore, adopting IaC implies developing code bases that consist of small, thoroughly tested, and well-documented pieces, because large systems with tightly coupled components are normally hard to change and easy to break.

All the core practices described by Morris [86] are important to understand how serverless applications are developed and secured. The above-mentioned principle that everything should be defined as code, in particular, is the most relevant to this project. In essence, this means that infrastructure definitions, which include both the necessary elements and their configurations [86, Ch. 4], should be managed via text files containing, as further clarified in the remainder of this section, either source code or structured configuration parameters. Therefore, IaC files will include several different pieces of information, e.g., packages to be installed, user accounts, and details of delivery pipeline software. The main advantage of this approach is that infrastructure code can be managed via a Version Control System (VCS), as if it were a standard application. However, it should also be observed that some closed-box tools do not fully support IaC [86, Ch. 4], as they cannot be fully configured by using an external configuration file. In addition, security implications have to be taken into account, as IaC files should not contain passwords or cryptographic keys [86, Ch. 7].

Since a significant amount of information is needed to manage a modern IT system, it is important to understand how the concept of defining everything as code can be put into practice. As illustrated in [86, Ch. 4], there are three different approaches, which rely on *declarative*, *imperative* and *domain-specific* languages. When a declarative approach is used, for instance through YAML [139] (Fig. 2.4), the focus is on what has to be achieved, but not how. The chief implication of this is that declarative code is seen by many practitioners as a configuration file, given that it does not include any logic. By contrast, as shown in Fig. 2.5, an imperative IaC file typically contains parameters as well as statements that implement checks, loops or similar programming constructs. Finally, domain-specific languages, which are normally implemented by using a declarative model [86, Ch. 4], are a way of providing developers and system administrators with a tool that help them to write and understand code. While several DSL-based frameworks, such as Puppet [57] and Terraform [34], exist, others, for instance Pulumi [106], allow defining infrastructure by using existing general-purpose programming languages.

```yaml
virtual_machine:
  name: my_application_server
  source_image: 'base_linux'
  cpu: 2
  ram: 2GB
  network: private_network_segment
  provision:
    provisioner: servermaker
    role: tomcat_server
```

Fig. 2.4 Example of declarative code for IaC [86, Ch. 4]

The IT industry has not yet identified a standardized approach to implementing IaC, which implies that infrastructure code bases currently rely on a mixture of tools and languages. As explained by Morris [86, Ch. 4], in fact, declarative code, supported or not by a DSL, is more

```
this_country = getArgument("country")
data_centers = CloudApi.find_data_centers(country: this_country)
full_ip_range = 10.2.0.0/16

vlan_number = 0
for $DATA_CENTER in data_centers {
  vlan = CloudApi.vlan.apply(
    name: "public_vlan_${DATA_CENTER.name}"
    data_center: $DATA_CENTER.id
    ip_range: Networking.subrange(
        full_ip_range,
        data_centers.howmany,
        data_centers.howmany++
    )
  )
}
```

Fig. 2.5 Example of imperative code for IaC [86, Ch. 4]

suitable to define the desired state of a system, particularly when there is not much variation in the expected outcomes, whilst imperative languages are a better choice to build libraries and abstraction layers.

## 2.4 Related work

The complexity of the serverless computing paradigm has prompted several different research activities. Among them, it is worth mentioning the work by Obetz *et al.* [91], which have recently presented a novel approach to adequately extending the traditional concept of *call graph* to a serverless environment. The latter, in fact, has specific features, which can be overlooked if a simple directed graph, where functions are represented as nodes and the program flow is described through edges, is adopted.

To support information flow and security analysis as well as automatic generation of documentation and code maintenance tasks, the work [91] introduces the idea of *extended service call graph*, which aims to provide a comprehensive description of the entire structure of a serverless application by incorporating nodes and edges associated with external platform services made available through the cloud-based execution environment. From an information security point of view, this approach is particularly advantageous to identify leakage of sensitive information to untrusted sinks.

However, obtaining an augmented graph of this kind implies introducing a new class of nodes representing platform services as well as external libraries, which are very frequently difficult to model. The tests executed by Obetz *et al.* [91] on the AWS Serverless Application Repository [28] show that the approach and their tool have yet to be further developed, as the authors focused their attention only on JavaScript applications implemented with AWS Lambda and a subset of typical platform services). Additional information about the method described in [91] is available in the follow-up paper [90] where the authors focus on the operational semantics needed to further develop their approach.

The work of Datta *et al.* [52], who have focused their attention on a method that aims at improving both static and dynamic analysis-based approaches, is also particularly relevant to this project. The authors of [52] have developed the framework Valve, which relies on the open source serverless computing platform OpenFaaS [93] to implement a *workflow-centric* approach to monitoring and authorizing information flows. Since serverless applications routinely make use of third-party components, controlling where data goes is undoubtedly challenging. In addition, several serverless security solutions offer only function-level protections, which implies that information flow and inter-function security violations are neglected [52].

Thanks to the adoption of *taint labels*, Valve attempts to solve this problem by providing runtime tracing and enforcement of policy-based information flows. This is achieved by a service that includes an *agent* and a *controller*. The agent is focused on monitoring API calls, which are recorded as taints, and disk access, with the latter being a necessary step to prevent information leakage. By contrast, the Valve controller module processes the taints generated by the agent to audit the information flow of the application, and orchestrates security policy enforcement, if configured to do so. It is worth noting that developer-specified authorization rules can also be incorporated, though these can introduce vulnerabilities if not properly checked. From a more practical perspective, it is also important to highlight that the Valve framework [52] has been conceived to protect against common classes of attacks with minimum runtime overhead and without requiring code modification, the latter being a key difference with the work of Alpernas *et al.* [5].

Sirone Jegan *et al.* [123] have recently improved the above-mentioned Valve tool [52] by proposing an extensible security framework for serverless applications called *SecLambda*. In addition to tracking the control flow of an application, its key feature is that it relies both on global application-related information as well as local function states. These, which are not processed by Valve, are used in SecLambda as an input to a *guard* module, which allows running additional code that performs security-specific tasks, e.g. credential protection and rate limiting of serverless functions. To achieve this goal, such functions are executed within a modified execution environment built on top of the gVisor container runtime [61], which aims at reducing the attack surface of a serverless function, e.g. by controlling the system calls [123]. Similarly to Valve, SecLambda also includes a *controller* module, which is responsible for coordinating and collecting function states from the guards, and for providing the guards with information about the global state of the application, thus facilitating the execution of sophisticated security tasks. Differently from Valve [52], SecLambda is extensible because the function states can be processed by user-defined code that can be added on a need basis.

Finally, it is worth mentioning that Trach *et al.* [135] have recently developed the secure serverless platform *Clemmys* by adopting a significantly different approach in comparison to [5, 52, 123]. In particular, the paper [135] describes a system that hardens a FaaS platform by relying on the hardware-based shielded execution technology Intel SGX [64], which has been the basis for more generic cloud security research as well [32]. The authors of [135] have implemented a low-throughput system that guarantees confidentiality and integrity of users' functions, which are executed within the SGX *enclave*, i.e., a secure region of memory address space. However, the deployment of Clemmys is not straightforward, as it requires SGX-specific optimizations and at least one additional software component for cryptographic key management.

# Chapter 3

# CodeQL queries for infrastructure code

After explaining the challenges of testing infrastructure code (Section 3.1) and introducing the open-source Serverless Framework (Section 3.2), the code analysis engine CodeQL (Section 3.3), and the code analysis platform LGTM (Section 3.4), the main objective of this chapter is to illustrate the CodeQL queries developed to test infrastructure code (Section 3.5).

## 3.1 The challenges of testing infrastructure code

As illustrated in Section 2.3, IaC can be described as a software engineering methodology that aims at managing infrastructure in an efficient manner. To this end, repeatable processes that rely on automated testing should be created. However, as explained in [86, Ch. 8], this is a challenging task because testing infrastructure code implies considering multiple aspects, such as scalability, functionality, compliance with standards, performance and, of course, security.

Moreover, when infrastructure code includes only hard-coded values, e.g., a range of IP addresses, testing it does not add much value to the quality of the code base, as a failed test in all probability points either to a problem with the deployment tool or to obsolete test cases, which do not contain up-to-date values. On the other hand, when IaC code includes some logic and dependencies, then the best way of conducting tests is to use a real instance of infrastructure, which is normally expensive and time-consuming. This explains the existence of tools such as LocalStack [80], Moto [87], and Azurite [85], which allow the developer to *mock* the APIs of cloud vendors by providing a functional *local* cloud stack, thus extending the traditional software testing concept of *test doubles* mentioned by Turnquist [136, Ch. 4] and in [108, Ch. 3].

Within the more specific context of the serverless paradigm, though, using simpler static analysis techniques is a valid option, as it is possible to develop tests that, albeit executed without running the software, can identify issues early in the development cycle. According to Morris [86, Ch. 9], this can be achieved with the following two approaches[1]:

- *Offline Static Code Analysis*[2]. The infrastructure code is checked without connecting to the infrastructure platform. This type of analysis can assess compliance with coding standards, and identify both errors and potential security issues. As shown in Table 3.1, there exist tools that perform these checks on infrastructure code, though not all of them are security-focused.
- *Static Code Analysis with API*. In this case, connections to the cloud platform are made in order to check for conflicts. For instance, if the infrastructure code relies on a specific database, connecting to the cloud platform allows verifying the existence of that database.

Offline static code analysis is also the approach adopted in this work. However, since none of the tools listed in Table 3.1 is focused on the Serverless Framework, a code search engine (CodeQL) and a compatible code analysis platform (LGTM) were used instead.

Table 3.1 Existing tools for offline static analysis of infrastructure code

| Tool | Environment | Security Features | Reference |
|---|---|---|---|
| cfn-lint | AWS CloudFormation | ✗ | [129] |
| cfn_nag | AWS CloudFormation | ✓ | [130] |
| tflint | Terraform | ✗ | [133] |
| tfsec | Terraform | ✓ | [134] |

## 3.2 The Serverless Framework

The Serverless Framework [121] is an open-source project that aims at facilitating and standardizing the deployment of serverless applications. As explained in [67, Ch. 5], in order to create the cloud dependencies necessary for the execution of a serverless application, the framework makes use of configuration files specifying a collection of resources, which can be referred to as infrastructure and include, amongst others, event handlers and external plug-ins.

[1]Morris [86, Ch. 9] also mentions syntax checking as possible approach, but this has not been considered in this project since it is not security-relevant.

[2]Morris [86, Ch. 9] refers to this approach also as *linting*, which is a term routinely used to indicate basic static code analysis [102]. However, as shown in the examples discussed in [108, Ch. 9], linting tools are generally not security-focused, so the author will not use this term to avoid ambiguity.

The chief advantage of the Serverless Framework is that it promotes the IaC paradigm. Adopting this tool, in fact, allows deploying a serverless application without directly interacting with the serverless environment-specific configuration dashboards, for instance AWS CloudFormation [30][3], which are typically implemented with a Graphical User Interface (GUI). However, it is important to be aware that the Serverless Framework ultimately interacts with these dashboards, which means that, in the case of AWS, the configuration file will be mapped into a CloudFormation template [30]. Consequently, as underlined in [67, Ch. 5], if the serverless environment relies on a dashboard or a template system which has limitations, these will also inevitably affect an application deployed via the Serverless Framework.

As regards the configuration file syntax, YAML was chosen by the framework developers because it is generally considered more readable than JSON and, in addition, it supports comments. It should also be noted that the Serverless Framework supports AWS, Microsoft Azure and Google Cloud Platform. Furthermore, it can be extended with JavaScript or TypeScript plug-ins, which, according to Katzer [67, Ch. 5], represent a very efficient way of enhancing the default functionality and sharing solutions among users.

## 3.3 The code analysis engine CodeQL

Originally developed by the company Semmle [118][4], CodeQL [115] is a semantic code analysis engine designed to query source code with a *code as data* strategy [114]. As further explained in the academic publications dedicated to this tool [55, 7, 54][5], this approach facilitates the identification of logical variants of a known software vulnerability, which is also referred to as *variant analysis*. Therefore, CodeQL can perform more comprehensive analyses in comparison with a linting tool, and, as confirmed by a number of case studies, e.g. [117, 113], support static analysis of software written for a variety of applications.

Even though describing in detail how the code search engine under discussion works is outside the scope of this report, it should be observed that the main concept behind CodeQL is that source code can be mapped into a database, which can then be queried with a specialized language. To achieve this, the structures present in the code to be analysed are processed by a dedicated piece of software called the *extractor* [71]. Unsurprisingly, as shown in the example [76], this process has to take into account syntax and semantic-related aspects that are language-specific. Consequently, each extractor works only with a designated

[3]For future research and development activities, it is worth noting that the reference [30] is part of a collection of AWS White Papers.

[4]Semmle joined GitHub in 2019 [112] and CodeQL is now available in a GitHub repository [41].

[5]Additional research papers that illustrate practical applications of CodeQL can be found at [44].

language, and, given their complexity, CodeQL currently supports only C, C++, C#, Go, Java, JavaScript, and Python.

Despite a well-structured documentation [39] and the availability of numerous online resources [49, 40, 51], learning a query language with complex features such as CodeQL is unavoidably time-consuming. For this reason, at the beginning of this research, the author considered using the open-source tool Semgrep [111], which, by contrast, allows implementing queries and rules in the same language as the source code to be analysed. While this appears to be an attractive feature, the project report by Nicolalde [88, Ch. 4] shows that Semgrep has several limitations and, therefore, was not chosen for this research activity.

## 3.4 The code analysis platform LGTM

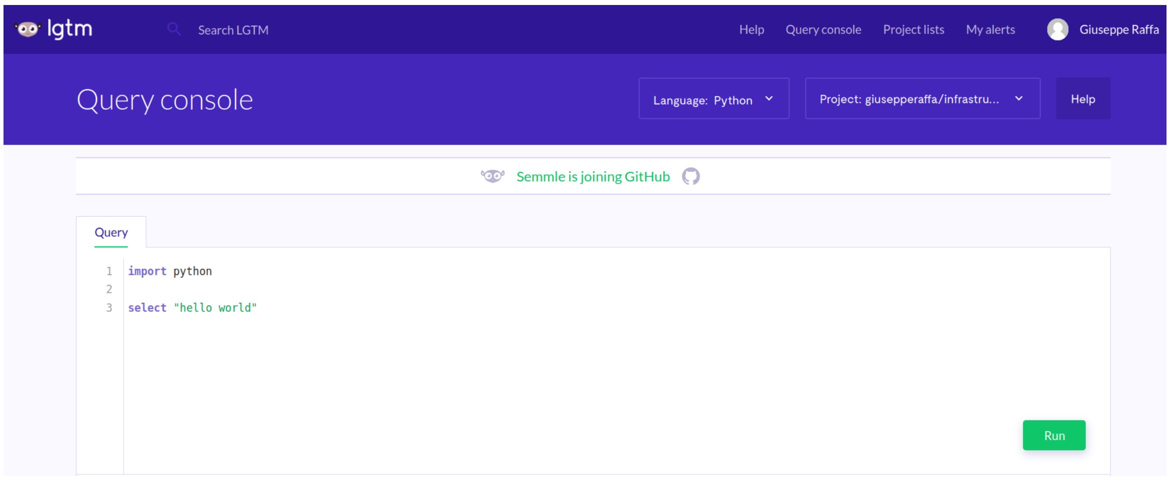
CodeQL has a companion tool called LGTM (Looks Good To Me) [79, 116], which can be described as a code analysis platform [118, 72]. Conceived to provide a powerful interface for the execution of CodeQL queries, LGTM includes an online query console [77] and a set of other tools that allow implementing a CI/CD pipeline.

The online query console, which is visible in Fig. 3.1, played a key role in this project, as it was used to prototype and interactively test all the developed CodeQL queries. As detailed in [78], using such interface is straightforward. Prior to executing a query specified within the dedicated text box, the user only needs to select the language of interest (e.g., Python) and a target project available within a supported version control system (e.g., GitHub)[6]. It should be observed though that there is a delay of up to 24 hours between the last commit in the repository to be scanned and when the corresponding code is available for analysis in LGTM [73]. This issue can be solved by integrating the latter in a CI/CD pipeline, but it has not been attempted as part of this project due to the experimental nature of the developed queries.

## 3.5 Infrastructure code CodeQL queries

The purpose of this section is to present the CodeQL queries developed for static analysis of infrastructure code. After introducing the adopted strategy (Section 3.5.1) and some implementation details (Section 3.5.2), their limitations are documented in Section 3.5.3. Finally, a summary that comprises a classification of the queries is provided in Section 3.5.4.

[6]Only logged-in users can submit queries. LGTM does not require a dedicated account though, as the version control system credentials can be used.



```
import python

select "hello world"
```

Fig. 3.1 LGTM query console

### 3.5.1 Strategy

After reviewing the generic list of vulnerabilities published by PureSec [107], which highlighted the importance of checking deployment and permissions configuration along with third party dependencies (Table 2.1), given the scope of this project (Section 1.2), the second step consisted of analysing relevant parts of the AWS documentation. In particular, the IAM-related security best practices recommended by Amazon [19] and details on access control [10, 11] were considered in addition to information about S3 buckets[7]. The latter included generic guidelines [25] as well as examples of user policies [26] and bucket policies [24].

Furthermore, as shown in Fig. 3.2, which summarizes the development process, the ten Serverless Framework YAML configuration examples [120] listed in Table 3.2 were studied along with infrastructure code samples in [67, Ch. 5]. As a result, three main sets of queries dedicated to checking IAM configurations, two of which focused on DynamoDB tables [15] and S3 buckets, were developed. In addition, queries targeting HTTP events, external plug-ins, and unprotected storage of environment variables [125][8] were also implemented. A complete list and a classification of the query files, which includes links to access the publicly available CodeQL code, is provided in Section 3.5.4.

Finally, as further detailed in Chapter 5, it should be observed that a total of 25 infrastructure code samples were also prepared by the author to test the developed queries.

[7]S3 (Simple Storage Service) buckets are a popular cloud storage solution offered by AWS [27].

[8]Relevant information about secure storage is available in the section Manage secrets in secure storage of [125].

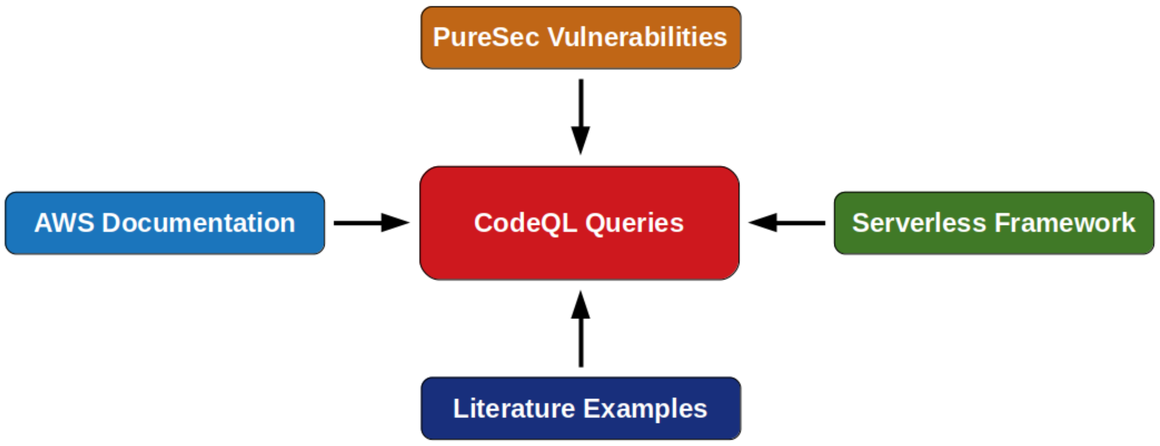


Fig. 3.2 Strategy adopted to develop the infrastructure code CodeQL queries

Table 3.2 YAML configuration files used to develop the infrastructure code CodeQL queries

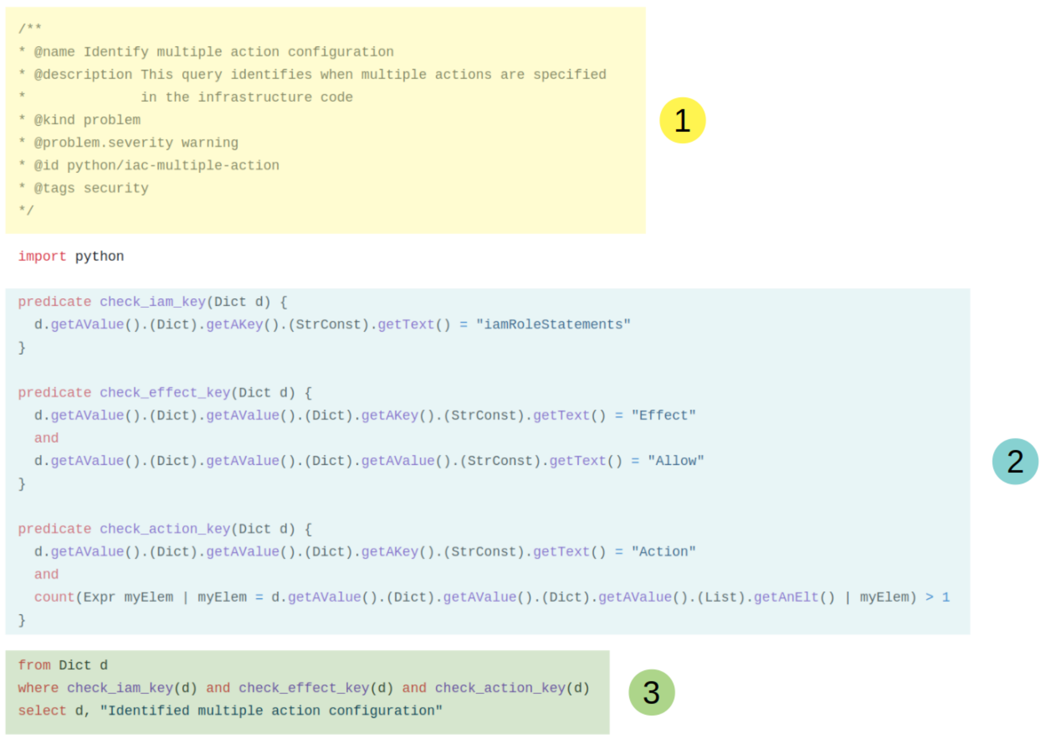
| Number | Application Name | GitHub Link |
|---|---|---|
| 1 | aws-rust-simple-http-endpoint | serverless.yml |
| 2 | aws-ruby-sqs-with-dynamodb | serverless.yml |
| 3 | aws-ruby-sinatra-dynamodb-api | serverless.yml |
| 4 | aws-ruby-simple-http-endpoint | serverless.yml |
| 5 | aws-ruby-line-bot | serverless.yml |
| 6 | aws-ruby-cron-with-dynamodb | serverless.yml |
| 7 | aws-python-telegram-bot | serverless.yml |
| 8 | aws-python-sqs-worker | serverless.yml |
| 9 | aws-python-simple-http-endpoint | serverless.yml |
| 10 | aws-python-scheduled-cron | serverless.yml |

### 3.5.2 Implementation

As regards the implementation of the queries, since CodeQL does not support YAML (Section 3.3), it was necessary to identify a way of transforming the configuration files contents into a data structure that the chosen code analysis engine could process. Considering the properties of Python dictionaries, which are among the core data types of the language and can be described as associative arrays, they were selected to map the YAML files to be statically analysed. Thanks to the parser included in the Python module `yaml`[9], the Serverless Inspector tool, which is presented in Chapter 5, can be used to automatically remap a set of YAML files into Python files containing the corresponding dictionaries.

[9]The configuration used for this project included Python 3.6.9 and the `yaml` module version 3.12.

The developed queries share the same architecture, which, as shown in the example of Fig. 3.3, features a SQL-like `select` clause preceded by one or more *predicates*. These allow checking whether a condition is true and can be included in a query as function calls. As detailed in [43], there are two types of predicates in CodeQL, depending on whether they return an explicit result, e.g., a string. Both types have been used in this project to facilitate the implementation of the queries.



```
/**
 * @name Identify multiple action configuration
 * @description This query identifies when multiple actions are specified
 *              in the infrastructure code
 * @kind problem
 * @problem.severity warning
 * @id python/iac-multiple-action
 * @tags security
 */

import python

predicate check_iam_key(Dict d) {
  d.getAValue().(Dict).getAKey().(StrConst).getText() = "iamRoleStatements"
}

predicate check_effect_key(Dict d) {
  d.getAValue().(Dict).getAValue().(Dict).getAKey().(StrConst).getText() = "Effect"
  and
  d.getAValue().(Dict).getAValue().(Dict).getAValue().(StrConst).getText() = "Allow"
}

predicate check_action_key(Dict d) {
  d.getAValue().(Dict).getAValue().(Dict).getAKey().(StrConst).getText() = "Action"
  and
  count(Expr myElem | myElem = d.getAValue().(Dict).getAValue().(Dict).getAValue().(List).getAnElt() | myElem) > 1
}

from Dict d
where check_iam_key(d) and check_effect_key(d) and check_action_key(d)
select d, "Identified multiple action configuration"
```

Fig. 3.3 Structure of the CodeQL queries, which include ❶ metadata, ❷ predicates and ❸ a `select` clause

CodeQL includes classes that enable inspecting Python core structures, such as dictionaries and lists. Since YAML files were mapped into Python dictionaries, the class `Dict` [37], which provides methods such as `getAKey()` and `getAValue()`, was particularly important. The implementation of the query conditions, i.e., the above-mentioned predicates, also relied on information retrieved by the AWS documentation, especially regarding the *actions* defined for DynamoDB tables [16] and S3 buckets [23], which indicate what an user is allowed to do on a cloud resource.

Finally, it should be observed that the analysis of the examples downloaded from Serverless Framework repository was particular important to identify an additional syntax supported by this open-source tool to specify IAM-related information. The latter can be specified by using the identifier `iamRoleStatements` or, alternatively, the three *nested* identifiers `iam`, role, and statements[10]. Depending on which syntax is used, the data structure corresponding to the YAML file is different. Consequently, some of the queries for infrastructure code had to be implemented to specifically support the nested syntax, and their file names, which are reported in Section 3.5.4, end with the string *nested*.

### 3.5.3 Limitations

Taking into account the experimental nature of this project and its time constraints, the developed infrastructure code queries have some limitations, which are documented in this section to support future research work.

#### Unrestricted Principal

None of the queries checks the configuration of the infrastructure code entry Principal, for which examples are available at [11, 24]. In particular, the AWS JSON files included in [24] show that the Principal entry can be unrestricted, i.e., set up to "*". As emphasized in [124], this configuration is insecure because it allows unauthorized access. In other words, it can lead to a service that will be publicly accessible unless an additional condition is specified to restrict access.

#### HTTP referer

A query dedicated to the HTTP referer issue described in [24] should be part of future versions of the library. Developers, in fact, have the option of protecting data stored in the cloud, e.g., an AWS S3 bucket, by granting access only to requests originated from specified websites. While occasionally useful, this mechanism should not be used to prevent unauthorized parties from accessing confidential information. When the referer is publicly known, in fact, HTTP requests containing it can be crafted by an attacker.

#### Multi-Factor Authentication

None of the developed queries inspects the Multi-Factor Authentication (MFA) set-up. As shown in [24, 22], the latter relies on Boolean flags, such as MultiFactorAuthAge and MultiFactorAuthPresent, so a query could detect if these have been set up to false by mistake.

[10]An example of nested IAM syntax can be found in this YAML file, which is included in one of the GitHub repositories described in Chapter 5 and listed in Table 5.1.

#### Commands embedded in the YAML file

During the analysis of the YAML files included in the aforementioned Serverless Framework repository, the author has identified some cases where the `Resource` is specified by using a command, for instance `!GetAtt`[11]. When this occurs, the Python YAML parser raises an exception, as information provided in this form cannot be processed. As a result, no CodeQL query currently included in the library can analyse configuration files that make use of this syntax.

#### Resource specification syntax

As shown in the AWS documentation [24], the entry `Resource` of the infrastructure code can identify a list of Amazon Resource Names (ARNs). However, at present, the implemented queries cannot process multiple resources, and they should be further developed to take into account additional variants of the basic ARN syntax explained by Katzer [67, Ch. 5] and in [8].

Furthermore, as illustrated in one the examples included in [67, Ch. 5][12], AWS supports an additional syntax that relies on a `resource` entry specified in the YAML file outside the IAM-related part of the infrastructure code. This syntax is not currently supported by the developed CodeQL queries.

#### Variety of data structures

The YAML syntax is generally very flexible. For instance, albeit not always mandatory, the YAML entries are sometimes preceded by a dash character "-". It should be observed that this has important implications for the data structure created by the Python YAML parser. The latter, in fact, when a dash is used, returns Python lists containing dictionaries instead of mapping the YAML file entries into dictionary keys[13]. Consequently, it is difficult to implement generic CodeQL queries that take into account all possible data structures.

### 3.5.4 Summary

The purpose of this section is to categorize the 23 infrastructure code CodeQL queries that were implemented as part of this project. It should be noted that the GitHub links

[11]An example is provided by this application, which is among those listed in Table 3.2.
[12]This particular example can be found in the *Resources* section of [67, Ch. 5].
[13]An example is provided by this YAML file, which is included in one of the GitHub repositories described in Chapter 5 and listed in Table 5.1.

included in the Tables 3.3, 3.4, 3.5, and 3.6 allow accessing the CodeQL code as well as its documentation, which is reported in the metadata section.

### Generic IAM-related queries

These four queries aim at identifying potentially insecure settings of the YAML file entries `Resource` and `Action` regardless of the particular cloud resource, e.g., a NoSQL database, they refer to. The GitHub links as well as their classification in terms of PureSec vulnerabilities (Table 2.1) are reported in Table 3.3.

Table 3.3 Generic IAM-related CodeQL queries

| GitHub Link | PureSec Risk |
|---|---|
| query_iac_multiple_action_nested.ql | SAS-3 / SAS-4 |
| query_iac_multiple_action.ql | SAS-3 / SAS-4 |
| query_iac_multiple_resources_nested.ql | SAS-3 / SAS-4 |
| query_iac_multiple_resources.ql | SAS-3 / SAS-4 |

### DynamoDB IAM-related queries

These eight queries, which are listed in Table 3.4, provide a DynamoDB-focused and, therefore, more specialized version of the generic IAM-related queries previously described. DynamoDB-specific information was taken into account to identify potential vulnerabilities, such as the presence of multiple actions that allow deleting a database table or part of it.

Table 3.4 DynamoDB IAM-related CodeQL queries

| GitHub Link | PureSec Risk |
|---|---|
| query_iac_dynamodb_any_action_nested.ql | SAS-3 / SAS-4 |
| query_iac_dynamodb_any_action.ql | SAS-3 / SAS-4 |
| query_iac_dynamodb_multiple_delete_like_action_nested.ql | SAS-3 / SAS-4 |
| query_iac_dynamodb_multiple_delete_like_action.ql | SAS-3 / SAS-4 |
| query_iac_dynamodb_multiple_read_like_action_nested.ql | SAS-3 / SAS-4 |
| query_iac_dynamodb_multiple_read_like_action.ql | SAS-3 / SAS-4 |
| query_iac_dynamodb_multiple_write_like_action_nested.ql | SAS-3 / SAS-4 |
| query_iac_dynamodb_multiple_write_like_action.ql | SAS-3 / SAS-4 |

### S3 IAM-related queries

This collection of eight queries, which can be accessed by following the GitHub links included in Table 3.5, was developed with the same approach as those focused on DynamoDB, but by using S3-specific information instead.

Table 3.5 S3 IAM-related CodeQL queries

| **GitHub Link** | **PureSec Risk** |
|---|---|
| query_iac_s3_any_action_nested.ql | SAS-3 / SAS-4 |
| query_iac_s3_any_action.ql | SAS-3 / SAS-4 |
| query_iac_s3_multiple_delete_like_action_nested.ql | SAS-3 / SAS-4 |
| query_iac_s3_multiple_delete_like_action.ql | SAS-3 / SAS-4 |
| query_iac_s3_multiple_read_like_action_nested.ql | SAS-3 / SAS-4 |
| query_iac_s3_multiple_read_like_action.ql | SAS-3 / SAS-4 |
| query_iac_s3_multiple_write_like_action_nested.ql | SAS-3 / SAS-4 |
| query_iac_s3_multiple_write_like_action.ql | SAS-3 / SAS-4 |

### Other queries

As shown in Table 3.6, the last group includes the three remaining queries, which were designed to identify different issues, such as HTTP events, external plug-ins, and unprotected storage of environment variables.

Table 3.6 Other infrastructure code CodeQL queries

| **GitHub Link** | **PureSec Risk** |
|---|---|
| query_iac_external_plugin.ql | SAS-6 |
| query_iac_functions_triggered_via_http.ql | SAS-1 |
| query_iac_unprotected_environment_information.ql | SAS-7 |

# Chapter 4

# CodeQL queries for application code

The CodeQL queries specifically developed for static analysis of application code are the focus of this chapter. After introducing the chosen strategy (Section 4.1) and detailing how this was translated into a suitable implementation (Section 4.2), the limitations of the implemented queries are discussed (Section 4.3). Finally, the chapter ends by explaining how to access the developed CodeQL code and its documentation (Section 4.4).

## 4.1 Strategy

This section is devoted to explaining the two-phase strategy employed to implement a set of CodeQL queries focused on serverless application code. Given the time constraints of this project, only the most serious vulnerability according to PureSec [107], i.e., function event data injection[1] was considered. Moreover, the scope of this research activity was limited to applications implemented in Python, which is one of the languages supported by CodeQL.

During the first phase, it was necessary to identify real samples of vulnerable code, which are introduced in Section 4.1.1. Initially, these were important to gain a better understanding of the problem at hand, and they were also subsequently used as test cases.

The second phase prior to the implementation of the queries consisted of studying the taint analysis-related features of CodeQL, which are summarised in Section 4.1.2. Differently from the previously described infrastructure code queries (Chapter 3), where the objective was to inspect a data structure, taint analysis played a key role in identifying injection-related vulnerabilities, as it marks inputs from untrusted *sources* as tainted and then detects if security-critical code, i.e., a *sink*, receives one of such inputs [59, Ch. 10][2].

[1]For simplicity, the author will refer to this vulnerability as injection.

[2]According to Gollmann [59, Ch. 10], there exist two types of taint analysis, i.e., *static* and *dynamic*. Only the former is of interest for this project, given its scope.

### 4.1.1 Code samples preparation

In order to prepare the code samples, several resources were consulted by the author. Among them, it is worth mentioning the PureSec list of vulnerabilities [107], which includes an example of injection, and the OWASP Serverless Top 10 Report [98], which contains two relevant attack scenarios[3], though only one of them is implemented in Python.

In addition, the four examples provided by Segal in [96] were very useful to obtain code samples showing both OS commands injection and the problems caused by lack of sanitization of data included in function-triggering events. The work by Podjarny *et al.* [103, Ch. 3] was also used as a reference[4], as it contains a section specifically dedicated to injection flaws.

In total, 13 code samples were prepared, which are listed in Table 4.1 along with the corresponding GitHub link[5] and the used sources. Moreover, it should be observed that the samples have been classified in terms of level of required customization. While eight of them were used in this project without any changes (i.e., no customization), two required some code modifications (i.e., a partial customization), and three were implemented by the author after studying the concepts illustrated in the relevant sources.

The set of fully customized samples, in particular, includes two focused on HTTP requests, i.e., `sample_python_http_get_params.py` and `sample_python_http_post_data.py`. For simplicity, the chosen approach consisted of preparing one sample dedicated to HTTP Get and another sample focused on HTTP Post, which were implemented by taking into account the features of the Python module `requests` [92, 89, 4]. Both samples include HTTP requests with event-originated unsanitized parameters, which, as suggested in [107], is an important case both in traditional web applications and in serverless functions. Among future enhancements of the work presented in this report, there could therefore be an additional sample with a HTTP request sent to a server after including unsanitized inputs from another HTTP request.

It is also noteworthy that the implementation of SQL-focused samples was considered at the beginning of this research. However, neither the concepts illustrated by Daly [65] nor the Amazon Aurora-specific examples provided by Mendonca [83] allowed the author to prepare generic and relevant code samples. Identifying a unique way of accessing a SQL database in a AWS Python application, in fact, appeared to be unfeasible, as this can be achieved by using several different libraries and DB connectors. For this

[3]Both the example attack scenarios of interest can be found in Section A1 of the document [98].

[4]The report [103] is also available on the HubSpot's Website.

[5]All the code samples are included in one of the GitHub repositories described in Chapter 5 and listed in Table 5.1.

reason, since injection affects both SQL and NoSQL databases [107], only the sample `sample_python_dynamodb_scanfilter_injection.py`, which replicates a specific case of DynamoDB injection [1], was implemented for demonstration purposes. As shown in the examples available in the AWS documentation [17], however, there are multiple ways of calling the `scan` method on a DynamoDB table, which implies that, similarly to a SQL database, developing a query that covers all possible code variants is in all probability unattainable.

Table 4.1 Prepared code samples including injection vulnerabilities (NC=No Customization, PC=Partial Customization, FC=Full Customization)

| **GitHub Link** | **Sources** | **NC** | **PC** | **FC** |
|---|---|---|---|---|
| sample_python_dynamodb_scanfilter_injection.py | [1, 119] | ✗ | ✗ | ✓ |
| sample_python_eval_function.py | [96, 98] | ✗ | ✓ | ✗ |
| sample_python_event_data_returned_not_sanitized.py | [96] | ✓ | ✗ | ✗ |
| sample_python_exec_function.py | [96, 98] | ✗ | ✓ | ✗ |
| sample_python_http_get_params.py | [107, 48] | ✗ | ✗ | ✓ |
| sample_python_http_post_data.py | [107, 48] | ✗ | ✗ | ✓ |
| sample_python_os_system_a.py | [103] | ✓ | ✗ | ✗ |
| sample_python_os_system_b.py | [96] | ✓ | ✗ | ✗ |
| sample_python_os_system_c.py | [96] | ✓ | ✗ | ✗ |
| sample_python_serverless_goat_example.py | [122] | ✓ | ✗ | ✗ |
| sample_python_subprocess_call.py | [98] | ✓ | ✗ | ✗ |
| sample_python_subprocess_check_output.py | [107] | ✓ | ✗ | ✗ |
| sample_python_subprocess_getoutput.py | [96] | ✓ | ✗ | ✗ |

### 4.1.2 Taint analysis with CodeQL

Thanks to a set of dedicated resources, CodeQL supports data flow and taint-tracking analyses, which are conducted by using *path* queries [70, 38, 42]. Taint-tracking can be considered an extended form of data flow analysis, because it detects code statements where data values are not necessarily preserved, but a potentially insecure object is still propagated. Depending upon the scope of taint-tracking, which dictates how much code is actually inspected, it is useful to distinguish between *global* and *local* taint analysis, which are both possible with CodeQL.

Among the resources provided by the latter, it is worth mentioning the Python-specific library `Concepts`, which includes classes such as `CodeExecution`, `FileSystemAccess`, `SqlExecution`, and `SystemCommandExecution` that can be used to identify injection flaws [47]. Moreover, the collection of CodeQL examples shown in Table 4.2, which employ an additional set of modules and classes[6], was an important starting point for the author to understand how to implement queries that could put into practice the strategy detailed in Section 4.1.

Table 4.2 Analysed injection flaws-related CodeQL examples

| **Example Number** | **GitHub Link** |
|---|---|
| 1 | Code injection |
| 2 | Deserializing untrusted input |
| 3 | Incomplete URL substring sanitization |
| 4 | Information exposure through an exception |
| 5 | SQL query built from user-controlled sources |
| 6 | URL redirection from remote source |
| 7 | Uncontrolled command line |
| 8 | Uncontrolled data used in path expression |

## 4.2 Implementation

After studying the relevant CodeQL resources (Section 4.1.2), a local taint analysis-based approach was adopted. Considering that serverless applications are typically implemented with a microservices-oriented and event-based architecture, the application code files normally contain a collection of small functions or *handlers* that are executed either synchronously or asynchronously (Fig. 2.1). A global taint analysis-based approach would therefore be significantly more complicated, as it would imply statically analysing, at least to some extent, serverless provider-managed code, which is not available to the end user. Moreover, local taint analyses feature a lower execution time, which encourages the developer to run them at coding time.

As far as the CodeQL resources used are concerned, the author has initially conducted experiments with the Code Injection example reported in Table 4.2. However, it was observed that the provided code was not able to detect injection flaws as expected. To the best of the best of the author's knowledge, this was probably due to the fact that the configuration class included in the example had to be extended (i.e., customized). A further review of the Python-specific CodeQL documentation [47, 45, 46] then highlighted that the local taint analyses of interest could also be implemented by using the modules `ApiGraphs` and `TaintTracking`. These, in fact, allow executing taint analysis on a function input arguments

[6]The CodeQL Search Page allows accessing the documentation of specific modules or classes very efficiently.

(*sources*), which contain event-related information, and identifying where such inputs could cause a security vulnerability (*sinks*).

The example provided in Fig. 4.1 shows that the architecture shared by all the developed queries. The are no significant structural changes in comparison with the infrastructure code case (Fig. 3.3), but it is worth noting that the taint-tracking resources have to be explicitly imported, and that predicates were used only in three out of seven application code queries (Section 4.4).

```
/**
* @name Identify calls to built-in functions eval() and exec() that can cause injection flaws
* @description This query identifies calls to built-in functions eval() and exec()
*              without input sanitization
* @kind problem
* @problem.severity warning
* @id python/application-eval-exec-function-calls
* @tags security
*/

import python
import semmle.python.dataflow.new.TaintTracking
import semmle.python.ApiGraphs

predicate check_builtin_function(API::Node myNode){
  myNode.toString().matches("%eval%") or myNode.toString().matches("%exec%")
}

from API::Node myFunction, DataFlow::ParameterNode p
where myFunction = API::builtin(_) and check_builtin_function(myFunction)
and TaintTracking::localTaint(p, myFunction.getACall().getArg(0))
select myFunction.getACall(), p.asExpr()
```

Fig. 4.1 Implemented application code query with taint-tracking-specific resources

## 4.3 Limitations

As mentioned in Section 4.1, the scope of the CodeQL queries discussed in this chapter was limited to a specific type of vulnerability (i.e., injection) and to AWS applications implemented in Python. However, it is also important to point out that there are additional limitations, which are documented in this section to provide a reference for future research.

### 4.3.1 Code variants

As mentioned in Section 4.1.1, the existence of numerous libraries and software components that allow establishing a connection with a cloud-based database implies that developing queries capable of covering all possible code variants is in practice unachievable. Moreover, it is important to highlight that one of the main reasons why Python is widely used is the flexibility of its syntax. The latter, however, can lead to an even higher number of code variants. This issue should always be considered while using the developed CodeQL queries, especially in the presence of false negatives (i.e., undetected security vulnerabilities).

### 4.3.2 Event inspection

One of the key elements of the implement strategy adopted for the application code queries is that these rely on the local taint analysis capabilities of CodeQL rather than inspecting the structure and contents of a given event. The latter approach is possible because serverless applications include functions where events are passed as input arguments. Furthermore, inspecting events in an attempt to recognize specific types and features would undoubtedly be helpful to implement more selective queries.

However, the structure of an event input depends on its type [21] and, in addition, the number of events that should be considered is high [65]. As a result, an event-focused approach would be difficult to implement and maintain. By contrast, although less selective and incapable of extracting event-specific information, the queries created as part of this project are more generic and less cumbersome to implement.

### 4.3.3 Unprocessed function inputs

The local taint analysis-based approach adopted in this project is ineffective when the function input arguments, which in most examples are called `event` and `context`, are unprocessed (i.e., never used). To further illustrate this point, let us consider the sample `sample_python_subprocess_call.py`. As explained in [98], this code sample is vulnerable because `subprocess.call` executes a command including a portion that can be indirectly modified by a malicious user by pre-uploading a file. Unfortunately, this cannot be detected by taint-tracking `event` and `context`. More generally, it should be observed that the function being considered only needs read and download permissions, but granting exclusively these does not prevent the attack, as long as uploading arbitrary files on the cloud-based repository is allowed. Therefore, to detect this kind of vulnerability, a static analyser should be capable of identifying anything that is somehow user-modifiable, which is not possible with the chosen approach.

### 4.3.4 Python format strings

The experiments conducted as part of this project show that CodeQL does not behave as expected when taint-tracking variables updated via Python string formatting, i.e., when there is, as shown in Fig. 4.2, a format-specifying string followed by a `format` method call[7]. The code sample files where the problem was detected are reported below along with a more detailed description:

- `sample_python_os_system_a.py`. It has been observed that the `os.system` call in this sample file is not found by the query `query_python_os_system.ql`. The `os.system` command relies on a Python format string.
- `sample_python_subprocess_check_output.py`. During the validation of the query `query_python_subprocess_functions.ql`, it has been ascertained that the latter does not detect the `subprocess.check_output` call contained in this sample file. The first input argument of this call includes a variable updated via string formatting.
- `sample_python_subprocess_check_output.py`. During the validation of the query `query_python_subprocess_shell_true.ql`, it has been observed that the latter is not able to find the `subprocess.check_output` call contained in this sample file. The first input argument of this call includes a variable updated via string formatting.

To address this issue, the author has analysed the CodeQL examples pertaining to Python format strings listed in Table 4.3. Moreover, experiments with the CodeQL modules `AdvancedFormattingCall` and `AdvancedFormatString` were conducted as well[8]. Given that no attempt was successful and that, as mentioned in [74], the LGTM parser was reviewed in 2020 to solve another format string related problem, it cannot be ruled out that this is a genuine CodeQL bug. However, the tests executed as part of this project should be further reviewed before drawing this conclusion.

Table 4.3 Examples of CodeQL queries focused on Python format strings

| **Example Number** | **Documentation Link** |
|---|---|
| 1 | py-str-format-missing-argument |
| 2 | py-str-format-missing-named-argument |
| 3 | py-str-format-mixed-fields |
| 4 | py-str-format-surplus-argument |
| 5 | py-str-format-surplus-named-argument |

[7]As illustrated in [66], strings can be formatted in Python also by using the "%" operator and the f-string syntax. However, to the best of the author's knowledge, the issue described here affects only strings formatted with the built-in method `format`.

[8]These modules are not available by default in LGTM, but they can both be imported with either `import AdvancedFormatting` or `import Expressions.Formatting.AdvancedFormatting`.

```python
def index(event, context):
    for record in event['Records']:
        sns_message = json.loads(record['Sns']['Message'])
        raw_email = sns_message['content']
        parser = email.message_from_string(raw_email)
        if parser.is_multipart():
            for email_msg in parser.get_payload():
                file_name = email_msg.get_filename()
                if not file_name:
                    continue
                if not file_name.endswith('.pdf'):
                    continue

                # export pdf attachment to /tmp
                pdf_file_path = os.path.join('/tmp', file_name)
                with open(pdf_file_path, "wb") as pdf_file:
                    pdf_file.write(email_msg.get_payload(decode=True))

                # extract text from pdf file
                cmd = "/var/task/lib/pdftotext {} -".format(pdf_file_path)

                pdf_content = subprocess.check_output(cmd, shell=True)
```

Fig. 4.2 Example of Python format string with the built-in `format` method

## 4.4 Summary

The aim of this section is to explain how to access the seven application code CodeQL queries that were developed as part of this research. Consistently with the approach adopted for the infrastructure code queries, Table 4.4 includes all the links to the relevant public GitHub repository. As previously mentioned (Section 3.5.4), the reader is encouraged to follow these links to access the queries' documentation, which can be found in the metadata section of the files.

Table 4.4 Application code CodeQL queries

| GitHub Link | PureSec Risk |
|---|---|
| query_python_dynamodb_scanfilter_injection.ql | SAS-1 |
| query_python_eval_exec_functions.ql | SAS-1 |
| query_python_event_data_returned_not_sanitized.ql | SAS-1 |
| query_python_os_system.ql | SAS-1 |
| query_python_requests_functions.ql | SAS-1 |
| query_python_subprocess_functions.ql | SAS-1 |
| query_python_subprocess_shell_true.ql | SAS-1 |

Finally, in line with the strategy devised for this part of the project, it should be noted that all the implemented application code queries have the same classification in terms of PureSec risk (Table 2.1).

# Chapter 5

# The Serverless Inspector tool

The primary objective of this chapter is to illustrate the architecture (Section 5.1) and the implementation (Section 5.2) of the Serverless Inspector (SI) tool, which allows running automated tests with the CodeQL queries introduced in Chapter 3 and Chapter 4. The results obtained in self-test mode, which provide a baseline for the validation of the implemented tool, are summarized in Section 5.3.

Finally, the developed queries were tested against a repository including a selection of Serverless Framework examples [120]. The results achieved in this case are presented in Section 5.4.

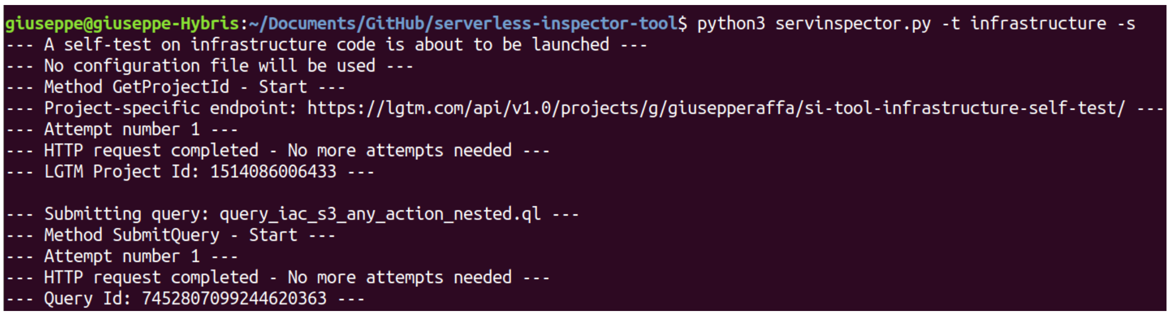
## 5.1 Architecture

The SI tool employs the infrastructure and application code queries developed as part of this project to enable automated static analysis of serverless applications. To facilitate its integration into an IDE, it has been developed in Python with a command-line interface, which implies that it can alternatively be executed, as shown in Fig. 5.1, by using an independent terminal window[1].

From an architectural point of view, as illustrated in Fig. 5.2, the tool provides an interface towards the LGTM public API [68], which allows testing the code included in a given repository with user-specified CodeQL queries. To achieve this, the SI includes a main script (`servinspector.py`), which implements the test execution logic, e.g., the cycle that allows submitting multiple queries, and a Python module (`lgtmreslib.py`) developed by the author to access the above-mentioned API. Separating this functionality from the

[1]The SI tool was developed and tested with a computer running Ubuntu Linux 18.04 LTS and Python 3.6.9. No tests were conducted with Windows or Mac OS.

main script, in fact, facilitates code maintenance and reuse, and it allows creating a generic framework that can be used to run *any* CodeQL query on a target repository.



```
giuseppe@giuseppe-Hybris:~/Documents/GitHub/serverless-inspector-tool$ python3 servinspector.py -t infrastructure -s
--- A self-test on infrastructure code is about to be launched ---
--- No configuration file will be used ---
--- Method GetProjectId - Start ---
--- Project-specific endpoint: https://lgtm.com/api/v1.0/projects/g/giusepperaffa/si-tool-infrastructure-self-test/ ---
--- Attempt number 1 ---
--- HTTP request completed - No more attempts needed ---
--- LGTM Project Id: 1514086006433 ---

--- Submitting query: query_iac_s3_any_action_nested.ql ---
--- Method SubmitQuery - Start ---
--- Attempt number 1 ---
--- HTTP request completed - No more attempts needed ---
--- Query Id: 7452807099244620363 ---
```

Fig. 5.1 Linux terminal showing a Serverless Inspector execution example

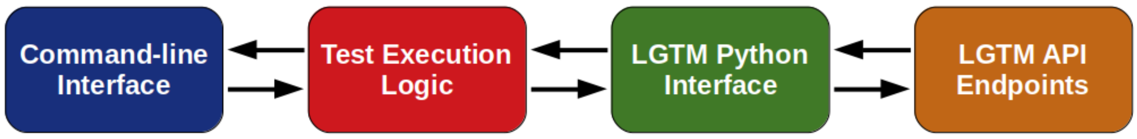


Fig. 5.2 Serverless Inspector high-level architecture

As detailed in Table 5.1, the tool code has been made available in the public GitHub repository `serverless-inspector-tool`. In addition, the SI makes use of four additional public repositories, also listed in Table 5.1, to support the following three operating modes:

- *Conversion Mode*. This is the mode to be selected in order to convert YAML files into Python dictionaries defined as literals. The dictionaries are stored in `.py` files. As explained in Section 3.5.2, such conversion is necessary prior to statically analysing infrastructure code, because CodeQL cannot directly process YAML files.

- *Self-test Mode*. When the self-test mode is chosen, the CodeQL queries, which are all stored within the folder `codeql`[2] of the `serverless-inspector-tool` repository, are used to test the files included in either `si-tool-application-self-test` or `si-tool-infrastructure-self-test`. As shown in Fig. 5.1, the self-test repository actually analysed has to be selected with the command-line option `-t`. Note that no external tool configuration file is used in self-test mode.

- *Test Mode*. If the test mode is selected, the CodeQL queries are executed to test one of the repositories included in the configuration file specified via the option `-f`.

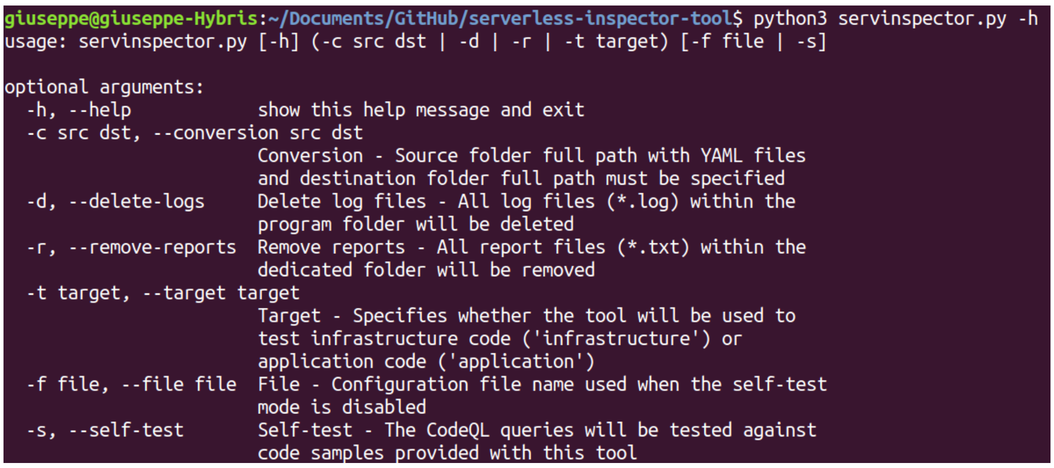
[2]The folders `application` and `infrastructure` within `codeql` help categorize the queries.

Similarly to the self-test mode, it is the option `-t` that determines the repository actually scanned. A YAML configuration file template, which currently includes the repositories tested as part of this project, can be found within the folder `config` of `serverless-inspector-tool`. Note that the target repositories are specified within the configuration file via their LGTM URLs and not their GitHub URLs [75].

A summary of all the command-line options, which can be displayed in the used terminal window by using the `-h` or `--help` option, is provided in Fig. 5.3. Finally, it is worth mentioning that, to improve the user interface, the main script includes a mechanism that detects when incompatible options have been specified.

Table 5.1 Serverless Inspector tool GitHub repositories (TO=Tool, IC=Infrastructure Code, AC=Application Code, ST=Self-test Mode, TM=Test Mode)

| GitHub Link | TO | IC | AC | ST | TM |
|---|---|---|---|---|---|
| serverless-inspector-tool | ✓ | ✗ | ✗ | ✗ | ✗ |
| si-tool-application-self-test | ✗ | ✗ | ✓ | ✓ | ✗ |
| si-tool-application-test | ✗ | ✗ | ✓ | ✗ | ✓ |
| si-tool-infrastructure-self-test | ✗ | ✓ | ✗ | ✓ | ✗ |
| si-tool-infrastructure-test | ✗ | ✓ | ✗ | ✗ | ✓ |


```
giuseppe@giuseppe-Hybris:~/Documents/GitHub/serverless-inspector-tool$ python3 servinspector.py -h
usage: servinspector.py [-h] (-c src dst | -d | -r | -t target) [-f file | -s]

optional arguments:
  -h, --help            show this help message and exit
  -c src dst, --conversion src dst
                        Conversion - Source folder full path with YAML files
                        and destination folder full path must be specified
  -d, --delete-logs     Delete log files - All log files (*.log) within the
                        program folder will be deleted
  -r, --remove-reports  Remove reports - All report files (*.txt) within the
                        dedicated folder will be removed
  -t target, --target target
                        Target - Specifies whether the tool will be used to
                        test infrastructure code ('infrastructure') or
                        application code ('application')
  -f file, --file file  File - Configuration file name used when the self-test
                        mode is disabled
  -s, --self-test       Self-test - The CodeQL queries will be tested against
                        code samples provided with this tool
```

Fig. 5.3 Serverless Inspector help

## 5.2 Implementation

After illustrating the architecture and the configuration options of the SI tool (Section 5.1), this section aims at succinctly describing its Python implementation. Before providing additional details, it is important to mention that the author has opted for an object-oriented approach and, consequently, used some of the language features that support this programming paradigm. For brevity, however, this report will assume that the reader is already familiar with such features, which are detailed, for example, in [3, Ch. 4] and in [33, Ch. 7].

The aforementioned main script `servinspector.py` contains a set of auxiliary functions. Among them, the most important one is `ProcessProgramInputs`, which relies on the Python standard library module `argparse` [53] to return an object encapsulating the user-specified configuration options. Such object is then passed to the instance of the class `TestLauncherCls`, which manages the logic of the test execution, e.g., by identifying the target repository, through its method `TestLauncherLogic`. The actual execution of the CodeQL queries, however, takes place thanks to the method `SubmitQueries`. The latter, after creating an instance of the interface class `LGTMAPIInterfaceCls`, included in the module `lgtmreslib`, cyclically submits all the relevant queries and checks for consistency what the LGTM interface returns, raising an exception when necessary[3].

The class `TestLauncherCls` also generates a log file[4] containing the execution time of each query and a test report, which will be stored within the folder `reports` of the `serverless-inspector-tool` repository. Note that the folder is not visible in GitHub as it will always contain files ignored by the version control system, but it will be created on the local version of the repository[5]. Both log and report files can be deleted by the user with the options `-d` and `-r` shown in Fig. 5.3.

As regards the above-mentioned `LGTMAPIInterfaceCls` class, it contains a collections of methods that have been implemented according to the LGTM API documentation on *query jobs* [69]. More precisely, after initializing the relevant API endpoints within the constructor, an instance of this class allows obtaining the LGTM project ID (method `GetProjectId`), which identifies the target repository, submitting a query (method `SubmitQuery`), checking the status of its execution (method `GetQueryJobStatus`), and, finally, downloading the test results (method `GetResultsSummary` and method `GetQueryJobResults`). It is important to highlight that each methods attempts to send an HTTP request multiple times up to a pre-defined maximum, in order to deal with network glitches or the temporary unavailability of the API endpoints. This mechanism is a noteworthy feature, because it greatly facilitates the use of the interface in client code, such as the script `servinspector.py` of the SI tool.

[3]This is achieved with the Python command `assert`.
[4]This functionality has been implemented by using the Python standard library module `logging` [2].
[5]More precisely, reports files will be generated within subfolders identified via a timestamp.

Finally, using the LGTM API is possible only if an access token is provided[6]. This is handled by the method `ExtractAccessTokenFromFile` of the class `TestLauncherCls`, which expects to find in the folder `config` a text file named `lgtm_access_token.txt` with this information reported in its first line.

## 5.3 Self-test mode results

The purpose of this section is to present the results obtained by launching the SI tool in self-test mode. These tests have been conducted to automatically validate the implemented infrastructure (Section 5.3.1) and application code queries (Section 5.3.2), thus providing a baseline for future research and development activities.

### 5.3.1 Infrastructure code queries

The validation results of the infrastructure code queries are summarized in Fig. 5.4, which shows the number of code samples where each query has raised a warning[7]. In addition to this aggregate result, Appendix A includes a set of tables that list the files where each query has detected a potential vulnerability.

As expected, the majority of the queries has raised a warning in only one or two files. By contrast, `query_iac_multiple_action` and `query_iac_multiple_action_nested` exhibit an aggregate result equal to eight and nine, respectively. This can be attributed to the fact that they are not selective, which implies that, differently from the more specialized ones, e.g., `query_iac_dynamodb_multiple_delete_like_action`, they attempt to identify issues that are not cloud resource-specific.

It should also be observed that the results presented in Fig. 5.4 do not take into account possible multiple occurrences, i.e., files where the queries detected more than one issue were counted only once. The analysis of the test reports, however, shows that only the query `query_iac_functions_triggered_via_http.ql` found two occurrences in one given file (`serverless_iac_multiple_functions_http_multiple_plugins.py`).

The query execution times were also logged, and are visible in Fig. 5.5, which includes figures that refer to all the 25 files in the repository `si-tool-infrastructure-self-test` (Table 5.1). The average of the execution times is 2.6 s, i.e., 0.1 s per sample file, if the value logged for `query_iac_unprotected_environment_information.ql` is ignored. In the

[6]Detailed information about LGTM access tokens can be found in the Managing access tokens section of the API documentation.

[7]The query names were abbreviated both in Fig. 5.4 and in Fig. 5.5 by removing the initial string `query_iac`.

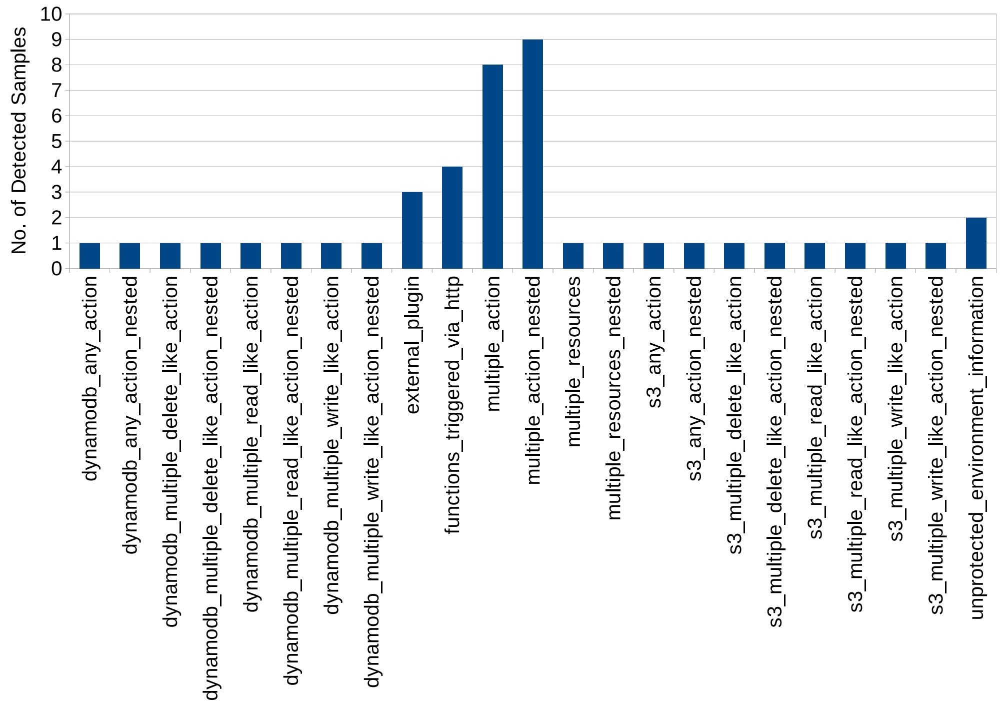


Fig. 5.4 Infrastructure code queries validation results

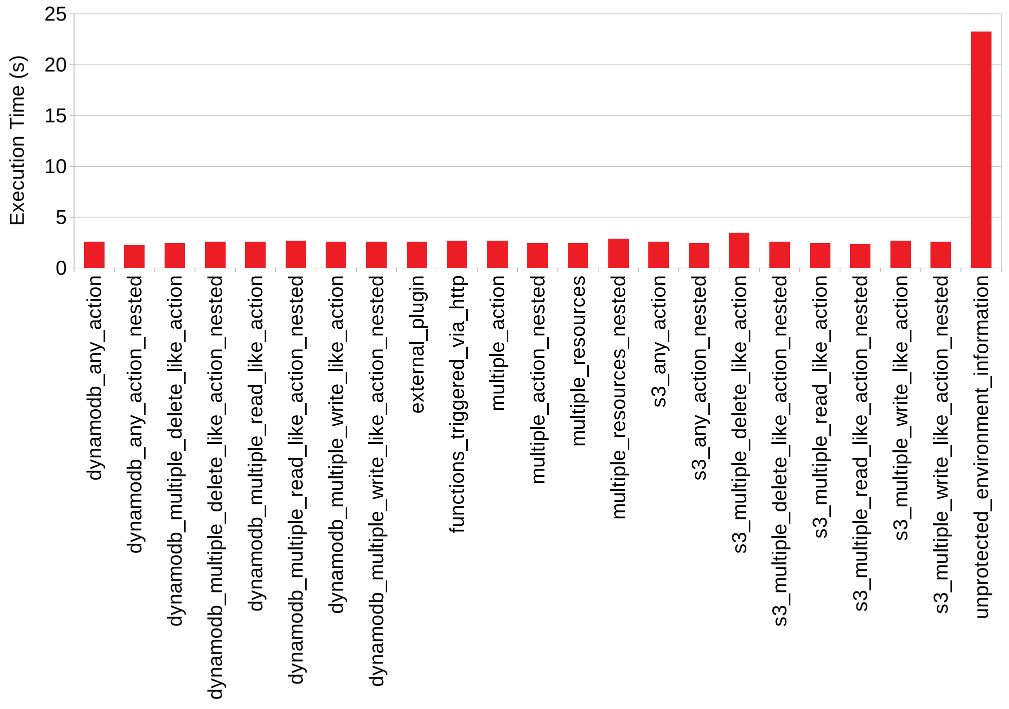


Fig. 5.5 Infrastructure code queries execution times

latter case, in fact, as inferred by the messages printed in the terminal window, the SI tool had to wait significantly longer for the end of the query execution.

### 5.3.2 Application code queries

The detailed validation results obtained for the application code queries are shown in Table 5.2, where the query and the code samples file names were abbreviated by removing the initial strings `query_python` and `sample_python`, respectively[8].

Table 5.2 Application code query vs code sample (✓=Warning Raised)

| **Application Code Query** | **dynamodb_scanfilter_injection.py** | **eval_function.py** | **event_data_returned_not_sanitized.py** | **exec_function.py** | **http_get_params.py** | **http_post_data.py** | **os_system_b.py** | **os_system_c.py** | **serverless_goat_example.py** | **subprocess_getoutput.py** |
|---|---|---|---|---|---|---|---|---|---|---|
| dynamodb_scanfilter_injection | ✓ | | | | | | | | | |
| eval_exec_functions | | ✓ | | ✓ | | | | | | |
| event_data_returned_not_sanitized | | | ✓ | | | | | | | |
| os_system | | | | | | | ✓ | ✓ | | |
| requests_functions | | | | | ✓ | ✓ | | | | |
| subprocess_functions | | | | | | | | | | ✓ |
| subprocess_shell_true | | | | | | | | | ✓ | |

It should be noted that only ten samples out of the 13 prepared for this project (Table 4.1) are included in Table 5.2. Due to the limitations explained in Sections 4.3.3 and 4.3.4, in fact, none of the developed queries can currently detect the vulnerabilities that are present in the following three samples:

- `sample_python_os_system_a.py`

[8]For consistency, the query names were abbreviated in the same manner in Fig. 5.6, which is introduced later in this section.

- `sample_python_subprocess_call.py`
- `sample_python_subprocess_check_output.py`

As regards the query execution times, they are shown in Fig. 5.6. Similarly to Fig. 5.5, the figures refer to all the 13 files of the repository `si-tool-application-self-test` (Table 5.1). Their average is 58.3 s, which means 4.5 s per sample file. The latter value is therefore more than 40 times higher than the one recorded during the infrastructure code query validation. While it is reasonable that the complexity introduced by the local taint analysis increases the execution times, it is important to emphasize that such a difference could also be partly due to the quality of the network connection as well as the load conditions of the LGTM engine[9].

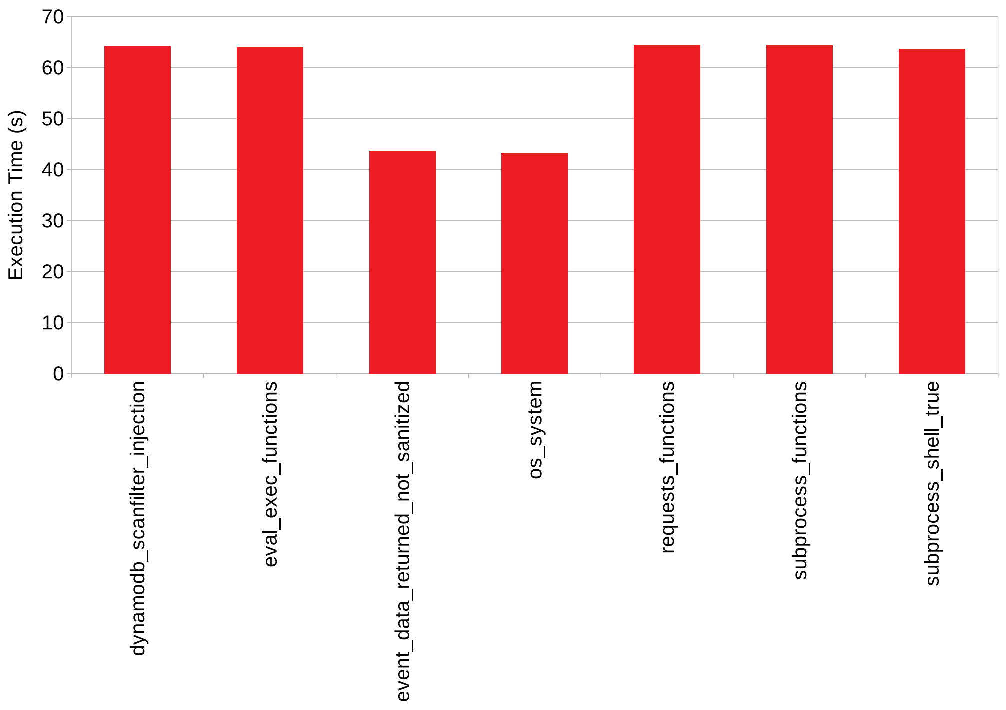


Fig. 5.6 Application code queries execution times

## 5.4 Tests with the Serverless Framework repository

The Serverless Framework example applications available in [120] were used to test more extensively the developed CodeQL queries. For each case, i.e., infrastructure (Section 5.4.1)

[9]All the tests with the SI tool in self-test mode were conducted on 4th August 2021, with the same computer and network connection, and scheduled one shortly after the other.

and application code queries (Section 5.4.2), this section details how the relevant test set was created and reports the obtained results.

As regards the execution times, in order to obtain their average values, each test was repeated three times. However, it was observed that the execution times recorded after the first test were significantly smaller. As also noted by the author during the initial manual tests, LGTM features a *caching* mechanism, which is triggered when an unmodified query is run against the same repository[10]. Therefore, it would be incorrect to present the average values of the execution times, and, similarly to the self-test mode results (Section 5.3), only the values logged during the first executions are reported.

## 5.4.1 Infrastructure code queries

After detailing how the used test set was prepared, this section presents the results obtained with the developed infrastructure code queries.

#### Test set preparation

A total of 89 AWS-specific examples were available in the Serverless Framework repository when this research began (May 2021). However, ten of them, which are listed in Table 3.2, were analysed prior to implementing the infrastructure code queries. Consequently, the remaining 79 were considered for the execution of the final tests described in this section.

Cloning the GitHub repository containing the targeted YAML files was the first step[11]. The Python script documented in Appendix B was then used to copy all the `serverless.yml` files to another folder, where they were renamed by appending the application name for traceability purposes. Taking into account the embedded commands issue mentioned in Section 3.5.3, the script attempted to map the YAML files into Python dictionaries, and finally moved those that could not be converted to another folder.

As further detailed in Appendix B, only two `serverless.yml` files could not be processed, which implies that the used test set comprised 77 samples.

[10]In this context, *same* repository indicates a repository where there has not been a new commit since the last execution of the query. It is also worth remembering that LGTM employs query identifiers, which are updated even when caching is used. As shown in Fig. 5.1, query identifiers are printed by the SI tool in the terminal window.

[11]The Serverless Framework repository [120] includes links to individual GitHub repositories, each dedicated to one specific application. However, since creating local copies of these repositories was not possible, the process described in this section began by cloning their Master Repository.

**Test results**

A total of 58 out of 77 samples, which correspond to 75.3% of the entire test set, were detected by the infrastructure code queries. Interestingly, as illustrated in Table 5.3, only four infrastructure code queries flagged potential vulnerabilities, and the number of files where warnings were raised ranges from 2 to 51. In addition, Table 5.4 shows that two queries reported multiple occurrences in some of the tested Serverless Framework examples.

The individual files flagged by the queries, along with the number of raised warnings on a per sample basis, are listed in a set of tables included in Appendix C.

Table 5.3 Number of code samples detected by the infrastructure code queries (Serverless Framework examples)

| **Infrastructure Code Query** | **No. of Detected Code Samples** |
|---|---|
| query_iac_external_plugin | 28 |
| query_iac_functions_triggered_via_http | 51 |
| query_iac_multiple_action_nested | 2 |
| query_iac_unprotected_environment_information | 7 |

Table 5.4 Number of code samples where multiple warnings were raised (Serverless Framework examples)

| **Infrastructure Code Query** | **No. of Code Samples** |
|---|---|
| query_iac_functions_triggered_via_http | 20 |
| query_iac_unprotected_environment_information | 2 |

The execution times are displayed in Fig. 5.7, where the query names were abbreviated by removing the initial string `query_iac`. In this case, after filtering out the two highest values, when the SI tool had to wait significantly longer before retrieving the results, the average is 30.0 s, i.e., approximately 0.4 s per file. Although higher than the value of 0.1 s measured in self-test mode (Section 5.3.1), the two results can be considered consistent, as the code samples included in the repository `si-tool-infrastructure-test` originate from real applications and are, therefore, generally more complex than those stored in the repository `si-tool-infrastructure-self-test`.

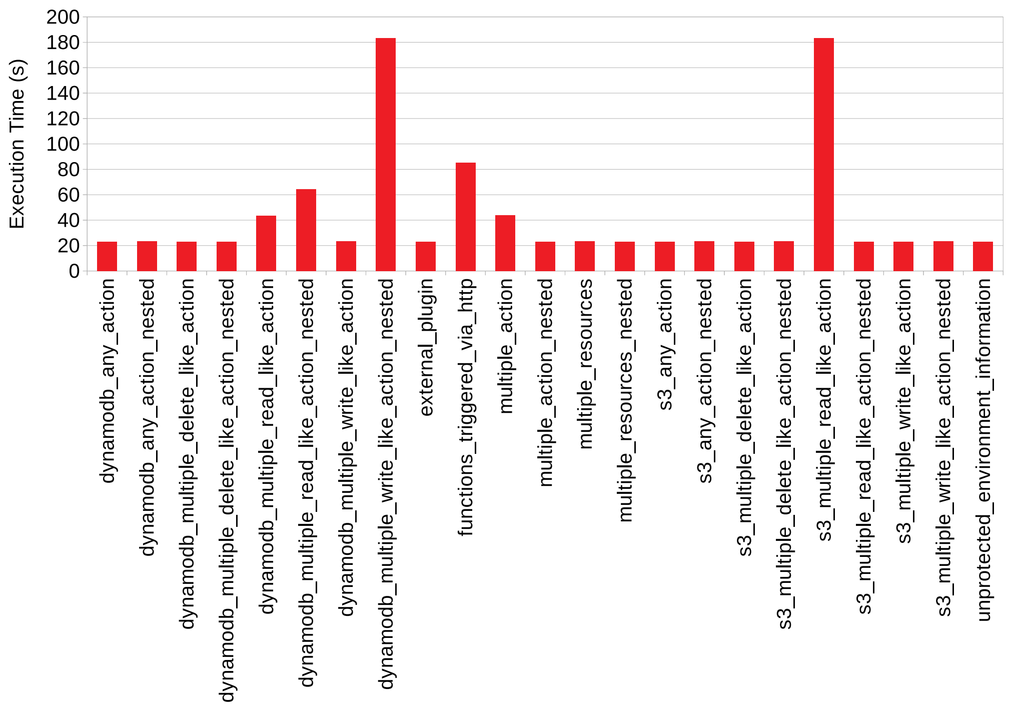


Fig. 5.7 Infrastructure code queries execution times (Serverless Framework tests)

## 5.4.2 Application code queries

This section begins by explaining how the tested application code files were retrieved from the Serverless Framework repository, and then ends with a summary of the query execution results.

### Test set preparation

At the beginning of this research activity, the Serverless Framework repository contained 16 AWS-specific examples of applications implemented in Python. Since some of them included more than one `.py` file, the final test set consisted of 38 samples. Note that their original locations are available in Appendix B, and that, differently from the infrastructure code case (Section 5.4.1), no Python script was used.

**Test results**

As shown in Table 5.5, only `query_python_event_data_returned_not_sanitized` identified potential vulnerabilities in five files[12]. It also noteworthy that in two cases multiple warnings were raised, with their values being six and two.

Table 5.5 Application code query vs code sample (Serverless Framework examples)

| **Application Code Query** | **bucket-aws-python-pynamodb-s3-sigurl.py** | **delete-aws-python-pynamodb-s3-sigurl.py** | **get-aws-python-pynamodb-s3-sigurl.py** | **lambda_handlers-aws-python-auth0-custom-authorizers-api.py** | **update-aws-python-pynamodb-s3-sigurl.py** |
|---|---|---|---|---|---|
| event_data_returned_not_sanitized | 1 | 1 | 1 | 6 | 2 |

The recorded execution times are shown in Fig. 5.8, where the query names were abbreviated by removing the initial string `query_python`. Note that, in this case, the vertical axis is in logarithmic scale, as an outlier equal to 183.3 s was measured for `query_python_dynamodb_scanfilter_injection.ql`. Without considering this value, the average of the execution times is 2.5 s, which means less than 0.1 s per file. Interestingly, this result is significantly different than the one recorded with application code queries in

[12]The name of the query was abbreviated in Table 5.5 for clarity of presentation.

self-test mode (Section 5.3.2), and it is more consistent with those recorded in other tests previously discussed.

This confirms that, as already mentioned, in addition to the complexity of the files being analysed, there are other factors that contribute to the query executions times, such as quality of the network connection and load conditions of the LGTM engine. Furthermore, although not quantified in this study, the author has observed that queries that have to retrieve results are frequently characterized by higher execution times in comparison with those that do not detect any issues.

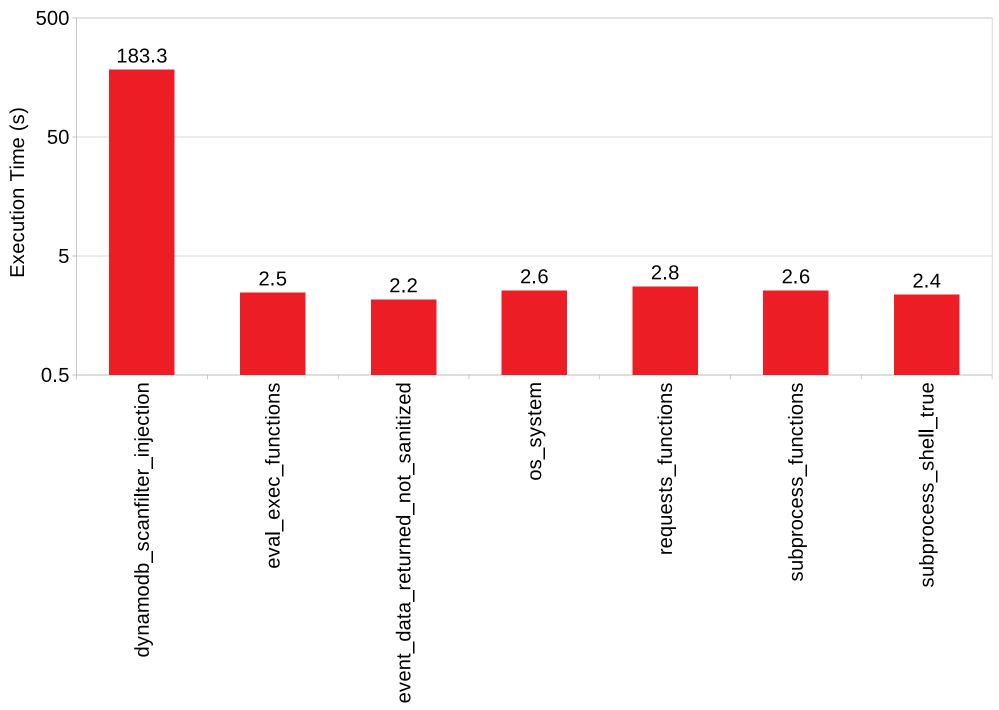


Fig. 5.8 Application code queries execution times (Serverless Framework tests)

# Chapter 6

# Conclusion

The main objective of this chapter is to present a summary of the achieved results (Section 6.1). In addition, a set of recommendations (Section 6.2) along with some suggestions for future research work (Section 6.3) are included as well.

## 6.1 Summary

In this work, an approach to securing serverless applications based on a semantic code search engine (CodeQL) has been explored. After identifying a set of relevant vulnerabilities, a collection of 30 CodeQL queries, focused on both infrastructure (Chapter 3) and application code (Chapter 4), were developed. In addition, to enable the automatic execution of the queries, a novel command-line tool, the Serverless Inspector (SI), was implemented by using the public API of the CodeQL-compatible analysis platform LGTM (Chapter 5). After validating the library of developed queries, the SI tool was also used to test 77 infrastructure code files and 38 application code files available in the open-source Serverless Framework examples repository.

In general, this work has proved that a semantic code search engine can be used to statically analyse serverless applications and improve their security posture. This is advantageous from a software development process standpoint because, differently from dedicated dynamic analysis frameworks recently proposed in other academic studies, e.g., [52], a tool such as CodeQL can easily be integrated into a modern, production-intent CI/CD pipeline. As regards more specifically the inspection of infrastructure code, however, the author would like to emphasize that the investigated approach has important limitations to be considered as well. As explained in Section 3.5.3, infrastructure code files can be mapped into a variety of data structures, and implementing queries that cover all cases is unattainable. Moreover, as further discussed in Section 6.2.2, CodeQL lacks some features, e.g., dynamic typing, that would be very helpful to handle numerous variants. As a result, while adopting the strategy described in Section 6.2.1 can mitigate this problem, and improving the 23 infrastructure code queries currently available is possible, alternative methods to inspect infrastructure code files should be assessed in future research work.

By contrast, the proposed application code queries indicate that CodeQL is better positioned to support the detection of security vulnerabilities in application code. Even though there are limitations caused by the number of variants in this case as well (Section 4.3), the local taint analysis resources provided by CodeQL are particularly suitable for serverless applications, which are typically implemented with a microservice-oriented architecture, i.e., with numerous, small, event-triggered functions. For completeness, it should be observed that CodeQL supports global taint analysis as well. Using the latter, however, in the context of serverless security does not appear to be viable, as a truly global taint analysis would have to take provider-managed code into account.

As regards the results obtained with the applications downloaded from the Serverless Framework examples repository, they provide interesting indications. The infrastructure code queries detected potential issues in 58 out of 77 samples, and the highest number of warnings was reported by `query_iac_functions_triggered_via_http.ql`. This suggests that identifying ways of protecting network-facing functions should be considered a priority, along with the security of external plug-ins, detected in 28 files (Table 5.3). In addition, the application code tests, despite flagging only five files, with a total of 11 warnings (Table 5.5), are indicative of the relevance of injection attacks caused by malicious event data, which should be further verified with additional tests (Section 6.3.3).

Finally, to complete the assessment of the proposed method, the SI tool was also used to measure the query execution times. In most cases, their average is less than one second per sample file, but much higher values have also been recorded, albeit very infrequently. Therefore, while additional factors, such as network connection quality and analysis engine load, need to be considered when using the SI tool, this research shows that they are not critical.

## 6.2 Recommendations

Considering the executed tests and their results, this section starts by providing recommendations that should be taken into account when integrating CodeQL into a CI/CD pipeline (Section 6.2.1). Furthermore, as detailed in Section 6.2.2, the author would like to point out that some CodeQL resources could be improved in future releases, whereas others would benefit from a more complete documentation.

### 6.2.1 Coding standards and approach to query execution

As explained in Section 3.5.3 and Section 4.3, the variety of infrastructure code-generated data structures and the number of application code variants are among the main problems highlighted by this research. Implementing generic CodeQL queries capable of covering all possible cases is unattainable, especially when a language featuring a highly flexible syntax, such as Python, is used. Moreover, it should be recognized that developing powerful CodeQL queries is a complex task, because some of its resources are not easy to use, and the absence of some features, e.g., dynamic typing, causes additional implementation problems, especially when hierarchical data structures, such as the nested Python dictionaries introduced in Section 3.5.2, have to be inspected.

Consequently, adopting coding standards for infrastructure as well as application code is particularly important. Compliant code could be statically analysed with CodeQL more confidently, as the queries would have to be implemented by focusing on what the standards allow, rather than a huge variety of coding variants. Taking into account the requirements of modern CI/CD pipelines, this process should be assisted by a code compliance-oriented preliminary analysis, which CodeQL can support as well. In other words, the approach explored in this project can add value to the development process of serverless applications if integrated into a CI/CD tool chain that first tests the code for compliance with the standards, and then executes the security-focused queries.

However, even if the above approach to query execution were adopted, some CodeQL limitations would remain, especially when it comes to inspecting complex data structures. Future research on alternative methods focused on infrastructure code security should therefore be encouraged.

### 6.2.2 CodeQL issues

While implementing the infrastructure code queries, the author observed that retrieving a Python dictionary value identified by a specific key was rather complicated. Since dictionaries are associative arrays, this operation should be straightforward. To address this issue, future releases of CodeQL could, for instance, include an additional, dedicated method within the class `DictItem` [37].

In addition, numerous experiments conducted during the development of the application code queries highlighted that using data flow or taint-tracking-specific resources along with generic ones, such as instances of the classes `AssignStmt`, `Call`, `CallNode`, and `ControlFlowNode` [37], is problematic. In most cases, incompatible types and lack of support for type casting were the reasons for failing to implement a working query. These

CodeQL resources should be reviewed to facilitate their integration and, in turn, the development of complex queries.

Finally, it is worth pointing out that CodeQL is capable of traversing nested Python dictionaries defined as literals. The query dedicated to identifying functions triggered via HTTP events (Table 3.6), for instance, can detect dictionaries with a *functions* key even when they are not at the highest hierarchical level of the inspected data structure. Therefore, CodeQL implements an implicit traversal of nested dictionaries, which, to the best of the author's knowledge, is an undocumented feature.

## 6.3 Future work

As previously mentioned, the library of CodeQL queries implemented as part of this project has some limitations. In addition to addressing these, the areas discussed in the remainder of this section should be considered for future research work.

### 6.3.1 Comprehensive static analysis of serverless applications

The CodeQL queries developed for this project are exclusively focused on either infrastructure or application code. However, adopting this approach implies establishing *a priori* what can cause security vulnerabilities, because the infrastructure code is inspected without any information about the application code, and vice versa. Unfortunately, this approach can cause false positives. As an example, one of the implemented queries (`query_iac_multiple_action.ql`) reports as a warning that multiple actions are specified in the Serverless Framework configuration file for a cloud-based resource. However, this set-up is legitimate if, for instance, the serverless function needs to read a DynamoDB table before modifying it.

Consequently, as illustrated in the example presented in [65], a comprehensive static analysis of a serverless application should link the results of the infrastructure code queries to those obtained with the application code queries. The first step toward achieving this would be the identification of an *action-permission mapping*, which should specify for each action on a database or a repository, for example, which permissions are actually needed. It should be noted that the AWS documentation on DynamoDB and S3 bucket actions consulted for this research [16, 23] does not include this information.

As regards possible implementations, filtering the results of infrastructure code queries to take into account the application code functionality could be the first option. Alternatively, it is also worth considering the automatic generation of infrastructure code queries following the analysis of the application code. To achieve this, customising the CodeQL engine by forking the repository available on GitHub [41] should be considered with a view to adding serverless-specific resources. Finally, it is important to highlight that the second suggested option would be based on a *dynamic* method. This would go well beyond the approach discussed in this report, which relies on ready-to-use CodeQL queries.

### 6.3.2 Analysis of existing tools

The analysis performed prior to the development of the infrastructure code queries (Section 3.1) indicates that there already exist tools that support static analysis of serverless applications, though not all of them are security-focused. The list presented in Table 3.1, however, cannot be considered exhaustive.

As shown in Table 6.1, the author has identified an additional set of commercial and open-source tools that should be analysed to gather requirements for future releases of the Serverless Inspector tool (Chapter 5).

Table 6.1 Additional commercial and open-source tools for serverless security

| Tool | Commercial | Open-source | References |
|---|---|---|---|
| AWS Config | ✓ | ✗ | [14, 107] |
| AWS Least Privilege | ✗ | ✓ | [128] |
| AWS WAF | ✓ | ✗ | [29, 9] |
| Imperva Serverless Protection | ✓ | ✗ | [63] |
| LambdaGuard | ✗ | ✓ | [131, 124] |
| Prisma Cloud | ✓ | ✗ | [100] |
| Serverless IAM Roles Plugin | ✗ | ✓ | [132] |
| Snyk Tools | ✓ | ✗ | [126] |

### 6.3.3 Extensive testing

Apart from the code samples prepared by the author, the implemented CodeQL queries were tested only with 77 infrastructure code files and 38 application code files, which were all downloaded from the Serverless Framework examples available at [120]. Therefore, a more extensive testing campaign would be needed, especially to evaluate the incidence of false positives.

To achieve this target, the AWS Serverless Repository [28] could be used for the application code queries. This repository, in fact, would provide JSON configuration files instead of YAML, which are Serverless Framework-specific. While mapping JSON data into Python dictionaries is certainly possible [81], the configuration files from the AWS repository could have a different structure, which implies that the current infrastructure code queries might not work. Alternatively, a test set could be obtained by using Serverless Framework-compatible applications available on GitHub. In this case, the GitHub Search API [58] should be used to identify relevant files.

# References

[1] Abhay Bhargav (2018). Dynamodb injection. [online] https://medium.com/appsecengineer/dynamodb-injection-1db99c2454ac.

[2] Abhinav Ajitsaria (2021). Logging in python. [online] https://realpython.com/python-logging/.

[3] Alchin, M. (2010). *Pro Python*. Apress, New York, NY, USA, first edition.

[4] Alex Ronquillo (2021). Python's requests library (guide). [online] https://realpython.com/python-requests/.

[5] Alpernas, K., Flanagan, C., Fouladi, S., Ryzhyk, L., Sagiv, M., Schmitz, T., and Winstein, K. (2018). Secure serverless computing using dynamic information flow control. *Proc. ACM Program. Lang.*, 2(OOPSLA).

[6] Amazon Web Services (2021). Aws solutions. [online] https://aws.amazon.com/.

[7] Avgustinov, P., de Moor, O., Jones, M. P., and Schäfer, M. (2016). QL: Object-oriented Queries on Relational Data. In Krishnamurthi, S. and Lerner, B. S., editors, *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:25, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[8] AWS Amazon Resource Names (2021). Amazon resource names (arns). [online] https://docs.aws.amazon.com/general/latest/gr/aws-arns-and-namespaces.html.

[9] AWS API Gateway (2021). Using aws waf to protect your apis. [online] https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-control-access-aws-waf.html.

[10] AWS API Gateway Access Control (2021). Controlling and managing access to a rest api in api gateway. [online] https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-control-access-to-api.html.

[11] AWS API Gateway IAM Permissions (2021). Control access to an api with iam permissions. [online] https://docs.aws.amazon.com/apigateway/latest/developerguide/permissions.html.

[12] AWS CloudWatch (2021). Amazon cloudwatch documentation. [online] https://docs.aws.amazon.com/cloudwatch/.

[13] AWS Cognito (2021). Amazon cognito documentation. [online] https://docs.aws.amazon.com/cognito/.

[14] AWS Config Tool (2021). Aws config documentation. [online] https://docs.aws.amazon.com/config/index.html.

[15] AWS DynamoDB (2021). Amazon dynamodb documentation. [online] https://docs.aws.amazon.com/dynamodb/index.html.

[16] AWS DynamoDB Actions (2021). Actions, resources, and condition keys for amazon dynamodb. [online] https://docs.aws.amazon.com/service-authorization/latest/reference/list_amazondynamodb.html.

[17] AWS DynamoDB Documentation (2021). Step 4: Query and scan the data. [online] https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GettingStarted.Python.04.html.

[18] AWS IAM (2021). What is iam? [online] https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html.

[19] AWS IAM Best Practices (2021). Security best practices in iam. [online] https://docs.aws.amazon.com/IAM/latest/UserGuide/best-practices.html.

[20] AWS Lambda (2021). Aws lambda documentation. [online] https://docs.aws.amazon.com/lambda/index.html.

[21] AWS Lambda Usage (2021). Using aws lambda with other services. [online] https://docs.aws.amazon.com/lambda/latest/dg/lambda-services.html.

[22] AWS MFA-protected API Access (2021). Configuring mfa-protected api access. [online] https://docs.aws.amazon.com/IAM/latest/UserGuide/id_credentials_mfa_configure-api-require.html.

[23] AWS S3 Actions (2021). Actions, resources, and condition keys for amazon s3. [online] https://docs.aws.amazon.com/service-authorization/latest/reference/list_amazons3.html.

[24] AWS S3 Bucket Policy Examples (2021). Bucket policy examples. [online] https://docs.aws.amazon.com/AmazonS3/latest/userguide/example-bucket-policies.html.

[25] AWS S3 Bucket Support (2021). How can i secure the files in my amazon s3 bucket? [online] https://aws.amazon.com/premiumsupport/knowledge-center/secure-s3-resources/.

[26] AWS S3 Bucket User Policy Examples (2021). User policy examples. [online] https://docs.aws.amazon.com/AmazonS3/latest/userguide/example-policies-s3.html.

[27] AWS S3 Documentation (2021). Amazon simple storage service documentation. [online] https://docs.aws.amazon.com/s3/index.html.

[28] AWS Serverless Application Repository (2021). Aws serverless application repository. [online] https://serverlessrepo.aws.amazon.com/applications.

[29] AWS WAF Documentation (2021). Aws waf - web application firewall. [online] https://aws.amazon.com/waf/.

[30] AWS White Paper (2021). Infrastructure as code. [online] https://d1.awsstatic.com/whitepapers/DevOps/infrastructure-as-code.pdf?did=wp_card&trk=wp_card.

[31] Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., and Suter, P. (2017). Serverless Computing: Current Trends and Open Problems. *arXiv e-prints*, page arXiv:1706.03178.

[32] Baumann, A., Peinado, M., and Hunt, G. (2014). Shielding applications from an untrusted cloud with haven. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 267–283, Broomfield, CO. USENIX Association.

[33] Beazley, D. (2009). *Python Essential Reference*. Addison-Wesley Professional, Upper Saddle River, NJ, USA, fourth edition.

[34] Brikman, Y. (2019). *Terraform: Up & Running*. O'Reilly Media, Inc., Sebastopol, CA, USA, second edition.

[35] Chester, J. (2021). *Knative in Action*. Manning Publications, Shelter Island, NY, USA, first edition.

[36] Cloud Native Computing Foundation (2021). Cncf wg-serverless whitepaper v1.0. [online] https://raw.githubusercontent.com/cncf/wg-serverless/master/whitepapers/serverless-overview/cncf_serverless_whitepaper_v1.0.pdf.

[37] CodeQL Class Dict Documentation (2021). Class dict. [online] https://codeql.github.com/codeql-standard-libraries/python/index.html#D.

[38] CodeQL Data Flow Analysis (2021). About data flow analysis. [online] https://codeql.github.com/docs/writing-codeql-queries/about-data-flow-analysis/.

[39] CodeQL Documentation (2021). Discover vulnerabilities across a codebase with codeql, our industry-leading semantic code analysis engine. [online] https://codeql.github.com/docs/.

[40] CodeQL GitHub Documentation (2021). Using codeql code scanning with your existing ci system. [online] https://docs.github.com/en/code-security/secure-coding/using-codeql-code-scanning-with-your-existing-ci-system.

[41] CodeQL GitHub Repository (2021). Codeql: the libraries and queries that power security researchers around the world. [online] https://github.com/github/codeql.

[42] CodeQL Path Queries (2021). Creating path queries. [online] https://codeql.github.com/docs/writing-codeql-queries/creating-path-queries/.

[43] CodeQL Predicates Documentation (2021). Predicates. [online] https://codeql.github.com/docs/ql-language-reference/predicates/.

[44] CodeQL Publications (2021). Academic publications. [online] https://codeql.github.com/publications/.

[45] CodeQL Python API Graphs (2021). Using api graphs in python. [online] https://codeql.github.com/docs/codeql-language-guides/using-api-graphs-in-python/.

[46] CodeQL Python Control Flow (2021). Analyzing control flow in python. [online] https://codeql.github.com/docs/codeql-language-guides/analyzing-control-flow-in-python/.

[47] CodeQL Python Data Flow (2021). Analyzing data flow in python. [online] https://codeql.github.com/docs/codeql-language-guides/analyzing-data-flow-in-python/.

[48] CodeQL Python Example (2021). Uncontrolled data used in path expression. [online] https://codeql.github.com/codeql-query-help/python/py-path-injection/.

[49] CodeQL Resources (2021). Learning codeql. [online] https://lgtm.com/help/lgtm/ql/learning-ql.

[50] CodeQL Visual Studio Code Extension (2021). Ide integration for lgtm. [online] https://lgtm.com/help/lgtm/ide-integration.

[51] Colin's ALM Corner Website (2021). Custom codeql. [online] https://colinsalmcorner.com/custom-codeql/.

[52] Datta, P., Kumar, P., Morris, T., Grace, M., Rahmati, A., and Bates, A. (2020). Valve: Securing function workflows on serverless computing platforms. In *Proceedings of The Web Conference 2020*, WWW '20, pages 939–950, New York, NY, USA. Association for Computing Machinery.

[53] Davide Mastromatteo (2021). How to build command line interfaces in python with argparse. [online] https://realpython.com/command-line-interfaces-python-argparse/.

[54] de Moor, O., Sereni, D., Verbaere, M., Hajiyev, E., Avgustinov, P., Ekman, T., Ongkingco, N., and Tibble, J. (2008). *.QL: Object-Oriented Queries Made Easy*, pages 78–133. Springer Berlin Heidelberg, Berlin, Heidelberg.

[55] de Moor, O., Verbaere, M., Hajiyev, E., Avgustinov, P., Ekman, T., Ongkingco, N., Sereni, D., and Tibble, J. (2007). Keynote address: .ql for source code analysis. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 3–16.

[56] Digital Guardian Website (2017). The data breach (amazon) bucket list. [online] https://digitalguardian.com/blog/data-breach-amazon-bucket-list.

[57] Frank, F., Alfke, M., Franceschi, A., Pastor, J. S., and Uphillis, T. (2017). *Puppet: Mastering Infrastructure Automation*. Packt Publishing, Birmingham, UK, first edition.

[58] GitHub Search API Documentation (2021). The github search api lets you search for the specific item efficiently. [online] https://docs.github.com/en/rest/reference/search.

[59] Gollmann, D. (2011). *Computer Security*. John Wiley & Sons, Chichester, UK, third edition.

[60] Google Cloud Platform (2021). Accelerate your transformation with google cloud. [online] https://cloud.google.com/.

[61] gVisor (2021). gvisor is an application kernel for containers that provides efficient defense-in-depth anywhere. [online] https://gvisor.dev/.

[62] Hambling, B., Samaroo, A., Morgan, P., Thompson, G., and Williams, P. (2015). *Software Testing: An ISTQB-BCS Certified Tester Foundation Guide*. BCS, The Chartered Institute for IT, Swindon, UK, third edition.

[63] Imperva Capability Brief (2021). Serverless protection for aws. [online] https://www.imperva.com/resources/datasheets/Imperva_Serverless_Protection_for_AWS_SolutionBrief_20201029_v101.pdf.

[64] Intel SGX (2021). Intel software guard extensions. [online] https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html.

[65] Jeremy Daly (2020). Event injection: Protecting your serverless applications. [online] https://www.jeremydaly.com/event-injection-protecting-your-serverless-applications/.

[66] Joanna Jablonski (2021). Python 3's f-strings: An improved string formatting syntax (guide). [online] https://realpython.com/python-f-strings/.

[67] Katzer, J. (2020). *Learning Serverless*. O'Reilly Media, Inc., Sebastopol, CA, USA, first edition.

[68] LGTM API Documentation (2021). Api for lgtm. [online] https://lgtm.com/help/lgtm/api/api-for-lgtm.

[69] LGTM API Query Jobs Documentation (2021). Query jobs. [online] https://lgtm.com/help/lgtm/api/api-v1#LGTM-API-specification-Query-jobs.

[70] LGTM Data Flow Paths Documentation (2021). Exploring data flow paths. [online] https://lgtm.com/help/lgtm/exploring-data-flow-paths.

[71] LGTM Database Generation (2021). Generating databases. [online] https://lgtm.com/help/lgtm/generate-database.

[72] LGTM Documentation (2021). About lgtm. [online] https://lgtm.com/help/lgtm/about-lgtm.

[73] LGTM Enterprise Documentation (2021). Administrator help - scheduling project analysis. [online] https://help.semmle.com/lgtm-enterprise/admin/help/scheduling-project-analysis.html.

[74] LGTM Issue Report (2021). Lgtm.com - false positive - python - unused variable in nested f string #2885. [online] https://github.com/github/codeql/issues/2885.

[75] LGTM Personalized Views (2021). Adding projects to lgtm. [online] https://lgtm.com/help/lgtm/adding-projects.

[76] LGTM Python Extraction (2021). Python extraction. [online] https://lgtm.com/help/lgtm/python-extraction.

[77] LGTM Query Console (2021). Query console. [online] https://lgtm.com/query.

[78] LGTM Query Console Documentation (2021). Using the query console. [online] https://lgtm.com/help/lgtm/using-query-console.

[79] LGTM Website (2021). Continuous security analysis. [online] https://lgtm.com/.

[80] LocalStack (2021). A fully functional local cloud stack. [online] https://localstack.cloud/.

[81] Lucas Lofaro (2021). Working with json data in python. [online] https://realpython.com/python-json/.

[82] Luo, L., Sanchez, D., Schäf, M., and Bodden, E. (2021). Ide support for cloud-based static analyses. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE '21, New York, NY, USA. Association for Computing Machinery.

[83] Marcilio Mendonca (2019). Use the data api to interact with an amazon aurora serverless mysql database. [online] https://aws.amazon.com/blogs/database/using-the-data-api-to-interact-with-an-amazon-aurora-serverless-mysql-database/.

[84] Microsoft Azure (2021). Power your vision on azure. [online] https://azure.microsoft.com/en-gb/.

[85] Microsoft Azure Documentation (2021). Use the azurite emulator for local azure storage development. [online] https://docs.microsoft.com/en-us/azure/storage/common/storage-use-azurite.

[86] Morris, K. (2020). *Infrastructure as Code*. O'Reilly Media, Inc., Sebastopol, CA, USA, second edition.

[87] Moto (2021). Moto: Mock aws services. [online] http://docs.getmoto.org/en/latest.

[88] Nicolalde, A. (2021). Security issues in serverless applications. Master's thesis, MSci (Hons) in Computer Science (Information Security), Royal Holloway, University of London.

[89] Nikhil Kumar (2021). Get and post requests using python. [online] https://www.geeksforgeeks.org/get-post-requests-using-python/.

[90] Obetz, M., Das, A., Castiglia, T., Patterson, S., and Milanova, A. (2020). Formalizing event-driven behavior of serverless applications. In Brogi, A., Zimmermann, W., and Kritikos, K., editors, *Service-Oriented and Cloud Computing*, pages 19–29, Cham. Springer International Publishing.

[91] Obetz, M., Patterson, S., and Milanova, A. (2019). Static call graph construction in AWS lambda serverless applications. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA. USENIX Association.

[92] Olivia Smith (2019). Performing an http request in python. [online] https://www.datacamp.com/community/tutorials/making-http-requests-in-python.

[93] OpenFaaS (2021). Serverless functions, made simple. [online] https://www.openfaas.com/.

[94] OpenLambda (2021). An open source serverless computing platform. [online] https://github.com/open-lambda/open-lambda.

[95] OpenWhisk (2021). Open source serverless cloud platform. [online] https://openwhisk.apache.org/.

[96] Ory Segal (2018). Serverless security - what are we up against? [online] https://www.youtube.com/watch?v=M7wUanfWs1c.

[97] OWASP Serverless Top 10 Project (2021). Owasp serverless top 10. [online] https://owasp.org/www-project-serverless-top-10/.

[98] OWASP Serverless Top 10 Report (2018). Owasp top 10: Serverless interpretation. [online] https://raw.githubusercontent.com/OWASP/Serverless-Top-10-Project/master/OWASP-Top-10-Serverless-Interpretation-en.pdf.

[99] Palo Alto Networks (2019). Palo alto networks completes acquisition of puresec. [online] https://www.paloaltonetworks.com/company/press/2019/palo-alto-networks-completes-acquisition-of-puresec.

[100] Palo Alto Networks Prisma Cloud (2021). Cloud security at its best. [online] https://www.paloaltonetworks.com/prisma/cloud.

[101] Panker, T. and Nissim, N. (2021). Leveraging malicious behavior traces from volatile memory using machine learning methods for trusted unknown malware detection in linux cloud environments. *Knowledge-Based Systems*, 226:107095.

[102] Perforce Software Website (2019). What is lint code? and why is linting important? [online] https://www.perforce.com/blog/qac/what-lint-code-and-why-linting-important.

[103] Podjarny, G. and Tal, L. (2019). *Serverless Security*. O'Reilly Media, Inc., Sebastopol, CA, USA, first edition.

[104] PortSwigger Website (2019). Alexa, hack my serverless technology – attacking web apps with voice commands. [online] https://portswigger.net/daily-swig/alexa-hack-my-serverless-technology-attacking-web-apps-with-voice-commands.

[105] PortSwigger Website (2021). Latest amazon security news. [online] https://portswigger.net/daily-swig/amazon.

[106] Pulumi (2021). Cloud engineering for everyone. [online] https://www.pulumi.com/.

[107] PureSec (2019). The ten most critical risks for serverless applications v1.0. [online] https://github.com/puresec/sas-top-10.

[108] Sale, D. (2014). *Testing Python*. John Wiley & Sons, Chichester, UK, first edition.

[109] Sankaran, A., Datta, P., and Bates, A. (2020). Workflow integration alleviates identity and access management in serverless computing. In *Annual Computer Security Applications Conference*, ACSAC '20, pages 496–509, New York, NY, USA. Association for Computing Machinery.

[110] Sarkar, A. and Shah, A. (2018). *Learning AWS*. Packt Publishing, Birmingham, UK, second edition.

[111] Semgrep Website (2021). Static analysis at ludicrous speed - find bugs and enforce code standards. [online] https://semgrep.dev/.

[112] Semmle Announcement (2019). Securing software together: Github + semmle. [online] https://blog.semmle.com/secure-software-github-semmle/.

[113] Semmle BlackLine Case Study (2021). Semmle at blackline: Securing the data integrity of financial data in the cloud. [online] https://semmle.com/case-studies/semmle-blackline-securing-data-integrity-financial-data-cloud.

[114] Semmle Blog (2019). Code as data: Advanced techniques for code analysis. [online] https://blog.semmle.com/code-as-data-code-analysis/.

[115] Semmle CodeQL Webpage (2021). Codeql. [online] https://semmle.com/codeql.

[116] Semmle LGTM Webpage (2021). Lgtm. [online] https://semmle.com/lgtm.

[117] Semmle NASA Case Study (2021). Semmle at nasa: Landing curiosity safely on mars. [online] https://semmle.com/case-studies/semmle-nasa-landing-curiosity-safely-mars.

[118] Semmle Website (2021). Securing software, together. [online] https://semmle.com/.

[119] Serverless Example With DynamoDB (2021). Aws application with python flask dynamodb api. [online] https://github.com/serverless/examples/blob/master/aws-python-flask-dynamodb-api/app.py.

[120] Serverless Framework Examples (2021). See real world serverless code and architecture examples. [online] https://www.serverless.com/examples/.

[121] Serverless Framework Website (2021). Zero-friction serverless development. [online] https://www.serverless.com/.

[122] Serverless Goat Python Application (2021). Lambda function python file. [online] https://github.com/motikan2010/Serverless-Goat-Python/blob/main/src/api/convert/lambda_function.py.

[123] Sirone Jegan, D., Wang, L., Bhagat, S., Ristenpart, T., and Swift, M. (2020). Guarding Serverless Applications with SecLambda. *arXiv e-prints*, page arXiv:2011.05322.

[124] Skyscanner Engineering Website (2019). Introducing lambdaguard — a security scanner for aws lambda. [online] https://medium.com/@SkyscannerEng/introducing-lambdaguard-a-security-scanner-for-aws-lambda-f5c6e23f8345.

[125] Snyk Best Security Practices (2019). 10 serverless security best practices. [online] https://snyk.io/blog/10-serverless-security-best-practices/.

[126] Snyk Website (2021). Develop fast - stay secure. [online] https://snyk.io/.

[127] Tal Melamed (2019). Alexa, hack my server(less) please. [online] https://i.blackhat.com/eu-19/Thursday/eu-19-Melamed-Alexa-Hack-My-Server-less-Please.pdf.

[128] Tool AWS Least Privilege (2021). Use aws x-ray to reach least privilege. [online] https://github.com/functionalone/aws-least-privilege.

[129] Tool cfn-lint (2021). Cloudformation linter. [online] https://github.com/aws-cloudformation/cfn-lint.

[130] Tool cfn_nag (2021). Linting tool for cloudformation templates. [online] https://github.com/stelligent/cfn_nag.

[131] Tool LambdaGuard (2021). Aws serverless security. [online] https://github.com/Skyscanner/LambdaGuard.

[132] Tool Serverless IAM Roles Per Function Plugin (2021). Serverless plugin for easily defining iam roles per function via the use of iamrolestatements at the function level. [online] https://github.com/functionalone/serverless-iam-roles-per-function.

[133] Tool tflint (2021). A pluggable terraform linter. [online] https://github.com/terraform-linters/tflint.

[134] Tool tfsec (2021). Security scanner for your terraform code. [online] https://github.com/aquasecurity/tfsec.

[135] Trach, B., Oleksenko, O., Gregor, F., Bhatotia, P., and Fetzer, C. (2019). Clemmys: Towards secure remote execution in faas. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR '19, pages 44–54, New York, NY, USA. Association for Computing Machinery.

[136] Turnquist, G. L. (2011). *Python Testing Cookbook*. Packt Publishing, Birmingham, UK, first edition.

[137] Ustiugov, D., Petrov, P., Kogias, M., Bugnion, E., and Grot, B. (2021). Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*. ACM.

[138] vHive (2021). Open-source framework for serverless experimentation. [online] https://github.com/ease-lab/vhive.

[139] YAML (2021). Yaml ain't markup language. [online] https://yaml.org/.

# Appendix A

# Infrastructure code queries validation

The purpose of this appendix is to document the validation of the developed infrastructure code queries, which was completed with the SI tool in self-test mode (Section 5.3.1).

## A.1 Detailed results tables

The detailed results obtained with the SI tool in self-test mode are shown in Tables A.1, A.2, A.3, and A.4, where the query and the code samples file names were abbreviated by removing the strings `query_iac` and `serverless_iac`, respectively.

Table A.1 Infrastructure code query vs code sample (Part 1 of 4, ✓=Warning Raised)

| **Infrastructure Code Query** | **dynamodb_any_action.py** | **dynamodb_any_action_nested.py** | **dynamodb_multiple_delete_like_action.py** | **dynamodb_multiple_delete_like_action_nested.py** | **dynamodb_multiple_read_like_action.py** | **dynamodb_multiple_read_like_action_nested.py** | **dynamodb_multiple_write_like_action.py** |
|---|---|---|---|---|---|---|---|
| dynamodb_any_action | ✓ | | | | | | |
| dynamodb_any_action_nested | | ✓ | | | | | |
| dynamodb_multiple_delete_like_action | | | ✓ | | | | |
| dynamodb_multiple_delete_like_action_nested | | | | ✓ | | | |
| dynamodb_multiple_read_like_action | | | | | ✓ | | |
| dynamodb_multiple_read_like_action_nested | | | | | | ✓ | |
| dynamodb_multiple_write_like_action | | | | | | | ✓ |
| dynamodb_multiple_write_like_action_nested | | | | | | | |
| external_plugin | | | | | | | |
| functions_triggered_via_http | | | | | | | |
| multiple_action | | | ✓ | | ✓ | | ✓ |
| multiple_action_nested | | | | ✓ | | ✓ | |
| multiple_resources | | | | | | | |
| multiple_resources_nested | | | | | | | |
| s3_any_action | | | | | | | |
| s3_any_action_nested | | | | | | | |
| s3_multiple_delete_like_action | | | | | | | |
| s3_multiple_delete_like_action_nested | | | | | | | |
| s3_multiple_read_like_action | | | | | | | |
| s3_multiple_read_like_action_nested | | | | | | | |
| s3_multiple_write_like_action | | | | | | | |
| s3_multiple_write_like_action_nested | | | | | | | |
| unprotected_environment_information | | | | | | | |

Table A.2 Infrastructure code query vs code sample (Part 2 of 4, ✓=Warning Raised)

| **Infrastructure Code Query** | **dynamodb_multiple_write_like_action_nested.py** | **function_environment_secret.py** | **multiple_action.py** | **multiple_action_nested.py** | **multiple_functions_http_multiple_plugins.py** | **multiple_resources.py** | **multiple_resources_nested.py** |
|---|---|---|---|---|---|---|---|
| dynamodb_any_action | | | | | | | |
| dynamodb_any_action_nested | | | | | | | |
| dynamodb_multiple_delete_like_action | | | | | | | |
| dynamodb_multiple_delete_like_action_nested | | | | | | | |
| dynamodb_multiple_read_like_action | | | | | | | |
| dynamodb_multiple_read_like_action_nested | | | | | | | |
| dynamodb_multiple_write_like_action | | | | | | | |
| dynamodb_multiple_write_like_action_nested | ✓ | | | | | | |
| external_plugin | | | | | ✓ | | |
| functions_triggered_via_http | | ✓ | | | ✓ | | |
| multiple_action | | | ✓ | | | ✓ | |
| multiple_action_nested | ✓ | | | ✓ | | | ✓ |
| multiple_resources | | | | | | ✓ | |
| multiple_resources_nested | | | | | | | ✓ |
| s3_any_action | | | | | | | |
| s3_any_action_nested | | | | | | | |
| s3_multiple_delete_like_action | | | | | | | |
| s3_multiple_delete_like_action_nested | | | | | | | |
| s3_multiple_read_like_action | | | | | | | |
| s3_multiple_read_like_action_nested | | | | | | | |
| s3_multiple_write_like_action | | | | | | | |
| s3_multiple_write_like_action_nested | | | | | | | |
| unprotected_environment_information | | ✓ | | | | | |

Table A.3 Infrastructure code query vs code sample (Part 3 of 4, ✓=Warning Raised)

| **Infrastructure Code Query** | **one_function_http_one_plugin.py** | **one_function_multiple_http_one_plugin.py** | **provider_environment_secret.py** | **s3_any_action.py** | **s3_any_action_nested.py** | **s3_multiple_delete_like_action.py** | **s3_multiple_delete_like_action_nested.py** |
|---|---|---|---|---|---|---|---|
| dynamodb_any_action | | | | | | | |
| dynamodb_any_action_nested | | | | | | | |
| dynamodb_multiple_delete_like_action | | | | | | | |
| dynamodb_multiple_delete_like_action_nested | | | | | | | |
| dynamodb_multiple_read_like_action | | | | | | | |
| dynamodb_multiple_read_like_action_nested | | | | | | | |
| dynamodb_multiple_write_like_action | | | | | | | |
| dynamodb_multiple_write_like_action_nested | | | | | | | |
| external_plugin | ✓ | ✓ | | | | | |
| functions_triggered_via_http | ✓ | ✓ | | | | | |
| multiple_action | | | | | | ✓ | |
| multiple_action_nested | | ✓ | | | | | ✓ |
| multiple_resources | | | | | | | |
| multiple_resources_nested | | | | | | | |
| s3_any_action | | | | ✓ | | | |
| s3_any_action_nested | | | | | ✓ | | |
| s3_multiple_delete_like_action | | | | | | ✓ | |
| s3_multiple_delete_like_action_nested | | | | | | | ✓ |
| s3_multiple_read_like_action | | | | | | | |
| s3_multiple_read_like_action_nested | | | | | | | |
| s3_multiple_write_like_action | | | | | | | |
| s3_multiple_write_like_action_nested | | | | | | | |
| unprotected_environment_information | | | ✓ | | | | |

Table A.4 Infrastructure code query vs code sample (Part 4 of 4, ✓=Warning Raised)

| **Infrastructure Code Query** | **s3_multiple_read_like_action.py** | **s3_multiple_read_like_action_nested.py** | **s3_multiple_write_like_action.py** | **s3_multiple_write_like_action_nested.py** |
|---|---|---|---|---|
| dynamodb_any_action | | | | |
| dynamodb_any_action_nested | | | | |
| dynamodb_multiple_delete_like_action | | | | |
| dynamodb_multiple_delete_like_action_nested | | | | |
| dynamodb_multiple_read_like_action | | | | |
| dynamodb_multiple_read_like_action_nested | | | | |
| dynamodb_multiple_write_like_action | | | | |
| dynamodb_multiple_write_like_action_nested | | | | |
| external_plugin | | | | |
| functions_triggered_via_http | | | | |
| multiple_action | ✓ | | ✓ | |
| multiple_action_nested | | ✓ | | ✓ |
| multiple_resources | | | | |
| multiple_resources_nested | | | | |
| s3_any_action | | | | |
| s3_any_action_nested | | | | |
| s3_multiple_delete_like_action | | | | |
| s3_multiple_delete_like_action_nested | | | | |
| s3_multiple_read_like_action | ✓ | | | |
| s3_multiple_read_like_action_nested | | ✓ | | |
| s3_multiple_write_like_action | | | ✓ | |
| s3_multiple_write_like_action_nested | | | | ✓ |
| unprotected_environment_information | | | | |

# Appendix B

# Serverless Framework test sets

This appendix provides additional information on the test sets obtained from Serverless Framework example repository. This includes the documentation of the Python script used to process the YAML files initially identified (Section 5.4.1).

## B.1 Python code

The Python script used to prepare the infrastructure code test set is documented in Fig. B.1 and Fig. B.2. As regards the results of its execution:

- In two cases the file `serverless.yml` was not found at the expected location in the repository[1], whereas in one case[2] the file extension was `.yaml` rather than `.yml`. All the files that could not be automatically processed were downloaded and then added manually to the test set.
- Due to `Resource` and `FunctionName` entries that included the `Fn::` syntax, which is not supported by the used Python YAML parser, two `serverless.yml` files could not be processed[3].

[1]The applications aws-node-fullstack and aws-node-shared-gateway included the YAML file in the folders `backend` and `gateway`, respectively.

[2]The application aws-node-github-check.

[3]The applications for which the YAML files were not processed are aws-node-ses-receive-email-body and aws-node-ses-receive-email-header.

```
# ====================
# Import Python modules
# ====================
import os
import shutil
import yaml

# ================
# Script parameters
# ================
# The following file must be stored within the same folder as this script.
ServExamplesTargetFileName = "infrastructure_test_set_examples.txt"
# Name of the folder containing all the Serverless Framework examples (root).
# This folder must be stored within the same folder as this script.
ExamplesRootFolderName = 'examples'
# Full path of the folder where all the serverless.yml files of interest
# will be stored. Files that cannot be parsed will subsequently be moved.
TestSetFolderFullPath = '/home/giuseppe/Desktop/Infrastructure_Code_Test_Set'
# Target file name. This is the file that will be looked for within the
# the example folders of interest.
TargetFileName = 'serverless.yml'
# Full path of the folder where all target files that cannot be parsed
# will be moved for further inspection.
RejectedFilesFolderFullPath = '/home/giuseppe/Desktop/Rejected_Files'

# ============================================================
# Upload file with Serverless Framework examples to be considered
# ============================================================
ServExamplesTargetSet = set(Line.split(os.sep)[-1].replace('\n','') for Line in
open(ServExamplesTargetFileName, mode='r'))
```

Fig. B.1 Python script executed to prepare the Serverless Framework test set used for the infrastructure code queries (Part 1 of 2)

```
# ========================================================
# Copy and rename all target files within the selected example
# ========================================================
for FolderName in (Elem for Elem in os.listdir(os.path.join(os.curdir, ExamplesRootFolderName)) if (Elem in
ServExamplesTargetSet)):
    print()
    try:
        print('--- Example folder %s being processed... ---' % FolderName)
        print('--- The target file is about to be copied... ---')
        shutil.copy2(os.path.join(os.curdir, ExamplesRootFolderName, FolderName, TargetFileName),
TestSetFolderFullPath)
        print('--- The target file is about to be renamed... ---')
        os.rename(os.path.join(TestSetFolderFullPath, TargetFileName), os.path.join(TestSetFolderFullPath,
os.path.splitext(TargetFileName)[0] + '-' + FolderName + os.path.splitext(TargetFileName)[1]))
    except Exception as Error:
        print('--- Exception raised - Details: ---')
        print('--- %s ---' % Error)

# ======================================================================
# Launch YAML parser and move all the renamed target files to a different folder
# ======================================================================
# This processing step allows separating the target files that cannot be parsed
# for further inspection.
for FileName in os.listdir(TestSetFolderFullPath):
    print()
    try:
        print('--- The file %s is about to be parsed... ---' % FileName)
        TargetFileObj = open(os.path.join(TestSetFolderFullPath, FileName), mode='r')
        InfrastructureDict = yaml.load(TargetFileObj)
        TargetFileObj.close()
    except Exception as Error:
        print('--- Exception raised while processing file %s - Details: ---' % FileName)
        print('--- %s ---' % Error)
        shutil.move(os.path.join(TestSetFolderFullPath, FileName), RejectedFilesFolderFullPath)
        print('--- The file %s has been moved to the specified folder ---' % FileName)
```

Fig. B.2 Python script executed to prepare the Serverless Framework test set used for the infrastructure code queries (Part 2 of 2)

## B.2 Infrastructure code queries test set

This section lists the Serverless Framework examples that were included in the test set used with the infrastructure code queries (Table B.1, B.2, and B.3).

Table B.1 Serverless Framework test set for infrastructure code queries (Part 1 of 3)

| Sample Number | GitHub Link |
|---|---|
| 1 | aws-ffmpeg-layer |
| 2 | aws-golang-auth-examples |
| 3 | aws-golang-dynamo-stream-to-elasticsearch |
| 4 | aws-golang-googlemap |
| 5 | aws-golang-http-get-post |
| 6 | aws-golang-rest-api-with-dynamodb |
| 7 | aws-golang-s3-file-replicator |
| 8 | aws-golang-simple-http-endpoint |
| 9 | aws-golang-stream-kinesis-to-elasticsearch |
| 10 | aws-java-simple-http-endpoint |
| 11 | aws-multiple-runtime |
| 12 | aws-node |
| 13 | aws-node-alexa-skill |
| 14 | aws-node-auth0-cognito-custom-authorizers-api |
| 15 | aws-node-auth0-custom-authorizers-api |
| 16 | aws-node-dynamic-image-resizer |
| 17 | aws-node-dynamodb-backup |
| 18 | aws-node-env-variables |
| 19 | aws-node-env-variables-encrypted-in-a-file |
| 20 | aws-node-express-api |
| 21 | aws-node-express-dynamodb-api |
| 22 | aws-node-fetch-file-and-store-in-s3 |
| 23 | aws-node-fullstack |
| 24 | aws-node-function-compiled-with-babel |
| 25 | aws-node-github-check |
| 26 | aws-node-github-webhook-listener |
| 27 | aws-node-graphql-and-rds |
| 28 | aws-node-graphql-api-with-dynamodb |
| 29 | aws-node-heroku-postgres |
| 30 | aws-node-iot-event |
| 31 | aws-node-mongodb-atlas |
| 32 | aws-node-oauth-dropbox-api |
| 33 | aws-node-puppeteer |
| 34 | aws-node-recursive-function |

Table B.2 Serverless Framework test set for infrastructure code queries (Part 2 of 3)

| **Sample Number** | **GitHub Link** |
|---|---|
| 35 | aws-node-rekognition-analysis-s3-image |
| 36 | aws-node-rest-api |
| 37 | aws-node-rest-api-mongodb |
| 38 | aws-node-rest-api-typescript |
| 39 | aws-node-rest-api-typescript-simple |
| 40 | aws-node-rest-api-with-dynamodb |
| 41 | aws-node-rest-api-with-dynamodb-and-offline |
| 42 | aws-node-s3-file-replicator |
| 43 | aws-node-scheduled-cron |
| 44 | aws-node-scheduled-weather |
| 45 | aws-node-serve-dynamic-html-via-http-endpoint |
| 46 | aws-node-serverless-gong |
| 47 | aws-node-shared-gateway |
| 48 | aws-node-signed-uploads |
| 49 | aws-node-simple-http-endpoint |
| 50 | aws-node-simple-transcribe-s3 |
| 51 | aws-node-single-page-app-via-cloudfront |
| 52 | aws-node-sqs-worker |
| 53 | aws-node-stripe-integration |
| 54 | aws-node-telegram-echo-bot |
| 55 | aws-node-text-analysis-via-sns-post-processing |
| 56 | aws-node-twilio-send-text-message |
| 57 | aws-node-twitter-joke-bot |
| 58 | aws-node-typescript-apollo-lambda |
| 59 | aws-node-typescript-kinesis |
| 60 | aws-node-typescript-nest |
| 61 | aws-node-typescript-rest-api-with-dynamodb |
| 62 | aws-node-typescript-sqs-standard |
| 63 | aws-node-upload-to-s3-and-postprocess |
| 64 | aws-node-vue-nuxt-ssr |
| 65 | aws-node-websockets-authorizers |
| 66 | aws-python |
| 67 | aws-python-alexa-skill |
| 68 | aws-python-auth0-custom-authorizers-api |
| 69 | aws-python-flask-api |
| 70 | aws-python-flask-dynamodb-api |
| 71 | aws-python-line-echo-bot |
| 72 | aws-python-pynamodb-s3-sigurl |
| 73 | aws-python-rest-api |

Table B.3 Serverless Framework test set for infrastructure code queries (Part 3 of 3)

| Sample Number | GitHub Link |
|---|---|
| 74 | aws-python-rest-api-with-dynamodb |
| 75 | aws-python-rest-api-with-faunadb |
| 76 | aws-python-rest-api-with-pymongo |
| 77 | aws-python-rest-api-with-pynamodb |

## B.3 Application code queries test set

This section lists the Serverless Framework examples that were included in the test set used with the application code queries (Table B.4).

Table B.4 Serverless Framework test set for application code queries

| Sample Number | GitHub Link |
|---|---|
| 1 | aws-python |
| 2 | aws-python-alexa-skill |
| 3 | aws-python-auth0-custom-authorizers-api |
| 4 | aws-python-flask-api |
| 5 | aws-python-flask-dynamodb-api |
| 6 | aws-python-line-echo-bot |
| 7 | aws-python-pynamodb-s3-sigurl |
| 8 | aws-python-rest-api |
| 9 | aws-python-rest-api-with-dynamodb |
| 10 | aws-python-rest-api-with-faunadb |
| 11 | aws-python-rest-api-with-pymongo |
| 12 | aws-python-rest-api-with-pynamodb |
| 13 | aws-python-scheduled-cron |
| 14 | aws-python-simple-http-endpoint |
| 15 | aws-python-sqs-worker |
| 16 | aws-python-telegram-bot |

# Appendix C

# Serverless Framework test results

The aim of this appendix is to provide detailed information about the results of the test conducted with the Serverless Framework example applications (Section 5.4).

## C.1 Infrastructure code queries

This section includes a set of tables that show, for each query, the files where the warnings were raised and their number. It should be noted that:

- The query and the Serverless Framework example names were abbreviated by removing the initial strings `query_iac` and `serverless-aws`, respectively.
- In Table C.5 and in Table C.6, the Serverless Framework example names were further abbreviated by the removing the string `node-` as well.
- For clarity of presentation, the infrastructure code queries that did not flag any issues were not included in the tables below.

Table C.1 Number of warnings raised by the infrastructure code queries (Part 1 of 8)

| Infrastructure Code Query | golang-auth-examples.py | golang-googlemap.py | golang-http-get-post.py | golang-rest-api-with-dynamodb.py | golang-simple-http-endpoint.py | java-simple-http-endpoint.py | multiple-runtime.py |
|---|---|---|---|---|---|---|---|
| external_plugin | | | | | | | |
| functions_triggered_via_http | 2 | 3 | 3 | 5 | 2 | 1 | 2 |
| multiple_action_nested | | | | | | | |
| unprotected_environment_information | | 1 | | | | | |

Table C.2 Number of warnings raised by the infrastructure code queries (Part 2 of 8)

| Infrastructure Code Query | node-auth0-cognito-custom-authorizers-api.py | node-auth0-custom-authorizers-api.py | node-dynamic-image-resizer.py | node-env-variables-encrypted-in-a-file.py | node-env-variables.py | node-express-api.py | node-express-dynamodb-api.py |
|---|---|---|---|---|---|---|---|
| external_plugin | | 1 | 1 | 1 | | | |
| functions_triggered_via_http | 2 | 2 | 1 | | | 1 | 1 |
| multiple_action_nested | | | | | | | 1 |
| unprotected_environment_information | | | | | 1 | | |

Table C.3 Number of warnings raised by the infrastructure code queries (Part 3 of 8)

| **Infrastructure Code Query** | **node-fullstack.py** | **node-function-compiled-with-babel.py** | **node-github-check.py** | **node-github-webhook-listener.py** | **node-graphql-and-rds.py** | **node-graphql-api-with-dynamodb.py** | **node-heroku-postgres.py** |
|---|---|---|---|---|---|---|---|
| external_plugin | | 1 | 1 | | 1 | | |
| functions_triggered_via_http | 1 | | 1 | 1 | 2 | 1 | 1 |
| multiple_action_nested | | | | | | | |
| unprotected_environment_information | | | | 1 | | | |

Table C.4 Number of warnings raised by the infrastructure code queries (Part 4 of 8)

| **Infrastructure Code Query** | **node-mongodb-atlas.py** | **node-oauth-dropbox-api.py** | **node-puppeteer.py** | **node-rekognition-analysis-s3-image.py** | **node-rest-api-mongodb.py** | **node-rest-api-typescript-simple.py** | **node-rest-api-typescript.py** |
|---|---|---|---|---|---|---|---|
| external_plugin | | 1 | 1 | | | 1 | 1 |
| functions_triggered_via_http | 1 | 2 | 1 | 1 | 4 | 1 | 5 |
| multiple_action_nested | | | | | | | |
| unprotected_environment_information | | | | | | | |

Table C.5 Number of warnings raised by the infrastructure code queries (Part 5 of 8)

| **Infrastructure Code Query** | **rest-api-with-dynamodb-and-offline.py** | **rest-api-with-dynamodb.py** | **rest-api.py** | **s3-file-replicator.py** | **scheduled-weather.py** | **serve-dynamic-html-via-http-endpoint.py** | **serverless-gong.py** |
|---|---|---|---|---|---|---|---|
| external_plugin | 1 | | | 1 | | | 1 |
| functions_triggered_via_http | 5 | 5 | 1 | | | 1 | 1 |
| multiple_action_nested | | | | | | | |
| unprotected_environment_information | | | | | 6 | | |

Table C.6 Number of warnings raised by the infrastructure code queries (Part 6 of 8)

| **Infrastructure Code Query** | **signed-uploads.py** | **simple-http-endpoint.py** | **single-page-app-via-cloudfront.py** | **sqs-worker.py** | **stripe-integration.py** | **telegram-echo-bot.py** | **text-analysis-via-sns-post-processing.py** |
|---|---|---|---|---|---|---|---|
| external_plugin | 1 | | 1 | 1 | | | |
| functions_triggered_via_http | 1 | 1 | | 1 | 1 | 1 | 1 |
| multiple_action_nested | | | | | | | |
| unprotected_environment_information | | | | | | | |

Table C.7 Number of warnings raised by the infrastructure code queries (Part 7 of 8)

| **Infrastructure Code Query** | **node-twilio-send-text-message.py** | **node-typescript-apollo-lambda.py** | **node-typescript-kinesis.py** | **node-typescript-nest.py** | **node-typescript-rest-api-with-dynamodb.py** | **node-typescript-sqs-standard.py** | **node-vue-nuxt-ssr.py** | **python-auth0-custom-authorizers-api.py** |
|---|---|---|---|---|---|---|---|---|
| external_plugin | | 1 | 1 | 1 | | 1 | 1 | |
| functions_triggered_via_http | 1 | 1 | 1 | 1 | 4 | 1 | 1 | 2 |
| multiple_action_nested | | | | | | | | |
| unprotected_environment_information | 3 | | | | | | | 1 |

Table C.8 Number of warnings raised by the infrastructure code queries (Part 8 of 8)

| **Infrastructure Code Query** | **python-flask-api.py** | **python-flask-dynamodb-api.py** | **python-line-echo-bot.py** | **python-pynamodb-s3-sigurl.py** | **python-rest-api-with-dynamodb.py** | **python-rest-api-with-faunadb.py** | **python-rest-api-with-pymongo.py** | **python-rest-api-with-pynamodb.py** |
|---|---|---|---|---|---|---|---|---|
| external_plugin | 1 | 1 | 1 | 1 | | 1 | 1 | 1 |
| functions_triggered_via_http | 1 | 1 | 1 | 5 | 5 | 5 | 4 | |
| multiple_action_nested | | 1 | | | | | | |
| unprotected_environment_information | | | | | | 1 | | |