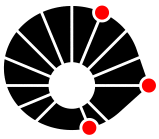

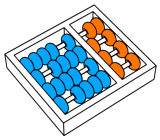

SLiTer: A Static Analysis Tool to Detect Security Smells in Terraform Configurations

Antonio Gabriel da Silva Fernandes
Breno Bernard Nicolau de França

Relatório Técnico - IC-PFG-23-40
Projeto Final de Graduação
2023 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

SLiTer: A Static Analysis Tool to Detect Security Smells in Terraform Configurations

Antonio Gabriel da Silva Fernandes Breno Bernard Nicolau de França

Abstract

Infrastructure as Code (IaC) is widely embraced for its ability to facilitate system infrastructure management, ensuring ease of modification and reproducibility. However, the inherent susceptibility of IaC configurations to security vulnerabilities necessitates specialized tools for code analysis. Building upon the work of Rahman *et al.*, who identified 7 security smells present in IaC scripts and introduced SLIC, a static analysis tool for identifying security smells in Puppet scripts, this paper presents SLiTer — a tool designed to detect the same security smells in Terraform files. By doing so, we developed two Rule Engines to serve distinct purposes: the first faithfully translated SLIC rules to establish a baseline, while the second incorporated modifications to enhance accuracy when applied to Terraform configurations. Evaluating SLiTer on 105 Terraform files from 15 directories revealed the most prevalent security smell as "Hard-coded secret," aligning with findings in the original work. SLiTer may prove valuable for practitioners seeking to identify general security smells in Terraform configurations, complementing other tools like Sonar or *tfparse* for provider-specific issues.

1 Introduction

Infrastructure as Code (IaC) is an approach to infrastructure automation based on software development practices [1]. Keeping all definitions for an application's infrastructure in configuration files makes the infrastructure more visible, allows for quick changes and testing, and, most importantly, makes the infrastructure reproducible. Practitioners consider IaC as a fundamental pillar to implement DevOps practices [2], and it is especially useful in deploying cloud-based infrastructures, which are ever more prevalent in Industry since they require reproducible and easily changeable configuration.

The IaC technology ecosystem is currently populated by a myriad of tools, often with overlapping purposes, with no single tool dominating the market [3]. Some of the most used IaC tools include Docker, Kubernetes, Chef, Ansible, Puppet, Terraform, AWS CloudFormation, and others. Some of these tools have more specific purposes, such as CloudFormation, which is specific for provisioning AWS services, while others aim to be expansible and deal with a wide range of use cases, such as Terraform. In summary, they all have their focuses, strengths, and weaknesses.

Although interest in IaC has been steadily growing, with technology giants such as Netflix and Mozilla adopting IaC in their products [4], there is still little research done regarding good security practices in IaC [2] [5]. IaC configurations can be susceptible to a

few types of security vulnerabilities that demand special attention [5], and thus it may be beneficial to use software tools to help verify that an infrastructure is secure.

In their work titled *The Seven Sins: Security Smells in Infrastructure as Code Scripts* [5], Rahman *et al.* identified seven types of security smells that frequently appear in IaC scripts, and developed a static analysis tool called Security Linter for Infrastructure as Code scripts (SLIC) that aims to detect such smells in Puppet scripts. However, since their tool is specifically built for the Puppet syntax, it cannot support IaC scripts made for other tools.

This project aims to develop a tool called Security Linter for Terraform (SLiTer), a static analysis tool for Terraform configurations that detects the same smells identified in [5]. We analyzed the original solution, adapted it for the Terraform technology, and performed tests based on open-source repositories, including Terraform configuration files.

The remaining sections of this work are organized as follows. Section 2 presents the background to IaC, Terraform, and security vulnerabilities. Section 3 presents the methodology used to build SLiTer. Section 4 describes SLiTer and how the detection rules were implemented. Section 5 details how we evaluated SLiTer and what results were obtained. Finally, Section 6 concludes this work.

2 Background

2.1 Infrastructure as Code

Infrastructure as Code (IaC) is the practice of describing deployments by means of machine-readable code [3]. That code can then be programmatically translated into infrastructure provisioning in a specific configuration. Interest in IaC has steadily grown, as shown in figure 1. This growth in the usage of IaC is mainly enabled by the popularization of public cloud computing, which allows practitioners to quickly provision and change the infrastructure of systems, with no concerns over the physical hardware involved.

The usage of IaC goes hand-in-hand with DevOps principles, which dictate a closer relationship between the development and operation of software, advocating for quick iterations and continuous integration.

2.2 Terraform

Terraform is an open source IaC tool created by HashiCorp [6]. It uses a declarative language called HashiCorp Configuration Language (HCL) to describe a desired infrastructure. One of its key features is the use of different providers that enable the expansion of Terraform to manage infrastructure across multiple public and private cloud providers, as well as virtualization platforms and on-premises servers.

Terraform projects are comprised of one or more Terraform (.tf) files. The basic unit in a Terraform file is a resource declaration, which specifies the state of a resource that must be provisioned and managed using Terraform providers. The files may also specify variables that may be changed upon applying the configuration to change certain parameters in

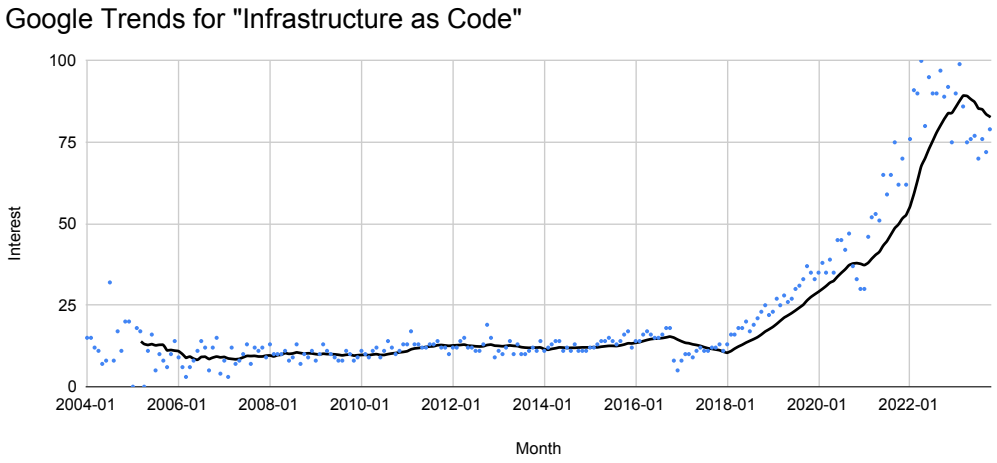

Figure 1: Google Trends data for the subject "Infrastructure as Code". Interest has been steadily rising since 2018.

deployed resources. They may also define output variables, which are values created during the execution of the configuration and must be given back to users.

Listing 1 shows an example of a Terraform project that provisions an Elastic Compute Cloud (EC2) instance on AWS. Lines 2 to 4 show the configuration of the AWS provider, specifying the region in which the instance will be provisioned. Lines 7 to 11 create an input variable called *instance_type* with the default value "t2.micro". Lines 14 to 17 effectively create the instance, specifying a name, an Amazon Machine Image (AMI), and using the value of the *instance_type* variable as one of the attributes. Finally, lines 20 to 22 define an output variable to inform the user about the public IP of the created instance.

```
1  # Define the AWS provider and specify your credentials and region
2  provider "aws" {
3    region = "us-east-1"
4  }
5
6  # Input variables
7  variable "instance_type" {
8    description = "The EC2 instance type"
9    type        = string
10   default     = "t2.micro" # Default instance type
11 }
12
13 # Create an EC2 instance
14 resource "aws_instance" "example" {
15   ami           = "ami-0dbc3d7bc646e8516"
16   instance_type = var.instance_type
17 }
18
19 # Output variable to display the public IP address of the created instance
20 output "public_ip" {
21   value = aws_instance.example.public_ip
22 }
```

Listing 1: Terraform configuration example

2.3 Code smells

Martin Fowler and Kent Beck introduced the concept of code smells in 1999 [7]. A code smell is a recurring coding pattern that does not necessarily cause an error but may indicate problems in the code that should be further investigated.

A security smell, as Rahman *et al.* described in their paper [5], is a recurring coding pattern indicative of security weakness and requires further inspection. A security smell may indicate the existence of a vulnerability in the software that malicious users could exploit.

An example of a security smell is the usage of HTTP without the Transport Layer Security (TLS). Without TLS, HTTP is susceptible to man-in-the-middle attacks, since the communication is unencrypted. However, the data may have already been encrypted before the transmission, so there would not be a security breach. Therefore, while the usage of HTTP without TLS does not necessarily mean a vulnerability, it should be handled carefully and may indicate the presence of a security issue.

2.4 Terraform Security Analysis Tools

Many tools in the market aim to find security issues in Terraform code via static analysis. Two of the most prominent solutions are Aqua Security's *tfsec* [8] and Sonar's [9] solution suite (SonarQube, SonarLint, and SonarCloud). Both solutions offer a wide range of rules that will be statically checked against input code, searching for security issues in several types of Terraform resources from different providers.

While these tools are excellent at finding more specific potential issues, SLiTer is more generalistic, and searches for issues that are not specific to Terraform but could be present in basically any IaC framework. *tfsec*, for example, has several rules searching for specific configuration problems in 36 AWS services, 14 Azure Cloud services, 9 Google Cloud Platform services, as well as CloudStack, Digital Ocean, Kubernetes, GitHub, OpenStack, and Oracle Cloud rules [10]. The only rule not related to a specific provider is a check for plaintext exposure of sensitive information [11], similar to the "Hard-coded secret" smell in SLIC and SLiTer.

Sonar has 51 rules used in static analysis for Terraform [12], most of which are also specific to a provider. However, there is a rule called "Track uses of 'TODO' tags", which is similar to the "Suspicious comments" smell in SLIC and SLiTer, and another called "Using clear-text protocols is security-sensitive", which is similar to the "Use of HTTP without TLS" smell. Other rules may have similar ideas to SLIC/SLiTer smells, but with very different and often provider-specific implementations, such as "Allowing public network access to cloud resources is security-sensitive," which is, in principle, similar to the "Invalid IP address binding" smell, but is implemented very differently and searches for specific resources from certain providers.

3 Methods

3.1 Understanding *The Seven Sins*

According to the authors of the baseline approach, the point of *The Seven Sins: Security Smells in Infrastructure as Code Scripts* [5] is to answer the following research questions:

- **RQ1:** What security smells occur in infrastructure as code scripts?
- **RQ2:** How frequently do security smells occur in infrastructure as code scripts?
- **RQ3:** What is the lifetime of the identified security smell occurrences for infrastructure as code scripts?
- **RQ4:** How do practitioners perceive the identified security smell occurrences?

To answer RQ1, they applied a qualitative analysis technique called descriptive coding [13] over 1726 Puppet configuration files, listing security smells present in those files and associating them with weaknesses defined in the Common Weakness Enumeration (CWE) [14]. The smells encountered in this step were separated into categories, each associated with a few CWE weaknesses. This process produced seven different smell categories: Admin by default, Empty password, Hard-coded secret, Invalid IP address binding, Suspicious comment, Use of HTTP without TLS, and Use of weak cryptography algorithms. The definitions of each smell are further detailed in Section 4.3.

To answer RQ2, the authors built SLIC, their static analysis tool. SLIC first uses a parser to turn the Puppet scripts into tokens, which are then fed through a Rule Engine that detects the presence of each smell. The Rule Engine uses syntactic information from the parser, describing what kind of tokens are present and the presence of certain string patterns within those tokens, to detect smells. The specific rules and string patterns will be detailed in section 4.3.

To evaluate SLIC, the authors built a test dataset of 140 scripts which were manually checked for smells by students. SLIC's results were then compared to the student's evaluation yielding an average precision and recall of 0.99.

The authors then applied SLIC on 4 datasets, containing Puppet scripts from GitHub, Mozilla, OpenStack, and Wikimedia repositories, respectively, adding up to a total of 15232 scripts from 293 repositories. The scripts were sourced from open source software (OSS) projects that followed specific criteria, such as a minimum number of contributors, commits per month, and proportion of IaC scripts relative to other content in the repository.

As a sanity check, the authors manually checked 250 of those scripts for smells and then compared their results to SLIC. The tool generated a few false positives and one false negative, especially in the "Hard-coded secret" and "Suspicious comment" smells, which presented 0.78 and 0.73 precision, respectively.

SLIC found occurrences of smells across all datasets, with 21202 total occurrences. The GitHub dataset had the largest density of smells per line of code and the largest proportion of scripts containing at least one smell. The most prevalent smell across all datasets was "Hard-coded secret", amounting to 80% of total occurrences, followed by "HTTP without

TLS" and "Suspicious comments" with about 8% of occurrences each. The other 4 smells combine to only about 4% of occurrences.

To answer RQ3, the authors applied SLIC on the same repositories over commits spanning several years and determined for how long the same smell with the same values would remain in the codebase. They determined that a smell can persist for as long as 98 months, allowing attackers to exploit potential security problems.

Finally, to answer RQ4, the authors randomly selected 1000 of the smells detected by SLIC and submitted a bug report explaining the smell and asking if contributors of the repository agreed to fix the smell instances. They received only 212 responses, and 69.8% agreed to fix the reported smells. "Use of weak cryptography algorithms" had the highest agreement ratio.

3.2 Attempts to Reproduce SLIC Results

The source code to SLIC was made available by the authors in a *figshare* repository [15]. The repository contains a ZIP file with the source code and 4 CSV files, one for each dataset used to answer RQ2, RQ3, and RQ4 of their work. However, the actual datasets, containing the Puppet scripts, are not available: the CSV files contain only the local names of the script files in the author's computer and how many occurrences of each smell there are in that file. There is also no easy way to find the original repositories, as there is no link in the CSV files, and the names have been altered. This made it, unfortunately, impossible to try to replicate the results described in the original work.

There were also issues in trying to run SLIC. The program is written in Python 2, which makes it much more difficult to correctly install and manage dependencies since pip no longer supports it as of version 20.3 [16]. It also depends on puppet-lint [17], which is a standalone tool that requires external installation, making the code even less portable.

Despite these issues, we ran SLIC on a few scripts and got the expected results. The program could identify a few smells on these test scripts, but we could not replicate the original authors' results without the original test repositories. The main utility of the SLIC source code was as a reference to how exactly the rules for smell checking were implemented so that we could replicate them in SLiTer.

The Python code checks files in the input directory, and all Puppet files detected are passed through the Rule Engine. There is code suggesting that at some point, the authors intended to also inspect Chef files using foodcritic [18], as shown in Listing 2, but for some reason decided not to include it in the final paper. The Puppet files are fed through puppet-lint, passing the custom rules defined in a ruby file as a parameter. The output from puppet-lint is then parsed and converted into the CSV outputs describing the number of occurrences and their locations.

```
1 #### CHEF ZONE
2 ### PUPPET LINT AND ITS RULES
3 CHEF_LINT_TOOL = 'foodcritic -I'
4 # CHEF_RULE_HARDCODE  = '../SPrules4pupp/no_hardcode_key.rb'
5 CHEF_ALL_RULES = '../SPrules4chef/my-rules/my_rules.rb'
6 CHEF_SWITCH_RULE = '../SPrules4chef/my-rules/special_missing_case.rb'
```

Listing 2: Lines 30-35 in the *constants.py* from the SLIC source, showing chef-related code.

3.3 Translating the Seven Sins to Terraform

After fully understanding the baseline solution, we developed SLiTer using Python 3.11. It follows the same basic premise as SLIC, feeding the scripts through a parser and then using a Rule Engine to detect smells. More details about how SLiTer works are explained in Section 4. The idea was to "translate" the rules considering Terraform specifications.

3.4 Mining Terraform Repositories

To test and evaluate SLiTer, we created a dataset of Terraform directories. The dataset comprises 15 directories sourced from 12 different OSS repositories, adding up to 105 Terraform files. The repositories were chosen based on the premise of using code from real-world projects, avoiding repositories with educational purposes and toy projects. Many repositories are from Terraform modules available on the Terraform Registry [19], while others are from open-source applications such as GitLab.

For acquiring the relevant Terraform directories from the repositories, we developed a Bash script called *git_sparse_clone.sh* that leverages the Git sparse-checkout feature [20] that makes it possible to clone only a specific directory in a repository. The script is available along with the SLiTer source code on GitHub [21].

3.5 Improving Original Rules

As the tool was being developed, it became clear that some changes could be made to improve the performance of the rules compared to the baseline. We then decided to make 2 Rule Engines: one, called the baseline, replicates the SLIC rules as closely as is reasonable given the language differences. The other has a few differences in the keywords used to detect rules and specific built-in cases that improve the overall accuracy, especially by lowering the number of false positives. Further details about the differences between the rule engines are available in section 4.

4 SLiTer

SLiTer (Security Linter for Terraform) is written in Python 3.11. The full SLiTer source code is available on GitHub [21]. It is composed of 4 modules:

- **main.py:** The program entry point, is responsible for listing the input directories, instantiating the Rule Engines, and generating the output files.
- **baseline_ruleengine.py:** Contains the `Baseline_RuleEngine` class, which implements the basic parse tree traversal functionality and the baseline rules, meant to follow the SLIC rules as closely as is reasonable.
- **sliter_ruleengine.py:** Contains the `SLiTer_RuleEngine` class, which inherits from the `Baseline_RuleEngine` and applies changes to improve the tool effectiveness.

REPO	SMELL	LOCATION
./terraform/gl-infra-packagecloud	ADMIN_BY_DEFAULT	cloudsql.tf.module.mysql[0].user_name

Table 1: Example of log entry for an Admin by Default smell occurrence.

- **hclparser.py:** Contains the `HCLParser` class, which takes a Terraform directory and returns to the Rule Engines the parse trees of all files in the form of dictionaries, as well as the comments extracted from the files.

The only dependencies of SLiTer outside of the Python Standard Library [22] are Docker [23] and the docker-py Python library [24]. The `HCLParser` uses Docker to run *hcl2json* [25], which converts the Terraform files into a JSON format which is then turned into a Python dictionary using Python's `json` module, to be verified by the Rule Engines. If unavailable in the local system, the *hcl2json* Docker image will be automatically pulled from DockerHub.

SLiTer currently sees all subdirectories within the `terraform/` directory in the repository as input. It will run through every file with the .tf extension in each of those subdirectories and produce four CSV files as output:

- **output_sliter.csv** and **output_baseline.csv**: How many occurrences of each smell are present in each input directory, as per the SLiTer Rule Engine and the Baseline Rule Engine, respectively. Each column corresponds to a smell type, and each line corresponds to an input directory.
- **log_sliter.csv** and **log_baseline.csv**: Where, in the parse tree, each smell occurrence was found, for the SLiTer Rule Engine and the Baseline Rule Engine, respectively. Each line corresponds to a smell occurrence and contains the input directory where the smell was found, the smell type, and the path to the smell location.

Table 1 and listing 3 show an example of how a smell may appear in a Terraform configuration, and how it shows in the log files when detected by SLiTer. In line 8, the code sets the default user as "admin", characterizing the Admin by Default smell. Thus, the path to the `user_name` attribute shows in the log file, as shown in table 1.

```
1  module "mysql" {
2    source  = "GoogleCloudPlatform/sql-db/google//modules/mysql"
3    version = "16.1.0"
4
5    name                 = "packagecloud"
6    random_instance_name = true
7
8    user_name = "admin"
9
10   db_name      = "packages_onpremise"
11   db_charset   = "utf8mb4"
12   db_collation = "utf8mb4_general_ci"
13   disk_size    = var.cloudsql_disk_size
14
15   ...
16
17 }
```

Listing 3: Example of the Admin by Default smell found in the *cloudsql.tf* file in the *gl-infra-packagecloud* test repository.

4.1 Parsing Terraform Files

The `HCLParser` class receives a Terraform directory with one or more .tf files, and its main purpose is to return two dictionaries via the `HCLParser.parse()`: one containing the parse tree for each file and another containing the comments extracted from the files.

The comments are extracted using regular expressions, with Python's `re` module. The expression detects all comment syntaxes allowed by Terraform [26]: both single-line comments beginning with `#` or `//` and multiline comments beginning with `/*` and ending with `*/`. If one of the comment starting sequences is inside a string literal in the Terraform code, delimited by double quotes (`"`), then it does not initiate a comment.

Through a two-step process, the `HCLParser` generates the parse trees from the Terraform files. First, it runs the files through *hcl2json*, creating a Docker container for each file. The output of those containers, a JSON file with the same structure as the original Terraform, is then transformed into a Python dictionary using Python's `json` module.

4.2 Traversing the Parse Tree

The Rule Engines, after using the `HCLParser` to get the comments and parse trees from the Terraform files, need to traverse those structures to search for smells. This is done in the `get_smells()` method. This process is done in two parts: first, the comments are traversed, searching for the Suspicious Comment smell, and then the parse trees are traversed, to find the other smell types. Comments are traversed sequentially and tested with the rules described in Section 4.3.5 to detect Suspicious Comments.

The parse trees are traversed using a recursive, depth-first approach. The nodes in the parse tree can be a dictionary, a list, or a leaf value (usually a string or number). The `Baseline_RuleEngine` class implements a method called `visit()`, which receives a node and, depending on the node type, calls either the `visit_dict()`, the `visit_list()` or the `visit_leaf()` method. Both `visit_dict()` and `visit_list()` recursively call `visit()` on their inner values while keeping track of the current path traversed in the tree with an attribute called `current_key`. When the algorithm reaches a leaf node, it tests its values to search for all 6 of the remaining smell types, and, if a smell is detected, the `current_key` is added to the list of occurrences of that smell. These methods can be seen in Listing 4.

4.3 Smells and Detection Rules

Rahman *et al.* identified seven types of security smells widely present in IaC scripts and developed their SLIC tool to detect them in Puppet scripts [5]. The smells are detected using custom rules passed to puppet-lint [17], using Ruby files. Several Ruby files are present in the source code made available by the authors, with names that suggest each file would contain one rule. However, the actual code only passes one file to puppet-lint, named *all_in_one.rb*, which contains the final version of the rules used in their work.

The rules in this file are not a one-to-one match with the smell types; sometimes, more than one rule composes a single smell. They also do not perfectly match the rules described in Section IV of the original paper [5], and the differences will be explained for each smell.

```python
def visit(self, node):
    if isinstance(node, dict):
        self.visit_dict(node)
    elif isinstance(node, list):
        self.visit_list(node)
    else:
        self.visit_leaf(node)

def visit_dict(self, node):
    for key, value in node.items():
        previous_key = self.current_key
        self.current_key += "." + key
        self.visit(value)
        self.current_key = previous_key

def visit_list(self, node):
    previous_key = self.current_key
    for i, item in enumerate(node):
        self.current_key = previous_key + f"[{i}]"
        self.visit(item)

def visit_leaf(self, node):
    location = self.current_key

    if isinstance(node, str):
        # Test for smell occurrences
```

Listing 4: *Baseline_RuleEngine* class methods implementing the tree traversal functionality

4.3.1 Admin by Default

This smell is the recurring pattern of default users as administrative users [5]. The authors associate this smell with CWE-250: Execution with Unnecessary Privileges. The authors claim this smell is detected by checking for the following pattern:

$$(isParameter(x)) \wedge (isAdmin(x.name) \wedge isUser(x.name)),$$

where $x.name$ is the name of the parameter (not the value), $isAdmin(x.name)$ checks whether the name contains the substring "admin" and $isUser(x.name)$ checks whether the name contains the substring "user". In practice, the actual logic in the `no_admin_by_default` check in the *all_in_one.rb* file is quite similar, however, it also triggers if the name of the parameter includes "user" as a substring and the *value* includes "admin" as a substring.

The rule for this smell in the Baseline Rule Engine follows this same idea, checking for the name to be a user and either the name or the value to be an admin, as in Listing 5.

```python
def test_admin_by_default(self, s: str) -> bool:
    latest_key = self.latest_key()
    constant_s = self.remove_variables(s).lower()
    return self.is_user(latest_key) and (self.is_admin(constant_s) or self.
        is_admin(latest_key))
```

Listing 5: Baseline Rule Engine check for the Admin by default smell.

The `is_admin()` method is the same as the original, merely checking for the "admin" substring. The `is_user()` method, however, returns true if the string contains "user" but *does not* contain the substring "provider", as shown in Table 2. This is done using the same

	User substrings	Negative user substrings	Admin substrings
SLIC	“user”	None	“admin”
Baseline Rule Engine	“user”	“provider”	“admin”
SLiTer Rule Engine	”user”	“provider”	“adm”, “root”, “superuser”

Table 2: Positive and negative substrings used to detect “user” and “admin” occurrences for the Admin by default smell in each Rule Engine.

`is_user()` method for both this smell and the Hard-coded secret smell, which makes this added check. In the 15 repositories used to test SLiTer, this addition does not change the occurrences found for this smell.

The SLiTer Rule Engine adds some extra words that may characterize the string as administrative: instead of only “admin”, the list is now “adm”, “root”, and “superuser”. The “adm” word also makes “admin” be detected, so this is, in fact, a superset of the original solution.

4.3.2 Empty Password

This smell is the recurring pattern of using a string of length zero for a password [5], making it very easy to guess. It is different from using no passwords, as other forms of authentication may be available that do not require a password. The authors relate this smell to CWE-258: Empty Password in Configuration File. The paper claims that the following rule detects this smell:

$$(isAttribute(x) \vee isVariable(x)) \wedge (length(x.value) == 0 \wedge isPassword(x.name)),$$

where $isPassword(x.name)$ is true if the name (not the value) of the attribute or variable contains one of the following substrings: “pwd”, “pass”, “password”. The actual `no_empty_pass` check used by SLIC actually works not only with strings of length zero but also with a string containing only a single space character (“ ”).

The rule for this smell in our Baseline Rule Engine works similarly, working with zero length or single space strings, as shown in listing 6. The `isPassword()` method, similarly to the `isUser()` method explained in section 4.3.1, returns True if one of the password substrings appears in the value, and the substrings ”provider” and ”passive” do not appear in it, as shown in table 3.

```
1 def test_empty_password(self, s: str) -> bool:
2         latest_key = self.latest_key()
3         if self.is_password(latest_key):
4             return (len(s) == 0) or (s == " ")
5         else:
6             return False
```

Listing 6: Baseline Rule Engine check for the Empty password smell.

The SLiTer rule engine slightly reworks this function so that, instead of only a single whitespace, any amount of whitespace is recognized as an empty string.

	Password substrings	Negative password substrings
SLIC	“pwd”, “password”, “pass”	None
Baseline Rule Engine	“pwd”, “password”, “pass”	“provider”, “passive”
SLiTer Rule Engine	“pwd”, “password”, “pass”	“provider”, “passive”

Table 3: Positive and negative substrings used to detect the empty password smell in each Rule Engine.

4.3.3 Hard-coded Secret

This smell is the recurring pattern of revealing sensitive information such as usernames and passwords as configurations in IaC scripts [5]. The authors consider three types of hard-coded secrets: passwords, user names, and private cryptography keys. The authors acknowledge that user names and SSH keys may be left in configuration files intentionally, and may not be sufficient to cause a security vulnerability, which makes these practices a security smell, and not an outright vulnerability. This smell is related to CWE-798: Use of Hard-coded Credentials and CWE-259: Use of Hard-coded Password. The paper uses the following rule to explain the detection of hard-coded secrets:

$$
\begin{aligned}
&((isAttribute(x) \vee isVariable(x)) \\
&\wedge(isUser(x.name) \vee isPassword(x.name) \vee isPvtKey(x.name)) \\
&\wedge(length(x.value) > 0))
\end{aligned}
$$

where $isUser(x.name)$ would work as described in section 4.3.1, searching for the “user” substring; $isPassword(x.name)$ would work as described in section 4.3.2, searching for the “password”, “pwd” or “pass” substrings; and $isPvtKey(x.name)$ would search for either “pvt” or “priv” followed by either “cert”, “key”, “rsa”, “secret” or “ssl”.

In practice, there is a total of 5 different checks implemented in the *all_in_one.rb* file related to this smell.

- **no_hardcode_secret_v1:** This check tests the following criteria:
  - The name of the variable contains one of the following substrings: “pwd”, “password”, “pass”, “key”, “crypt”, “secret”, “certificate”, “cert”, “ssh_key”, “md5”, “rsa”, “ssl”, “dsa”, “user”.
  - The variable’s name does not contain the substrings “passive” or “provider.
  - The value of the variable has a length greater than 1.
- **no_hardcode_secret_v2:** Checks if the value of the variable contains the substring “ssh-rsa”, likely checking for SSH private keys, as well as a length greater than zero.
- **no_hardcode_secret_uname:** Checks if the variable’s name contains “user”, does not contain “provider”, and the value has a length greater than 1.
- **no_hardcode_secret_password:** Checks if the name of the variable contains “pwd”, “pass” or “password”, and does not contain “provider” or ”passive”, and the value has a length greater than 1.

- **no_hardcode_secret_key:** Checks if:
  - The name of the variable contains one of the following substrings: “key”, “crypt”, “secret”, “certificate”, “cert”, “ssh_key”, “md5”, “rsa”, “ssl”, “dsa”.
  - The variable’s name does not contain the substring “provider”.
  - The value of the variable has a length greater than 1.

It appears the no_hardcode_secret_v1 check was meant to encompass all secret types, while no_hardcode_secret_v2 is an inclusion to detect SSH keys by the value, and the other three checks are meant to break down the findings of the first main check. However, if the source code made available is the final version actually used to get to the findings in the paper, all of the checks are contributing to the ”Hardcoded secret” count equally, which would make most of the occurrences counted twice.

In the Baseline Rule Engine for SLiTer, the check combines no_hardcode_secret_v1 and no_hardcode_secret_v2, checking for the same substrings. The logic for the rule is shown in Listing 7.

```
def test_hard_coded_secret(self, s: str) -> bool:
    latest_key = self.latest_key()
    constant_s = self.remove_variables(s).lower()
    if len(constant_s) > 0:
        is_secret = self.is_user(latest_key) or self.is_password(latest_key)
        is_secret = is_secret or self.is_pvt_key(latest_key) or ("ssh-rsa"
            in constant_s)
        return is_secret
    else:
        return False
```

Listing 7: Baseline Rule Engine check for the Hard-coded secret smell.

The SLiTer Rule Engine makes key changes to this smell’s detection. First, the substrings used to detect private keys are changed, with the most relevant change being removing the substring “key”, which generated too many false positives. Also, a Terraform-specific condition was added: SLiTer checks for the usage of the `tls_private_key` resource type. This resource is a quick way to create a PEM formatted private key, meant for easily bootstrapping throwaway development environments, and should not be used in production environments [27]. Therefore, it fits well within this smell, possibly exposing a private key in production. Lastly, the check will discard an occurrence if the latest part of the path to the scanned value is “description”. This is to avoid the descriptions of variables in Terraform, stating that “this is a private SSH key”, for example, triggering the smell.

4.3.4 Invalid IP Address Binding

This smell is the recurring pattern of assigning the IP address 0.0.0.0 for a database server or a cloud service/instance [5]. This may cause issues by allowing the instances to be accessible from any network. The authors correlate this smell to CWE-284: Improper Access Control. The rule presented in the paper for this smell is the following:

$$((isVariable(x) \vee isAttribute(x)) \wedge isInvalidBind(x.value)),$$

where $isInvalidBind(x.value)$ simply checks if "0.0.0.0" is a substring of the value. The actual `no_full_binding` check passed to puppet-lint by SLIC flags every occurrence of the "0.0.0.0" string found by the linter.

The rule for detecting this smell works the same in the Baseline and SLiTer Rule Engines, which flags every time the substring "0.0.0.0" appears in a value, as shown in listing 8.

```
1 def test_invalid_IP_binding(self, s: str) -> bool:
2     return ("0.0.0.0" in s.lower())
```

Listing 8: Baseline Rule Engine check for the Invalid IP address binding smell.

4.3.5 Suspicious comment

This smell is the recurring pattern of putting information in comments about defects, missing functionality, or system weakness [5]. This may give hints to malicious users about possible vulnerabilities. These are indicated by keywords such as "TODO" or "FIXME" at the beginning of comments. The paper claims the following rule is used for this smell:

$$(isComment(x) \wedge (hasWrongWord(x) \vee hasBugInfo(x))),$$

where $hasWrongWord(x)$ searches for the substrings "bug", "hack", "fixme", "later", "later2" and "todo"; and $hasBugInfo(x)$ searches for the patterns "bug[#nt]*[0-9]+" and "show bug\.cgi?id=[0-9]+". In the actual `no_susp_comments` check, the list of keywords is slightly expanded to include "ticket", "launchpad" and "to-do", and the check does not consider comments with the word "debug". The patterns from $hasBugInfor(x)$ are not present, but since they both contain the word "bug" they are already detected anyway.

The check in the Baseline Rule Engine is the same, containing the same keywords and logic. The SLiTer Rule Engine changes the word list, removing duplicates (such as "later2", which is already detected by "later") and including "pending", "missing" and "note".

4.3.6 Use of HTTP without TLS

This smell is the recurring pattern of using HTTP without the Transport Layer Security (TLS) [5]. This may make the communication susceptible to man-in-the-middle attacks. The authors correlate this smell to CWE-319: Cleartext Transmission of Sensitive Information. The rule described in the paper for this smell is as follows:

$$((isAttribute(x) \vee isVariable(x)) \wedge isHTTP(x.value)),$$

where $isHTTP(x.value)$ checks for the presence of substring "http:". The actual `no_http` check used by SLIC, however, checks for the substring "http://", with two slashes in the end. The rules in the Baseline and the SLiTer Rule Engines work the same way, searching for "http://", as shown in Listing 9.

```
1 def test_HTTP_without_TLS(self, s: str) -> bool:
2     return ("http://" in s.lower())
```

Listing 9: Baseline Rule Engine check for the HTTP without TLS smell.

4.3.7 Use of Weak Cryptography Algorithms

This smell is the recurring pattern of using weak cryptography algorithms, such as MD5 and SHA-1, for encryption purposes [5]. Those algorithms are not appropriate for encryption and are susceptible to attacks. The authors associate this smell with CWE-327: Use of a Broken or Risky Cryptographic Algorithm and CWE-326: Inadequate Encryption Strength. The rule presented in the paper for this smell is the following:

$$(isFunction(x) \wedge usesWeakAlgo(x.name)),$$

where $usesWeakAlgo(x.name)$ searches for the substrings "md5" and "sha1". This is accurate to check `no_md5` used by SLIC, which uses those same keywords.

The Baseline Rule Engine implements the rule similarly to SLIC, using only those two keywords. The SLiTer Rule Engine, however, expands that list based on the Open Worldwide Application Security Project's (OWASP) recommendations for testing for weak encryption [28]. The expanded list is: "md4", "md5", "rc4", "rc2", "blowfish", "sha1", "sha-1", "sha_1", " des ", "_des_", "-des-". Some variations of the names were added to account for ways they may appear in code, to avoid false positives, and other algorithms were added.

4.4 Implementation Challenges

One of the biggest challenges in developing SLiTer was finding an appropriate parser for Terraform. Early in development, we used *tfparse* [29], since it seems like a well-used parser and is directly available as a Python module. However, using this parser brought several problems. *tfparse* performs variable resolution wherever it can, so if a smell is present in the default value of a variable, it will also be present in every location where this variable is used. Also, if a resource is declared with a "count" meta-argument, determining that multiple instances of this resource should be created, *tfparse* displays that quantity of resources in the parse tree, instead of simply representing a single node with a "count" attribute. Combining those two features means that, if a variable contains a smell, and that variable is used in a resource definition with a "count" of 1000, that will make this same smell, with a single point of origin, appear 1000 times in the parse tree. This creates too much noise and makes it difficult to accurately identify how many smells are present in a project.

Our solution was to adopt *hcl2json*, which despite not being available directly as a Python module, requiring the extra step of running it through Docker (or installing it locally), makes an accurate and simple translation of the Terraform files into JSON. It simply represents variable references without resolving them and does not create multiple instances of a resource with a "count" meta-argument.

Another issue encountered was finding a way to parse the comments in Terraform. Many tools easily available to extract comments from source code, such as *comment_parser* [30], only support a few specific languages, and HCL/Terraform is usually unavailable. Comment detection is also not so simple due to the existence of both single-line and multi-line comments, and the fact that characters that would typically begin a comment do not do so if they are inside a literal string. Ultimately, our solution of using different groupings of regular expressions to match string literals and match comments was inspired by the solution employed in *comment_parser*'s source code to extract C-style comments.

5 Evaluation

5.1 Overview of the Test Repositories

The 105 Terraform files used to test and evaluate SLiTer are divided into 15 Terraform directories, which come from 12 different repositories. Eight are hosted on GitHub and the other four on GitLab. All the repositories hosted on GitLab are from the GitLab company production infrastructure. Four of the GitHub projects are Terraform modules available on the Terraform Registry, and the other four are from other open-source projects. Figure 2 shows the source distribution.

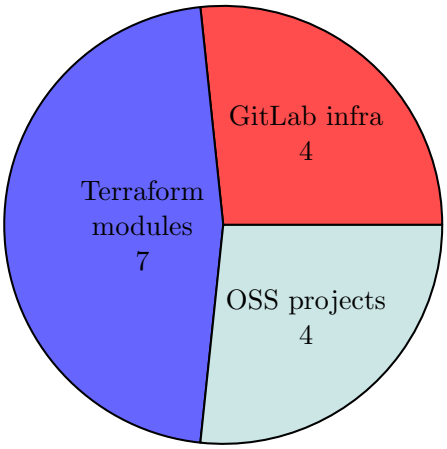

Figure 2: Distribution of the origins of the Terraform test directories.

5.2 Results

Tables 4 and 5 show the Baseline and SLiTer Rule Engines' outputs for the 15 directories. For both engines, at least one occurrence of each smell was found, except for the "Empty Password" smell. The most frequent smell was "Hard-coded Secret". For the Baseline, there were six directories in which zero smells were detected; for the SLiTer, there were 7.

These results are in line with the original work, which also found "Hard-coded secrets" to be the most prevalent smell. However, SLiTer found a much higher proportion of "Invalid IP Bindings", and a smaller proportion of "HTTP without TLS" than SLIC.

The only difference between the engines' results is that the SLiTer Rule Engine found four less hard-coded secrets and four more suspicious comments than the Baseline. The hard-coded secret differences are all due to removing the keyword "key" from the list of words used to detect private keys. All of the hard-coded secrets that are no longer detected were false positives, as exemplified in Listing 10, in which the Baseline Rule Engine detected line 8 as a hard-coded secret due to the "key" keyword.

The suspicious comments the SLiTer Rule Engine detected, but the Baseline did not, are all due to the inclusion of the keyword "note" as one of the possible words that indicate a suspicious comment. An example can also be found in Listing 10, at line 3.

Test Directory	Admin by Default	Empty Password	Hard-coded Secret	Invalid IP Binding	Suspicious Comments	HTTP without TLS	Weak Crypto. Algo.
lacework-terraform-aws-iam-role/	0	0	0	0	0	0	0
gl-infra-aws-nessus-scanner/	0	0	0	1	0	0	0
terraform-aws-modules-terraform-aws-eks/	0	0	2	2	3	0	3
grafana-loki/	0	0	0	0	0	0	0
gl-infra-packagecloud/	1	0	1	0	0	0	0
apache-pulsar/	0	0	0	7	0	1	0
gruntwork-io-internal-lb/	0	0	0	0	0	0	0
babbel-terraform-aws-lambda-with-inline-code/	0	0	2	1	0	0	1
samuelclay-NewsBlur/	0	0	1	0	0	0	0
jpetazzo-ampernetacle/	0	0	2	5	1	0	0
gl-infra-dns-record/	0	0	0	0	0	0	0
gl-infra-vault/	0	0	20	0	0	1	0
gruntwork-io-network-lb/	0	0	0	0	0	0	0
gruntwork-io-http-lb/	0	0	0	0	0	0	0
gruntwork-io-google-lb/	0	0	2	0	0	0	0
Total	1	0	30	16	4	2	4

Table 4: Output for the Baseline Rule Engine, showing the number of occurrences for each smell in each test repository.

Test Directory	Admin by Default	Empty Password	Hard-coded Secret	Invalid IP Binding	Suspicious Comments	HTTP without TLS	Weak Crypto. Algo.
lacework-terraform-aws-iam-role/	0	0	0	0	0	0	0
gl-infra-aws-nessus-scanner/	0	0	0	1	0	0	0
terraform-aws-modules-terraform-aws-eks/	0	0	1	2	7	0	3
grafana-loki/	0	0	0	0	0	0	0
gl-infra-packagecloud/	1	0	1	0	0	0	0
apache-pulsar/	0	0	0	7	0	1	0
gruntwork-io-internal-lb/	0	0	0	0	0	0	0
babbel-terraform-aws-lambda-with-inline-code/	0	0	1	1	0	0	1
samuelclay-NewsBlur/	0	0	0	0	0	0	0
jpetazzo-ampernetacle/	0	0	2	5	1	0	0
gl-infra-dns-record/	0	0	0	0	0	0	0
gl-infra-vault/	0	0	20	0	0	1	0
gruntwork-io-network-lb/	0	0	0	0	0	0	0
gruntwork-io-http-lb/	0	0	0	0	0	0	0
gruntwork-io-google-lb/	0	0	1	0	0	0	0
Total	1	0	26	16	8	2	4

Table 5: Output for the SLiTer Rule Engine, showing the number of occurrences in each test repository.

```
1  module "kms" {
2    source  = "terraform-aws-modules/kms/aws"
3    version = "1.1.0" # Note - be mindful of Terraform/provider version
         compatibility between modules
4
5    create = local.create && var.create_kms_key && local.
         enable_cluster_encryption_config # not valid on Outposts
6
7    description             = coalesce(var.kms_key_description, "${var.
         cluster_name} cluster encryption key")
8    key_usage               = "ENCRYPT_DECRYPT"
9    deletion_window_in_days = var.kms_key_deletion_window_in_days
10
11   ...
12
13 }
```

Listing 10: False positive for the "Hard-coded Secret" smell found by the Baseline Engine in the *terraform-aws-modules-terraform-aws-eks* directory, line 8 due to the "key" keyword.

5.3 Limitations

The dataset used to achieve these results is relatively small and comes from various sources. This may impact the generalizability of these findings, and it could prove beneficial for future works to apply SLiTer to larger and more diverse datasets.

Although SLiTer was developed based on SLIC and the findings in *The Seven Sins* [5], comparing their results is difficult because of two main factors. First, the simple fact of switching to a different domain of IaC, from Puppet to Terraform, can already cause inherent changes in the way that practitioners deal with security issues, and so different problems may appear more or less frequently. Also, the dataset used to achieve the results in this paper is much smaller than the original, which may cause the results to be more biased.

The idea of searching for smells based on certain patterns with keywords is also somewhat restrictive, and other, more sophisticated approaches that take more context into consideration may lead to better results. SLiTer could also benefit from further investigation on optimizing the keywords used to improve its accuracy. Also, as Rahman *et al.* explained in their work, there may be other relevant smells besides the 7 detected by SLIC and SLiTer.

SLiTer is also subject to the limitations of its dependencies, most notably *hcl2json* [25], which, as its creator states, may not always create a perfect translation of the Terraform file into JSON, especially if the target files use static analysis.

6 Conclusion

IaC is a great tool for ensuring that a system's infrastructure is easy to change and reproducible. However, IaC configurations are susceptible to vulnerabilities that demand specific attention, making it beneficial to use specialized tools to check the code for possible security issues. In *The Seven Sins: Security Smells in Infrastructure as Code Scripts*, Rahman *et*

al. developed SLIC, a static analysis tool that aims to identify 7 different types of security smells in Puppet scripts. Based on their work, we created SLiTer, a static analysis tool that detects the same smells in Terraform files. We developed two Rule Engines: one to serve as a baseline, faithfully translating the SLIC rules, and another with modifications to improve the tool's accuracy on Terraform. We mined 15 Terraform directories for testing SLiTer, adding up to 105 Terraform files. We found that the most prevalent smell in our test files was "Hard-coded secret", following the results found in the original work.

SLiTer could be very useful for practitioners in checking for these more generalistic security smells in Terraform configurations, while also using another tool like Sonar or *tfparse* for searching for provider-specific issues.

Future work on SLiTer could investigate other improvements that could be made to improve its accuracy, by testing it on a larger dataset. Other smells could also be added, as SLiTer is built, so changing the rules and adding new ones is fairly simple.

References

[1] K. Morris, *Infrastructure as Code*. O'Reilly Media, 2020. [Online]. Available: https://books.google.com.br/books?id=UW4NEAAAQBAJ

[2] A. Rahman, R. Mahdavi-Hezaveh, and L. Williams, "A systematic mapping study of infrastructure as code research," *Information and Software Technology*, vol. 108, pp. 65–77, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584918302507

[3] M. Guerriero, M. Garriga, D. A. Tamburri, and F. Palomba, "Adoption, support, and challenges of infrastructure-as-code: Insights from industry," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 580–589.

[4] C. Parnin, E. Helms, C. Atlee, H. Boughton, M. Ghattas, A. Glover, J. Holman, J. Micco, B. Murphy, T. Savor, M. Stumm, S. Whitaker, and L. Williams, "The top 10 adages in continuous deployment," *IEEE Software*, vol. 34, no. 3, pp. 86–95, 2017.

[5] A. Rahman, C. Parnin, and L. Williams, "The seven sins: Security smells in infrastructure as code scripts," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 164–175.

[6] Y. Brikman, *Terraform: Up and Running*. O'Reilly Media, 2022. [Online]. Available: https://books.google.com.br/books?id=dFaKEAAAQBAJ

[7] M. Fowler and K. Beck, *Refactoring: Improving the Design of Existing Code*, ser. Addison-Wesley object technology series. Addison-Wesley, 1999. [Online]. Available: https://books.google.com.br/books?id=UTgFCAAAQBAJ

[8] aquasecurity/tfsec: Security scanner for your terraform code. [Online]. Available: https://github.com/aquasecurity/tfsec

[9] Clean code: Writing clear, readable, understandable, & reliable quality code — sonar. [Online]. Available: https://www.sonarsource.com

[10] tfsec. [Online]. Available: https://aquasecurity.github.io/tfsec/v1.28.4/

[11] Secret/sensitive data should not be exposed in plaintext. [Online]. Available: https://aquasecurity.github.io/tfsec/v1.28.4/checks/general/secrets/no-plaintext-exposure/

[12] Terraform static code analysis. [Online]. Available: https://rules.sonarsource.com/terraform/

[13] J. Saldana, *The Coding Manual for Qualitative Researchers*. SAGE Publications, 2015. [Online]. Available: https://books.google.com.br/books?id=ZhxiCgAAQBAJ

[14] Cwe - common weakness enumeration. [Online]. Available: https://cwe.mitre.org

[15] A. Rahman, "The Seven Sins: Security Smells in Infrastructure as Code Scripts," 1 2019. [Online]. Available: https://figshare.com/articles/dataset/Dataset_The_Seven_Sins-Security_Smells_in_Infrastructure_as_Code_Scripts/6943316

[16] Release process - pip documentation v23.3.1. [Online]. Available: https://pip.pypa.io/en/stable/development/release-process/

[17] puppetlabs/puppet-lint: Check that your puppet manifests conform to the style guide. [Online]. Available: https://github.com/puppetlabs/puppet-lint

[18] Foodcritic/foodcritic: Lint tool for chef cookbooks. [Online]. Available: https://github.com/foodcritic/foodcritic

[19] Terraform registry. [Online]. Available: https://registry.terraform.io

[20] Git - git-sparse-checkout documentation. [Online]. Available: https://git-scm.com/docs/git-sparse-checkout

[21] bilbosf/sliter: Static analysis tool to detect security smells in terraform scripts. [Online]. Available: https://github.com/bilbosf/SLiTer

[22] The python standard library — python 3.11.6 documentation. [Online]. Available: https://docs.python.org/3.11/library/index.html

[23] Docker - accelerated container application development. [Online]. Available: https://www.docker.com

[24] docker/docker-py: A python library for the docker engine api. [Online]. Available: https://github.com/docker/docker-py

[25] tmccombs/hcl2json: Convert hcl2 to json. [Online]. Available: https://github.com/tmccombs/hcl2json

[26] Syntax - configuration language — terraform — hashicorp developer. [Online]. Available: https://developer.hashicorp.com/terraform/language/syntax/configuration#comments

[27] tls_private_key — resources — hashicorp/tls — terraform — terraform registry. [Online]. Available: https://registry.terraform.io/providers/hashicorp/tls/latest/docs/resources/private_key

[28] Wstg - v4.2 — owasp foundation. [Online]. Available: https://owasp.org/www-project-web-security-testing-guide/v42/4-Web_Application_Security_Testing/09-Testing_for_Weak_Cryptography/04-Testing_for_Weak_Encryption

[29] cloud-custodian/tfparse: python extension for terraform hcl parsing. [Online]. Available: https://github.com/cloud-custodian/tfparse

[30] jeanralphaviles/comment_parser: Python module to extract comments from source code files of various types. [Online]. Available: https://github.com/jeanralphaviles/comment_parser