Latest updates: https://dl.acm.org/doi/10.1145/3551349.3556945

RESEARCH-ARTICLE

GLITCH: Automated Polyglot Security Smell Detection in Infrastructure as Code

NUNO SAAVEDRA, University of Lisbon, Lisbon, Lisbon, Portugal

JOÃO F. FERREIRA, University of Lisbon, Lisbon, Lisbon, Portugal

Open Access Support provided by:

University of Lisbon

PDF Download
3551349.3556945.pdf
17 February 2026
Total Citations: 23
Total Downloads: 785

Published: 10 October 2022

Citation in BibTeX format

ASE '22: 37th IEEE/ACM International Conference on Automated Software Engineering
October 10 - 14, 2022
MI, Rochester, USA

ASE '22: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (October 2022)
https://doi.org/10.1145/3551349.3556945
ISBN: 9781450394758

GLITCH: Automated Polyglot Security Smell Detection in Infrastructure as Code

Nuno Saavedra
nuno.saavedra@tecnico.ulisboa.pt
INESC-ID and IST, University of Lisbon
Lisbon, Portugal

João F. Ferreira
joao@joaoff.com
INESC-ID and IST, University of Lisbon
Lisbon, Portugal

ABSTRACT

Infrastructure as Code (IaC) is the process of managing IT infrastructure via programmable configuration files (also called IaC scripts). Like other software artifacts, IaC scripts may contain security smells, which are coding patterns that can result in security weaknesses. Automated analysis tools to detect security smells in IaC scripts exist, but they focus on specific technologies such as Puppet, Ansible, or Chef. This means that when the detection of a new smell is implemented in one of the tools, it is not immediately available for the technologies supported by the other tools — the only option is to duplicate the effort.

This paper presents an approach that enables consistent security smell detection across different IaC technologies. We conduct a large-scale empirical study that analyzes security smells on three large datasets containing 196,755 IaC scripts and 12,281,251 LOC. We show that all categories of security smells are identified across all datasets and we identify some smells that might affect many IaC projects. To conduct this study, we developed GLITCH, a new technology-agnostic framework that enables automated polyglot smell detection by transforming IaC scripts into an intermediate representation, on which different security smell detectors can be defined. GLITCH currently supports the detection of nine different security smells in scripts written in Ansible, Chef, or Puppet. We compare GLITCH with state-of-the-art security smell detectors. The results obtained not only show that GLITCH can reduce the effort of writing security smell analyses for multiple IaC technologies, but also that it has higher precision and recall than the current state-of-the-art tools.

CCS CONCEPTS

• **Software and its engineering → Software configuration management and version control systems**; **Software maintenance tools**; • **Security and privacy → Software and application security**.

KEYWORDS

devops, infrastructure as code, security smells, Ansible, Chef, Puppet, intermediate model, static analysis

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASE '22, October 10–14, 2022, Rochester, MI, USA
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3556945

ACM Reference Format:
Nuno Saavedra and João F. Ferreira. 2022. GLITCH: Automated Polyglot Security Smell Detection in Infrastructure as Code. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3551349.3556945

1 INTRODUCTION

Infrastructure as Code (IaC) is a process which has been progressively gaining more adoption in the DevOps world since it facilitates the provision of scalable and reproducible environments. In current practice, the use of IaC scripts is essential to efficiently maintain servers and development environments. For example, according to Rahman et al. [18], Intercontinental Exchange (ICE), a Fortune 500 company, maintains 75% of its 20,000 servers using IaC scripts. The use of IaC scripts has helped ICE decrease the time needed to provision development environments from 1–2 days to 21 minutes.

Despite its many benefits, IaC scripts may contain defects that can have serious implications. For instance, due to bugs in their IaC scripts, GitHub experienced an outage of their DNS infrastructure [2] and Amazon Web Services lost around 150 million USD after issues with their S3 billing system [5]. To address this, there has been an effort by the research community to categorize and identify defects, and in particular, so-called security smells, which are coding patterns that can result in security weaknesses [15, 18–20].

Even when a security smell does not lead to a security breach, it deserves attention and inspection. It is thus important to develop automated methods that can assist developers identifying security smells in their IaC scripts. Two influential automated tools developed by the research community are SLIC [18], which supports seven types of security smells in Puppet[1] scripts, and SLAC [19], which supports nine types of security smells in Chef[2] scripts and six types in Ansible[3] scripts. These tools are very valuable, since they cover a wide range of security smells and three of the major IaC technologies. However, their implementations are separate and involve substantial duplication. If one wishes to implement the detection of a new smell, one has to develop a different implementation for each of the IaC technologies supported. Consequently, it is often the case that the detection of security smells is inconsistent for different IaC technologies. Figure 1a presents part of a Chef script with no security smells taken from the project Vagrant Chef for CakePHP.[4] For this example, SLAC reports a false positive: a non-existent security smell of type *Hard-coded secret*. On the other

[1] https://puppet.com
[2] https://www.chef.io
[3] https://www.ansible.com
[4] https://github.com/FriendsOfCake/vagrant-chef/blob/288336e506a5009ed93c06a784fa93e30a27040c/cookbooks/percona/recipes/server.rb#L28

hand, if we consider the same script in Puppet (Figure 1b), SLIC will not report any security smell. Surprisingly, inconsistencies exist even when considering the same tool: SLAC will not report any security smell when considering the same script in Ansible (this happens because SLAC uses separate code for Ansible and Chef).

These inconsistencies would not occur if we had polyglot defect prediction and debugging environments for IaC, a direction recently proposed by Alnafessah et al. [1]. Also, a problem that has been observed by Guerriero et al., after interviews with IaC experts, is that the IaC technology ecosystem is currently very scattered and heterogeneous [3]. Guerriero et al. also identified the need to develop more IaC development tools, such as static analysis tools and security-related tools. It is thus clear that it would be beneficial to develop unifying methods that can reduce inconsistencies.

This paper presents an approach that enables consistent security smell detection across different IaC technologies. We conduct a large-scale empirical study that analyzes security smells on three large datasets containing Ansible, Chef, and Puppet scripts. We show that all categories of security smells are identified across all datasets and we identify some smells that might affect many IaC projects. To conduct this study, we developed GLITCH, a new technology-agnostic framework that enables automated polyglot smell detection by transforming IaC scripts into an intermediate representation, on which different security smell detectors can be defined. GLITCH currently supports the detection of nine different security smells and it can analyze scripts written in Puppet, Ansible, or Chef. We compare GLITCH with the state-of-the-art security smell detectors SLIC [18] and SLAC [19]. The results obtained not only show that GLITCH can reduce the effort of writing security smell analyses for multiple IaC technologies, but also that it has higher precision and recall than the current state-of-the-art tools.

Contributions. Our main contributions are: **(1)** A new intermediate representation that can be used to model IaC scripts and on which security smell detection rules can be defined. **(2)** The implementation of a framework called GLITCH that is able to transform IaC scripts written in Ansible, Chef, or Puppet into the new intermediate representation, and that supports the detection of nine security smells. We show that the average precision and recall values of GLITCH are substantially better than the average precision and recall values of state-of-the-art tools. **(3)** An empirical study that investigates how frequently security smells occur in IaC scripts. We consider Ansible, Chef, and Puppet scripts. We use three large datasets containing 196,755 IaC scripts and 12,281,251 LOC. We show that all categories of security smells are identified across all datasets and we identify some smells that might affect many IaC projects. **(4)** A replication package containing all the datasets used in this work, including three oracle datasets that were manually annotated. We tried to use replication packages from other authors, but they were lacking data. As a result, to the best of our knowledge, our replication package is the first to be complete and available. It is available as a Docker container at **https://doi.org/10.6084/m9.figshare.19726603.v2**

GLITCH is open source and available from GitHub:

https://github.com/sr-lab/GLITCH

Characteristic	Ansible	Chef	Puppet
Conf. Setup	Push	Pull	Pull
Add. Agent	No	Yes	Yes
Syntax	YAML	Ruby	Puppet DSL
Exec. Order	Procedural	Procedural	Declarative
Atomic Unit	Task	Resource	Resource
Code Structure	Roles - Playbooks - - Plays - - - Tasks	Cookbooks - Recipes - - Resources	Modules - Manifests - - Classes - - - Resources

Table 1: Summary of Ansible, Chef and Puppet's characteristics.

2 BACKGROUND AND RELATED WORK

We focus on IaC tools for configuration management of services. The main reason is that the ecosystem around this category of tools is heterogeneous with several technologies widely adopted. Guerriero et al. [3] listed four technologies in this category that are adopted by industry experts: Ansible, Chef, Puppet, and Saltstack. Out of these four, three of them had an adoption rate greater than 29%: Puppet with 29.5%, Chef with 36.3%, and Ansible with 52.2% (note that industry experts can adopt more than one technology simultaneously). To maximize impact of our work, we focus on these three technologies.

2.1 Ansible, Chef, and Puppet Scripts

We provide a brief background on Ansible, Chef, and Puppet scripts. Table 1 summarizes and compares some relevant characteristics of these technologies. There are two types of *configuration setups* for IaC technologies: push and pull. In a push configuration setup, the sysadmin commands a centralized server, able to connect to every node, to provide the configuration to a set of nodes. In a pull configuration setup, each node periodically contacts the server to retrieve the latest configuration for that particular machine. Technologies may require an *additional agent* to be installed in every node. The agent is a program that runs as a background service and is capable of doing the necessary operations in the nodes (e.g., updates). Regarding, *syntax*, IaC technologies use different programming languages. Ansible uses YAML, Chef uses Ruby, and Puppet uses a domain-specific language (DSL). Using a programming language like Ruby allows complex programs to be written. However, it may be more difficult to abstract the concepts being represented. Ansible and Chef encourage a procedural style, which means that scripts follow, in order, a sequence of instructions specified by practitioners. On the other hand, Puppet uses a declarative style, in which practitioners specify the desired state and it is up to the Puppet tool to decide how the state is achieved. Regarding atomic units and code structure, while Ansible considers the notion of Task as the atomic unit, both Chef and Puppet use the notion of Resource. In Ansible, configurations are managed using Playbooks, which are decomposed into Plays that define Tasks. In the case of Chef, configurations are defined as Cookbooks, which are decomposed into Recipes specifying Resources. Puppet structures the configurations using Modules that contain configuration files called Manifests. Resources are specified in Classes, which are named blocks used to configure larger chunks of functionality.

```
server_root_password = node['mysql']['server_root_password']
execute 'set-mysql-root' do
  command <<-EOH
    mysqladmin -u root password #{server_root_password}
    mysql -uroot -p#{server_root_password} -e (...)
  EOH
  only_if "/usr/bin/mysql -u root -e 'show databases;'"
end
```

(a) Part of a Chef script (from Vagrant Chef for CakePHP)

```
$server_root_password = $facts['mysql']['server_root_password']
exec { 'set-mysql-root':
  command => @("COMMAND"/L)
    mysqladmin -u root password ${server_root_password}
    mysql -uroot -p${server_root_password} -e (...)
  | COMMAND,
  only_if => "/usr/bin/mysql -u root -e 'show databases;'"
}
```

(b) Same part of a Chef script rewritten in Puppet

Figure 1: Inconsistencies in state-of-the-art tools: SLAC reports false positive "Hard-coded secret" for script (a); SLIC does not report any security smell for script (b).

2.2 Security Smells in IaC Scripts

Several catalogs and categories of code smells for IaC scripts have been proposed. Sharma et al. [26] created a configuration smells catalog for Puppet scripts with 13 implementation and 11 design configuration smells. Schwarz et al. [24] extended the research done by Sharma et al. by applying the detection of IaC smells to Chef scripts. Rahman and Williams [20] characterized defective IaC scripts by extracting text features from faulty scripts. Rahman and Williams [21] identified 10 source code properties that correlate with defective IaC scripts. Rahman et al. [15] proposed a defect taxonomy for IaC scripts that includes eight categories. In another work, Rahman et al. [16] identified five development anti-patterns for IaC scripts. Focusing on security smells, Rahman et al. [17] concluded, after a systematic mapping study with 32 IaC-related publications, that there is a need for more research studies focused on defects and security flaws for IaC. Rahman et al. [18] identified seven types of security smells that are indicative of security weaknesses in Puppet scripts. Rahman et al. [19] later replicated this study for Ansible and Chef scripts, identifying two additional security smells.

Like Rahman et al. [18, 19], we also focus on security smells. However, to the best of our knowledge, we are the first to provide a method for enabling consistent security smell detection across different IaC technologies. We consider the following nine security smells (we adapt the descriptions by Rahman et al. [19]): **Admin by default (CWE-250 [11])**: This smell is the recurring pattern of specifying default users. The smell can violate the "principle of least privilege" property [14]. **Empty password (CWE-258 [11])**: The smell is the recurring pattern of using a string of length zero for a password. **Hard-coded secret (CWE-259, CWE-798 [11])**: This smell is the recurring pattern of revealing sensitive information, such as user name and passwords in IaC scripts. **Unrestricted IP Address (CWE-284 [11])**: This smell is the recurring pattern of assigning the address 0.0.0.0 for a database server or a cloud service/instance. Binding to the address 0.0.0.0 may cause security concerns as this address can allow connections from every possible network [13]. **Suspicious comment (CWE-546 [11])**: This smell is the recurring pattern of putting information in comments about the presence of defects, missing functionality, or weakness of the system (e.g., "TODO" and "FIXME"). **Use of HTTP without SSL/TLS (CWE-319 [11])**: : This smell is the recurring pattern of using HTTP without the Transport Layer Security (TLS) or Secure Sockets Layer (SSL). Such use makes the communication between two entities less secure [22]. **No integrity check (CWE-353 [11])**: This smell is the recurring pattern of downloading content from the Internet and not checking the downloaded content using checksums or gpg signatures. **Use of weak cryptography algorithms (CWE-326, CWE-327 [11])**: This smell is the recurring pattern of using weak cryptography algorithms, namely, MD5 and SHA-1, for encryption purposes. **Missing Default in Case Statement (CWE-478 [11])**: This smell is the recurring pattern of not handling all input combinations when implementing a case conditional logic.

2.3 Related Work

Several studies have been published on code quality and security coding practices for IaC scripts. For example, Jiang and Adams [7] conducted an empirical study on the co-evolution of IaC scripts and other software artifacts. They found that the IaC scripts are coupled tightly with the other files in a project. Hanappi et al. [4] introduced a conceptual framework for asserting reliable convergence in configuration management. Van der Bent et al. [28] proposed a code quality model for Puppet and validated it with experts.

In terms of analysis tools for IaC scripts, Hanappi et al. [4] propose a tool that detects idempotence and convergence related issues in a set of existing Puppet scripts. Schwarz et al. [24] picked smells from the catalog proposed by Sharma et al. [26] and convert them into detection rules for Foodcritic, a static code analysis tool designed for Chef. Sotiropoulos et al. [27] propose a tool for detecting faults regarding ordering violations and notifiers in Puppet scripts. Lepillet et al. [10] propose Häyä, a tool that uses dataflow graph analysis to detect intra-update sniping vulnerabilities in CloudFormation templates. More relevant for our work are the tools SLIC and SLAC, which are focused on security smells. SLIC, developed by Rahman et al. [18], detects seven types of security smells in Puppet scripts and SLAC, developed by Rahman et al. [19], detects nine types of security smells in Chef scripts and six types in Ansible scripts. Our tool extends the state-of-the-art by providing the first IaC-technology-agnostic framework that can be used to unify tools such as SLIC and SLAC, facilitating the detection of security smells in different IaC technologies. When compared with the security smells supported by SLIC and SLAC, we also identify two additional types of smell in Puppet scripts (*Missing default case statement* and *No integrity check*) and two additional types in Ansible scripts (*Admin by default* and *Use of weak cryptographic algorithms*).

Finally, some analysis tools for IaC use intermediate representations [6, 25, 27] to describe file-system manipulations done by IaC scripts. Shambaugh et al. [25] translated IaC scripts to the intermediate representation by mapping types of resources to their

Table 2: Correspondence between the abstract concepts and the concepts in each IaC technology.

Abstract Concepts	Ansible	Chef	Puppet
Modules	Roles	Cookbooks	Modules
Unit Blocks	Playbooks	Recipes	Manifests, Classes
Atomic Units	Tasks	Resources	Resources

filesystem operations. Sotiropoulos et al. [27] used system calls executed by each resource in a IaC script to automatically map the resources to the filesystem operations, which were represented in the intermediate language. To the best of our knowledge, our work is the first that translates scripts of different IaC technologies into an intermediate representation.

3 INTERMEDIATE REPRESENTATION

In order to achieve a technology-agnostic framework, we use an intermediate representation. Our representation is able to capture similar concepts from different IaC technologies, while assuring it is expressive enough to apply analyses that identify security smells. Figure 2 describes the abstract syntax of our intermediate representation. We follow an object-oriented approach with a hierarchical structure. As the top-level structure, the intermediate representation can model a *Project*, a *Module*, or a *Unit block*. Projects represent a generic folder that may contain several modules and unit blocks. This structure allows us to represent the high-level code structures described in Table 1 from Ansible, Chef and Puppet. Table 2 shows the relation between the high-level code structures in each IaC technology and the abstract concepts in our intermediate representation. As the table shows, it is possible to find similar structures in the different technologies. *Modules* are the top component from each structure and they agglomerate the scripts necessary to execute a specific functionality. Modules are file system folders, usually with a specific organization (e.g. a role in Ansible usually has a *tasks* and a *vars* folder where, respectively, the tasks and variables for the role are defined). *Unit Blocks* correspond to the IaC scripts themselves or to a group of atomic units. For instance, in Puppet, we can agglomerate resources in classes. Finally, *Atomic Units* are the building block of IaC scripts. Atomic units define the system components we want to change and the actions we want to perform on them. As shown in Figure 2, unit blocks can have *attribute* definitions, *variable* definitions, and *conditions*. Atomic units have attribute definitions. When values in attribute and variable definitions use variable references, the field *has_variable* is true. For instance, in Figure 1b, the definition of the variable `$server_root_password` has as value a reference to the variable `$facts['mysql']['server_root_password']`, setting the field *has_variable* in our intermediate representation to true. Figure 3 shows a graph-based visualization of how our intermediate representation models the scripts in Figure 1a and Figure 1b.

4 SECURITY SMELL DETECTION

In Table 3, we define the rules used by GLITCH to detect security smells. The formalism used to define rules is similar to the one used by SLIC [18] and SLAC [19]. The functions *isAttribute*, *isVariable*, *isComment*, *isAtomicUnit*, and *isConditionStatement* verify the type of instance being analyzed (e.g., if the node x is an Attribute node,

```
<S> ::= <project>
      | <module>
      | <unitblock>

<project> ::=
    Project {
        name: <str>,
        modules: <module>*
        blocks: <unitblock>*
    }

<module> ::=
    Module {
        name: <str>,
        blocks: <unitblock>*
    }

<condition> ::=
    ConditionStatement {
        type: IF | SWITCH
        condition: <str>,
        else_statement: <condition>,
        is_default: <bool>
    }

<comment> ::=
    Comment {
        content: <str>
    }

<unitblock> ::=
    UnitBlock {
        name: <str>,
        atomic_units: <atomicunit>*,
        variables: <variable>*,
        attributes: <attributes>*
        comments: <comment>*,
        conditions: <condition>*,
        unit_blocks: <unitblock>*
    }

<atomicunit> ::=
    AtomicUnit {
        name: <str>,
        type: <id>,
        attributes: <attribute>*
    }

<attribute> ::=
    Attribute {
        name: <id>,
        value: <value>,
        has_variable: <bool>
    }

<variable> ::=
    Variable  {
        name: <id>,
        value: <value>,
        has_variable: <bool>
    }

- - - - - - - - - - - - - - - - - - - - - - - - -

    <value> ::= <str> | <number> | <bool> | <value>* | <id>
    <id> ::= ;sequence of alphanumerics which starts with a letter
    <str> ::= "<character>*"    <number> ::= ;integer or double
    <bool> ::= True | False
```

Figure 2: Abstract syntax of our intermediate representation.

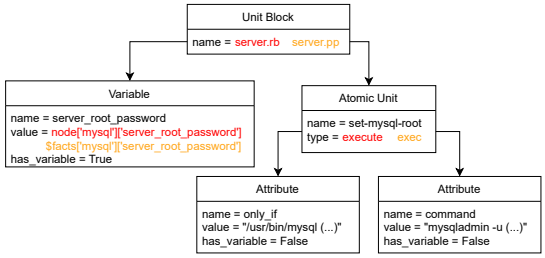

Figure 3: Graph-based representation of the scripts in Figure 1 using our intermediate representation. In black and red: the representation of the Chef script from Figure 1a. In black and orange: the representation of the Puppet script from Figure 1b.

isAttribute(x) is true). Each node in our representation is referred by the variable x. We traverse the nodes using a depth-first search (DFS). We start in the initial node (a Project, a Module or a Unit Block) and then we execute the DFS considering each collection inside the node as its children. Each node may have more than one security smell, and so every rule is applied, even if a smell was already identified for that node. Previous nodes have no influence in the analyses of other nodes. The function *hasDownload* goes through a list of attributes and verifies if for at least one of them *isDownload(x.value)* is true. The same goes for the function *hasChecksum* but instead of using *isDownload*, it uses *isChecksum*. The function *isDefault* is a recursive function that returns true if a default branch is found in the case statement, and false otherwise. The remaining functions are defined in Table 4. These functions verify if any of the string patterns described are present in the values they receive.

The GLITCH framework allows the definition of different configurations to identify security smells. These configurations change the keywords in the (disjunctive) string patterns for each function defined in Table 4. In the table, we describe the configuration used by the improved version of GLITCH to which we will refer in Section 5. Configurations allow users to tweak the tool to best suit the needs of the IaC developers and to better adapt to each IaC technology. GLITCH is implemented in Python and it currently supports the analysis of Ansible, Chef, and Puppet scripts. Our implementation transforms the original scripts into our intermediate representation and then attempts to detect security smells as described above. To parse the Ansible scripts we used the *ruamel.yaml* package[5] for Python. The Chef scripts were parsed using Ripper,[6] a script parser for Ruby. We developed a parser for Ripper's output using a package called *ply*.[7] Finally, for Puppet scripts, we developed our own parser[8] using the same *ply* package. We decided to develop our parser since we did not find any other good options to parse Puppet DSL in Python.

5 EVALUATION

This section describes the evaluation of GLITCH.

5.1 Research Questions

RQ1. **[Abstraction]** Can our intermediate representation model IaC scripts and support automated detection of security smells?

RQ2. **[Accuracy and Performance]** How does GLITCH compare with existing state-of-art tools for detecting security smells in terms of accuracy and performance?

RQ3. **[Frequency]** How frequently do security smells occur in IaC scripts?

5.2 Datasets

This section describes how we constructed the datasets used for our evaluation. Since we consider Ansible, Chef, and Puppet scripts, our first step was to attempt to obtain the same datasets as used in the studies involving SLIC and SLAC [18, 19]. We got hold of the publicly available datasets[9] and Docker image[10], and we observed that only the oracle for Ansible was available. We thus contacted the first author of the studies mentioned above, who very kindly shared with us a Puppet dataset almost identical to the one used in the empirical study using SLIC (there were small differences in the number of Puppet scripts contained in the dataset). We constructed oracle datasets for Chef and Puppet as these oracle datasets were not available as part of Rahman et al.'s replication packages. We further contacted the first author about the availability of the oracle datasets and learned that these datasets reside in computing clusters to which the first author no longer has access to. Given this, we decided to reuse their oracle for Ansible and the Puppet dataset, and to construct new oracles for Chef and Puppet, and new IaC datasets for Ansible and Chef.

[5]https://pypi.org/project/ruamel.yaml/
[6]https://github.com/ruby/ruby/tree/master/ext/ripper
[7]https://github.com/dabeaz/ply
[8]https://github.com/Nfsaavedra/puppetparser
[9]https://doi.org/10.6084/m9.figshare.8085755
[10]https://hub.docker.com/repository/docker/akondrahman/slic_ansible

5.2.1 IaC datasets. To perform an empirical study of security smells in Ansible, Chef, and Puppet scripts, we require three datasets of IaC scripts, one for each technology. As mentioned above, we reused Rahman et al.'s Puppet dataset [18], which is composed of four different sub-datasets. Three datasets are constructed using repositories collected from three organizations: Mozilla (MOZ), Openstack (OST), and Wikimedia (WIK). The fourth dataset is constructed from repositories hosted on GitHub (GH).

For Ansible and Chef, we created two new datasets by selecting OSS repositories from GitHub. As described in previous research [12], OSS repositories need to be curated. We apply the same criteria that Rahman et al. [18] used to construct their Puppet sub-datasets extracted from GitHub (except that we consider all the available repositories created between 2012 and 2022):

Criterion 1: At least 11% of the files belonging to the repository must be IaC scripts. This follows from a Jiang and Adams's study [7], where it was observed that in OSS repositories, a median of 11% of the files are IaC scripts. The rationale is to collect repositories that contain sufficient amount of IaC scripts for analysis. **Criterion 2**: The repository is not a clone. **Criterion 3**: The repository must have at least two commits per month. This is based on Munaiah et al. [12], who used the threshold of at least two commits per month to determine which repositories have enough software development activity. **Criterion 4**: The repository has at least 10 contributors. Similar to Rahman et al. [18], we assume that this criterion may help us to filter out irrelevant repositories.

Table 5 presents the number of repositories, the number of IaC scripts, and the number of LOC in the three IaC datasets. The Ansible dataset was constructed from 681 repositories and contains 108,509 Ansible scripts (5,180,747 LOC). The Chef dataset was constructed from 439 repositories and contains 70,939 Chef scripts (6,071,035 LOC). The Puppet dataset was constructed from 293 repositories and contains 17,307 Puppet scripts (1,029,469 LOC). When considering the three IaC datasets as a whole, there are 1413 repositories with 196,755 IaC scripts. In total, there are 12,281,251 LOC.

5.2.2 Oracles. To determine the accuracy of GLITCH and to compare it with other tools, we require three oracle datasets, one for each IaC technology considered. In what follows, we describe how we selected the IaC scripts included in each oracle and how we annotated the datasets.

File collection. For the Ansible oracle, we reused Rahman et al.'s oracle [19], which contains 81 IaC scripts. We constructed new oracle datasets for Chef and Puppet. To ensure that the size of the three oracles was similar, based on the size of the Ansible oracle dataset, we decided to create oracles with exactly 80 IaC scripts. To select the files, we wrote a Python script that kept selecting a random file from the respective IaC dataset described in the previous subsection while the desired size was not achieved. For each file, we ran GLITCH and either SLAC (if the file was a Chef script) or SLIC (if the file was a Puppet script). We kept track of the number of security smells reported and their respective categories. If, after analyzing a file, the file contained a smell of a category that up to that point had less than 5 reports, then the file was included in the oracle dataset. Table 6 presents the number of IaC scripts and the number of LOC in the three oracle datasets.

Table 3: Rules to detect security smells used by GLITCH.

Smell Name	Rule
Admin by default	$(isAttribute(x) \lor isVariable(x)) \land (isUser(x.name) \lor isRole(x.name)) \land isAdmin(x.value) \land \neg x.has_variable$
Empty password	$(isAttribute(x) \lor isVariable(x)) \land isPassword(x.name) \land length(x.value) == 0$
Hard-coded secret	$(isAttribute(x) \lor isVariable(x)) \land (isPassword(x.name) \lor isSecret(x.name) \lor isUser(x.name)) \land \neg x.has_variable$
Invalid IP address binding	$(isAttribute(x) \lor isVariable(x)) \land isInvalidBind(x.value)$
Suspicious comment	$isComment(x) \land hasWrongWords(x.content)$
Use of HTTP without TLS	$(isAttribute(x) \lor isVariable(x)) \land isURL(x.value) \land hasHTTP(x.value) \land \neg hasHTTPWhiteList(x.value)$
No integrity check	$(isAtomicUnit(x) \land hasDownload(x.attributes) \land \neg hasChecksum(x.attributes)) \lor ((isAttribute(x) \lor isVariable(x)) \land isCheckSum(x.name) \land (x.value == "no" \lor x.value == "false"))$
Use of weak crypto alg.	$(isAttribute(x) \lor isVariable(x)) \land isWeakCrypt(x.value) \land \neg hasWeakCryptWhiteList(x.name) \land \neg hasWeakCryptWhiteList(x.value)$
Missing default case statement	$isConditionStatement(x) \land x.is_default == False \land \neg isDefault(x.else_statement)$

Table 4: String patterns used in the GLITCH's rules. These are configurable. The configuration shown is the one used by the improved version of GLITCH.

Function	String Pattern
isUser()	"user", "uname", "username", "login", "userid", "loginid" (...)
isRole()	(the config is empty for this function)
isAdmin()	"admin", "root"
isPassword()	"pass", "pwd", "password", "passwd", "passno", "pass-no" (...)
isSecret()	"auth_token", "authetication_token", "secret", "ssh_key" (...)
isInvalidBind()	"0.0.0.0"
hasWrongWords()	"bug", "debug", "todo", "hack", "solve", "fixme" (...)
hasHTTP()	"http"
hasHTTPWhiteList()	"localhost", "127.0.0.1"
isDownload()	"(http|https|www).*iso$", "(http\|https\|www).*tar.gz$" (...)
isCheckSum()	"gpg", "checksum"
isWeakCrypt()	"md5", "sha1", "arcfour"
hasWeakCryptWhiteList()	"checksum"

Annotating the oracle datasets. After collecting the scripts that make the oracle datasets, we manually annotated them, identifying security smells. Despite the use of analysis tools in the file selection process described above, we guaranteed that the location of the security smells was not disclosed. In other words, at the annotation stage we only had access to the files, but not the reports. We did this to reduce bias in the annotation process. The Ansible oracle dataset was already annotated, but since the numbers of smell occurrences did not match the numbers reported in Rahman et al.'s study [19], we decided to reannotate the dataset. To annotate the oracle datasets, we used closed coding [23], where three raters identified security smells and their agreement was checked. In total, there were seven raters involved. One of the raters was the first author. For each of the three IaC technologies, we recruited two postgraduate students who had experience with IaC and/or cybersecurity. They were given access to: the 80 files in the oracle datasets, a general description of the IaC technology, and a description of the nine security smells considered. For each report, raters identified the name of the file, the category of the security smell, and the line where it occurs; they collated this information in a CSV file.

We then manually inspected the three CSV files produced for each oracle dataset, and we decided to keep only the classifications where at least two raters agreed. Table 7 shows the agreement distribution for each dataset. We only consider the lines of code where at least one rater identified a smell. The percentage values shown are for the cases where there was no agreement, two raters agreed, or all the raters agreed. When a rater did not identify a smell identified by other rater, we considered the label "none" to be attributed. The results on the table demonstrate that at least two raters agreed on the great majority of subjects: 99.1% in Ansible, 93.9% in Chef, and 95.1% in Puppet. We calculated the agreement distribution instead of other statistics, such as Cohen's Kappa or Krippendorff's alpha, since these statistics consider the probability of chance agreement. We argue that, since our annotation task includes finding the smells in the scripts, the likelihood of chance agreement is significantly reduced.

After this process, we obtained: an oracle of 44 Ansible security smells categorized as shown in Table 8 and with 69 files with no smells; an oracle of 105 Chef security smells categorized as shown in Table 9 and with 43 files with no smells; and an oracle of 65 Puppet security smells categorized as shown in Table 10 and with 52 files with no smells.

5.3 Accuracy of GLITCH

To determine the accuracy of GLITCH, we ran it for the oracle datasets. We also ran SLIC for the Puppet oracle dataset and SLAC for the other two oracle datasets. We measured precision and recall of each tool. Since it is easy to configure GLITCH (see Section 4), we used two versions of GLITCH for each oracle dataset: one version was configured to behave similarly to SLIC (or SLAC), and the other was an improved version. As described in Section 4, the difference between the two versions is on the keywords for each function in Table 4: one uses the keywords used by SLIC (or SLAC) and the other configuration was tweaked by us. In the tables below, we use the headers GLITCH (SLIC) and GLITCH (SLAC) to refer to GLITCH configured to behave similarly to SLIC and SLAC, respectively. The header GLITCH refers to the improved version of GLITCH that uses the rules shown in Table 4.

Tables 8, 9, and 10 report the accuracy results for Ansible, Chef, and Puppet, respectively. We use N/I to denote that the detection of a certain smell is not implemented (e.g., SLAC does not detect the smell *Admin by default* for Ansible scripts); N/A to denote that a certain smell cannot occur (e.g., Ansible does not have switch statements, so the smell *Missing default case statement* does not apply); and N/D to denote that the tool does not report any security smell or to denote that there are no occurrences of a given smell (see, for example, the recall value of GLITCH for the *Use of weak crypto algorithm* in Table 8). To facilitate comparison between tools and IaC technologies, we decided to keep all the rows in these tables, even when there are no smell occurrences or when its detection is not implemented.

5.3.1 Accuracy results for the Ansible oracle dataset. As shown in Table 8, GLITCH configured to behave similarly to SLAC has the same precision and recall as SLAC (same average). There is a small discrepancy in the recall values for *No Smell*. This happens because SLAC detects one *No integrity check* smell in an Ansible script where no smells should be detected. The difference between both tools is that GLITCH enforces detection of *No integrity check*

Table 5: Attributes of IaC Datasets.

			Puppet			
Attribute	**Ansible**	**Chef**	**GH**	**MOZ**	**OST**	**WIK**
Repository count	681	439	219	2	61	11
Total IaC scripts	108,509	70,939	10,009	1613	2,840	2,845
Total LOC (IaC scripts)	5,180,747	6,071,035	610,122	66,367	217,843	135,137

Table 6: Attributes of Oracle Datasets.

Attribute	**Ansible**	**Chef**	**Puppet**
Total IaC scripts	81	80	80
Total LOC (IaC scripts)	4,185	4,630	4,367

Table 7: Agreement distribution for the oracle datasets (%).

	Ansible	**Chef**	**Puppet**
No agreement	0.9	6.1	4.9
2 raters agreed	86.4	73.6	78.6
3 raters agreed	12.7	20.3	16.5

smells only on *Atomic Unit* nodes, while SLAC ignores the type of node, which leads SLAC to detect this type of smell in the definition of a variable.

Regarding the improved version of GLITCH, the average precision improves from 67% to 77% and recall improves from 79% to 87%. There are also improvements regarding files with no smells. We can also see that it supports the smell *Admin by default* with perfect precision and recall. GLITCH keeps the values of precision and recall when they were already 100%. It also improves the precision for *Hard-coded secret* by 10 percentage points (from 32% to 42%); for *Suspicious comment* by 8 percentage points (from 67% to 75%); and for *Use of HTTP without TLS* by 24 percentage points (from 71% to 95%). Recall for *Suspicious comment* improved from 67% to 100%. The only case where improvements do not occur is for the smell *No integrity check*, where the single occurrence is not detected (note that SLAC did not detect it either). This happens because the occurrence of this smell is regarding a URL referring to an YAML file, which GLITCH does not consider (i.e., the string pattern *isDownload()* shown in Table 4 does not contain URLs that end with `.yml`). Finally, the worst precision value is for the smell *Hard-coded secret* (42%). This happens mainly because the string patterns *isSecret()*, *isPassword()*, and *isUser()* are the ones with more possibilities, thus increasing the probability of having false positives. Some of the possibilities are keywords such as "user", which result in a higher number of false positives.

5.3.2 Accuracy results for the Chef oracle dataset. Table 9 shows that when GLITCH is configured to behave similarly to SLAC, it actually obtains better results than SLAC: the average precision improves 28 percentage points (from 49% to 77%) and the average recall improves 16 percentage points (from 60% to 76%). There are also improvements regarding files with no smells. Contributing to these improvements is the substantial increase in precision for the smells *Empty password* and *No integrity check*. Regarding the first smell, this is because SLAC wrongly treats variables as empty values; regarding the second, GLITCH searches for links in the values of variables and attributes, while SLAC is searching for links on a line-by-line basis.

When compared to GLITCH configured to behave similarly to SLAC, the improved version maintains the average precision and increases the average recall by 10 percentage points (76% to 86%). When compared to SLAC, the results for all smells improve, except for *Invalid IP address binding* and *Use of HTTP without TLS*, where the results are the same, and for *Suspicious comment*, where the precision decreases. This decrease in precision is because GLITCH uses a larger set of keywords (this is similar to what caused the low precision for the smell *Hard-coded secret* when analyzing the Ansible oracle dataset). This is also why the worst precision value is for the smell *Hard-coded secret*. The worst recall value is for the smell *Admin by default* (41%). This happens because there are some scripts in the dataset that configure the execution of MySQL commands. The commands executed as root, such as the following, were considered by the raters as a security smell: `cmd = "mysql -uroot ..."`. However, for this smell, GLITCH only considers the value of attributes or variables that define users (e.g. `user: root`).

5.3.3 Accuracy results for the Puppet oracle dataset. Similar to what was described above, Table 10 shows that when GLITCH is configured to behave similarly to SLIC, it also obtains better results than SLIC: the average precision improves 8 percentage points (from 60% to 68%) and the average recall improves 10 percentage points (from 72% to 82%). Contributing to this is the fact that GLITCH detects smells of type *Missing default case statement* with a high precision. Also, the precision for the smell *Empty password* is noticeable higher (GLITCH reports no false positives). This is because GLITCH seems to deal better with variables. There are also improvements regarding files with no smells.

When compared to GLITCH configured to behave similarly to SLIC, the improved version maintains the average precision and improves the average recall by 3 percentage points (82% to 85%). The precision and recall for *No Smell* decreased 1 and 6 percentage points, respectively. We can see that for the smell *Admin by default* many more true positives are identified, but there are some false positives. There were no reports for the smell *No integrity check*. Precision and recall improved or remained the same for all the smells, except for *Suspicious comment*. Similar to what happened with the Chef oracle dataset, the precision values for the smells *Hard-coded secret* and *Suspicious comment* are low due to the use of more keywords.

5.4 Security Smells Frequency

Using GLITCH, we performed an empirical study to quantify the prevalence of security smells in Ansible, Chef, and Puppet. Similar studies were performed by Rahman et al. [18] (for Puppet scripts using SLIC) and Rahman et al. [19] (for Ansible and Chef scripts using SLAC). Here, the goal is to use GLITCH and investigate whether there are any noticeable differences. The IaC datasets used are described in Section 5.2 and their attributes shown in Table 5. This means that, when considering the three IaC datasets as a whole, this empirical study considers 1413 repositories with 196,755 IaC scripts. In total, we analyze 12,281,251 LOC.

Table 8: GLITCH vs SLAC: Accuracy for the Ansible Oracle Datasets (N/I - Not implemented, N/A - Not applicable, N/D - No data)

		Original Oracle					
		SLAC		GLITCH (SLAC)		GLITCH	
Smell Name	**Occurr.**	**Precision**	**Recall**	**Precision**	**Recall**	**Precision**	**Recall**
Admin by default	7	N/I	N/I	N/I	N/I	1.00	1.00
Empty password	1	1.00	1.00	1.00	1.00	1.00	1.00
Hard-coded secret	8	0.32	1.00	0.32	1.00	0.42	1.00
Invalid IP address binding	1	1.00	1.00	1.00	1.00	1.00	1.00
Suspicious comment	6	0.67	0.67	0.67	0.67	0.75	1.00
Use of HTTP without TLS	20	0.71	1.00	0.71	1.00	0.95	1.00
No integrity check	1	0.00	0.00	0.00	0.00	0.00	0.00
Use of weak crypto alg.	0	N/I	N/I	N/I	N/I	N/D	N/D
Missing default case statement	0	N/A	N/A	N/A	N/A	N/A	N/A
No smell	69	0.98	0.87	0.98	0.88	1.00	0.94
Average		0.67	0.79	0.67	0.79	0.77	0.87

Table 9: GLITCH vs SLAC: Accuracy for the Chef Oracle Datasets (N/I - Not implemented, N/A - Not applicable, N/D - No data)

		Original Oracle					
		SLAC		GLITCH (SLAC)		GLITCH	
Smell Name	**Occurr.**	**Precision**	**Recall**	**Precision**	**Recall**	**Precision**	**Recall**
Admin by default	37	0.00	0.00	N/D	0.00	0.94	0.41
Empty password	4	0.00	0.00	1.00	0.75	1.00	0.75
Hard-coded secret	13	0.13	0.54	0.20	0.69	0.20	0.69
Invalid IP address binding	7	1.00	1.00	1.00	1.00	1.00	1.00
Suspicious comment	4	0.80	1.00	0.80	1.00	0.40	1.00
Use of HTTP without TLS	13	0.71	0.92	0.71	0.92	0.71	0.92
No integrity check	6	0.20	0.33	1.00	0.33	1.00	1.00
Use of weak crypto alg.	1	0.25	1.00	0.25	1.00	0.50	1.00
Missing default case statement	20	1.00	0.45	1.00	0.95	1.00	0.95
No smell	43	0.85	0.79	0.95	0.93	0.95	0.88
Average		0.49	0.60	0.77	0.76	0.77	0.86

Table 10: GLITCH vs SLIC: Accuracy for the Puppet Oracle Datasets (N/I - Not implemented, N/A - Not applicable, N/D - No data)

		Original Oracle					
		SLIC		GLITCH (SLIC)		GLITCH	
Smell Name	**Occurr.**	**Precision**	**Recall**	**Precision**	**Recall**	**Precision**	**Recall**
Admin by default	14	N/D	0.00	N/D	0.00	0.81	0.93
Empty password	5	0.60	0.60	1.00	1.00	1.00	1.00
Hard-coded secret	11	0.10	0.73	0.14	0.82	0.14	0.82
Invalid IP address binding	6	1.00	1.00	1.00	1.00	1.00	1.00
Suspicious comment	9	0.75	1.00	0.60	1.00	0.39	1.00
Use of HTTP without TLS	5	0.38	1.00	0.42	1.00	0.45	1.00
No integrity check	1	N/I	N/I	N/I	N/I	N/D	0.00
Use of weak crypto alg.	4	0.43	0.75	0.50	0.75	0.57	1.00
Missing default case statement	10	N/I	N/I	0.83	1.00	0.83	1.00
No smell	52	0.95	0.71	0.98	0.77	0.97	0.71
Average		0.60	0.72	0.68	0.82	0.68	0.85

Similar to previous studies, the first step was to determine the occurrences of security smells for each IaC script. We then calculated the two following metrics:

- **Smell density**: frequency of a given security smell for every 1,000 LOC [9, 19]. For a given smell x,

$$SmellDensity(x) = \frac{\text{Total occurrences of } x}{\text{Total line count for all scripts}/1000}$$

- **Proportion of scripts (Script%)**: percentage of scripts that contain at least one occurrence of smell x.

5.4.1 Occurrences. Looking at Table 11, we observe that all categories of security smells are identified across all datasets. Overall, GLITCH detects 76,015 security smells for Ansible, 19,455 for Chef, and 18,151 for Puppet. GLITCH identifies fewer security smells in the Chef dataset than SLIC. On the other hand, GLITCH identifies more security smells than SLIC in the Ansible and Puppet datasets (76,015 vs 67,078 and 18,151 vs 12,884). When using GLITCH for

Table 11: Smell Occurrences. (N/I - Not implemented, N/A - Not applicable, N/D - No data)

	Ansible		Chef		Puppet							
					GH		MOZ		OST		WIK	
	SLAC	GLITCH	SLAC	GLITCH	SLIC	GLITCH	SLIC	GLITCH	SLIC	GLITCH	SLIC	GLITCH
Admin by default	N/I	10,222	248	1,821	34	1,201	4	30	35	172	6	136
Empty password	1,973	1,432	115	303	131	348	20	20	24	127	36	66
Hard-coded secret	47,735	45,325	15,100	7,763	5,608	6,236	394	592	1,751	2,172	858	1,114
Invalid IP address binding	914	2,033	499	603	179	96	20	26	90	45	41	18
Suspicious comment	10,498	10,749	2,267	4,343	868	1,802	202	285	309	965	343	609
Use of HTTP without TLS	4,812	3,393	2,507	2,281	934	703	52	31	453	163	164	111
No integrity check	1,146	1,359	1,662	304	N/I	44	N/I	0	N/I	4	N/I	3
Use of weak crypto alg.	N/I	1,502	76	147	227	109	48	28	27	18	26	21
Missing default case statement	N/A	N/A	702	1,890	N/I	527	N/I	210	N/I	36	N/I	83
Combined	67,078	76,015	23,176	19,455	7,981	11,066	740	1,222	2,689	3,702	1,474	2,161

Table 12: Smell density (per KLOC). (N/I - Not implemented, N/A - Not applicable, N/D - No data)

	Ansible		Chef		Puppet							
					GH		MOZ		OST		WIK	
	SLAC	GLITCH	SLAC	GLITCH	SLIC	GLITCH	SLIC	GLITCH	SLIC	GLITCH	SLIC	GLITCH
Admin by default	N/I	1.97	0.04	0.30	0.06	1,97	0.06	0.45	0.16	0.79	0.04	1.00
Empty password	0.38	0.28	0.02	0.05	0.21	0.57	0.30	0.30	0.11	0.58	0.27	0.49
Hard-coded secret	9.21	8.75	2.49	1.28	9.19	10.22	5.94	8.92	8.04	9.97	6.35	8.24
Invalid IP address binding	0.18	0.39	0.08	0.10	0.29	0.16	0.30	0.39	0.41	0.21	0.30	0.13
Suspicious comment	2.03	2.07	0.37	0.72	1.42	2.95	3.04	4.29	1.42	4.43	2.54	4.51
Use of HTTP without TLS	0.93	0.65	0.41	0.38	1.53	1.15	0.78	0.47	2.08	0.75	1.21	0.82
No integrity check	0.22	0.26	0.27	0.05	N/I	0.07	N/I	0.00	N/I	0.02	N/I	0.02
Use of weak crypto alg.	N/I	0.29	0.01	0.02	0.37	0.18	0.72	0.42	0.12	0.08	0.19	0.16
Missing default case statement	N/A	N/A	0.12	0.31	N/I	0.86	N/I	3.16	N/I	0.17	N/I	0.61
Combined	12.95	14.66	3.81	3.21	13.07	18.13	11.14	18.40	12.34	17.00	10.90	15.98

Table 13: Proportion of Scripts (Script%) with at Least One Smell. (N/I - Not implemented, N/A - Not applicable, N/D - No data)

	Ansible		Chef		Puppet							
					GH		MOZ		OST		WIK	
	SLAC	GLITCH	SLAC	GLITCH	SLIC	GLITCH	SLIC	GLITCH	SLIC	GLITCH	SLIC	GLITCH
Admin by default	N/I	5.7	0.2	1.7	0.3	3.6	0.2	1.5	1.1	5.2	0.2	4.0
Empty password	0.8	0.4	0.2	0.3	1.1	2.5	0.6	0.9	0.7	3.5	0.4	1.1
Hard-coded secret	18.3	13.9	7.7	5.2	18.2	20.2	9.9	12.3	24.6	31.3	17.0	19.1
Invalid IP address binding	0.7	0.7	0.5	0.4	1.4	0.7	0.7	0.6	2.8	1.4	1.4	0.6
Suspicious comment	5.4	5.4	2.6	3.8	5.3	8.9	8.6	11.0	7.0	13.5	9.1	13.7
Use of HTTP without TLS	2.3	1.6	1.8	1.8	5.1	3.7	1.5	0.9	8.2	3.1	3.8	2.5
No integrity check	0.8	0.9	1.4	0.4	N/I	0.4	N/I	0.0	N/I	0.1	N/I	0.1
Use of weak crypto alg.	N/I	0.6	0.1	0.2	1.6	0.8	1.4	0.4	0.8	0.5	0.5	0.4
Missing default case statement	N/A	N/A	0.9	2.1	N/I	2.9	N/I	9.9	N/I	1.0	N/I	1.8
Combined	23.8	19.6	11.4	10.4	25.5	29.6	18.0	27.5	32.5	40.1	26.8	31.5

Ansible and Puppet, the three most dominant security smells are *Hard-coded secret*, *Admin by default*, and *Suspicious comment*. For Chef, the three most dominant security smells are *Hard-coded secret*, *Suspicious comment*, and *Use of HTTP without TLS*.

5.4.2 Smell density. Table 12 shows the smell density for the three datasets. Overall, GLITCH detects 14.66 security smells per 1,000 LOC in Ansible scripts, 3.21 in Chef scripts, and an average of 17.38 in Puppet scripts. For all datasets, the dominant security smell is *Hard-coded secret*, followed by *Suspicious comment*. Given that the precision values for these smells tend to be the lowest (see Section 5.3), this suggests that many of these are false positives. The third most dominant security smell differs across the three datasets: for Ansible, it is *Admin by default* (1.97); for Chef, it is *Use of HTTP without TLS* (0.38); and for Puppet, it is *Admin by default* when considering the GitHub dataset (1.97), *Missing default case statement* when considering the Mozilla dataset (3.16), and *Admin by default* when considering the Openstack and Wikimedia datasets (0.79 and 1.00, respectively).

5.4.3 Proportion of Scripts (Script%). Table 13 shows, for the three datasets, the proportion of scripts with at least one occurrence of a smell. For Ansible, GLITCH detects at least one of the eight identified security smells in 19.6% of the total scripts. For SLIC, the percentage is 23.8%, but note that SLIC only supports six security

smells. This is not very different from the values obtained by Rahman et al. [19], where the percentages obtained with SLIC were 25.3% and 29.6% for their GitHub and Openstack datasets, respectively. For Chef, GLITCH detects at least one of the nine identified security smells in 10.4% of the total scripts. For SLIC, the percentage is slightly higher at 11.4%. Here, we note a more noticeable discrepancy with Rahman et al.'s study [19]: the percentages obtained with SLIC were 20.5% and 30.4% for their GitHub and Openstack datasets, respectively. For Puppet, in the GitHub, Mozilla, OpenStack, and Wikimedia datasets, GLITCH detects at least one of the nine identified security smells in, respectively, 29.6%, 27.5%, 40.1%, and 31.5% of the total scripts. These percentages are slightly higher than those obtained for SLIC.

For all datasets, the dominant security smell is *Hard-coded secret*, followed by *Suspicious comment*. Given that the precision values for these smells tend to be the lowest (see Section 5.3), this suggests that many of these are false positives. However, there is an exception: for Ansible, the second most dominant smell is *Admin by default* (5.7%); since the accuracy of GLITCH for this smell is high, this suggests that there is a substantial number of Ansible scripts that are affected by this problem. The third most dominant security smell differs across the three datasets: for Ansible, it is *Suspicious comment* (5.4%); for Chef, it is *Missing default case statement* (2.1%); and for Puppet, it is *Use of HTTP without TLS* when considering the GitHub dataset (3.7%), *Missing default case statement* when considering the Mozilla dataset (9.9%), *Admin by default* when considering the Openstack and Wikimedia datasets (5.2% and 4.0%, respectively). We note that the high accuracy of GLITCH for the smell *Missing default case statement*, suggests that a substantial number of scripts in the Mozilla dataset are affected by this problem.

5.4.4 Execution times. The execution times of GLITCH, SLIC, and SLAC for the three datasets are shown in Table 14 (in seconds). These times were obtained in a server machine running Debian 10, with 4 Intel(R) Xeon(R) CPU E5-2630 v2 @ 2.60GHz, 64GB RAM, and with a Toshiba MG03ACA100 hard drive. We executed 5 runs for each pair tool/dataset and averaged the obtained execution times. Each run was executed in its own Docker container created from the Docker image we provide in the replication package. Runs from the same set of 5 runs were executed simultaneously. GLITCH is much quicker than SLIC and SLAC when running on Chef or Puppet scripts (speedups vary from 9.14× to 32.07×). SLIC and SLAC respectively call *puppet-lint*[11] and *foodcritic*[12] to analyze each Puppet or Chef script. The overhead of creating a new system process for each script analyzed and other non-related analyses performed by *puppet-lint* and *foodcritic* are the main reason for the slower execution times. However, when compared to SLAC, GLITCH takes more than double the time to run on the Ansible dataset. This happens because we parse Ansible scripts using *ruamel.yaml*, a Python package slower than the popular *yaml* package, but with the advantage of saving comments in the AST.

[11] https://github.com/rodjek/puppet-lint
[12] http://www.foodcritic.io/

Table 14: The average execution times between 5 runs (seconds).

			Puppet			
Tool	**Ansible**	**Chef**	**GH**	**MOZ**	**OST**	**WIK**
SLIC/SLAC	797	76,153	2,615	380	915	866
GLITCH	1,668	8,335	86	14	35	27
Speedup	0.48	9.14	30.41	27.14	26.14	32.07

6 DISCUSSION

In this section, we answer the research questions listed in Section 5.1, we discuss the practical implications of our findings, and we outline potential threats to the validity of our work.

6.1 Answers to Research Questions

Given the findings reported in the previous section, we answer the research questions posed in Section 5.1 as follows:

Answer to RQ1 [Abstraction]. *Can our intermediate representation model IaC scripts and support automated detection of security smells?* Yes. We demonstrate that our intermediate representation can model scripts written in different IaC technologies, with our current implementation supporting Ansible, Chef, and Puppet. We also define and implement nine rules that operate on the intermediate representation and that can be used to detect security smells. New rules can be easily created and existing rules can be easily changed. We evaluate our implementation with three large datasets containing 196,755 IaC scripts and 12,281,251 LOC. This strongly suggests that the intermediate representation is robust enough to support a large variety of IaC scripts.

Answer to RQ2 [Accuracy and Performance]. *How does GLITCH compare with existing state-of-art tools for detecting security smells in terms of accuracy and performance?* As shown in Tables 8, 9, and 10, the average precision and recall values of GLITCH are substantially better than the average precision and recall values of SLIC and SLAC. For Puppet, average precision and average recall improved by 8 and 13 percentage points, respectively. For Ansible, average precision improved 10 percentage points and average recall improved 8 percentage points. For Chef, the improvement was more expressive: the average precision and average recall improved by 28 and 26 percentage points, respectively. In terms of performance, as Table 14 shows, GLITCH is much faster analyzing Chef and Puppet scripts than tools such as SLIC or SLAC (speedups vary from 9.14× to 32.07×). For Ansible, GLITCH takes more than twice as long than SLAC, but it can still analyze IaC scripts in an acceptable amount of time (e.g., it took us around 28 minutes to analyze more than 5M LOC).

Answer to RQ3 [Frequency]. *How frequently do security smells occur in IaC scripts?* All categories of security smells are identified across all datasets considered in this work. For Ansible, GLITCH detects at least one of the eight identified security smells in 19.6% of the total scripts. For Chef, it detects at least one of the nine identified security smells in 10.4% of the total scripts. For Puppet, in the GitHub, Mozilla, OpenStack, and Wikimedia datasets, GLITCH detects at least one of the nine identified security smells in, respectively, 29.6%, 27.5%, 40.1%, and 31.5% of the total scripts.

In general, the most dominant security smell is *Hard-coded secret*, followed by *Suspicious comment*. Given that the precision values for these smells tend to be the lowest (see Section 5.3), this suggests that many of these are false positives. For Ansible, the second most dominant smell is *Admin by default* (5.7%). For Chef and for the Mozilla dataset of Puppet scripts, the third most dominant smell is *Missing default case statement* (2.1% and 9.9%). Since the accuracy of GLITCH for these smells is high, this suggests that there is a substantial number of Ansible and Chef scripts that are affected by these problems.

6.2 Practical Implications and Challenges

The main practical implication of this work is that it is now possible to implement new rules to detect code smells that can be immediately applied to a variety of IaC technologies. Some of the rules currently implemented have very high precision and recall, and have been used to identify a considerable number of smells in our study. This suggests that IaC practitioners can benefit if they focus first on smells of those specific categories (e.g., *Admin by default* and *Missing default case statement*). Also, during the development of this work it became clear that there are no open replication packages that IaC researchers and practitioners can use. Therefore, we constructed a new open-source replication package that can be used by the community.

We identify three main challenges: **(1) Quality.** This challenge is about increasing the precision and recall of GLITCH. For example, the definitions of some rules (e.g., those that use many keywords) still report a considerable number of false positives (e.g. *Hard-coded secret*). Future work should be invested in improving the quality of the rules that GLITCH implements. Addressing this challenge is perhaps an important step toward real-life adoption of GLITCH. **(2) Scope.** This challenge is about extending GLITCH to support more IaC technologies and to detect more vulnerabilities. For example, it would be interesting to extend GLITCH to support Terraform and to support the detection of faults regarding ordering violations [27] or intra-update sniping vulnerabilities [10]. By addressing this challenge, we will be in a better position to provide a more precise characterization of the expressiveness of the smell detection engine. **(3) Development process.** This challenge is about integrating these tools into the development process, thus contributing to real-life adoption. The following could bring added value: integration with continuous integration (CI) processes (e.g., GitHub actions), integration with popular IDEs, interactive reports (e.g., highlight vulnerable code), and explainable warnings. Since GLITCH is much faster than other state-of-the-art tools for analyzing Chef and Puppet scripts, it becomes more appealing to integrate GLITCH as part of a CI workflow [8].

6.3 Threats to Validity

A threat to conclusion validity is that the identification of security smells in the oracle datasets are susceptible to the subjectivity of the raters. We mitigated this by using three raters, with two of them not being authors of the paper and with experience in IaC technologies and/or cybersecurity. Also, we only kept the classifications where at least two raters agreed.

A threat to internal validity is that, due to the complexity and generality of GLITCH, there may exist implementation bugs in the codebase. We extensively tested the tool to mitigate this risk. Furthermore, all our code and datasets are publicly available for other researchers and potential users to check the validity of the results. Finally, the detection accuracy of GLITCH depends on the rules that we have provided in Table 3. These rules are heuristic-driven and can result in false positives and false negatives.

A threat to external validity is that, since we focus on Ansible, Chef, and Puppet scripts, our findings may not be generalizable to other IaC technologies. Moreover, in its current form, our internal representation might not be rich enough to detect other categories of security smells not considered in this paper. We mitigated this risk by ensuring that the concepts modeled by the intermediate representation are as general as possible and by choosing to demonstrate its validity using three different IaC technologies that, as shown in Table 1, have different characteristics (procedural vs declarative, different configuration setup, etc.). Also, the classification of security smells used is subject to practitioner interpretation and their relevance may vary from one practitioner to another. To mitigate this, we followed classifications established by previous work [18, 19]. Finally, all the datasets used in our work are from open-source projects and not from proprietary sources.

7 CONCLUSION

This paper shows that it is possible and beneficial to consistently detect security smells across different IaC technologies. We conducted a large-scale empirical study where we consider nine security smells documented in the literature. We found that all categories of security smells are identified across all datasets. We identified some smells that might affect many IaC projects.

An outcome of this work is GLITCH, a new technology-agnostic framework that allows polyglot security smell detection in IaC scripts, by transforming them into a new intermediate representation on which different security smell detectors can be defined. GLITCH currently supports the detection of nine different security smells in Puppet, Ansible, or Chef scripts. Our evaluation not only shows that GLITCH can reduce the effort of writing security smell analyses for multiple IaC technologies, but also that it has higher precision and recall than the current state-of-the-art tools.

All our code and datasets are publicly available. We argue that GLITCH and the datasets that we created and made available in our replication package are very valuable assets for driving reproducible research in the analysis of IaC scripts.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers, whose comments and corrections have led to significant improvements. We would also like to thank Akond Rahman, who very kindly provided access to datasets used in the evaluation of the tools SLIC and SLAC. The first author is funded by the Advanced Computing/EuroCC MSc Fellows Programme, which is funded by EuroHPC under grant agreement No 951732. This project was supported by national funds through FCT under project UIDB/50021/2020, and by project ANI 045917 funded by FEDER and FCT.

REFERENCES

[1] Ahmad Alnafessah, Alim Ul Gias, Runan Wang, Lulai Zhu, Giuliano Casale, and Antonio Filieri. 2021. Quality-Aware DevOps Research: Where Do We Stand? *IEEE Access* 9 (2021), 44476–44489.

[2] James Fryman. 2014. DNS outage post mortem. https://github.blog/2014-01-18-dns-outage-post-mortem/ Accessed: 3 May 2022.

[3] Michele Guerriero, Martin Garriga, Damian A Tamburri, and Fabio Palomba. 2019. Adoption, support, and challenges of infrastructure-as-code: Insights from industry. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 580–589.

[4] Oliver Hanappi, Waldemar Hummer, and Schahram Dustdar. 2016. Asserting reliable convergence for configuration management scripts. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 328–343.

[5] Rebecca Hersher. 2017. Amazon and the $150 Million typo. https://www.npr.org/sections/thetwo-way/2017/03/03/518322734/amazon-and-the-150-million-typo?t=1651588365675 Accessed: 3 May 2022.

[6] Katsuhiko Ikeshita, Fuyuki Ishikawa, and Shinichi Honiden. 2017. Test suite reduction in idempotence testing of infrastructure as code. In *International Conference on Tests and Proofs*. Springer, 98–115.

[7] Yujuan Jiang and Bram Adams. 2015. Co-evolution of infrastructure and source code-an empirical study. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 45–55.

[8] Xianhao Jin and Francisco Servant. 2021. What helped, and what did not? An Evaluation of the Strategies to Improve Continuous Integration. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 213–225.

[9] John C Kelly, Joseph S Sherif, and Jonathan Hops. 1992. An analysis of defect densities found during software inspections. *Journal of Systems and Software* 17, 2 (1992), 111–117.

[10] Julien Lepiller, Ruzica Piskac, Martin Schäf, and Mark Santolucito. 2021. Analyzing Infrastructure as Code to Prevent Intra-update Sniping Vulnerabilities.. In *TACAS (2)*. 105–123.

[11] MITRE. 2022. CWE-Common Weakness Enumeration. https://cwe.mitre.org/index.html.

[12] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating github for engineered software projects. *Empirical Software Engineering* 22, 6 (2017), 3219–3253.

[13] Pars Mutaf. 1999. Defending against a Denial-of-Service Attack on TCP.. In *Recent Advances in Intrusion Detection*.

[14] National Institute of Standards and Technology. 2014. Security and Privacy Controls for Federal Information Systems and Organizations. https://www.nist.gov/publications/security-and-privacy-controls-federal-information-systems-and-organizations-including-0.

[15] Akond Rahman, Effat Farhana, Chris Parnin, and Laurie Williams. 2020. Gang of eight: A defect taxonomy for infrastructure as code scripts. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 752–764.

[16] Akond Rahman, Effat Farhana, and Laurie Williams. 2020. The 'as code'activities: development anti-patterns for infrastructure as code. *Empirical Software Engineering* 25, 5 (2020), 3430–3467.

[17] Akond Rahman, Rezvan Mahdavi-Hezaveh, and Laurie Williams. 2019. A systematic mapping study of infrastructure as code research. *Information and Software Technology* 108 (2019), 65–77.

[18] Akond Rahman, Chris Parnin, and Laurie Williams. 2019. The seven sins: Security smells in infrastructure as code scripts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 164–175.

[19] Akond Rahman, Md Rayhanur Rahman, Chris Parnin, and Laurie Williams. 2021. Security smells in ansible and chef scripts: A replication study. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 1 (2021), 1–31.

[20] Akond Rahman and Laurie Williams. 2018. Characterizing defective configuration scripts used for continuous deployment. In *2018 IEEE 11th International conference on software testing, verification and validation (ICST)*. IEEE, 34–45.

[21] Akond Rahman and Laurie Williams. 2019. Source code properties of defective infrastructure as code scripts. *Information and Software Technology* 112 (2019), 148–163.

[22] Eric Rescorla et al. 2000. HTTP over TLS. RFC 2818, May.

[23] Johnny Saldaña. 2021. *The coding manual for qualitative researchers*. sage.

[24] Julian Schwarz, Andreas Steffens, and Horst Lichter. 2018. Code smells in infrastructure as code. In *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*. IEEE, 220–228.

[25] Rian Shambaugh, Aaron Weiss, and Arjun Guha. 2016. Rehearsal: A configuration verification tool for puppet. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 416–430.

[26] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. 2016. Does your configuration code smell?. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 189–200.

[27] Thodoris Sotiropoulos, Dimitris Mitropoulos, and Diomidis Spinellis. 2020. Practical fault detection in Puppet programs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 26–37.

[28] Eduard Van der Bent, Jurriaan Hage, Joost Visser, and Georgios Gousios. 2018. How good is your puppet? an empirically defined and validated quality model for puppet. In *2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 164–174.