Polyglot Code Smell Detection for Infrastructure as Code with GLITCH

Nuno Saavedra*, João Gonçalves*, Miguel Henriques*, João F. Ferreira* and Alexandra Mendes†
*INESC-ID and IST, University of Lisbon, Lisbon, Portugal
†HASLab / INESC TEC & Faculty of Engineering, University of Porto, Porto, Portugal

***Abstract*—This paper presents GLITCH, a new technology-agnostic framework that enables automated polyglot code smell detection for Infrastructure as Code scripts. GLITCH uses an intermediate representation on which different code smell detectors can be defined. It currently supports the detection of nine security smells and nine design & implementation smells in scripts written in Ansible, Chef, Docker, Puppet, or Terraform. Studies conducted with GLITCH not only show that GLITCH can reduce the effort of writing code smell analyses for multiple IaC technologies, but also that it has higher precision and recall than current state-of-the-art tools. A video describing and demonstrating GLITCH is available at: https://youtu.be/E4RhCcZjWbk.**

***Index Terms*—devops, infrastructure as code, code smells, security smells, design smells, implementation smells, Ansible, Chef, Docker, Puppet, Terraform, intermediate model, static analysis**

I. Introduction

Infrastructure as Code (IaC) is the process of managing IT infrastructure via programmable configuration files (also called IaC scripts). In recent years, several tools for detecting code smells in IaC scripts have been proposed [1]–[6]. These tools are very valuable, since they cover a wide range of code smells and several 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 code smells is inconsistent for different IaC technologies. For example, Figure 1 presents a line of code with a comment taken from the project *puppet-foreman* by The Foreman[1]. For this example, Schwarz et al.'s tool [2] detects the *Long Statement* smell because the line has exactly 140 characters, and the tool reports the smell for lines with 140 characters or more. However, if we use the tool Puppeteer [1] to analyze the same line in Puppet, the smell will not be detected since Puppeteer only detects the *Long Statement* smell for lines with more than 140 characters. Even though this example might be considered a minor problem, it shows that having separate implementations for code smell analysis can easily lead to inconsistent code smell detection. Ensuring consistency is particularly important for projects that use more than one IaC technology.

To address this problem, we present GLITCH, a technology-agnostic framework that enables automated polyglot smell detection by transforming IaC scripts into an intermediate representation, on which different code smell detectors can be defined. GLITCH currently supports the detection of nine security smells and nine design & implementation smells in scripts written in Ansible, Chef, Docker, Puppet, or Terraform. A previous study compared GLITCH with state-of-the-art security smell detectors [7]. In this paper we introduce an extended version of GLITCH that supports 9 previously unreported design & implementation code smell detectors and that extends the original tool with support for Docker and Terraform. We also present preliminary results on the detection of design & implementation smells for Ansible, Chef, and Puppet.

The envisioned users of GLITCH are DevOps engineers and system administrators who have to develop or maintain IaC scripts. GLITCH is particularly helpful in environments where multiple IaC technologies are being used, which happens in many organizations. Moreover, since we created and make available three large datasets containing 196,756 IaC scripts (with a total of 12,281,383 LOC), and three oracle datasets for security smells (one for each IaC technology supported by GLITCH), we argue that GLITCH can also be used by researchers interested in software quality of IaC scripts.

GLITCH is open-source and is available online at: https://github.com/sr-lab/GLITCH. A Docker container to replicate the study on design & implementation smells presented in this paper is available at: https://doi.org/10.6084/m9.figshare.21407058.v1.

II. GLITCH

This section describes GLITCH, providing an overview of the intermediate language and the methodology for smell detection. It also provides information about GLITCH's implementation and how it can be used. Figure 2 shows an overview of GLITCH's architecture.

A. Intermediate Language

The intermediate representation used by GLITCH captures similar concepts from different IaC technologies. The representation uses a hierarchical structure, as shown in Figure 2. Projects represent a generic folder that may contain several modules and unit blocks. *Modules* are the top component from each structure and they agglomerate the scripts necessary to execute a specific functionality (corresponding to roles in Ansible, cookbooks in Chef, and modules in Puppet

[1]https://github.com/theforeman/puppet-foreman/blob/1d09876d7838bcd133add6266f4ba19b936ccb6c/manifests/init.pp#L57

```
# $unattended_url::              URL hosts will retrieve templates from during build (normally http as many installers don't support https)
```

Fig. 1: Line of code taken from puppet-foreman by The Foreman. Issues with state-of-the-art tools: Schwarz et al.'s tool [2] reports the smell "Long Statement" since the line has exactly 140 characters. Puppeteer [1] does not report the smell since its rule only considers lines with more than 140 characters.

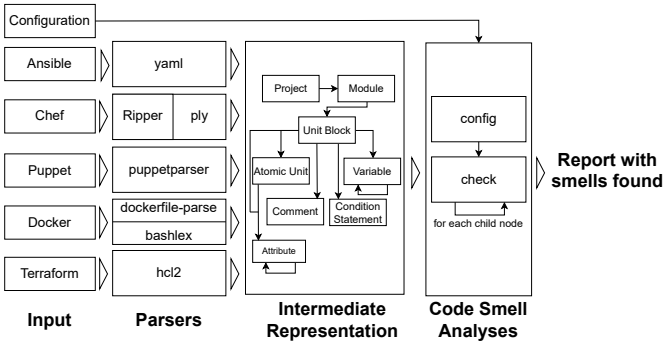


Fig. 2: GLITCH's architecture overview.

and Terraform). *Unit Blocks* correspond to the IaC scripts themselves or to a group of atomic units (corresponding to playbooks in Ansible, recipes in Chef, scripts and build stages in Docker, classes in Puppet, and configuration files in Terraform). Finally, *Atomic Units* define the system components we want to change and the actions we want to perform on them (corresponding to tasks in Ansible, bash commands in Docker, and resources in Chef, Puppet, and Terraform). Figure 3 shows a graph-based visualization of how our intermediate representation models the scripts shown in Figure 3a and Figure 3b. More details about the intermediate representation, including its abstract syntax, can be found in the authors' study on security smells [7]. We note that minor changes were performed to GLITCH's original intermediate representation: since in technologies such as Ansible, Puppet, and Terraform, we can have nested attributes and variables, we adapted the intermediate representation to consider these nested constructs.[2]

B. Smell Detection

To implement a new analysis in GLITCH, developers must extend the abstract class *RuleVisitor*. The *RuleVisitor* class has an abstract method for each component in our intermediate representation. The abstract methods should be implemented to return the list of smells detected on a component. Since the developers have the freedom to implement the abstract methods as they need, they are not bounded to a specific analysis approach. For example, in a previous study focused on security smells [7], we used a rule-based approach to detect the smells. However, to implement the analyses for the design & implementation smells, we used an algorithmic approach.

For each Visitor, GLITCH traverses the nodes in the intermediate representation using a depth-first search (DFS). Starting in the initial node (a Project, a Module, or a Unit Block), it executes the DFS considering each collection inside the node as its children. Each node may have more than one code smell, and so every analysis is applied, even if a smell was already identified for that node. GLITCH allows the definition of different configurations to identify code smells. In a Visitor, the developers can define a list of variables, whose values can be changed according to the configuration passed to the tool. Use cases include modifying keywords used for pattern detection or defining a technology-specific token. Configurations allow users to tweak the tool to best suit the needs of the IaC developers and to better adapt to each IaC technology. If specific behaviour for a technology is required, the Visitors select, in their constructor and according to the technology, objects that check a certain smell, which are called later in the implementation of the methods to check the components of the intermediate representation (see Figure 4).

C. Implementation and Usage

GLITCH is implemented in Python and it currently supports the analysis of Ansible, Chef, Docker, Puppet, and Terraform scripts. It transforms the original scripts into its intermediate representation and then attempts to detect code smells as described above. To parse Ansible scripts, it uses the *ruamel.yaml* package[4] for Python. The Chef scripts are parsed using Ripper,[5] a script parser for Ruby. We developed a parser for Ripper's output using a package called *ply*.[6] For Docker scripts, we use the packages *dockerfile-parse*[7] and *bashlex*[8]. For Puppet scripts, we developed our own parser[9] using the same *ply* package. Finally, for Terraform scripts, we use a sligthly modified version of the *python-hcl2* package.[10]

Using GLITCH: GLITCH provides a command-line interface. Besides the path of the file or folder to analyze, other relevant available options are:

- **`--tech [ansible|docker|chef|puppet|terraform]`**: The IaC technology in which the scripts analyzed are written. This option is required.

[2]An updated description of the intermediate representation and its abstract syntax is available in GLITCH's Wiki: https://github.com/sr-lab/GLITCH/wiki/3.-Intermediate-representation

[3]https://github.com/wikimedia/operations-puppet-cdh4/blob/eb24f44669bf6b83fa0ee37bfd1742fb2e297d4c/manifests/hive/metastore/mysql.pp#L45

[4]https://pypi.org/project/ruamel.yaml/

[5]https://github.com/ruby/ruby/tree/master/ext/ripper

[6]https://github.com/dabeaz/ply

[7]https://pypi.org/project/dockerfile-parse

[8]https://pypi.org/project/bashlex

[9]https://github.com/Nfsaavedra/puppetparser

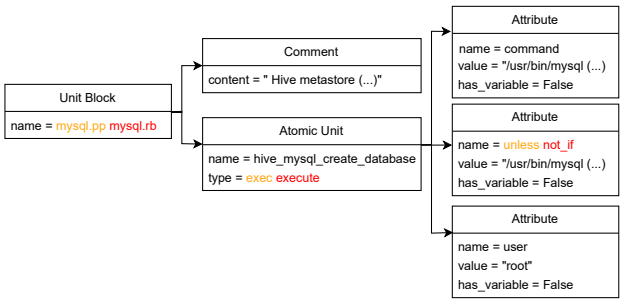
[10]https://github.com/joaotgoncalves/python-hcl2

```
# Hive metastore MySQL database need a (...)
exec { 'hive_mysql_create_database':
  command => "/usr/bin/mysql (...)",
  unless => "/usr/bin/mysql (...)",
  user => 'root',
}
```

(a) Part of a Puppet script from a CDH module by Wikimedia[3].

```
# Hive metastore MySQL database need a (...)
execute 'hive_mysql_create_database' do
    command "/usr/bin/mysql (...)"
    not_if "/usr/bin/mysql (...)"
    user 'root'
end
```

(b) Script above written in Chef.



(c) Graph-based representation of the scripts on the left using our intermediate representation. In black and orange: the representation of the Puppet script from Figure 3a. In black and red: the representation of the Chef script from Figure 3b.

Fig. 3: Translation of scripts to GLITCH's intermediate representation.

```
if tech == Tech.ansible:
    self.imp_align = \
        DesignVisitor.AnsibleImproperAlignmentSmell()
elif tech == Tech.puppet:
    self.imp_align = \
        DesignVisitor.PuppetImproperAlignmentSmell()
(...)
errors += self.imp_align.check(u, u.path)
```

Fig. 4: Implementation of specific behaviour when creating new rules.

- **`--smells [design|security]`**: Type of smells being analyzed. Currently it supports nine security smells [7] and nine design & implementation smells.
- **`--config PATH`**: The path for a config file. Otherwise the default config will be used.
- **`--tableformat [prettytable|latex]`**: The presentation format of the tables that show stats about the analysis.
- **`--csv`**: This flag produces the output in CSV format.

The last two options are particularly useful for researchers who need to analyze datasets of IaC scripts and generate CSV data that can be automatically analyzed or tables that can be directly added to research papers.

Adding support for new IaC technologies: To add a new technology, one needs to create a new subclass of the abstract class *Parser* and to implement the abstract methods that map the constructs of that technology to the intermediate representation (these methods are *parse_file*, *parse_folder*, and *parse_module*). After that, the developer only needs to change the command-line tool to consider the new technology.

III. Evaluation

Already conducted studies: In a previous study focused on security smells for Ansible, Chef, and Puppet [7], we used GLITCH to analyze three large datasets containing 196,756 IaC scripts and 12,281,383 LOC. That study demonstrated that GLITCH is robust enough to support a large variety of IaC scripts. It also showed that GLITCH has higher precision and recall than current state-of-the-art tools.

TABLE I: Comparison of **G**LITCH with **P**uppeteer [1] and **S**chwarz et al.'s tool [2]. T_1/T_2 is the percentage of smells detected by T_1 that are also detected by T_2.

Smells	P/G (%)	G/P (%)	S/G (%)	G/S (%)
Avoid comments	-	-	100.0	100.0
Duplicate block	-	-	100.0	97.6
Improper alignment	98.4	89.9	33.3	42.9
Long resource	-	-	82.4	82.4
Long statement	100.0	91.4	100.0	100.0
Misplaced attribute	100.0	100.0	97.8	97.8
Multifaceted abstraction	-	-	100.0	57.1
Too many variables	-	-	40.0	80.0
Unguarded variable	100.0	92.3	-	-
Average	99.6	93.4	81.7	82.2

Ongoing studies: We are currently conducting empirical studies similar to Saavedra and Ferreira's study [7], but focused on the new features implemented in GLITCH: the new design & implementation smells and the two new IaC technologies, Docker and Terraform. Due to space limitations, in this paper we focus on presenting preliminary results on the detection of design & implementation smells for Ansible, Chef, and Puppet. We use the same three datasets mentioned above; readers interested in the datasets' attributes should consult Table 5 of Saavedra and Ferreira's paper [7].

In Table I, we compare GLITCH to two state-of-the-art tools that detect design & implementation smells in Puppet [1] and Chef [2]. We do not compare GLITCH to an Ansible tool since, to the best of our knowledge, GLITCH is the **first** tool that detects the design & implementation smells considered in our study in Ansible scripts. We ran the tools on a subset of the datasets considered. The subset was created by randomly selecting 20 files for each smell, with the smell being detected on each of the selected files. This resulted in a total of 80 Puppet files and 160 Chef files. Afterwards, we compared the results of GLITCH with the results of the other two tools when considering these files. We identified smells with the same path, category, and location as reported by each tool. In some cases, the tools do not output the same line number, although they fundamentally detect the same smell. For these cases, we

had to manually inspect the files and check whether the tools agree, which was the reason why we only selected a subset of files. The replication package has a script that automatically solves the cases we found on this subset of files.

GLITCH can detect almost every smell detected by Puppeteer [1] (first column). The value for *Improper alignment* is slightly below 100% because GLITCH does not consider the alignment in hashes[11] since these structures, when used as values, are still represented as strings in our intermediate representation. The second column shows that Puppeteer detects a lower percentage of the smells identified by GLITCH. For *Improper alignment*, the reason is that GLITCH, in contrast to Puppeteer, follows the Puppet style guides,[12] which state that the hash rocket for attributes in a resource should be **only one space** ahead of the longest attribute name.

When verifying if GLITCH was able to detect the smells identified by the tool developed by Schwarz et al. [2], there are two smells with lower percentage values: **(1)** *Improper alignment* and **(2)** *Too many variables* (third column). This happens because **(1)** Schwarz et al.'s tool presents false positives for attributes with names such as *variables* and *attributes* because they have structured values which are indented in the lines following the name of the attribute; **(2)** GLITCH does not consider variable references when calculating the ratio between variables and lines of code. Comparing the ability of Schwarz et al.'s tool to detect the smells found by GLITCH (fourth column), there are two smells with a lower percentage: **(1)** *Improper alignment* and **(2)** *Multifaceted abstraction*. The main reasons for the lower values are: with respect to **(1)**, GLITCH detects true positives that are not detected by the other tool and some false positives also undetected by the other tool (the false positives are due to problems when handling blocks, such as conditionals, inside atomic units); with respect to **(2)**, GLITCH finds true positives that the other tool does not, since Schwarz et al.'s tool does not handle multi-line strings and ignores the pipe character ("|").

IV. Related Work

To the best of our knowledge, GLITCH is the only polyglot code smell detector for IaC scripts. It unifies other tools such as SLIC [3], which detects seven security smells in Puppet scripts, and SLAC [4], which identifies nine in Chef scripts and six in Ansible scripts (using separate implementations). Compared to SLIC and SLAC, GLITCH identifies two more smells in Puppet scripts and two more in Ansible scripts. Other relevant tools are *Puppeteer* [1], which detects design configuration smells, and Schwarz et al.'s tool [2], which detects design & implementation smells. More recent work includes GASEL [6], which takes into consideration control-flow and data-flow information for security smell detection in Ansible scripts. GASEL presents better recall and precision than GLITCH for some smells, but it focuses on a single IaC technology compared to five currently supported by GLITCH.

Finally, some analysis tools for IaC use intermediate representations [8]–[10] to describe file-system manipulations done by IaC scripts. However, to the best of our knowledge, GLITCH is the only smell detector that uses an intermediate representation, allowing a technology-agnostic approach.

V. Conclusion

GLITCH has already proven to be a practical and versatile tool. Despite its intermediate representation being remarkably simple, it is expressive enough to enable the implementation of both security and design & implementation code smells. Furthermore, supporting multiple IaC technologies requires minimal effort, thus reducing the effort of writing code smell analyses for multiple IaC technologies.

Future work includes i) the creation of oracle datasets of design & implementation smells so that we can measure precision and recall, as we have done in previous studies [7]; ii) the refinement of existing rules and the implementation of new rules to detect more smells; iii) the extension of GLITCH to take control-flow and data-flow information into account, so that we can improve precision and recall [6].

Acknowledgments

Thanks to Akond Rahman for the datasets used in the evaluation of the tools SLIC and SLAC and to Carolina Pereira for her help creating the video demonstrating GLITCH. The first author was funded by the Advanced Computing/EuroCC MSc Fellows Programme (EuroHPC grant agreement No 951732). This project was supported by national funds through FCT under project UIDB/50021/2020.

[11] https://puppet.com/docs/puppet/latest/lang_data_hash.html

[12] https://puppet.com/docs/puppet/latest/style_guide.html

References

[1] T. Sharma, M. Fragkoulis, and D. Spinellis, "Does your configuration code smell?" in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2016, pp. 189–200.

[2] J. Schwarz, A. Steffens, and H. Lichter, "Code smells in infrastructure as code," in *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*. IEEE, 2018.

[3] 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)*. IEEE, 2019, pp. 164–175.

[4] A. Rahman, M. R. Rahman, C. Parnin, and L. Williams, "Security smells in ansible and chef scripts: A replication study," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 1, 2021.

[5] S. Reis, R. Abreu, M. d'Amorim, and D. Fortunato, "Leveraging practitioners' feedback to improve a security linter," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[6] R. Opdebeeck and A. Zerouali, "Control and data flow in security smell detection for infrastructure as code: Is it worth the effort?" in *Proc. of the 20th Int. Conf. on Mining Software Repositories (MSR 2023)*, 2023.

[7] N. Saavedra and J. Ferreira, "GLITCH: Automated Polyglot Security Smell Detection in Infrastructure as Code," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, Preprint available: https://arxiv.org/abs/2205.14371.

[8] K. Ikeshita, F. Ishikawa, and S. Honiden, "Test suite reduction in idempotence testing of infrastructure as code," in *International Conference on Tests and Proofs*. Springer, 2017, pp. 98–115.

[9] R. Shambaugh, A. Weiss, and A. Guha, "Rehearsal: A configuration verification tool for puppet," in *PLDI*, 2016.

[10] T. Sotiropoulos, D. Mitropoulos, and D. Spinellis, "Practical fault detection in Puppet programs," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 26–37.