Polyglot Code Smell Detection for Infrastructure as Code with GLITCH

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 a process that facilitates the provision of scalable and reproducible environments. Due to its wide use and importance, there have been several efforts to automatically detect code smells in IaC scripts. However, and even though many code smells are technology-agnostic, most efforts tend to focus on a specific IaC technology. 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.

To address this challenge, we present GLITCH, a new technology-agnostic framework that enables automated polyglot smell detection by transforming IaC scripts into an intermediate representation, on which different smell detectors can be defined. GLITCH currently supports the detection of nine different security smells and nine design & implementation smells in scripts written in Puppet, Ansible, or Chef. Studies conducted with GLITCH 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 current state-of-the-art tools. A video describing and demonstrating GLITCH is available at: https://youtu.be/E4RhCcZjWbk.

CCS CONCEPTS

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

KEYWORDS

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

ACM Reference Format:
Nuno Saavedra and João F. Ferreira. 2022. Polyglot Code Smell Detection for Infrastructure as Code with GLITCH. In *ASE '22: 37th IEEE/ACM International Conference on Automated Software Engineering, October 10–14, 2022, Ann Arbor, Michigan, United States.* ACM, New York, NY, USA, 4 pages. https://doi.org/XXXXXXX.XXXXXXX

ASE '22, October 10–14, 2022, Ann Arbor, Michigan, United States

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 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, 2, 4, 5]. 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 security smells is inconsistent for different IaC technologies. For example, Figure 1a presents part of a Chef script with design & implementation smells taken from the project *chef-bcpc* by Bloomberg.[1] For this example, Schwarz et al's tool [4] detects the *Long Resource* smell because the *execute* method has more than 7 LoC. However, if we use the tool Puppeteer [5] to analyze the corresponding script in Puppet (Figure 1b), which has the same problem, the smell will not be detected because Puppeteer does not implement the detection of this smell.

To address these issues, we present GLITCH, a new 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 different security smells and nine different design & implementation smells in scripts written in Puppet, Ansible, or Chef. Previous experiences compared GLITCH with state-of-the-art security smell detectors [3]. In this paper, we further compare GLITCH with state-of-the-art design & implementation smell detectors. All the results obtained so far 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 current state-of-the-art tools.

The envisioned users of GLITCH are 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 as a Docker container at[2]: https://drive.google.com/file/d/181nMX5MIzch056mlTPDt_7rtvmqLvqxu/view?usp=sharing

[1] https://github.com/bloomberg/chef-bcpc/blob/6707d90d5aebd1b971f24759ffbdc9218c0a3af6/chef/cookbooks/bcpc/recipes/rally.rb#L79

[2] **Note to reviewers:** in the final version of this paper, we will include a link to GLITCH's GitHub repository and upload the Docker image to DockerHub.

```
execute 'install rally in virtualenv' do
  environment env
  retries 3
  user 'rally'
  command <<-EOH
    virtualenv --no-download #{venv_dir} -p /usr/bin/python3
    . #{venv_dir}/bin/activate
    pip install 'pip>=19.1.1'
    pip install 'decorator<=4.4.2'
    pip install 'jinja2<3.0.0'
    pip install rally-openstack==(...)
  EOH
  not_if "rally --version | grep (...)"
end
```

(a) Part of a Chef script from chef-bcpc by bloomberg

```
exec { 'install rally in virtualenv':
  environment => $env,
  tries       => 3,
  user        => 'rally',
  command     => [
    'virtualenv --no-download ${venv_dir} -p /usr/bin/python3',
    '. ${venv_dir}/bin/activate',
    "pip install 'pip>=19.1.1'",
    "pip install 'decorator<=4.4.2'",
    "pip install 'jinja2<3.0.0'",
    "pip install rally-openstack==(...)"
  ],
  unless      => "rally --version | grep (...)"
}
```

(b) Snippet on the left written in Puppet

Figure 1: Issues with state-of-the-art tools: Schwarz et al's tool [4] reports the smell "Long Resource" for script (a); Puppeteer [5] does not report the smell because it does not implement it (b).

2 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.

2.1 Intermediate Language

The intermediate representation used by GLITCH is able to capture similar concepts from different IaC technologies, while assuring it is expressive enough to apply analyses that identify code smells. The representation uses a hierarchical structure. At the top-level, it 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 high-level code structures commonly present in IaC technologies, such as Ansible, Chef and Puppet. *Modules* are the top component from each structure and they agglomerate the scripts necessary to execute a specific functionality (they correspond to roles in Ansible, cookbooks in Chef, and modules in Puppet). 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. They correspond to playbooks in Ansible, recipes in Chef, and classes in Puppet. 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. They correspond to tasks in Ansible, and resources in both Chef and Puppet. Unit blocks can have *attribute* definitions, *variable* definitions, and *conditions*. Atomic units have attribute definitions. When values in attribute and variable definitions use variables, the field *has_variable* is true. Figure 2 shows a graph-based visualization of how our intermediate representation models the scripts shown in Figure 1a and Figure 1b. More details about the intermediate representation, including its abstract syntax, can be found in the authors' study on security smells [3].

2.2 Smell Detection

GLITCH uses a rule-based approach to detect code smells. The formalism used to define rules is similar to the one used by SLIC [1] and SLAC [2]. For example, the security smell *Admin by default*, which consists in specifying default users as administrative users

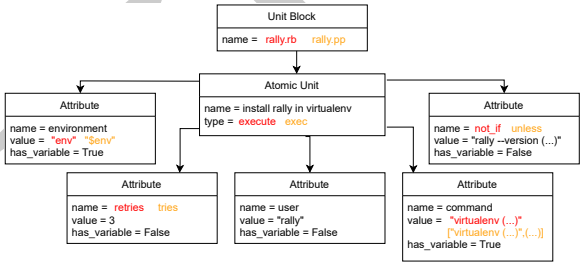

Figure 2: Graph-based representation of the scripts in Figure 1 using our intermediate representation. In black and red: the representation of the Chef script from Figure 1a. In black and orange: the representation of the Puppet script from Figure 1b.

is captured in the following rule:

$$(isAttribute(x) \vee isVariable(x)) \wedge (isUser(x.name) \vee isRole(x.name)) \wedge isAdmin(x.value) \wedge \neg x.has\_variable$$

The rule can be read as: if an attribute or variable is defining a user or role and its value represents an hard-coded user with elevated privileges, then the smell is detected. Here, x denotes a node in the intermediate representation. The functions *isAttribute* and *isVariable* verify the type of instance being analyzed. 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 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. In the example above, the function *isUser*, *isRole*, and *isAdmin* are string patterns that can vary according to each IaC technology. GLITCH allows the definition of different configurations to identify code smells. These configurations change the keywords used in the available string patterns. Configurations allow users to tweak the tool to best suit the needs of the IaC developers and to better adapt to each IaC technology.

2.3 Implementation and Usage

GLITCH is implemented in Python and it currently supports the analysis of Ansible, Chef, and Puppet scripts. Our implementation

```
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)
```

Figure 3: Implementation of specific behaviour when creating new rules.

transforms the original scripts into our intermediate representation and then attempts to detect code smells as described above. To parse the Ansible scripts we use the *ruamel.yaml* package[3] for Python. The Chef scripts are parsed using Ripper,[4] a script parser for Ruby. We developed a parser for Ripper's output using a package called *ply*.[5] Finally, for Puppet scripts, we developed our own parser[6] 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.

Using GLITCH. GLITCH provides a command-line interface that receives a path of the file or folder to analyze and supports multiple options. Some relevant available options are:

- **`-tech [ansible|chef|puppet]`**: The IaC technology in which the scripts analyzed are written. This option is required.
- **`-smells [design|security]`**: The type of smells being analyzed. Currently it supports nine security smells [3] and nine design & implementation smells.
- **`-dataset`**: This flag is used if the folder being analyzed is a dataset. A dataset is a folder with subfolders to be analyzed.
- **`-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 (e.g., Table 3 was automatically generated by GLITCH).

Adding new rules. To create new rules, the developers only need to implement a new Visitor that checks each component of the intermediate representation. 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 3). An illustrative example of this can be seen in the file */GLITCH/glitch/analysis/design.py*, which is available in the artefact provided.

3 EVALUATION

In a previous study focused on security smells [3], 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. In this paper, we present new results on the detection of design & implementation smells. We use the same three datasets, whose attributes are presented in Table 1.

In Table 2, we compare GLITCH to two state-of-the-art tools that detect design & implementation smells in Puppet [5] and Chef [4] scripts. We ran the tools on the datasets described in Table 1, but for Table 2, we only considered a subset of files. 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 (e.g., one shows the line number of the atomic unit, the other shows the line number of the attribute). 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.

We verified that GLITCH is able to detect almost every smell detected by Puppeteer [5] (first column). The value for *Unguarded variable* is lower because GLITCH does not consider the names of variables defined in other files. For the other smells, the main reason was that GLITCH incorrectly parsed a file or was unable to parse the file. The second column shows that Puppeteer detects a lower percentage of the smells identified by GLITCH. For *Improper alignment*, the main reason is that GLITCH, in contrast to Puppeteer, follows the Puppet style guides,[7] which state that the hash rocket for attributes in a resource should be *only one space* ahead of the longest attribute name. For the other smells, we were not able to reach a conclusion for why Puppeteer was not able to detect them, however the smells identified by GLITCH, from our perspective, are true positives.

When verifying if GLITCH was able to detect the smells identified by the tool developed by Schwarz et al. [4], there are three smells with lower percentage values: **(1)** *Improper alignment*, **(2)** *Too many variables*, and **(3)** *Long resource* (third column). The main reasons for each smell are: **(1)** the 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; **(3)** problems related to parsing in GLITCH. Comparing the ability of Schwarz et al.'s tool to detect the smells found by GLITCH (fourth column), there are five smells with a lower percentage: **(1)** *Improper alignment*; **(2)** *Avoid comments*; **(3)** *Long statement*; **(4)** *Multifaceted abstraction*; **(5)** *Duplicate block*. The main reasons for the lower values are: **(1)** GLITCH detects true positives that were not detected by the other tool and GLITCH has some problems when handling blocks, such as conditionals, inside atomic units; **(2)** GLITCH finds all comments but the Schwarz et al.'s tool does not; **(3) and (5)** for these smells, the Schwarz et al.'s tool considers only the files inside specific folders, ignoring other folders that may contain IaC scripts; **(4)** GLITCH finds true positives that

[3] https://pypi.org/project/ruamel.yaml/
[4] https://github.com/ruby/ruby/tree/master/ext/ripper
[5] https://github.com/dabeaz/ply
[6] https://github.com/Nfsaavedra/puppetparser
[7] https://puppet.com/docs/puppet/latest/style_guide.html

Table 1: Attributes of IaC Datasets. The GLITCH columns have the values for each attribute considering the code GLITCH was able to analyze.

	Ansible		Chef		Puppet							
					GH		MOZ		OST		WIK	
Attribute	**Total**	**GLITCH**	**Total**	**GLITCH**	**Total**	**GLITCH**	**Total**	**GLITCH**	**Total**	**GLITCH**	**Total**	**GLITCH**
Repository count	681	681	439	439	219	219	2	2	61	61	11	11
IaC scripts	108,510	80,952	70,939	29,723	10,009	9,776	1,613	1,613	2,840	2,840	2,845	2,845
LOC (IaC scripts)	5,180,879	3,491,033	6,071,035	1,638,539	610,122	531,577	66,367	66,367	217,843	217,843	135,137	135137

Table 2: Comparison of GLITCH to state-of-the-art tools (Puppeteer [5] and Schwarz et al.'s tool [4]). #{x ∩ y}/#y represents the fraction of smells detected by y that are also present in x.

Smells	$\frac{\#\{\text{GLITCH} \cap \text{Puppeteer}\}}{\#\text{Puppeteer}}(\%)$	$\frac{\#\{\text{GLITCH} \cap \text{Puppeteer}\}}{\#\text{GLITCH}}(\%)$	$\frac{\#\{\text{GLITCH} \cap \text{Schwarz}\}}{\#\text{Schwarz}}(\%)$	$\frac{\#\{\text{GLITCH} \cap \text{Schwarz}\}}{\#\text{GLITCH}}(\%)$
Avoid comments	-	-	79.7	49.2
Duplicate block	-	-	95.9	58.5
Improper alignment	97.4	88.1	31.1	44.1
Long resource	-	-	55.1	94.1
Long statement	100.0	91.4	91.5	55.1
Misplaced attribute	98.4	100.0	96.6	97.7
Multifaceted abstraction	-	-	91.7	56.3
Too many variables	-	-	40.0	80.0
Unguarded variable	92.0	82.1	-	-
Average	97.0	90.4	72.7	66.9

Table 3: Smell occurrences, Smell Density (SD), and Proportion of Scripts (PS) for the Puppet GitHub (GH) dataset

Smell	**Occurrences**	**SD (Smell/KLoC)**	**PS (%)**
Long statement	1689	3.18	7.5
Improper alignment	6449	12.13	12.6
Too many variables	291	0.55	3.0
Duplicate block	2066	3.89	3.2
Unguarded variable	1460	2.75	3.6
Avoid comments	20633	38.81	30.1
Long Resource	470	0.88	3.1
Multifaceted Abstraction	625	1.18	4.3
Misplaced attribute	439	0.83	2.9

the other tool does not because of issues in their implementation that we do not fully understand.

Table 3 shows the results of running GLITCH to detect design & implementation smells on the Puppet Github dataset listed in Table 1. The smells *Avoid comments*, *Improper alignment*, and *Duplicate block* are the three more frequent smells. For lack of space, we do not present the results for the other datasets, but they are available in our replication package (Docker container).

4 CONCLUSION

This paper presents GLITCH, a new tool that enables automated polyglot code smell detection by transforming IaC scripts into an intermediate representation. GLITCH currently supports the detection of nine different security smells and nine design & implementation smells in scripts written in Puppet, Ansible, or Chef. Its robustness, precision, and performance fared positively in a previous large empirical study focused on security smells. This paper presented new data on applying GLITCH to detect design & implementation smells. The results obtained suggest that GLITCH improves the detection of code smells, while 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 [3]; ii) the refinement of existing rules and the implementation of new rules to detect more smells; iii) the extension of GLITCH to support other IaC technologies (e.g., Terraform); iv) the improvement of the current parsers in GLITCH.

ACKNOWLEDGMENTS

We would like to thank Akond Rahman, who very kindly provided access to datasets used in the evaluation of the tools SLIC and SLAC. Also, thanks to Carolina Pereira for her help creating the video demonstrating GLITCH, and to Alexandra Mendes for valuable comments that improved this paper. 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.

REFERENCES

[1] 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.

[2] 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.

[3] Nuno Saavedra and João F. Ferreira. 2022. GLITCH: an Intermediate-Representation-Based Security Analysis for Infrastructure as Code Scripts. Submitted for publication. Preprint available: https://joaoff.com/publication/2022/ase/.

[4] 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.

[5] 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.