TÉCNICO LISBOA

Polyglot Code Smell Detection for Infrastructure as Code

Nuno Filipe Marques Saavedra da Silva

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor: Prof. João Fernando Peixoto Ferreira

Examination Committee

Chairperson: Prof. Pedro Miguel dos Santos Alves Madeira Adão
Supervisor: Prof. João Fernando Peixoto Ferreira
Member of the Committee: Prof. Rui Filipe Lima Maranhão de Abreu

November 2022

“If you keep practicing, you can do anything.”
— Elmo

Acknowledgments

Numa dissertação, o produto final somos nós. Foi esta uma das lições que o meu orientador João me ensinou. Uma tese é feita de perseverança, coragem, busca pelo conhecimento, aprendizagem sobre nós e sobre o que nos rodeia. O conhecimento que a ciência adquire com uma dissertação é o resultado do crescimento de um indivíduo não só academicamente, mas como a nível pessoal. Acredito que grande parte do nosso crescimento vem das relações com os outros, e por essa razão, aproveito este espaço para agradecer às pessoas que me ajudaram ao longo do meu percurso.

Começo por agradecer às professoras Carla Morais e Isabel Silva. A professora Carla fez crescer o meu gosto por matemática e sempre me apoiou ao longo da minha jornada no ensino básico. É um verdadeiro modelo de boa professora. A professora Isabel abraçou toda a minha turma do secundário como uma mãe, conseguia espalhar carinho nas suas aulas. Além disso, criava interesse no meu desinteresse por Português.

Um obrigado ao Miguel Mascarenhas Filipe pelo feedback que me deu na definição do problema descrito neste documento. Uma boa visão da indústria foi essencial no desenvolvimento do meu trabalho. Também gostava de agradecer à Alexandra Mendes pelos comentários que ajudaram a melhorar partes desta tese.

Um grande obrigado ao Luís Revez e à Cassilda Nunes por toda a amizade durante a minha passagem como bolseiro do Departamento de Engenharia Informática. Foi tempo onde aprendi muito e onde a ajuda e amizade deles foi essencial na minha integração.

Gostava também de agradecer ao Gonçalo Moura do Núcleo de Desenvolvimento Académico pela amizade, por todo o apoio que me deu ao longo da tese, por calmamente ouvir todas as minhas reclamações, e pelas preciosas sugestões que me deu ao longo deste desafio.

Um grande obrigado ao meu orientador Prof. João Ferreira. O João foi meu professor nas disciplinas de Engenharia de Software e Linguagens de Programação, onde o vi como uma pessoa extremamente empenhada em tornar aulas universitárias entusiasmantes e interativas, mantendo a qualidade e precisão da informação. Agredeço-lhe por todo o tempo que dedicou à minha tese e por todas as sugestões que me deu, não só a nível deste trabalho, mas também a nível pessoal, e que fizeram de mim uma pessoa melhor. Finalmente, obrigado pela amizade que espero manter por longos anos.

Um grande obrigado também aos colegas e amigos que ganhei durante o meu percurso univer-

sitário. Em especial, agradeço àqueles que desde o ínicio me acompanharam, apoiaram, ensinaram, e com quem partilhei algumas das minhas melhores memórias: ao André Silva pela perícia de me iluminar, mesmo nos dias mais sombrios; ao Bernardo Conde por me conseguir levantar com humor de qualidade quando estou mais em baixo, pelos seus dotes culinários, e por ter sido o meu primeiro amigo no IST; ao David Martins por todo ele ser um exemplo a seguir de grandeza, de força, e de empenho, pelo riso constante, e por me incentivar a procurar patamares mais altos; ao Pedro "Cenoura" Carrott por ter sido o meu grande parceiro neste percurso académico – os trabalhos e momentos com ele foram os mais *danks* – e pela oportunidade de me deixar conhecer a sua brilhante cabeça; ao Pedro Matono por me ensinar os conceitos de perseverança, superação e liberdade, pelas discussões que ganhámos juntos, e por me mostrares que se pode ser feliz atrás de um computador; ao Tiago Manuel Severino Gonçalves por produzir boa disposição suficiente para um país inteiro, pela capacidade de carregar o peso dos outros, e pelos seus grandes abraços. Também gostava de agradecer aos três, que apesar de terem chegado mais tarde, também fazem parte deste bonito grupo de amigos: aos meus díscipulos David Belchior e Eduardo Espadeiro de quem espero muito sucesso académico e pessoal; ao Rodrigo Antunes por ser o exemplo a seguir de um homem a quem se pode chamar rei. Obrigado por terem feito os meus dias na universidade mais facilmente suportáveis e conto com vocês para o resto da minha vida.

O mais especial agradecimento à minha namorada Carolina Pereira. Conheci-te pouco antes de começar a aventura que foi esta tese e desde esse momento que a minha vida mudou imenso. Sou uma pessoa mais motivada, mais feliz, mais completa. Sempre que me senti em baixo ou que quis desistir, estiveste lá para me puxar para cima, para me amar, para dizer-me que sou capaz, para me meter um sorriso na cara que me fazia continuar com confiança. Este trabalho também pertence ao teu apoio. Obrigado por estares aqui e por acreditares incondicionalmente em mim. Continua a ver a nossa vida com esses bonitos olhos.

Finalmente, agradeço a toda a minha família. Em especial, às três pessoas que me criaram e viram crescer: o meu pai Rogério, a minha mãe Dulce, e o meu irmão Bruno. Obrigado ao meu pai por toda a calma que me transmitiu ao longo da vida, pelos sacrifícios que fez por mim, por todo o carinho e amor, por ser um grande exemplo de um homem trabalhador e o melhor pai. Espero ao longo da minha vida conseguir seguir o teu exemplo. Obrigado à minha mãe por toda a preocupação que tem por mim, por todo o carinho e amor, pelas noites sem dormir a cuidar de mim em pequeno, por ser a melhor mãe e por me ter ensinado a ler, escrever, fazer contas, e a ser uma pessoa da qual me consigo orgulhar. És a pessoa mais forte que conheço, mãe. Obrigado ao meu irmão por todo o cuidado que tem comigo, por me proteger, pelos conselhos, pelas discussões, pelo amor, por me fazer rir mesmo quando eu não estava tão bem, por ser o meu melhor amigo e o meu segundo pai. Obrigado por ouvires as minhas dissertações de pensamentos aleatórias sobre esta tese. Agradeço em especial a estas três pessoas

porque sem elas, não só não tinha conseguido acabar esta tese, como não tinha conseguido o percurso que tive até aqui, da qual me orgulho muito. Espero que estejam orgulhosos de mim e que se continuem a orgulhar daqui para a frente.

Thanks to the anonymous reviewers of the papers based on this thesis, whose comments and corrections have led to significant improvements. I would also like to thank Akond Rahman, who very kindly provided access to datasets used in the evaluation of the tools SLIC and SLAC. This work was partially funded by the Advanced Computing/EuroCC MSc Fellows Programme, which is funded by EuroHPC under grant agreement No 951732.

Abstract

Infrastructure as Code (IaC) is the process of managing IT infrastructure via programmable configuration files (also called IaC scripts). IaC has progressively gained more adoption in the DevOps landscape. Even so, IaC is not a silver bullet; akin to other software artifacts, IaC scripts can suffer from bugs. Automated analysis tools to detect smells in IaC scripts exist, however, they focus on specific technologies such as Ansible, Chef, or Puppet. 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.

Since the IaC technologies ecosystem is very scattered, we address the generalization problem and we present GLITCH, a technology-agnostic framework that enables automated detection of IaC smells. GLITCH allows 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 security smells and nine design & implementation smells. We compare GLITCH with state-of-the-art smell detectors. For the security smells, the results show that GLITCH can reduce the effort of writing security smell analyses for multiple IaC technologies and it obtains higher precision and recall than the current state-of-the-art tools. For the design & implementation smells, we concluded that GLITCH has enough information in its intermediate representation to detect technology-agnostic smells detected by state-of-the-art tools.

Keywords

Infrastructure as Code, Intermediate Representation, Static Analysis, DevOps, Code Smells, Security Smells

Resumo

Infraestrutura como Código (IaC) é o processo de gerir infraestruturas informáticas através de ficheiros de configuração programáveis. IaC tem sido cada vez mais adotado no ambiente de DevOps. Apesar disso, como em outros artefactos de software, scripts de IaC podem ter bugs. Ferramentas de análise automática para detetar problemas em scripts de IaC existem, no entanto, estas focam-se em tecnologias específicas como Ansible, Chef, ou Puppet. Isto quer dizer que quando a deteção de um novo problema é implementada numa ferramenta, esta não está imediatamente disponível para as tecnologias suportadas por outras ferramentas — a única opção é duplicar o esforço.

Como o ecossistema de tecnologias IaC é muito disperso, considerámos importante resolver o problema da generalização e para isso criámos a framework GLITCH. A GLITCH permite a deteção automática de *code smells* em múltiplas linguagens ao transformar scripts de IaC numa representação intermédia, sobre a qual diferentes detetores de problemas podem ser definidos. A GLITCH suporta atualmente a deteção de nove problemas de segurança e nove problemas de design e implementação. Para os problemas de segurança, os resultados que obtivemos não só mostram que a GLITCH permite reduzir o esforço de escrever análises de segurança para múltiplas tecnologias de IaC, como também que obtém precisões e revocações mais elevadas que o estado da arte. Para os problemas de design e implementação, concluímos que a GLITCH tem informação suficiente na sua representação intermédia para detetar problemas agnósticos à tecnologia que são detetados por outras ferramentas do estado da arte.

Palavras Chave

Infraestrutura como Código, Representação Intermédia, Análise Estática, DevOps, Problemas em Código, Problemas de Segurança

Contents

List of Figures

List of Tables

List of Algorithms

1

Introduction

Contents

Infrastructure as Code (IaC) is a tactic that has been progressively gaining more adoption in the DevOps landscape given its many advantages, such as enabling scalable and reproducible environments or the traceability provided by version control software (e.g., git[1]). Guerriero et al. [1] define Infrastructure as Code as "*the DevOps tactic of managing and provisioning infrastructure through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools*". The use of IaC scripts is essential to efficiently maintain servers and development environments. Rahman et al. state an example which illustrates the importance of IaC in large companies [4]:

> *Fortune 500 companies, such as Intercontinental Exchange (ICE), use IaC scripts to maintain their development environments. For example, ICE, which runs millions of financial transactions daily, maintains 75% of its 20000 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.*

Although Infrastructure as Code has many benefits, by using it we are introducing a new class of possible issues — bugs in IaC scripts. For instance, due to bugs in their IaC scripts, GitHub experienced an outage of their DNS infrastructure [5] and Amazon Web Services lost around 150 million USD after issues with their S3 billing system [6]. As in software development, coding involves humans and humans are prone to make mistakes. Since one of the main problems in software engineering is the introduction of defects in code, a lot of effort has been made by the scientific community and companies in the industry to develop tools that try to avoid, identify or even automatically repair bugs. In recent years, efforts have been made in the same direction for IaC. For instance, Rahman et al. developed a tool, called SLIC, capable of identifying possible security issues in Puppet[2] scripts [4]. Sotiropoulos et al. developed a prototype that tries to identify missing dependencies between resources and services defined in a Puppet script [7]. Other examples include tools that find bugs and vulnerabilities in other IaC technologies, such as, Ansible[3], Chef[4], and CloudFormation[5] [8–10].

Existing tools that detect bugs in IaC code are very valuable. However, the great majority of work in this area has a limitation: even though some solutions may be easily extendable, they tend to be focused on a single technology. For instance, all the solutions mentioned above focus solely on a single IaC technology. In a recent study, which describes the state of the art related to DevOps research, Alnafessah et al. identified the need to surpass this limitation. In particular, they state that "*a research question is how to develop holistic and polyglot defect prediction and debugging environments for IaC*" [11].

[1] https://git-scm.com/
[2] https://puppet.com/
[3] https://www.ansible.com/
[4] https://www.chef.io/
[5] https://aws.amazon.com/cloudformation/

The importance of surpassing the technology limitation comes from the diversity of tools used in IaC. By interviewing IaC experts, Guerriero et al. concluded that "*the IaC technology ecosystem is currently very scattered, heterogeneous and not fully understood, with no single tool dominating the market*" [1]. In the same study, the authors identify the developers' need for IaC development tools, such as IDEs, static analysis tools, and security-related tools. Considering these statements, it would be valuable to have an artifact capable of assembling multiple analysis techniques and generalizing them for different IaC technologies.

1.1 Work Objectives

Our main goal is to create a framework in which we can develop technology-agnostic analyses for IaC scripts. Although more technologies can be implemented in the future, we focus our attention on technologies that deal with the configuration management of services, such as Ansible, Chef, and Puppet. Our choice is related to the heterogeneity of tools for this purpose, and because the three tools mentioned are the most adopted IaC tools (see Section 2.1.2). In our framework, researchers, developers, and sysadmins should be able to develop new analyses, represent new IaC concepts, or introduce new technologies. The IaC concepts will be the components of an intermediate representation to which the analyses are applied. Scripts from the supported IaC technologies will be translated to the intermediate representation, allowing the framework to apply the same analyses to IaC scripts from different technologies. We will answer the following research questions:

RQ1: Is it possible to create a model which abstracts different IaC technologies? Are the concepts expressed in the model relevant enough to apply different analyses from the literature to it?

RQ2: What limitations do we find when creating a model which abstracts IaC concepts between different technologies?

RQ3: Are we able to use our framework to obtain similar results to the analyses in the state-of-the-art?

Framework requirements. We want our framework to be easily extendable in three different aspects: analyses, the intermediate representation, and the IaC technologies supported. It must be designed with software engineering principles that fulfill the extensibility goals. The intermediate representation should allow the introduction of new IaC concepts without disrupting other work already present in the framework. It must also be able to capture similar concepts from different technologies while assuring

it is expressive enough to allow the execution of analyses from the literature. Information to allow the backtracking of smells to the original source code should be present in the framework's intermediate representation. Finally, we want to allow the users to fine-tune the analyses to their needs.

1.2 Contributions

The main contributions, which address our work objectives, can be summarized as follows:

1. A new intermediate representation that can be used to model IaC scripts and on which code smell analyses can be implemented. We focus on rule-based analyses with a high prevalence of pattern-detection techniques.

2. The framework GLITCH that is able to transform IaC scripts written in Ansible, Chef, or Puppet into the new intermediate representation and supports the detection of smells on this representation.

3. The implementation of nine security smells and nine design & implementation smells in GLITCH. For the security smells, we show that when compared with other state-of-the-art tools, GLITCH has higher precision and recall.

4. Oracle datasets of security smells for Ansible, Chef, and Puppet. We also created a dataset of Ansible scripts with over five million lines of code and a dataset of Chef scripts with over six million lines of code.

5. An empirical study that investigates how frequently security, design, and implementation smells occur in IaC. 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.

6. Replication packages publicly available as Docker containers and archived online with permanent links. These replication packages contain large datasets of IaC scripts, oracle datasets for security smells that were manually annotated, and ready-to-use tools to detect security and design & implementation smells. To the best of our knowledge, these are the first assets that enable truly reproducible research on this topic.

7. A Visual Studio Code[6] extension for GLITCH.

We hope that GLITCH becomes a standard to develop research about script analyses in IaC. GLITCH is free and open-source, so the scientific community is able to contribute to enriching our framework. As the intermediate representation gets richer, more complex and accurate analyses can be applied to it.

[6] https://code.visualstudio.com/

As more IaC technologies are supported, more value will be given to analyses developed in GLITCH. The source code for GLITCH can be found at: https://github.com/sr-lab/GLITCH.

Research Papers Parts of the work presented in this thesis were used in two papers co-authored by my supervisor Prof. João F. Ferreira: *GLITCH: Automated Polyglot Security Smell Detection in Infrastructure as Code* which was accepted for publication in ASE 2022[7] and awarded with an Artifact Evaluation Award of Reusability [12]; and *Polyglot Code Smell Detection for Infrastructure as Code with GLITCH* [13], which is a tool paper that will be submitted to ICSE 2023[8].

1.3 Novelty of our solution

To the best of our knowledge, we are the first to develop a polyglot code smell detection framework for IaC scripts. The publication of our paper [12] in ASE 2022 is proof of the novelty of our solution. There are some tools, such as Tortoise [14] and SLIC [4] which could be extended to other technologies. Rahman et al. extended the study done with SLIC by creating a new tool, called SLAC, which also identifies security issues in IaC scripts. The new tool is capable of analyzing Ansible and Chef scripts instead of Puppet [10]. However, the authors were not able to reuse previous components from SLIC and had to develop a new *Parser* and *Rule Engine*, which are the two components that build SLIC and SLAC. Even inside the new tool, Ansible and Chef are handled separately, which leads to inconsistencies [12]. As SLAC, some other tools try to handle more than one IaC technology. Checkov[9] is a policy-based static analysis tool which is capable of analyzing scripts from multiple IaC tools (e.g., Kubernetes[10] and Terraform[11]). Although Checkov considers more than one technology and the model it uses has some abstracted concepts (e.g., attributes and a graph representing connections between nodes), policies must be created for each tool. We consider it is important to mention that these tools are only focused on a single type of analysis, and more flexible models would be necessary to extend them to other analyses. We can think of two main reasons why the technology barrier has not been addressed yet. First, IaC is a recent trending topic. By using data from Google Trends, Rahman et al. identified that "*interest in IaC has increased steadily after 2015*" [15] (Figure 1.1 show the updated graphic until 2022 from which Rahman et al. achieved their conclusions). Secondly, at first glance, the ratio between benefits and difficulties to solve the technology limitation may not look appealing to the scientific community. However, we consider the benefits to be extremely valuable to the industry, since the IaC technology ecosystem is very scattered, and research could become even more relevant if generalized for more

[7] https://conf.researchr.org/home/ase-2022
[8] https://conf.researchr.org/home/icse-2023
[9] https://github.com/bridgecrewio/checkov
[10] https://kubernetes.io/
[11] https://www.terraform.io/

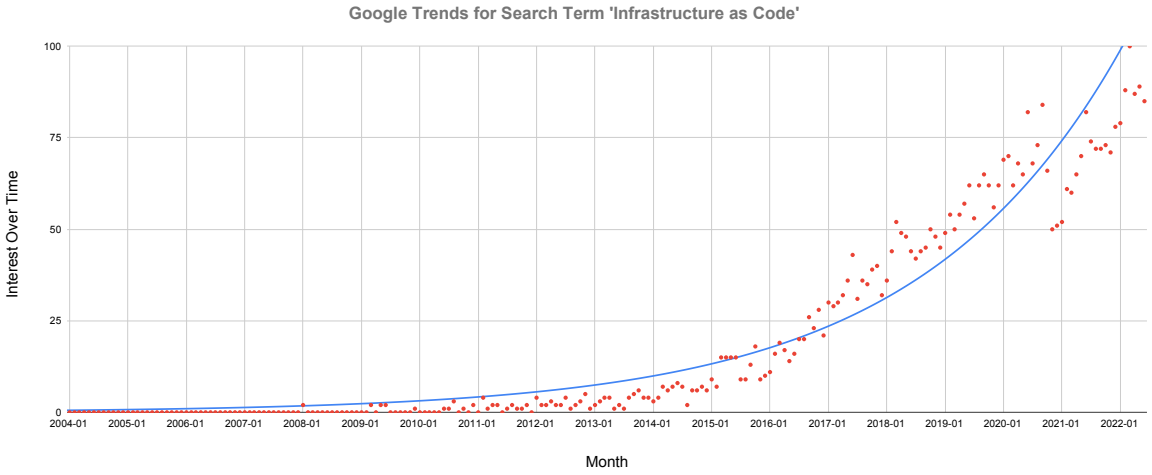

Figure 1.1: Interest in IaC as a search topic from 2004 until 2022 based on Google Trends data.

than one tool.

1.4 Document outline

Throughout this document, we explore the IaC ecosystem, propose a solution to the problem of polyglot code smell detection in IaC scripts, and evaluate our solution. We begin in Chapter 2 by introducing the necessary IaC background, describing the current state-of-the-art in code smell detection for IaC scripts, and exploring usages of intermediate representations in Software Engineering. In Chapter 3, we describe in-depth our framework GLITCH and its intermediate representation. Chapter 4 and Chapter 5 provide two empirical studies that evaluate our approach with GLITCH and investigate how frequently code smells occur in IaC. Finally, Chapter 6 concludes the document by summarizing our achievements, answering our research questions, and discussing future directions.

2

Background & Related Work

Contents

```yaml
---
- name: Update web servers
  hosts: webservers
  remote_user: root

  tasks:
  - name: Ensure apache is at the latest version
    ansible.builtin.apt:
      name: apache2
      state: latest
  - name: Create default page file
    ansible.builtin.copy:
      dest: /var/www/html/index.html
      content: "Welcome, {{ ansible_default_ipv4.address }}!"
  - name: Start apache service
    ansible.builtin.service:
      name: apache2
      enabled: yes
      state: started
```

Figure 2.1: Ansible Playbook which installs Apache, creates an HTML file to be provided, and enables the Apache service.

In this chapter, we present the motivation that led us to try to solve the problem of polyglot code smell detection in IaC scripts and the background and related work necessary to understand the solution we present. We start in Section 2.1 by giving context about what is Infrastructure as Code, its benefits, an overview of the technologies ecosystem, and a description of the most adopted configuration management of services technologies — Ansible, Chef, and Puppet. Section 2.3 highlights the importance of support tools in software engineering and IaC, and describes the state-of-the-art tools to analyze IaC scripts. Finally, in Section 2.4, we describe the importance of intermediate representations in a variety of Computer Science areas, and how these representations are applied.

2.1 Infrastructure as Code

Infrastructure as Code (IaC) is the process of managing IT infrastructure using configuration files that can be written as regular code. In the modern world, where applications require high scalability and availability, IaC proves to be necessary by decreasing maintenance costs, reducing the risks of manual misconfiguration or inconsistencies by human error, and allowing faster deployments and problem-solving. The ability to version control the configuration of a system is one good example of the advantages of this approach since it enables all the beneficial practices already explored in software engineering. For instance, source code history allows implicit communication in a team and simplifies the process of pinpointing when and why a configuration error was introduced. One simple example to understand the power of IaC is the following: imagine that system administrators (sysadmins) have ten machines with SSH access and they want them to work as web servers. To achieve their goal, the sysadmins will need to install the Apache HTTP server[1], create an HTML file to show on the web page, and start the Apache service. Usually, they would connect to each machine, install the Apache

[1] https://httpd.apache.org/

package, create the file and start the service manually. However, this process may take some time, it is repetitive and it is prone to human error (e.g., inconsistencies between machines). Instead, it is possible to automate this procedure using an IaC tool, such as Ansible. Ansible allows to configure machines by executing a list of tasks in each one of them. The configuration is done from a central machine that connects by SSH to each host and executes the commands to complete the defined tasks. The sysadmins could define the program in Figure 2.1 and execute it with Ansible to achieve their goal. Section 2.1.2.A describes in more depth how Ansible works and its features.

2.1.1 Case studies

Case studies that highlight the benefits of IaC to companies have been addressed in the literature. Rahman et al. identified four studies that describe how IaC benefited information technology organizations [10]:

> *For example, the use of IaC scripts helped the National Aeronautics and Space Administration to reduce its multi-day patching process to 45 minutes. Using IaC scripts, application deployment time for the Borsa Istanbul, Turkey, stock exchange reduced from ∼10 days to an hour. With IaC scripts, Ambit Energy increased their deployment frequency by a factor of 1,200. The Enterprise Strategy Group surveyed practitioners and reported the use of IaC scripts to help IT organizations gain 210% in time savings and 97% in cost savings on average.*

Many other relevant examples can be found. Edgenuity is an organization that provides online learning and teacher resources. The company decided to change to the cloud at the beginning of 2020 and choose Chef to leverage this process. By using IaC scripts, what once would take the company's DevOps team 4 to 5 days to deploy, would now take less than 3 hours. The usage of Chef also allowed Edgenuity to scale from supporting 500,000 connections to 5 million when the COVID-19 pandemic hit while minimizing disruption to the students [16]. Germany's Federal Office for Agriculture and Food used Ansible to decrease the time used in IT management and configuration by more than 50% [17]. Jewelers Mutual Insurance Company collaborated with Zivra to integrate Puppet into their workflow. They were able to reduce the time to create an environment of about 30 servers from 4 to 6 weeks to under 1 day and increased consistency across their heterogeneous environments [18].

From the case studies above, we consider there are two main takeaways. First, we can conclude that IaC technologies are widely used in the industry and with important benefits to companies adopting them. Secondly, we can confirm the heterogeneity of the IaC ecosystem, since we have organizations choosing different technologies for the same purpose. Although Ansible, Chef, and Puppet focus on the same class of problems, they have differences between them. These differences will be further

explained in Section 2.1.2. Organizations, after analyzing the different options, end up choosing the technology which they consider more fit to their goals. Edgenuity chose Chef "*because it was the only automated application packaging and delivery solution that was technology agnostic*" [16]. Germany's Federal Office for Agriculture and Food states as reasons to use Ansible: "*firstly because of the quality of Red Hat's support, and secondly because the Red Hat Ansible Tower solution works best with our existing Red Hat environment*" [17]. Finally, Jewelers Mutual Insurance Company decided to work with Puppet "*because whether you're a developer or sysadmin, Puppet is intuitive and the learning curve is not steep*" [18]. There is no standard technology to solve every IaC problem, so, it is important to generalize support tools to multiple technologies.

2.1.2 IaC Technologies Overview

Table 2.1: Categorization of IaC tools based on Guerriero et al.'s study [1].

Category	IaC Tools
Container Orchestration	Kubernetes, Nomad[a], Docker Swarm[b], Apache Mesos[c]
VM Management	Vagrant[d]
Configuration Management of Services	Ansible, Chef, Puppet, Saltstack[e]
Service Orchestration	Terraform, Pulumi[f], CloudFormation, Apache Brooklyn[g]
Image/Container Builder	Packer[h], Docker[i]

[a] https://www.nomadproject.io/ [b] https://docs.docker.com/engine/swarm/
[c] https://mesos.apache.org/ [d] https://www.vagrantup.com/ [e] https://saltproject.io/
[f] https://www.pulumi.com/ [g] https://brooklyn.apache.org/ [h] https://www.packer.io/
[i] https://www.docker.com/

The IaC ecosystem is composed of a high variety of technologies. Guerriero et al., in their study about the adoption, support, and challenges of IaC, state that the ecosystem "*is currently characterized by a plethora of different and often overlapping (in terms of their goals) tools and languages*" [1]. According to their study, no technology has more than 60% adoption and there are 10 technologies with at least 20% adoption, which shows the lack of homogeneity when choosing IaC technologies. The adoption of technologies with similar purposes can be understood by the trade-offs between them. These trade-offs can make a technology more appealing to some solutions than another with alike functionality. Ansible and Chef are examples of two technologies with similar purposes; however, one main difference that we may find is the configuration language. Chef uses the Ruby DSL, which is "*aimed more at advanced developers rather than those with little to no programming experience*". Ansible uses YAML, which "*is relatively easy to learn, regardless of your prior experience*". Given its simplicity, YAML is less powerful

and, for that reason, it is not able to handle tasks as complex as Ruby DSL [19]. The trade-off between the learning curve and the complexity of tasks a technology can achieve is important when deciding which one will be used for solving a certain problem. In the IaC ecosystem, there are also technologies that fulfill different purposes. In the same study, Guerriero et al. mention that "*developers tend to select a stack of tools, each having a different purpose, whose combination enables full IaC development*". For instance, Terraform is usually used to provision infrastructure components, and Ansible deals with the configuration management of services. We may use Terraform to create virtual machines and provide their configuration with Ansible (e.g., install an application in each one).

Considering the diversity of IaC technologies with different purposes, it is important to categorize them. Table 2.1 summarizes the categorization done by Guerriero et al. in their study [1]. In addition to the examples provided by the authors, the table contains other technologies which we consider fitting to these categories. The names of the technologies we added are underlined in the table.

In our work, we focus our attention on technologies that configure and manage services. Two reasons led us to select this class of technologies. First, the existence of scientific work to create analyses for scripts in these technologies allow us to test our framework by replicating these analyses and comparing the results to the original studies (as shown in Chapters 4 and 5) [2–4, 7, 10, 14]. Secondly, technologies that deal with the configuration management of services appear to have an ecosystem with more heterogeneity and with more balanced adoption between them. In the study conducted by Guerriero et al., four technologies in this category were mentioned to be adopted by the industry (Ansible, Chef, Puppet, and Saltstack) [1]. Out of these four, three of them had at least more than 29% adoption by the IaC experts interviewed (Puppet – 29.5% / Chef – 36.3% / Ansible – 52.2%). Those 3 technologies are the ones we further explore.

2.1.2.A Ansible

Ansible is an open-source infrastructure automation tool that uses a declarative language, called YAML, to automate IT tasks. As described in Ansible's documentation, the tool is able to "*configure systems, deploy software, and orchestrate more advanced IT tasks such as continuous deployments or zero downtime rolling updates*" [20].

Technology architecture. Ansible, unlike other configuration management technologies, works with a push configuration setup. IaC technologies usually have two types of machines: a server or set of servers where the configuration coded by sysadmins is maintained, and the nodes to be configured. In a push configuration setup, the sysadmin commands the server to provide the configuration to a set of nodes, instead of the nodes asking the server for their configuration. Without direct user interaction with the nodes and with no initiative coming from them, push configuration technologies do not necessarily

```
1  [webservers]
2  www1.example.com http_port=80
3  www2.example.com http_port=303
4
5  [webservers:vars]
6  max_requests_per_child=808
7
8  [dbservers]
9  db0.example.com
10 db1.example.com
```

Figure 2.2: Example of an Ansible Inventory with 2 tags (webservers and dbservers) and 4 hosts (ww1, ww2, db0, and db1).

need an agent installed in each slave machine. That is the case with Ansible. A server (machine with access to all nodes), which maintains the intended configuration, accesses each machine, usually through SSH, and pushes small programs, called Ansible modules, which execute the necessary actions to reach the desired state in the node. The list of nodes to be configured is maintained in an inventory. An inventory is usually an INI file where nodes are listed and can be grouped by being assigned a certain tag. An example of an inventory is shown in Figure 2.2.

Syntax and code structure. To define the desired state, sysadmins need to write code to achieve it. Ansible code is organized into four different hierarchical levels: roles, playbooks, plays and tasks. Ansible allows to group functionality into roles. Roles are folders that contain content such as tasks, variables, static files, modules, or templates. The functionality of a role can then be reused in different playbooks by including the role in a play or task. A playbook is a file written in YAML which contains instructions to configure our nodes. A playbook is composed of a set of plays. Each play links a set of hosts to a set of tasks. Hosts can be identified by a tag to which they belong (selecting a group of nodes) or by their hostname. Tasks are the atomic building block of an Ansible configuration. A task has two parts: an optional field with the task name, which is useful because the name will be printed when the task is executed; the action the task itself will run, which is defined by selecting an Ansible module and providing the arguments that achieve the state we want. Ansible also allows to group tasks in a block. Blocks can be seen as special tasks where the single argument of their module is a list of other tasks to be executed. For instance, blocks can be used to skip a group of tasks if a certain condition is not met.

Variables/Attributes. Ansible uses variables to manage differences between systems. Variables can be assigned a different value for each node, using the same name. These values can then be used in playbooks to change the behavior of tasks according to the node they are being applied to. We can define variables in a variety of ways, including creating them in playbooks or the inventory. Figure 2.2 shows the definition of a variable in the inventory, called *http_port*, with a different value for each web server. In the same figure, we can also see the definition of a variable, called *max_requests_per_child*, which, instead of node specific, is defined for the group of nodes (*webservers*). Ansible has special

variables related to information about the remote systems (nodes). These variables are called facts and are automatically gathered by Ansible. Facts can give information such as the IP address or the operating system of a node.

Execution order. By default, Ansible will execute the tasks in the order they are defined in the playbook. However, it is possible to define tasks that only run when notified. These tasks are called handlers and are triggered by a change in another task. For instance, we could define a task that changes a configuration file for a package, notifying a handler to restart the related service only if the content in the file changes.

Example and final remarks. Figure 2.1 is an example of an Ansible Playbook with a single play containing three different tasks. In this case, the tasks use the *apt*, *copy*, and *service* modules, which are built-in modules from Ansible. The playbook uses a fact to add the IP address of the node to the content of the HTML file. All the information presented is based on Ansible's documentation [20].

2.1.2.B Chef

Chef has more than one product that helps IT automation: Chef Infra, Chef Habitat, Chef InSpec, and Chef Automate. We focus our attention on Chef Infra since this technology is the one with direct correspondence to Ansible and Puppet. According to Chef's documentation, "*Chef Infra automates how infrastructure is configured, deployed, and managed across your network, no matter its size.*" [21].

Technology architecture. Such as Ansible, Chef has servers, which store configurations, and nodes to be configured, in this case, called clients. A particularity of Chef is the existence of a workstation, which corresponds to the everyday computer the sysadmin works with. Chef Workstation is a set of tools, installed in the sysadmin's computer, that allow to test Chef code and interact with other components of Chef Infra. Namely, *knife* is a command line tool capable of interacting with the server, which allows operations, such as uploading configurations from a workstation. In contrast to Ansible, Chef Infra works with a pull configuration setup, requiring a client to be installed in each node. Periodically, each client contacts the server to retrieve the latest configuration for that particular machine, and, if it differs from the current one, the new instructions are applied to the node. Pull configuration setups have the advantage of automatically converging to the desired state when new nodes are added. If a workstation has SSH access to a node, we can add that node using the *knife* tool. This operation consists of installing the Chef client in the new machine and adding the node to the inventory maintained in the server.

```
1  package 'apache2' do
2    action :install
3  end
4
5  file '/var/www/html/index.html' do
6    content "Welcome! I am " + node['ipaddress']
7    action :create
8  end
9
10 service 'apache2' do
11   action [ :enable, :start ]
12 end
```

Figure 2.3: Chef Recipe which installs Apache, creates an HTML file to be provided, and enables the Apache service.

Syntax and code structure. As with any IaC technology, Chef also needs us to code the infrastructure configuration. The configurations are organized in fundamental units called cookbooks. Cookbooks are folders created with the *chef* command-line tool, which contain all the necessary content to define a certain scenario. We consider of particular interest recipes and attributes. Recipes are written using a programming language called Ruby and they define the steps necessary to configure part of a system. Recipes can have helper code, but their focus is on the definition of resources. We can see resources as the equivalent of a task in Ansible. A resource is identified by its name and its type (package, service, file, etc.). The action and properties associated with each resource define the desired state for the selected configuration item.

Variables/Attributes. Attributes are another component of a cookbook. An attribute is a key-value pair linked to a node. They can be defined in attribute files or recipes. An example of an attribute that is linked by default to every node is the IP address. Attributes are relevant because we can use them to modify the definition of a resource accordingly to the node being configured.

Execution order. For the execution order in Chef, we should consider the order of resources and the order of recipes. Resources in each recipe are executed in the order they are defined in the file. However, similarly to Ansible, Chef allows a resource to notify another resource when its state changes. For that, we must identify the resource to be notified of, the action to perform, and when to perform it. A resource can also subscribe to another resource, taking action if the state of the resource being listened to changes. Recipes run in the order defined in a run-list. A run-list is an ordered list of recipes defined by the user for each node.

Example and final remarks. Figure 2.3 is an example of a Chef recipe. In the example, we have three resources with three different types: package, file, and service. We are configuring a web server that will provide the content in the *index.html* file. Each node configured by this recipe will show a different page since we are using the attribute *ipaddress* to set the content of the file. The tools provided in Chef, the language used to define the policies, and the ability to create helper code in Ruby make

Chef's environment more complex than Ansible's environment. All the information presented is based on Chef's documentation [21].

2.1.2.C Puppet

Puppet has a lot of similarities with Chef since the language it uses to program the configurations is based on Ruby and the technology architecture is similar. The Puppet's documentation describes Puppet as "*a tool that helps you manage and automate the configuration of servers*" [22].

Technology architecture. Puppet has the same participants as the previous two technologies. There are servers, usually called masters in Puppet context, and there are the nodes, called agents. Masters maintain the code and data to configure a node or a group of nodes. The data is managed by a component called Hiera, which will be better explained further ahead. As in Chef, Puppet uses a pull configuration setup. Each node must have the Puppet Agent installed which communicates with the master, at regular intervals, to retrieve the latest configuration for that specific machine. Master and nodes communicate by HTTPS using SSL certificates managed by the master. Nodes are added to the deployment by submitting a certificate signing request (CSR) to the master, which has to be accepted by the admins. The agents use a library, called Facter, to collect information about themselves, information to which we call facts. When a node asks for its configuration, it also sends its facts, which will be used, together with the code and data the server has, to compile a catalog. A catalog describes the desired state of a specific agent node. The agent receives its correspondent catalog and enforces the state described there. Finally, the node reports back to the server with information such as the events that occurred, metrics about the run, or the status of the resources in the node. Data such as reports, facts, catalogs, and node information is stored in a component called PuppetDB. "*Storing data in PuppetDB allows Puppet to work faster and provides an API for other applications to access Puppet's collected data*" [22]. For instance, the data can be used by an analysis tool to assess vulnerabilities or other problems in the infrastructure.

Syntax and code structure. To write code in Puppet, we use Puppet's Domain Specific Language (DSL), which is based on Ruby. By using a DSL instead of Ruby itself, Puppet provides a more simplistic language and forces the users to a resource-driven approach. Since the structure of the programs is well-defined and much more limited, analysis tools tend to be easier to develop. In Puppet, we organize our code into modules. Modules, which can be seen as the same as cookbooks in Chef, are directories containing all the necessary content to manage a specific functionality in our infrastructure. Examples of content are configuration files, static files, tests, parameter defaults, or examples of how to use the module's functionality. Configuration files are called manifests. Manifests can contain conditional logic,

but their main goal is the definition of resources — the atomic building block of Puppet. The syntax for resources and their concept is similar to Chef. A resource is declared by specifying a type, a title, and a set of attributes, which describe the desired state for that aspect of the system. We can group resources into classes. Classes are named blocks that can configure larger chunks of functionality. Besides the benefit of being able to reuse groups of resources usually used together without having to write them every time, classes can receive parameters to change the behavior of those resources. It is also possible to create a hierarchy of classes, which is a useful concept from object-oriented programming.

Variables/Attributes. Manifests can contain references to values that are specific to a node or group of nodes. We will call those references node attributes. When compiling a catalog, node attributes are replaced by the value corresponding to the node asking for its configuration. There are two types of node attributes: facts and Hiera key-value pairs. Facts, as explained above, are collected by the Facter and contain information such as the IP address or the operating system. Hiera allows the creation of hierarchies of attribute files. For instance, we could have attribute files specific to each node and attribute files for nodes with a certain operating system. Hiera could first search for an attribute in the node-specific file and then, if the attribute does not have a value there, in the correspondent operating system file. These files, written in YAML, contain the key-value pairs. Using Hiera allows us to detach data from code and gives us an easy way to change the behavior of a manifest, accordingly to the node they are applied to.

Execution order. Puppet applies resources in the order they are defined in their manifest, but only if the resource has no implicit relationship with another resource. We are able to specify relationships between resources and the technology is responsible for defining the execution order that fulfills those requirements.

Example and final remarks. Figure 2.4 is an example of a Puppet manifest that configures a web server. The manifest has a class, called *webserver*, which contains the three resources necessary to configure it (install the Apache package, create the HTML file and start the Apache service). The class has a parameter, called *content*, which defines the text that will be presented on the page. On line 17, we declare the class and set the content, enabling the resources in the class to be executed. On line 8, we use a fact to add the IP address of the node to the page. All the information presented is based on Puppet's documentation [22].

```
class webserver ($content = "") {
  package { 'apache2':
    ensure => present,
  }

  file { '/var/www/html/index.html':
    ensure  => file,
    content => "${content} ${::facts['ipaddress']}",
  }

  service { 'apache2':
    ensure => running,
    enable => true,
  }
}

class{'webserver':
  content => "Hello from"
}
```

Figure 2.4: Puppet Manifest which installs Apache, creates an HTML file to be provided, and enables the Apache service.

2.1.2.D Differences between tools.

IaC Tool	Ansible	Chef	Puppet
Configuration Setup	Push	Pull	Pull
Atomic Unit	Task	Resource	Resource
Add. agent	No	Yes	Yes
Syntax	YAML	Ruby	Puppet DSL
Execution Order	Procedural	Procedural	Declarative
Code Structure	Roles - Playbooks - - Plays - - - Tasks	Cookbooks - Recipes - - Resources	Modules - Manifests - - Classes - - - Resources

Table 2.2: Summary of differences between Ansible, Chef, and Puppet.

Table 2.2 summarizes the differences between the 3 tools we described. It is important to mention the difference we consider between a procedural execution order and a declarative one. Procedural means that the code order is always respected and the developer is responsible for ordering the operations accordingly to what he expects. Declarative means that the developer only needs to define requirements for the execution order and the program is responsible to find the right execution to fulfill those requirements.

2.2 Code Smells in IaC

Like other software artifacts, IaC scripts can suffer from bugs. These bugs may be the result of introducing one or more code smells into the code base. Fowler defines a code smell as “a surface indication that usually corresponds to a deeper problem in the system” [23]. Schwarz et al. define code smells

as “flaws in code which may lead to problems” [3]. In this section, we describe two different classes of code smells in IaC: security smells and design & implementation smells, which are the classes that we implemented in GLITCH and that we studied in Chapters 4 and 5, respectively.

2.2.1 Security Smells

Security smells are coding patterns that can result in security weaknesses. Even when a security smell does not lead to a security breach, it deserves attention and inspection. We focus our attention on the security smells studied by Rahman et al. [4, 10] (we adapted the descriptions by the authors [10]):

- **Admin by default (CWE-250 [24])**: This smell is the recurring pattern of specifying default users. The smell can violate the “principle of least privilege” property, which recommends practitioners to design and implement a system in a manner so that by default the least amount of access necessary is provided to any entity [25].
- **Empty password (CWE-258 [24])**: The smell is the recurring pattern of using a string of length zero for a password.
- **Hard-coded secret (CWE-259, CWE-798 [24])**: This smell is the recurring pattern of revealing sensitive information, such as user names and passwords in IaC scripts.
- **Unrestricted IP Address (CWE-284 [24])**: 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 [26].
- **Suspicious comment (CWE-546 [24])**: 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 [24])**: : 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 [27].
- **No integrity check (CWE-353 [24])**: 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 [24])**: 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 [24])**: This smell is the recurring pattern of not handling all input combinations when implementing a case conditional logic.

2.2.2 Design & Implementation Smells

Sharma et al. define design smells in IaC scripts as "quality issues in the module design or structure of a configuration project". The same authors define implementation smells as "quality issues such as naming convention, style, formatting, and indentation in configuration code" [2]. Sharma et al. used the knowledge of traditional software engineering and best practices associated with code quality management to leverage the creation of a configuration smells catalog for Puppet scripts [3]. The catalog is composed of thirteen implementation (e.g., improper alignment of arrows or long statements) and eleven design configuration smells (e.g., insufficient modularization or duplicate block). Schwarz et al. introduced new code smells for IaC based on the field of software engineering, such as *Long Resource* and *Too many Attributes*, and categorized design & implementation smells as technology agnostic, technology dependent, or technology specific [3]. Since our work is to generalize the detection of smells to multiple IaC technologies, we focus on the technology-agnostic smells identified by Schwarz et al. [3]. We define these nine design & implementation smells as follows (our definitions are based on Sharma et al. and Schwarz et al.'s [2, 3] definitions):

- **Avoid Comments**: Comments other than licensing information on the first lines should be avoided. Based on Fowler, comments are often used as a deodorant to bad code [28].
- **Duplicate Block**: Blocks of statements occurring more than once and above a certain size threshold may indicate a missing abstraction. According to Fowler, avoiding repetition leads to good design [29].
- **Improper Alignment**: The source code is not aligned according to the technology's style guide or tabulation characters are used. Albayrak and Davenport conducted a study with Java code which revealed that the presence of indentation defects decreases readability and significantly reduces the detection of functional defects in the code [30]. Tabulation characters should be avoided to minimize differences in the output between different coding environments.
- **Long Resource**: Atomic units in IaC should not surpass a threshold of lines of code. Since the size of atomic units is limited by their number of attributes, only atomic unit types that may contain source code in the values of their attributes (e.g., *bash* and *exec*) are considered. The Long Resource smell maps to the Long Method smell by Fowler [28] but it is adapted to the IaC domain.
- **Long Statement**: Code statements that are too long and usually do not fit on the screen.
- **Misplaced attribute**: The order of the attributes inside an atomic unit should follow the technology's style guide. Both Chef and Puppet mention the order of attributes in their style guides [31,32].
- **Multifaceted abstraction**: Each abstraction should follow the single responsibility principle [33]. In IaC, each abstraction should only specify the properties of a single piece of software.

- **Too many variables** (Schwarz et al. call it Too many attributes): A high density of variables should be avoided to simplify the code and maximize its maintainability. This smell is derived from a combination of the smells Long Parameter List and Speculative Generality by Fowler [28].
- **Unguarded Variable**: Variables should be enclosed in braces when being interpolated in a string.

2.3 Development Tools and Code Analysis

Tools that help the development of code and management of infrastructures are getting increasingly more valuable. As companies grow, there are more developers involved and code is changed faster. Code bases tend to get bigger, and so, more difficult to maintain. For instance, in January 2015, Google code base had "*approximately two billion lines of code in nine million unique source files*", and "*on a typical work day, (...) 16,000 changes to the codebase*" were made [34]. Managing this kind of scale requires a lot of human resources to code and make reviews, which leads to high costs for the companies. With this rate of change, even with all the possible caution from developers, it is impossible to not introduce bugs into the code base. Some of these bugs may cause major faults. For instance, on June 2021, Fastly, a cloud-network provider with clients such as the New York Times and The Guardian, experienced a global outage due to an undiscovered software bug triggered by a valid customer configuration change [35]. On October 2021, Facebook experienced an outage that caused products such as Messenger, WhatsApp, and Instagram to become globally unavailable. The outage was caused by a command which was supposed to check the available capacity of Facebook's global backbone network but instead took down all its connections. Santosh Janardhan from Facebook stated that: "*our systems are designed to audit commands like these to prevent mistakes like this, but a bug in that audit tool prevented it from properly stopping the command*" [36]. The statement not only shows the need for tools that support these operations but also the confidence developers and sysadmins have in these tools to help them. Examples directly related to bugs in IaC scripts are the outage of GitHub's DNS infrastructure in 2014 [5], and issues in Amazon Web Services' S3 billing system which made the company lose around 150 million USD in 2017 [6].

A lot of effort has been made by the scientific community and the industry to develop tools that try to support developers in their actions. For instance, IDEs (Integrated Development Environments), such as IntelliJ IDEA[2], help to avoid bugs by giving code suggestions, analyzing and highlighting possible mistakes, supplying powerful refactoring tools, and other supporting features. These features allow developers to focus on bugs related to functionality instead of worrying so much about simpler bugs, such as typos or a variable name that the developer missed changing in a refactor. Plugins for IDEs allow the addition of complex analyses to the coder's workflow. EcoAndroid is an example of a plugin for IntelliJ

[2] https://www.jetbrains.com/idea/

IDEA and Android Studio that suggests automated refactorings for reducing the energy consumption of Java Android applications [37, 38]. Bots that execute code analyses in a Continuous Integration pipeline are another popular approach to reduce the introduction of bugs into the code base. For example, Repairnator[3] is a bot that assembles multiple program repair approaches, coming from academic research, to create automatic and human-competitive patches to Java programs [39].

The increasing relevance of IaC and its parallelism with "regular" code led researchers to investigate how to develop analyses that identify or repair bugs in IaC scripts. We divide research in this area into two groups: identifying faulty IaC scripts, and detecting and repairing IaC issues. Inside these groups, there are multiple approaches explored.

2.3.1 Identifying faulty IaC scripts

The objective of identifying faulty scripts is to give the users tips about where to spend their time improving code that might lead to issues in the future. Tools with this principle in mind should return the probability of a script being faulty. Techniques to identify faulty IaC scripts tend to rely on machine learning methods.

Rahman and Williams extracted characteristics of defective IaC scripts by using qualitative analysis [40]. The qualitative analysis was applied to text features that appeared in faulty scripts. The text features were obtained by applying text mining techniques to convert Puppet scripts into tokens (words in the script). The characteristics associated with defective scripts that were found are file-system operations, infrastructure provisioning, and managing user accounts. The authors applied two techniques to the tokens that generate new features capable of being used as input to predictive models: the BOW technique, and the TF-IDF technique. The Random Forest (RF) technique was used to build the models. The training datasets were created by crawling open-source projects, generating the features, and manually labeling defective scripts. 10-fold cross-validation was used to evaluate the models. These models were able to obtain median F-Measure values between 0.70 and 0.74 depending on the dataset and text feature extraction technique [40].

Rahman and Williams, in another work, followed a similar approach but with a focus on source code properties, such as lines of code, number of attributes, or URL occurrences [41]. They found that using these properties as input to predictive models outperformed the BOW technique from the previous work. The authors tried to use other techniques to build the models, such as Logistic Regression (LR) and Naive Bayes (NB). Depending on the dataset and evaluation metric, different techniques were better, but the Random Forest technique was outperformed in the great majority of the experiments. Rahman and Williams found that the properties with the strongest correlation to a script being defective are the number of lines of code and hard-coded strings [41].

[3] https://github.com/eclipse/repairnator

Palma et al. complemented the two previous works by addressing the problem that datasets with IaC scripts tend to have a number of defective samples much smaller than non-defective ones [42]. To minimize the consequences of unbalanced datasets, the authors used different techniques to build the models. These techniques use novelty detection, which means that the models are trained using only non-defective samples. Defective samples are identified as anomalies that differ from the data provided to the model (novelties). For the dataset used, the RF technique had a precision of 0.08 while these techniques had a precision of up to 0.86, showing much better results. The authors focused on Ansible instead of Puppet, but stated that they planned to "*investigate how novelty detection generalizes on software defect datasets from different configuration management languages*" [42]. The dataset used was built by the *RADON framework* created by Palma et al. [9].

Palma et al. developed a framework called *RADON Framework for IaC Defect Prediction* which is "*a fully integrated Machine-Learning-based framework that allows for repository crawling, metrics collection, model building, and evaluation*" [9]. The study conducted with the RADON framework concluded that, when considering individual projects/repositories as the training base for a model, the Random Forest technique outperformed other techniques, such as Support Vector Machine and Logistic Regression, regardless of the metrics used or the project's characteristics. The authors also concluded that IaC-oriented metrics (e.g., lines of code or number of attributes) achieve better results than process metrics (e.g., the total number of lines added or the number of developers that changed a file) or delta metrics (amount of change in a file). The *RADON Framework* is capable of categorizing IaC scripts as possibly faulty or not, without manual intervention, by analyzing commits and issues in the project's repository. Palma et al. show their interest to extend their approach to other IaC technologies and languages [9].

2.3.2 Detecting and repairing IaC issues

Approaches that try to detect and repair IaC issues try to minimize faults caused by human error. These tools give more confidence to the artifacts being deployed, increase awareness of the users about what they should not do, and analyze a large number of scripts when the scale does not allow humans to do it. These approaches are more diverse than the ones to identify faulty IaC scripts.

Rahman et al. applied qualitative analysis on Puppet scripts to identify seven security smells, such as *Hard-coded secret* and *Use of HTTP without TLS* [4]. The qualitative analysis allowed them to detect patterns for each security smell. These patterns were used to create the rules for SLIC, a tool to detect security smells in Puppet scripts. The authors divided SLIC into two components: a parser and a rule engine. The parser goes through an IaC script and returns a set of tokens to which the rule engine applies its rules based on pattern matching. The authors submitted 1000 randomly selected occurrences of security smells identified by SLIC and out of the 104 responses they got, 64.4% of the

practitioners agreed with the smells identified [4]. Rahman et al. created a new tool, called SLAC, to replicate the previous work to Ansible and Chef [10]. The authors explored two new security smells: Missing default in case statement and No integrity check [10].

Sharma et al. used *puppet-lint*[4] to detect implementation smells and developed a tool, called Puppeteer, to detect design smells [2]. Using these tools, the authors analyzed 4621 Puppet repositories with 8.9 million lines of code to answer questions such as the distribution of maintainability smells in configuration code. They found *Improper Alignment* to be the most detected implementation smell and *Insufficient Modularization* the most detected design smell [2]. Schwarz et al. extended the research done by Sharma et al. by applying the detection of IaC smells to another technology, namely Chef. The authors developed a tool to detect code smells in Chef scripts and conducted a study by executing the tool on around 3200 Chef cookbooks. The results allowed to conclude that "*these smells are adequate to be used to investigate the quality of IaC in general*" [3].

As in the work to identify security smells [4], Rahman et al. applied qualitative analysis, but instead of code snippets, the authors used defect-related commits to identify defect categories for IaC scripts [43]. They identified eight categories, with *configuration data-related defects* being the most frequent category, and *idempotency* being the least frequent one. The authors used the information from the qualitative analysis to create empirical rules that automatically identify defect categories in enhanced commit messages (ECMs). ECMs are the combination of commit messages with the bug report descriptions that are linked to the commits (e.g., issues). These rules were used to build a tool, called ACID, with an average precision and recall across all categories of 0.84 and 0.96, respectively [43].

Chen et al. took a different approach to identify error patterns [44]. In their paper, the authors extracted error-fix-induced code changes from historical commits, and then used a clustering algorithm, namely HDBSCAN, to group similar code changes. By manually analyzing the clusters, they identified error patterns in the scripts, which were grouped into categories, such as operating system-related errors and file-related errors. The authors used the error patterns to propose a set of rules which were implemented in a tool called *Puppet Analyzer* [44].

Borovits et al. created an approach, called DeepIaC, to identify inconsistencies between task names and task bodies of Ansible scripts by leveraging word embedding and deep learning. The models, created using Convolutional Neural Networks (CNNs), classify a task as consistent or inconsistent and were able to obtain an accuracy between 0.785 and 0.915 [45].

A problem that may emerge in IaC scripts is missing dependencies between resources. Shambaugh et al. addressed this problem by creating a tool, called Rehearsal, that checks if Puppet manifests are deterministic and idempotent [46]. The authors translate IaC scripts into an intermediate language that models resources as the description of their file-system transformations. Logical formulas are created

[4] http://puppet-lint.com/

based on the semantics of each language's primitive and of the manifest's resource graph (representation of the valid sequences to apply the resources). By providing these formulas to an SMT solver, the solver decides if the program is deterministic or not. However, since the analysis is static, some transformations can not be identified, namely in manifests where the *exec* resource is used to run embedded shell scripts [46]. Sotiropoulos et al. took a different approach to find missing dependencies, which allows to overcome the issue with embedded shell scripts [7]. Their method collects the system calls invoked by a Puppet program and uses them to model the file-system transformations in an intermediate representation, called *FStrace*. The Puppet resources are linked to the correspondent system calls, which allows inferring the relationships between resources by interpreting the semantics of the FStrace program. These relationships are compared to the ones in the dependency graph, which is defined in the Puppet manifest, to check if there are missing dependencies [7]. Their work also identifies missing notifiers, which were not covered in Shambaugh et al.'s work [46].

Weiss et al. created a tool, called Tortoise, that aims to avoid configuration drifts in Puppet scripts when sysadmins use the shell to fix configuration errors [14]. Tortoise records system calls and file system changes caused by shell commands. The tool uses the recorded information and the original Puppet manifest to build a model in a language with primitive operations that manipulate files. The model is translated to logical formulas that are solved by an SMT solver, generating possible patches to Puppet scripts. Constraints coming from the manifest are considered as soft constraints, and the ones coming from shell updates are seen as hard constraints. This allows to update the manifests while trying to minimize changes. The patches are then ranked in a way that favors repairs with fewer changes. By doing experiments with 42 scenarios where the manifests would need repair, the authors identified that "*the highest-rank repair Tortoise synthesized was the correct repair 76% of the time, and the correct repair was in the top five Tortoise-synthesized repairs 100% of the time*" [14].

Ikeshita et al. used a mixed approach that uses both test suites and static verification to check if a Chef program is idempotent [47]. The approach uses an intermediate model that describes file-system manipulations. The model goes through a static verification, capable of reducing the number of necessary tests to check the idempotence of the program [47].

2.4 Intermediate Representations in Software Engineering

Intermediate representations allow the creation of abstractions useful for a variety of purposes. For instance, an intermediate representation may allow the abstraction of concepts to make the job of reasoning about an analysis easier. Another example is the usage of an intermediate representation to link new information to a subset of an existing representation. In our work, we use an intermediate representation to create a common structure to which scripts from multiple technologies can be translated. Many

areas of research in Computer Science benefit from the usage of intermediate representations, and we describe some of them below.

2.4.1 Usage of intermediate representations in IaC

As described in Section 2.3.2, some analysis tools for IaC use intermediate representations [7, 46, 47]. Namely, these tools use intermediate representations to describe file-system manipulations executed by IaC scripts. The manipulations include the creation and removal of files or directories, the creation of links, or the renaming of a file. Shambaugh et al. translated IaC scripts to an intermediate representation by mapping types of resources to their file-system operations [46]. Sotiropoulos et al. used system calls executed by each resource in an IaC script to automatically map the resources to their file-system operations, which were represented in an intermediate language [7]. Even though the main goal of these authors was not the translation of scripts from multiple IaC technologies to these intermediate representations, that would be possible and would allow the execution of the same analyses in other technologies. In Section 3.3.1, we explain why we did not use these intermediate representations in our work.

2.4.2 Usage of intermediate representations in other domains

Besides IaC, other areas of Software Engineering use intermediate representations to achieve their goals. In this section, we describe examples of intermediate representation usage in refactoring tools, software-defined networks, and formal verification of programs.

Silva et al. proposed a language-agnostic tool, called RefDiff 2.0, to identify refactoring operations in source code [48]. The tool is capable of identifying refactors in Java, JavaScript, and C. More languages can be supported by adding plugins to the system. The authors' approach was to create an intermediate representation, called Code Structure Tree (CST), that abstracts the specificities of each programming language. A CST is a tree-like structure that is focused on coarse-grained code elements, such as classes or functions, and the relationships between them. The nodes of the tree contain the following information about a code region: identifier, namespace, parameters list, tokenized source code, tokenized source code of the body, and node type. The node types vary with the programming language's characteristics. For example, in C, the authors defined *file* and *function* as the only node types, while in JavaScript the type *class* also exists. Although different types between languages exist, the analysis rules do not consider specific types, but only the relations of equality or inequality between node types, following the language-agnostic approach. When it comes to relationships, the CST can represent calls (e.g., a method calling another method) or a hierarchy (e.g., a class containing a method or a class extending another class) between nodes. All the analyses are then executed using the CST, which allows

working with different programming languages by only implementing the translator to this representation. Using a dataset of real refactorings in Java, RefDiff 2.0 got a precision of 0.96 and a recall of 0.80, which although not better than the state-of-the-art, it achieved similar enough results considering the language-agnostic approach. The evaluation for JavaScript and C obtained a precision of 0.91 and 0.88 and a recall of 0.88 and 0.91, respectively for each language [48].

Also related to refactoring, Koppel et al. developed an approach to build source-to-source transformations that runs on multiple programming languages by using a new way of representing programs, called incremental parametric syntax [49]. The authors state that intermediate representations do not work for source-to-source transformations since information is lost, which ends up creating unsuitable programs for humans to read. Their approach decomposes languages into generic and language-specific parts, allowing transformations to only run on the generic parts. Programmers can exclusively implement generic nodes for the parts of each language they need for a certain transformation, and the code related to other parts remains unchanged, maintaining the code readable. Although the authors mention that intermediate representations do not work for their use case, they state that these representations work for writing code analyses and code-generators [49].

With the advance of software-defined networking (SDN) in network management, many network programming languages (NPLs) have been proposed that allow operators to very efficiently program network data planes (NDPs). An NDP is the layer of a network device responsible for the forwarding of packages. In SDNs, the NDPs functionality is delivered through software. Considering the variety of NPLs and NDPs, Li et al. proposed an intermediate representation to express NPLs, called Network Transaction Automaton (NTA) [50]. The creation of this representation addresses two problems mentioned by the authors: the lack of interoperability between programs written in different languages, and the dependent evolution of NPLs bound to specific NDPs. To achieve compatibility between each NPL and all the NDPs and vice-versa, an architecture was proposed with NTA as its core. NPLs have a translator (*frontend*) to the intermediate language, and NDPs have a compiler (*backend*) from NTA to their correspondent representation. This architecture allows NPLs and NDPs to evolve independently. An NTA is an automaton with edges associated with network transactions. Network transactions contain the necessary information to model the classes of NPL semantics considered by the authors. These transactions can be represented as three-tuples with the next hop to be forwarded, the consumption of network resources, and a stateful operation capable of checking and updating node variables. Programs are translated to NTAs which are then composed using customized automaton operations. The composition of programs, which can be written in different languages, enables interoperability between NPLs. Li et al. showed that NTA can express the semantics of 6 NPLs used in the industry and is capable of efficiently compose the translated programs without semantic loss [50].

Another field of Computer Science where intermediate representations are present is in the formal

verification of programs. Programs written in regular programming languages, like Java and C, are translated to intermediate verification languages (IVLs), such as Boogie [51] and Why3 [52]. The intermediate language works as a separation between the backend and frontend of the formal verification. Leino described both components of this architecture: "*front end is concerned with breaking down the semantics of given source-language programs into the more primitive operations of the intermediate language, and the back end is concerned with encoding the meaning of the intermediate program as efficient theorem-prover input*" [53].

3

GLITCH: A Framework for Polyglot Smell Detection in IaC

Contents

In this chapter, we describe our framework GLITCH, a technology-agnostic framework that enables automated detection of IaC smells. We start by explaining the motivation behind GLITCH and describe its architecture. In Section 3.3, we present our new intermediate representation to abstract IaC scripts and explain why representations from the literature do not fulfill the requirements for GLITCH. Sections 3.4 and 3.5 describe the IaC technologies and analyses already implemented in GLITCH. Finally, in Section 3.6, we discuss the implementation of GLITCH, explain how to extend our framework, and explore the Visual Studio Code extension we created for GLITCH.

The source code for GLITCH can be found here: https://github.com/sr-lab/GLITCH.

3.1 Motivation

As discussed in Section 1.3, the interest in Infrastructure as Code has increased steadily since 2015. However, we still have few analysis tools to help develop IaC, and the ones we have do not address the scattered IaC technology environment (discussed in 2.1.2). Another problem is that the tools developed in academic research tend to ignore usability concerns because the focus is on fast prototyping to prove an idea. For instance, SLIC and SLAC [4, 10] do not have a proper command-line tool or another type of user interface which makes the usage of those tools harder.

To solve the problems above, we decided to create the framework GLITCH. GLITCH allows the creation of new analyses that generalize to multiple IaC technologies. The generalization increases the number of available analyses to each technology while promoting developments in research by increasing the impact new studies might have. GLITCH promotes usability since the framework already provides an easily extendable command-line tool. Finally, GLITCH increases the consistency between analyses for different technologies.

For instance, both SLIC and SLAC 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 3.1a presents part of a Chef script with no security smells taken from the project Vagrant Chef for CakePHP.[1] For this example, SLAC reports a false positive: a non-existent security smell of type Hard-coded secret. On the other hand, if we consider the same script in Puppet (Figure 3.1b), 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.

[1] https://github.com/FriendsOfCake/vagrant-chef/blob/288336e506a5009ed93c06a784fa93e30a27040c/cookbooks/

```
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 3.1: 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).

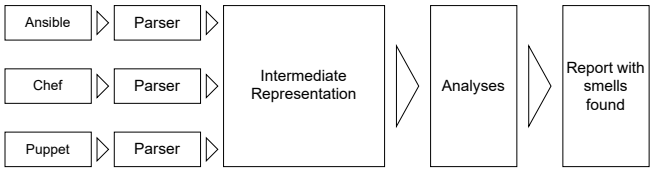

Figure 3.2: A diagram of a simplified version of GLITCH's architecture.

3.2 Architecture Overview

Figure 3.2 enumerates the main components of GLITCH's architecture. The architecture considers three components: **(1)** a parser for each technology which will transform the scripts into our intermediate representation (see Section 3.6.1); **(2)** the intermediate representation to which different analyses can be applied (see Section 3.3); **(3)** the different analyses developed on our framework (see Section 3.5). GLITCH receives as input the scripts of the supported IaC technologies (see Section 3.4) and outputs a report with the smells found by each enabled analysis.

3.3 Intermediate Representation

In this section, we explore the intermediate representation used by GLITCH. In the following subsections, we explain why we decided to create a new intermediate representation and describe it.

3.3.1 Motivation

When we considered the problem of polyglot analyses for IaC, the solution we found was to create an intermediate representation. As shown in Section 2.4, other authors found the same solution to similar problems. We decided to create an intermediate representation from scratch since we were not able to

`percona/recipes/server.rb#L28`

find in the literature a representation capable of abstracting concepts present in IaC scripts from different technologies.

In Section 2.4.2, we describe intermediate representations used in other domains of Computer Science. However, we are not able to reuse these representations since they miss necessary information from the IaC domain. For instance, let us consider the intermediate representation developed by Silva et al. [48] called Code Structure Tree (CST). We chose the study by Silva et al. since we consider it to be the pair problem/solution with more similarities to our own. Each CST node represents a code element in the source code of one of the supported programming languages. As described in Section 2.4.2, the nodes have the following information: identifier, namespace *(optional)*, node type, parameters list *(optional)*, tokenized source code, and tokenized source code of the body. The relationships between nodes can represent calls (e.g., a method calling another method) or a hierarchy (e.g., a class contains a method or a class extends another class). Let us also consider the security smell Admin by Default described in Section 2.2.1. This security smell is the recurring pattern of specifying default users with more privileges than required and it is detected in GLITCH by the following rule:

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

In the rule above, x refers to a node in the representation, and the dot notation is used to access node attributes. The functions *isAttribute* and *isVariable* check the node type, and the remaining functions check for patterns on a string. Considering the attributes in the nodes of a CST, we could map the attribute *name* to the identifier, the node type exists and can be checked, however, the concept of *value* is not present, which means the rule to detect the Admin by Default smell could not be expressed using a CST. Another example is if we consider the resource “apache2” in Figure 2.3, a CST would not capture the resource type “service” since the representation is focused on relationships between nodes instead of their content. However, as we will mention in Chapter 5, the detection of some smells requires information about the resource type. To retrieve this type of information, we need to take into account the constructs of IaC scripts, which excludes the usage of representations from other domains.

Even though we can not use representations from other domains, in Section 2.4.1 we describe intermediate representations used for IaC. These representations describe file-system manipulations executed by IaC scripts and map them to the resources defined in the source code. We did not use these representations since their focus is on the effects of a script in the file system, ignoring the structure and content of the scripts. For instance, if we consider the rule defined for the smell Admin by Default, these representations would not have the required information to implement our rule.

To conclude, we decided to create a new intermediate representation since, to the best of our knowledge, no representation exists that abstracts IaC concepts in a structured and hierarchical way that allows the implementation of rules such as the one defined for the smell Admin by Default.

```
<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>
    }

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

<unitblock> ::=
    UnitBlock {
        name: <str>,
        path: <str>,
        type: SCRIPT | TASKS | VARS |
            BLOCK | UNKNOWN
        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>
    }

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

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
<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 3.3: Abstract syntax of our intermediate representation.

	Ansible	**Chef**	**Puppet**
Modules	Roles	Cookbooks	Modules
Unit Blocks	Playbook	Recipe	Class
Atomic Units	Task	Resource	Resource

Figure 3.4: The table describes what component of each IaC tool corresponds to a component of the intermediate representation.

3.3.2 Abstract Syntax

Figure 3.3 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. It is common in IaC technologies that a folder for each project is created and it has a recommended structure[2]. Table 3.4 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 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

[2]Best practice for Ansible:
https://docs.ansible.com/ansible/latest/user_guide/sample_setup.html#sample-directory-layout

```
resource "aws_instance" "app_server" {
  ami           = "ami-830c94e3"
  instance_type = "t2.micro"

  tags = {
    Name = "ExampleAppServerInstance"
  }
}
```

Figure 3.5: Terraform resource that creates an AWS EC2 instance.

usually has a *tasks* and *vars* folder where, respectively, the tasks and variables for the role are defined[3]). Unit Blocks correspond to the IaC scripts themselves or a group of atomic units. For instance, in Puppet, we can agglomerate resources in classes. In Ansible, we can define IaC scripts that only define tasks or variables. The field type allows us to distinguish the type of a unit block. 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. Finally, IaC technologies may define concepts similar to regular programming languages, such as Condition Statements and Comments, which are also considered in our intermediate representation. As shown in Figure 3.3, unit blocks can have attribute definitions, variable definitions, and conditions. Atomic units have attribute definitions. When the field value in attribute and variable definitions contains variable references, the flag has_variable is set to true.

It is important to mention that even though we focus our attention on technologies for the configuration and management of services, namely, Ansible, Chef, and Puppet, other categories of IaC technologies share similar constructs. Figure 3.5 shows how to define a resource in Terraform[4] to create an AWS EC2 instance[5]. If we implemented a Terraform parser to our intermediate representation, the resource in Figure 3.5 would be translated to an atomic unit with the name “app_server”, type “aws_instance”, and three attributes.

Figures 3.6 and 3.7 show a graph-based visualization of how our intermediate representation models the scripts in Figure 2.1 and Figure 2.3, respectively. The translation of scripts to our intermediate representation generates a tree of relationships between IaC constructs. As described in our abstract syntax and as represented in both figures, the nodes of our intermediate representation carry information about themselves, such as the type of atomic units.

3.4 Supported IaC Technologies

As described in Section 1.1, we focused our attention on technologies that perform configuration management of services. In GLITCH, we implemented parsers from IaC scripts to our intermediate representation for Ansible, Chef, and Puppet. As stated in Section 2.1.2, these three technologies are the

[3] https://docs.ansible.com/ansible/latest/user_guide/sample_setup.html#sample-directory-layout
[4] https://www.terraform.io/
[5] https://aws.amazon.com/ec2/

```
---
- name: Update web servers
  hosts: webservers
  remote_user: root

  tasks:
  - name: Ensure apache is at the latest version
    ansible.builtin.apt:
      name: apache2
      state: latest
  - name: Create default page file
    ansible.builtin.copy:
      dest: /var/www/html/index.html
      content: "Welcome, {{ ansible_default_ipv4.address }}!"
  - name: Start apache service
    ansible.builtin.service:
      name: apache2
      enabled: yes
      state: started
```

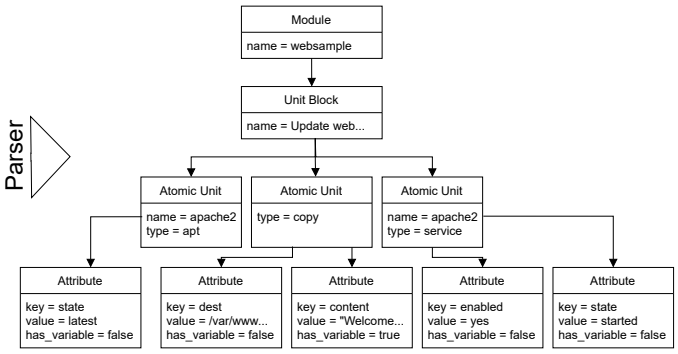

Figure 3.6: Example of the translation of an Ansible script (Figure 2.1) to our intermediate representation.

```
package 'apache2' do
  action :install
end

file '/var/www/html/index.html' do
  content "Welcome! I am " + node['ipaddress']
  action :create
end

service 'apache2' do
  action [ :enable, :start ]
end
```

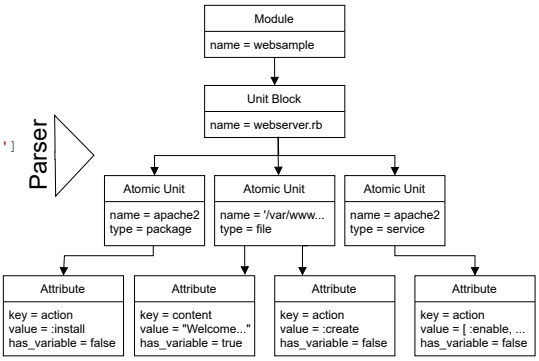

Figure 3.7: Example of the translation of a Chef script (Figure 2.3) to our intermediate representation.

most adopted in the category of configuration management of services. In the future, GLITCH can be extended to support more technologies by following the steps in Section 3.6.5.

3.5 Supported Analyses

In this section, we describe how GLITCH detects the code smells described in Sections 2.2.1 and 2.2.2. We focus on rule-based analyses with a high prevalence of pattern-detection techniques. This type of technique is commonly used in the literature, as is the case with the SLIC and SLAC tools [4, 10].

3.5.1 Security smells

We implemented in GLITCH the nine security smells described in Section 2.2.1. Table 3.1 defines the rules used by GLITCH to detect those security smells. The formalism used to define rules is similar to the one used by SLIC [4] and SLAC [10]. 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, *isAttribute(x)* is true). Each node in our representation is referred to by the variable x. We traverse the

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

Table 3.1: Rules to detect security smells used by 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”

Table 3.2: String patterns used in the GLITCH's rules. These are configurable. The configuration shown is the one used by the improved version of GLITCH.

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 do not influence the analyses of other nodes. The function *hasDownload* goes through a list of attributes and verifies if 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 3.2. 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 3.2. In the table, we describe the configuration used by the improved version of GLITCH to which we will refer in Chapter 4. In Section 3.6.7, we describe in more depth how configurations in GLITCH are defined.

3.5.2 Design & Implementation smells

We implemented in GLITCH the nine design & implementation smells described in Section 2.2.1. There are differences in the detection techniques of these smells between the works by Sharma et al. [2] and Schwarz et al. [3]. We based our implementation on the techniques applied by Schwarz et al. [3]. Appendix A describes in depth the techniques we implemented in GLITCH to detect the nine design & implementation smells described.

3.6 Implementation

In this section, we focus on the implementation aspects of our framework GLITCH. We developed our framework using object-oriented programming. For the programming language, we chose Python since it is one of the most popular programming languages [54], enables fast development, and has popular packages for machine learning, which usage is very frequent in static analysis and program repair. Figure 3.8 describes the class model of GLITCH's implementation. We developed a package for each of the components in Figure 3.2: *parsers*, *inter*, and *analysis*. The usage flow between packages is the same as the one in the components from Figure 3.2. Although in the following sections we summarize how to extend GLITCH, we recommend developers to check GLITCH's documentation: https://github.com/sr-lab/GLITCH/wiki

3.6.1 Parsers

In order to transform IaC scripts to our intermediate representation, first, we need to parse the scripts to get a representation that we can manipulate in Python. To parse the Ansible scripts we use the *ruamel.yaml* package[6] for Python. The *ruamel.yaml* package allows to save comments in the AST, which is the reason why we choose it instead of the *yaml* package[7]. The Chef scripts are parsed using Ripper[8], a script parser for Ruby. To transform the output to Python objects, we developed a parser for Ripper's output using a package called *ply*[9]. Finally, for Puppet scripts, we developed our parser[10] 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. After we obtain the Python representation, we map it to our intermediate representation.

[6] https://pypi.org/project/ruamel.yaml/
[7] https://pypi.org/project/PyYAML/
[8] https://github.com/ruby/ruby/tree/master/ext/ripper
[9] https://github.com/dabeaz/ply
[10] https://github.com/Nfsaavedra/puppetparser

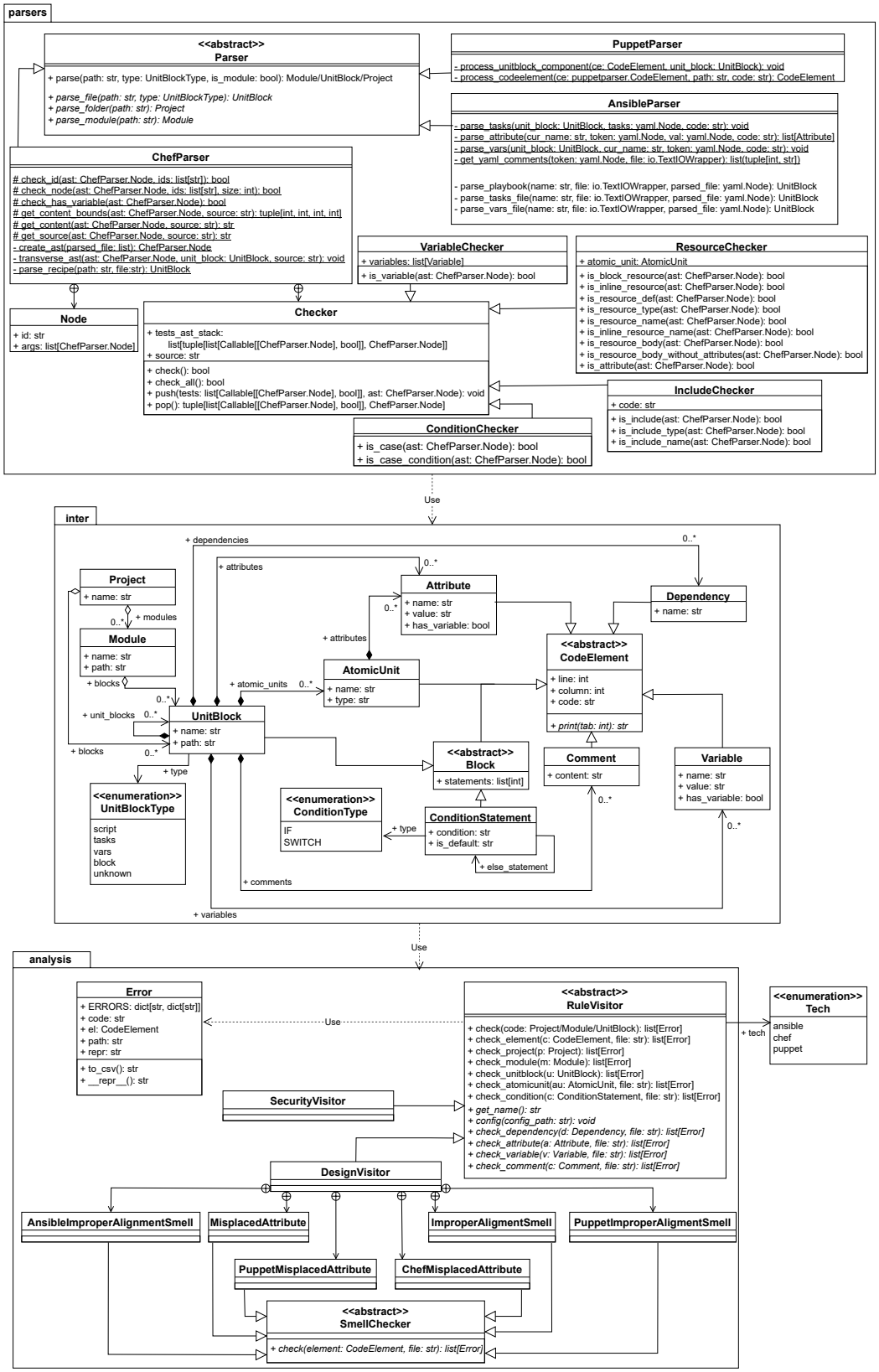

Figure 3.8: The UML Class diagram of the GLITCH framework.

3.6.2 Intermediate Representation

Figure 3.8 describes the class model used to implement our intermediate representation. Every element besides Project and Module inherits from CodeElement or from its subclass Block. The class CodeElement retains information about the element in the source code, namely, the line, column, and code snippet that corresponds to that element. Currently, the implementation does not consider the *column* field. The *line* and *code* fields allow us to map the errors found in the intermediate representation back to the original source code and present the error to the user.

3.6.3 Command-line Tool

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 [12] (described in Section 2.2.1) and nine design & implementation smells [13] (described in Section 2.2.2). If omitted, every smell implemented in the tool is analyzed.
- `--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., Tables 5.3, 5.4, 5.5, 5.6, 5.7 and 5.8 were automatically generated by GLITCH with some manual styling).

3.6.4 Methodology to add new analyses

To create new analyses, the developers need to implement a new subclass of RuleVisitor (see Figure 3.8). The new subclass should implement the abstract methods to check components of the intermediate representation and override the others if necessary (e.g., different way to check a Unit Block). If

```
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 3.9: Implementation of specific behavior when creating new analyses.

specific behavior for a technology is required, the visitors select, in their constructor and according to the technology, instances of the subclasses of SmellChecker. The subclasses implement the method *check* that verifies if a certain smell is detected on a component. The method *check* is called later in the implementation of the methods to check the components of the intermediate representation (see Figure 3.9). The class PuppetImproperAlignmentSmell, which implements the smell Improper Alignment for Puppet only, is an illustrative example of the implementation of specific behavior.

We also need to add identification codes and descriptions for the new types of smells on the *ERRORS* constant present in the Error class.

3.6.5 Methodology to add new IaC technologies

To add a new IaC technology, the developers need to create a new subclass of Parser. The methods *parse_file*, *parse_folder*, and *parse_module* need to be implemented and should return the translation of the script or set of scripts to our intermediate representation. The new technology must be added to the enumeration Tech and the developer should add to the command-line tool the parser that corresponds to the new technology.

3.6.6 Methodology to extend the intermediate representation

To extend the intermediate representation, developers need to create a class for the component being added. The class should inherit from CodeElement or, if the new component can contain statements from the technology being abstracted, from Block. The parsers for the IaC technologies should be changed to consider the new component if the component exists on that technology. Finally, the class RuleVisitor must be changed to include the new component. Default behavior to check the component should be implemented on RuleVisitor or behavior to every existent subclass of RuleVisitor should be added.

3.6.7 Analyses Configuration

The GLITCH framework allows to define configurations in INI files. Configurations allow users to tweak the tool to best suit the needs of the IaC developers and to better adapt to each IaC technology. One

```
def config(self, config_path: str):
    config = configparser.ConfigParser()
    config.read(config_path)
    SecurityVisitor.__WRONG_WORDS = json.loads(config['security']['suspicious_words'])
    SecurityVisitor.__PASSWORDS = json.loads(config['security']['passwords'])
    SecurityVisitor.__USERS = json.loads(config['security']['users'])
```

Figure 3.10: Example of the implementation of a *config* method for a subclass of RuleVisitor.

example of usage is to define the keywords that trigger the detection of a certain smell. Configurations are loaded by the *config* method implemented in each subclass of RuleVisitor. Figure 3.10 shows an example of how to implement a *config* method using the package *configparser*[11]. A default configuration is defined for GLITCH and should be changed if a RuleVisitor depends on a new configuration field.

3.6.8 Execution statistics and Output

At the end of a run, GLITCH outputs execution statistics and the smells found for the given input. Execution statistics include the number of files and lines of code analyzed, and the number of occurrences, smell density, and proportion of scripts for each smell. The metrics *smell density* and *proportion of scripts* are explained in Section 4.1.3. The smells found are described by the line where they occurred, a snippet of code, and the name of the smell, possibly with a small explanation.

3.6.9 Integration Tests

GLITCH has integration tests for each implemented smell in each technology. The tests use the framework *unittest*[12] from Python. Each test has its corresponding IaC script with smells previously identified by the developer. The tests compare if the smells identified by GLITCH are the same as the ones manually identified.

3.6.10 Visual Studio Code extension

Sadowski et al. mention the importance of including detection tools in the developers' workflow [55]. For that reason, we created a Visual Studio Code extension for GLITCH, which is available here:
https://marketplace.visualstudio.com/items?itemName=sr-lab.glitch-iac

The extension uses visual feedback to alert developers to the lines where smells were detected by GLITCH (see Figure B.1). By hovering the line with the mouse, developers can get more information about the smell, namely the name of the smell found and a small explanation about it. The extension has settings to disable the analyses, define the path for an analyses configuration, the technology to analyze, and the types of smells to detect.

[11] https://pypi.org/project/configparser/
[12] https://docs.python.org/3/library/unittest.html

4

Security Smells in IaC: An Empirical Study

Contents

As described in Section 3.5.1, we implemented the nine security smells studied by Rahman et al. [4, 10] in GLITCH. In this chapter, we evaluate the accuracy of GLITCH in the detection of security smells and compare its accuracy to state-of-the-art tools. We also conduct a large-scale empirical study that analyzes security smells on three large datasets containing 196,755 IaC scripts and 12,281,251 LOC. Finally, we discuss the obtained results. The contents of this chapter were published at ASE 2022 [12].

A replication package containing all the datasets used, including the three oracle datasets that were manually annotated, is available here:

https://doi.org/10.6084/m9.figshare.19726603.v2

4.1 Evaluation

We aim to answer the following research questions:

RQ4.1. **[Abstraction]** Can our intermediate representation model IaC scripts and support automated detection of security smells?

We are interested in determining whether our intermediate representation is capable of abstracting relevant information from IaC scripts written in different IaC languages so that one can define security smell detectors on it.

RQ4.2. **[Accuracy and Performance]** How does GLITCH compare with existing state-of-art tools for detecting security smells in terms of accuracy and performance?

We are interested in comparing the accuracy and performance of GLITCH with the accuracy of existing tools, such as SLIC [4] and SLAC [10].

RQ4.3. **[Frequency]** How frequently do security smells occur in IaC scripts?

We are interested in characterizing how frequently security smells are present in IaC scripts. This research question was addressed by Rahman et al. [4] for Puppet and by Rahman et al. [10] for Ansible and Chef. They used different tools for answering this question. Here, we want to use GLITCH to investigate whether there are any noticeable differences.

4.1.1 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 [4, 10]. We got hold of the publicly available datasets[1] and Docker image[2], and we observed that only the oracle for Ansible was available. We thus contacted the

[1] https://doi.org/10.6084/m9.figshare.8085755
[2] https://hub.docker.com/repository/docker/akondrahman/slic_ansible

			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	1,613	2,840	2,845
Total LOC (IaC scripts)	5,180,747	6,071,035	610,122	66,367	217,843	135,137

Table 4.1: Attributes of IaC Datasets.

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

Table 4.2: Attributes of Oracle Datasets.

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.

4.1.1.A 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 [4], 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 [56], OSS repositories need to be curated. We apply the same criteria that Rahman et al. [4] 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 [57], 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 a 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. [56], 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. [4], we assume that this criterion may help us to filter out irrelevant repositories.

Table 4.1 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.

4.1.1.B 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 [10], 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. After the minimum number of reports for each smell was achieved, the remaining files were added to the dataset without restrictions. Table 4.2 presents the number of IaC scripts and the number of LOC in the three oracle datasets.

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

	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

Table 4.3: Agreement distribution for the oracle datasets (%).

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 [10], we decided to reannotate the dataset. To annotate the oracle datasets, we used closed coding [58], 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 4.3 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 another 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 4.4 and with 69 files with no smells; an oracle of 105 Chef security smells categorized as shown in Table 4.5 and with 43 files with no smells; and an oracle of 65 Puppet security smells categorized as shown in Table 4.6 and with 52 files with no smells.

4.1.2 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 the precision and recall of each tool. Precision refers to the fraction of correctly identified smells among the total identified security smells, as determined by each tool. Recall refers to the fraction of correctly identified smells that have been retrieved by each tool over the total amount of security smells. Since it is easy to configure GLITCH (see Section 3.6.7), 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 3.5.1, the difference between the two versions is in the keywords for each function in Table 3.2: 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 3.2.

Tables 4.4, 4.5, and 4.6 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 4.4). 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.

4.1.2.A Accuracy results for the Ansible oracle dataset

As shown in Table 4.4, 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 the detection of No integrity check 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 a YAML file, which GLITCH does not consider (i.e., the string pattern *isDownload()* shown in Table 3.2 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.

4.1.2.B Accuracy results for the Chef oracle dataset

Table 4.5 shows that when GLITCH is configured to behave similarly to SLAC, it actually obtains better results than SLAC: the average precision improves by 28 percentage points (from 49% to 77%) and the average recall improves by 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`).

4.1.2.C Accuracy results for the Puppet oracle dataset

Similar to what was described above, Table 4.6 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

		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 w/o 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 crypt. alg.	0	N/I	N/I	N/I	N/I	N/D	N/D
Missing def. case stat.	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 4.4: GLITCH vs SLAC: Accuracy for the Ansible Oracle Datasets (N/I - Not implemented, N/A - Not applicable, N/D - No data)

high precision. Also, the precision for the smell Empty password is noticeably 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 by 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.

4.1.3 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. [4] (for Puppet scripts using SLIC) and Rahman et al. [10] (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 4.1.1 and their attributes are shown in Table 4.1. 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.

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:

		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 w/o 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 crypt. alg.	1	0.25	1.00	0.25	1.00	0.50	1.00
Missing def. case stat.	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 4.5: GLITCH vs SLAC: Accuracy for the Chef 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 w/o 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 crypt. alg.	4	0.43	0.75	0.50	0.75	0.57	1.00
Missing def. case stat.	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

Table 4.6: GLITCH vs SLIC: Accuracy for the Puppet Oracle Datasets (N/I - Not implemented, N/A - Not applicable, N/D - No data)

- **Smell density**: frequency of a given security smell for every 1,000 LOC [10, 59]. 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.

	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 bind.	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 w/o 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 crypt. alg.	N/I	1,502	76	147	227	109	48	28	27	18	26	21
Missing def. case stat.	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 4.7: 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	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 w/o 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 crypt. 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 def. case stat.	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 4.8: 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	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 w/o 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 crypt. 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 def. case stat.	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

Table 4.9: Proportion of Scripts (Script%) with at Least One Smell. (N/I - Not implemented, N/A - Not applicable, N/D - No data)

4.1.3.A Occurrences

Looking at Table 4.7, 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 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.

4.1.3.B Smell density

Table 4.8 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 4.1.2), 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).

4.1.3.C Proportion of Scripts (Script%)

Table 4.9 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 SLAC, the percentage is 23.8%, but note that SLAC only supports six security smells. This is not very different from the values obtained by Rahman et al. [10], where the percentages obtained with SLAC 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 SLAC, the percentage is slightly higher at 11.4%. Here, we note a more noticeable discrepancy with Rahman et al.'s study [10]: the percentages obtained with SLAC 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 4.1.2), this suggests

			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

Table 4.10: The average execution times between 5 runs (seconds).

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.

4.1.3.D Execution times

The execution times of GLITCH, SLIC, and SLAC for the three datasets are shown in Table 4.10 (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[3] and foodcritic[4] 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.

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

4.2 Discussion

In this section, we answer the research questions listed in Section 4.1, and we outline potential threats to the validity of our work.

4.2.1 Answers to Research Questions

Given the findings reported in the previous section, we answer the research questions posed in Section 4.1 as follows:

Answer to RQ4.1 [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 RQ4.2 [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 4.4, 4.5, and 4.6, 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 by 10 percentage points and average recall improved by 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 4.10 shows, GLITCH is much faster at analyzing Chef and Puppet scripts than tools such as SLIC or SLAC (speedups vary from 9.14$\times$ to 32.07$\times$). For Ansible, GLITCH takes more than twice as long as 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 RQ4.3 [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 4.1.2), this sug-

gests that many of these are false positives. For Ansible, the second most dominant smell is Admin by default (5.7%). For Chef and 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.

4.2.2 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 3.1. 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 2.2, 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 [4, 10]. Finally, all the datasets used in our work are from open-source projects and not from proprietary sources.

5

Design & Implementation Smells in IaC: An Empirical Study

Contents

In this chapter, we study the use of GLITCH to detect design & implementation smells in IaC scripts. We use the same three IaC datasets as in Chapter 4, whose attributes are presented in Table 4.1. Table 5.1 also presents the values for each attribute considering only the code GLITCH was able to analyze in this study. These values were obtained from GLITCH's output. The content of this chapter is unpublished, but it is part of a tool paper that will be submitted soon [13]. We focus our evaluation on the following questions:

RQ5.1. **[Expressiveness]** Is it possible to implement in GLITCH the detection of design & implementation smells present in state-of-the-art tools and obtain similar results?

RQ5.2. **[Frequency]** How frequently do design & implementation smells occur in IaC scripts?

The replication package for the experiments in Sections 5.1 and 5.2 is available here: https://doi.org/10.6084/m9.figshare.21407058.v1

5.1 Comparing GLITCH to state-of-the-art tools

We are interested in comparing GLITCH to the two state-of-the-art tools that detect design & implementation smells: Puppeteer [2], which detects smells in Puppet, and Schwarz et al's tool, which detects smells in Chef. To achieve this goal, we created two oracle datasets from the datasets described in Table 5.1. The oracle datasets were created by randomly selecting 20 files for each smell, with the smell being detected on each of the selected files by Puppeteer if we were handling Puppet scripts or Schwarz et al's tool if we were handling Chef scripts. This resulted in a total of 80 Puppet files and 160 Chef files. Afterward, we executed GLITCH and the two state-of-the-art tools on these oracle datasets and obtained the results shown in Table 5.2. 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, and the other shows the line number of the attribute). For these cases, we had to manually inspect the files and check

				Puppet			
Attribute		**Ansible**	**Chef**	**GH**	**MOZ**	**OST**	**WIK**
Repository Count	**Total**	681	439	219	2	61	11
	GLITCH	681	439	219	2	61	11
IaC scripts	**Total**	108,509	70,939	10,009	1,613	2,840	2,845
	GLITCH	102,780	29,723	9,776	1,613	2,840	2,845
LOC (IaC scripts)	**Total**	5,180,747	6,071,035	610,122	66,367	217,843	135,137
	GLITCH	4,873,597	1,638,539	531,577	66,367	217,843	135,137

Table 5.1: Attributes of IaC Datasets. The GLITCH rows have the values for each attribute considering the code GLITCH was able to analyze.

Smells	$\frac{\#\{\text{GLITCH} \cap \text{Puppeeteer}\}}{\#\text{Puppeeteer}}(\%)$	$\frac{\#\{\text{GLITCH} \cap \text{Puppeeteer}\}}{\#\text{GLITCH}}(\%)$	$\frac{\#\{\text{GLITCH} \cap \text{Schwarz}\}}{\#\text{Schwarz}}(\%)$	$\frac{\#\{\text{GLITCH} \cap \text{Schwarz}\}}{\#\text{GLITCH}}(\%)$
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

Table 5.2: Comparison of GLITCH to state-of-the-art tools (Puppeteer [2] and Schwarz et al.'s tool [3]). $\#\{\mathbf{x} \cap \mathbf{y}\}/\#\mathbf{y}$ represents the fraction of smells detected by y that are also present in x.

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.

We verified that GLITCH can detect almost every smell detected by Puppeteer [2] (first column). The value for Improper alignment is slightly below 100% because GLITCH does not consider the alignment in hashes[1] 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,[2] 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 conclude 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. [3], there are two smells with lower percentage values: **(1)** Improper alignment and **(2)** Too many variables (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. 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: **(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 true positives that the other tool does not, since in the detection of this smell, the Schwarz et al.'s tool does not handle multi-line strings and ignores the pipe character "|".

[1] https://puppet.com/docs/puppet/latest/lang_data_hash.html
[2] https://puppet.com/docs/puppet/latest/style_guide.html

5.2 Design & Implementation Smells Frequency

Tables 5.3 to 5.8 show the results of running GLITCH to detect design & implementation smells on the datasets listed in Table 5.1.

Occurrences Overall, GLITCH detects 580,713 design & implementation smells for Ansible, 217,052 for Chef, and 61,766 for Puppet. For Ansible and Chef, the three most frequent smells are Avoid comments, Duplicate block, and Long statement. Considering the four Puppet datasets, the most frequent smells for Puppet are Avoid Comments, Improper Alignment, and Duplicate Block.

Smell density Overall, GLITCH detects 119.16 design & implementation smells per 1,000 LOC in Ansible scripts, 132.47 for Chef, and an average of 67.10 for Puppet. For Ansible and Chef, the most dominant smells are Avoid comments, Duplicate block, and Long statement. In the Puppet Github and Puppet Openstack datasets, the most dominant smells are Avoid Comments, Improper Alignment, and Duplicate Block. For the Puppet Mozilla and Wikimedia datasets, the smell Long Statement replaces Improper Alignment in the most dominant smells.

Proportion of Scripts (Script%) GLITCH detects at least one design & implementation smell in 51.16% of Ansible scripts and 57.67% of Chef scripts. For Puppet, in the GitHub, Mozilla, OpenStack, and Wikimedia datasets, GLITCH detects at least one of the smells in, respectively, 45.85%, 54.25%, 49.65%, and 52.97% of the total scripts. In Ansible and Puppet, even though the smell Duplicate block has more occurrences than Long statement, the proportion of scripts where Duplicate block is detected is lower. A similar case happens in Chef where the smell Duplicate block is the second most frequent smell, but it is only detected on 5.52% of scripts. Contrariwise, the smell Improper alignment, which is the fourth most frequent smell, is detected on 6.98% of scripts. One possible reason for the lower proportion of scripts for Duplicate Block is that the detection of a Duplicate Block implies at least two occurrences of this type of smell in the same script.

5.3 Discussion

In this section, we answer the research questions posed at the beginning of Chapter 5 and outline potential threats to the validity of our work.

5.3.1 Answers to Research Questions

Given the results reported in Sections 5.1 and 5.2, we answer the research questions as follows:

RQ5.1. **[Expressiveness]** *Is it possible to implement in GLITCH the detection of design & implementation smells present in state-of-the-art tools and obtain similar results?* Yes. We demonstrate that GLITCH can detect an average across all smells of 99.6% of the smells identified by Puppeeter and 81.7% of the smells identified by Schwarz et al.'s tool. We also demonstrate that Puppeteer detects 93.4% of GLITCH's smells and that Schwarz et al.'s tool detects 82.2% of the smells identified by GLITCH. The main reasons for disagreements are false positives/negatives by the state-of-the-art tools and lack of granularity in the intermediate representation of GLITCH, which is one of the future directions of our research (see Section 6.2). Overall, the agreement between GLITCH and the state-of-the-art tools is high.

RQ5.2. **[Frequency]** *How frequently do design & implementation smells occur in IaC scripts?* GLITCH detects at least one design & implementation smell in 51.16% of Ansible scripts, 57.67% of Chef scripts, and an average of 50.58% of Puppet scripts between the four datasets. The high frequency of the smell Avoid comments suggests that comments may be used as a deodorant to bad code [28] in IaC scripts. When we consider the Chef scripts in our datasets, the smell Long statement is detected on 13.21% of scripts and Improper Alignment on 6.98%. For the Puppet GitHub dataset, the smell Long statement is detected on 7.48% of scripts and Improper Alignment on 12.62%. These values suggest that the usage of style linters in the development of Chef and Puppet scripts may need to increase. The smell Duplicate block is the second most frequent smell across all technologies and the smell Multifaceted abstraction is detected on 6.22% of Ansible scripts, 5.14% of Chef scripts, and an average between the four Puppet datasets of 3.84% of scripts. The high frequency of both these smells suggests that abstractions should be used more often in IaC scripts.

5.3.2 Threats to Validity

A threat to conclusion validity is that the subset of scripts used to compare GLITCH to the state-of-the-art tools was created by randomly selecting files for each smell, with the smell being detected on each of the selected files, which may create bias. Instead of using GLITCH, we used the state-of-the-art tools to detect the smells. Since our only goal was to compare GLITCH to these tools, we argue that the bias is significantly reduced. 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. A threat to external validity is that since we only compare GLITCH to Puppeeteer and Schwarz et al's tool, GLITCH may not be able to replicate other implementations of these smells or implement other classes of smells. Also, since we focus on Ansible, Chef, and Puppet scripts, our findings may not be generalizable to other IaC technologies.

Smell	Occurrences	Smell density (Smell/KLoC)	Proportion of scripts (%)
Avoid comments	427,686	87.76	39.76
Duplicate block	88,430	18.14	7.94
Improper alignment	480	0.10	0.13
Long Resource	6,441	1.32	3.24
Long statement	39,075	8.02	10.43
Misplaced attribute	0	0.00	0.00
Multifaceted Abstraction	14,182	2.91	6.22
Too many variables	299	0.06	0.29
Unguarded variable	0	0.00	0.00
Combined	580,713	119.16	48.93

Table 5.3: Smell occurrences, Smell Density (SD), and Proportion of Scripts (PS) for the Ansible dataset.

Smell	Occurrences	Smell density (Smell/KLoC)	Proportion of scripts (%)
Avoid comments	186,916	114.07	47.29
Duplicate block	10,434	6.37	5.52
Improper alignment	4,187	2.56	6.98
Long Resource	688	0.42	1.74
Long statement	8,250	5.03	13.21
Misplaced attribute	3,099	1.89	5.92
Multifaceted Abstraction	2,551	1.56	5.14
Too many variables	190	0.12	0.64
Unguarded variable	0	0.00	0.00
Combined	217,052	132.47	57.67

Table 5.4: Smell occurrences, Smell Density (SD), and Proportion of Scripts (PS) for the Chef dataset.

Smell	Occurrences	Smell density (Smell/KLoC)	Proportion of scripts (%)
Avoid comments	22,208	41.78	31.20
Duplicate block	2,066	3.89	3.23
Improper alignment	6,471	12.17	12.62
Long Resource	170	0.32	1.21
Long statement	1,689	3.18	7.48
Misplaced attribute	441	0.83	2.88
Multifaceted Abstraction	648	1.22	4.47
Too many variables	297	0.56	3.04
Unguarded variable	1,586	2.98	4.01
Combined	35,868	67.48	45.85

Table 5.5: Smell occurrences, Smell Density (SD), and Proportion of Scripts (PS) for the Puppet GitHub (GH) dataset.

Smell	Occurrences	Smell density (Smell/KLoC)	Proportion of scripts (%)
Avoid comments	4,113	61.97	48.73
Duplicate block	356	5.36	4.40
Improper alignment	103	1.55	4.22
Long Resource	40	0.60	1.67
Long statement	193	2.91	6.70
Misplaced attribute	1	0.02	0.06
Multifaceted Abstraction	53	0.80	2.85
Too many variables	48	0.72	2.98
Unguarded variable	16	0.24	0.68
Combined	4,969	74.86	54.25

Table 5.6: Smell occurrences, Smell Density (SD), and Proportion of Scripts (PS) for the Puppet Mozilla (MOZ) dataset.

Smell	Occurrences	Smell density (Smell/KLoC)	Proportion of scripts (%)
Avoid comments	8,132	37.33	34.82
Duplicate block	535	2.46	4.54
Improper alignment	545	2.50	10.74
Long Resource	61	0.28	1.90
Long statement	314	1.44	7.75
Misplaced attribute	56	0.26	1.41
Multifaceted Abstraction	218	1.00	4.47
Too many variables	180	0.83	6.34
Unguarded variable	122	0.56	2.50
Combined	10,263	47.12	49.65

Table 5.7: Smell occurrences, Smell Density (SD), and Proportion of Scripts (PS) for the Puppet OpenStack (OST) dataset.

Smell	Occurrences	Smell density (Smell/KLoC)	Proportion of scripts (%)
Avoid comments	9,541	70.60	46.33
Duplicate block	257	1.90	1.90
Improper alignment	236	1.75	5.98
Long Resource	40	0.30	1.16
Long statement	348	2.58	7.56
Misplaced attribute	1	0.01	0.04
Multifaceted Abstraction	124	0.92	3.59
Too many variables	51	0.38	1.79
Unguarded variable	30	0.22	0.98
Combined	10,666	78.94	52.97

Table 5.8: Smell occurrences, Smell Density (SD), and Proportion of Scripts (PS) for the Puppet Wikimedia (WIK) dataset.

6

Conclusion

Contents

6.1 Conclusions

In this thesis, we present the first automated polyglot code smell detection framework for IaC – GLITCH. GLITCH reduces the effort to write code smell analyses for multiple IaC technologies and avoids inconsistencies between implementations.

We start by explaining what Infrastructure as Code (IaC) is and describe its ecosystem. IaC suffers from the same problems as traditional software engineering, namely, bugs in scripts, which led us to study state-of-the-art analyses to detect problems in IaC scripts. In our study of these analyses, we discovered that they all have a problem in common; they are implemented only to a single technology. However, the IaC technology ecosystem is scattered and for that reason it is important that code analyses are implemented for multiple technologies. We reached the question *"How can we implement polyglot analyses for IaC scripts?"*. The idea we found to solve this problem was to create an intermediate representation to abstract IaC scripts on which analyses would execute. To gain confidence in our idea, we studied approaches in the IaC domain and in other domains of Computer Science that use intermediate representations to solve similar problems.

We present our solution called GLITCH, a new technology-agnostic framework that allows polyglot smell detection in IaC scripts, by transforming them into a new intermediate representation on which different smell detectors can be defined. GLITCH currently supports the detection of nine security smells and nine design & implementation smells in Puppet, Ansible, or Chef scripts. We developed a Visual Studio Code extension that allows developers to have immediate visual feedback on the smells detected by GLITCH. To ascertain the value of our framework, we conducted a study for the two classes of smells that we implemented.

In our study about security smells, 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. The study 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 the nine security smells documented in the literature. We found that all categories of security smells are identified across all datasets, and we identified some smells that might affect many IaC projects. Some of the rules for security smells 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).

We also implemented the detection of nine design & implementation smells in GLITCH and compared it to existing state-of-the-art tools. We checked the proportion of output given by GLITCH that is equivalent to the output given by other tools. We were able to conclude that GLITCH can express the same detection techniques as other tools and obtain similar results. We conducted a large-scale

empirical study where we consider the nine design & implementation smells that we implemented. We found that around 51% of all IaC scripts suffers from at least one of the design & implementation smells with the most frequent smells being Avoid comments, Duplicate block, and Long statement for Ansible and Chef, and Avoid comments, Improper alignment, and Duplicate block for Puppet. We conclude that the usage of style linters may need to increase and that abstractions should be used more often in IaC scripts.

The main practical implication of this thesis 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. 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 open-source replication packages that can be used by the community. We argue that GLITCH and the datasets that we created and made available in our replications packages are very valuable assets for driving reproducible research in the analysis of IaC scripts.

Finally, we answer our research questions as follows:

RQ1: *Is it possible to create a model which abstracts different IaC technologies? Are the concepts expressed in the model relevant enough to apply different analyses from the literature to it?*

Yes, our intermediate representation can abstract different IaC technologies. We implemented analyses in GLITCH to detect nine security smells and nine design & implementation smells, which proves that the concepts expressed in our model are relevant enough to implement different analyses from the literature. We also 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.

RQ2: What limitations do we find when creating a model which abstracts IaC concepts between different technologies?

One limitation is the representation of technology-specific aspects (i.e. aspects that can not be abstracted between multiple IaC technologies). We try to mitigate this limitation with configurations that can be easily changed between technologies (see Section 3.6.7). For instance, the detection of the smell Multifaceted abstraction requires knowing which types of atomic units execute shell scripts. These types of atomic units change between technologies and for that reason GLITCH has a configuration to define them. Also, we allow the definition of technology-specific behavior for the detection of a smell. Another limitation is related to the granularity of the model. Currently, the intermediate representation used by GLITCH only considers high-level structures. However, some analyses require more knowledge about low-level elements. For instance, the smell Improper alignment can be found on Puppet hashes, but since GLITCH represents all values as strings, we are not able to detect it. Future work should increase the granularity of the intermediate representation.

RQ3: Are we able to use our framework to obtain similar results to the analyses in the state-of-the-art?

Yes. For security smells, we were able to obtain higher precision and recall than state-of-the-art tools. For design & implementation smells, GLITCH was able to detect the majority of the smells that were detected by state-of-the-art tools. The empirical studies conducted on both classes of smells are in agreement with studies in the literature.

6.2 Future Work

We identify three main challenges to address in the future:

1. **Quality.** This challenge is about increasing the precision and recall of GLITCH. For security smells, 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. Recent work by Reis et al. has improved some of the rules for security smells and presented three new rules [60]. Since the authors' implementation only considers Puppet, it would be interesting to implement these improvements on GLITCH. Addressing this challenge is perhaps the most important step toward real-life adoption of GLITCH.

2. **Scope.** This challenge is about extending GLITCH to support more IaC technologies and 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 [7] or intra-update sniping vulnerabilities [8]. Also, exploring automated repair techniques is an interesting avenue for future work. The extension of the intermediate representation to be more granular is another important aspect to consider. Every value contained in the current components of the intermediate representation is represented as a string, but we should have a more structured approach for these values (e.g., we should model lists, dictionaries, integers, and operations between values). 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. Beyond the extension for Visual Studio Code that we already developed, the following could bring added value: integration with continuous integration (CI) processes (e.g., GitHub actions), integration with popular IDEs, 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 [61].

Bibliography

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

[2] 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.

[3] 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, pp. 220–228.

[4] 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.

[5] J. Fryman, “DNS outage post mortem,” Jan 2014, accessed: 3 May 2022. [Online]. Available: https://github.blog/2014-01-18-dns-outage-post-mortem/

[6] R. Hersher, “Amazon and the $150 Million typo,” Mar 2017, accessed: 3 May 2022. [Online]. Available: https://www.npr.org/sections/thetwo-way/2017/03/03/518322734/amazon-and-the-150-million-typo?t=1651588365675

[7] 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.

[8] J. Lepiller, R. Piskac, M. Schäf, and M. Santolucito, “Analyzing Infrastructure as Code to Prevent Intra-update Sniping Vulnerabilities.” in *TACAS (2)*, 2021, pp. 105–123.

[9] S. D. Palma, D. Di Nucci, F. Palomba, and D. A. Tamburri, "Within-project defect prediction of Infrastructure-as-Code using product and process metrics," *IEEE Transactions on Software Engineering*, pp. 1–1, 2021.

[10] 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, pp. 1–31, 2021.

[11] A. Alnafessah, A. U. Gias, R. Wang, L. Zhu, G. Casale, and A. Filieri, "Quality-Aware DevOps Research: Where Do We Stand?" *IEEE Access*, vol. 9, pp. 44 476–44 489, 2021.

[12] 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.

[13] ——, "Polyglot Code Smell Detection for Infrastructure as Code with GLITCH," 2022, Submitted for publication. Preprint available: https://www.nuno.saavedra.pt/articles/2022_ASE_TOOL_GLITCH_preprint.pdf.

[14] A. Weiss, A. Guha, and Y. Brun, "Tortoise: Interactive system configuration repair," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 625–636.

[15] A. Rahman, R. Mahdavi-Hezaveh, and L. Williams, "A systematic mapping study of Infrastructure as Code research," *Information and Software Technology*, vol. 108, pp. 65–77, 2019.

[16] Progress Chef, "Case Study - A Lesson in Digital transformation in the Midst of a Global Pandemic," 2020, https://www.chef.io/customers/edgenuity.

[17] Red Hat, "Customer Case Study - German Federal Office speeds I.T. management by 50% with Red Hat Ansible Tower," 2018, https://www.ansible.com/hubfs/Images/resources/rh-ble-case-study-f13497wg-201810-en_1.pdf?hsLang=en-us.

[18] Puppet Labs, Zivra, "Case Study - Jewelers Mutual Insurance Company collaborated with Zivra to speed up the creation of environments for maximum efficiency," 2019, https://puppet.com/resources/customer-story/jewelers-mutual.

[19] S. Bruce, "Ansible vs Chef: Which configuration management tool is best?" Dec 2020. [Online]. Available: https://careerkarma.com/blog/ansible-vs-chef/

[20] Ansible, Inc., "Ansible Documentation," 2021, visited on 2021-12-27. [Online]. Available: https://docs.ansible.com/

[21] Progress Chef, “Chef Documentation,” 2021, visited on 2021-12-29. [Online]. Available: https://docs.chef.io/

[22] Puppet Labs, “Puppet Documentation,” 2021, visited on 2021-12-29. [Online]. Available: https://puppet.com/docs

[23] M. Fowler, “CodeSmell,” Feb 2006, accessed: 22 October 2022. [Online]. Available: https://martinfowler.com/bliki/CodeSmell.html

[24] MITRE, “CWE-Common Weakness Enumeration,” 2022, https://cwe.mitre.org/index.html.

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

[26] P. Mutaf, “Defending against a Denial-of-Service Attack on TCP.” in *Recent Advances in Intrusion Detection*, 1999.

[27] E. Rescorla *et al.*, “HTTP over TLS,” 2000, rFC 2818, May.

[28] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[29] ——, “Avoiding repetition [software design],” *IEEE Software*, vol. 18, no. 1, pp. 97–99, 2001.

[30] Ö. Albayrak and D. Davenport, “Impact of maintainability defects on code inspections,” in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, 2010, pp. 1–4.

[31] Progress Chef, “Ruby Guide,” 2022, visited on 2022-08-27. [Online]. Available: https://docs.chef.io/ruby/

[32] Puppet Labs, “The Puppet language style guide,” 2022, visited on 2022-08-27. [Online]. Available: https://puppet.com/docs/puppet/latest/style_guide.html

[33] R. C. Martin, J. Newkirk, and R. S. Koss, *Agile software development: principles, patterns, and practices*. Prentice Hall Upper Saddle River, NJ, 2003, vol. 2.

[34] R. Potvin and J. Levenberg, “Why Google stores billions of lines of code in a single repository,” *Communications of the ACM*, vol. 59, no. 7, pp. 78–87, 2016.

[35] N. Rockwell, “Summary of June 8 outage,” Jun 2021. [Online]. Available: https://www.fastly.com/blog/summary-of-june-8-outage

[36] "More details about the October 4 outage." [Online]. Available: https://engineering.fb.com/2021/10/05/networking-traffic/outage-details/

[37] A. Ribeiro, J. F. Ferreira, and A. Mendes, "EcoAndroid: An Android Studio plugin for developing energy-efficient Java mobile applications," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2021, pp. 62–69.

[38] R. B. Pereira, J. F. Ferreira, A. Mendes, and R. Abreu, "Extending EcoAndroid with Automated Detection of Resource Leaks," in *2022 IEEE/ACM 9th International Conference on Mobile Software Engineering and Systems (MobileSoft)*, 2022, pp. 17–27.

[39] M. Monperrus, S. Urli, T. Durieux, M. Martinez, B. Baudry, and L. Seinturier, "Repairnator patches programs automatically," *Ubiquity*, vol. 2019, no. July, pp. 1–12, 2019.

[40] A. Rahman and L. Williams, "Characterizing defective configuration scripts used for continuous deployment," in *2018 IEEE 11th International conference on software testing, verification and validation (ICST)*. IEEE, 2018, pp. 34–45.

[41] ——, "Source code properties of defective Infrastructure as Code scripts," *Information and Software Technology*, vol. 112, pp. 148–163, 2019.

[42] S. D. Palma, M. Mohammadi, D. Di Nucci, and D. A. Tamburri, "Singling the odd ones out: a novelty detection approach to find defects in Infrastructure-as-Code," in *Proceedings of the 4th ACM SIGSOFT International Workshop on Machine-Learning Techniques for Software-Quality Evaluation*, 2020, pp. 31–36.

[43] A. Rahman, E. Farhana, C. Parnin, and L. Williams, "Gang of eight: A defect taxonomy for infrastructure as code scripts," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 752–764.

[44] W. Chen, G. Wu, and J. Wei, "An approach to identifying error patterns for Infrastructure as Code," in *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2018, pp. 124–129.

[45] N. Borovits, I. Kumara, P. Krishnan, S. D. Palma, D. Di Nucci, F. Palomba, D. A. Tamburri, and W.-J. van den Heuvel, "DeepIaC: deep learning-based linguistic anti-pattern detection in IaC," in *Proceedings of the 4th ACM SIGSOFT International Workshop on Machine-Learning Techniques for Software-Quality Evaluation*, 2020, pp. 7–12.

[46] R. Shambaugh, A. Weiss, and A. Guha, "Rehearsal: A configuration verification tool for puppet," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016, pp. 416–430.

[47] 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.

[48] D. Silva, J. Silva, G. J. D. S. Santos, R. Terra, and M. T. O. Valente, “RefDiff 2.0: A multi-language refactoring detection tool,” *IEEE Transactions on Software Engineering*, 2020.

[49] J. Koppel, V. Premtoon, and A. Solar-Lezama, “One tool, many languages: language-parametric transformation with incremental parametric syntax,” *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–28, 2018.

[50] H. Li, P. Zhang, G. Sun, C. Hu, D. Shan, T. Pan, and Q. Fu, “An Intermediate Representation for Network Programming Languages,” in *4th Asia-Pacific Workshop on Networking*, 2020, pp. 1–7.

[51] K. R. M. Leino, “This is boogie 2,” *manuscript KRML*, vol. 178, no. 131, p. 9, 2008.

[52] J.-C. Filliâtre and A. Paskevich, “Why3—where programs meet provers,” in *European symposium on programming*. Springer, 2013, pp. 125–128.

[53] K. R. M. Leino, “Program proving using intermediate verification languages (IVLs) like Boogie and Why3,” in *Proceedings of the 2012 ACM conference on High integrity language technology*, 2012, pp. 25–26.

[54] S. O’Grady, “The RedMonk Programming Language Rankings: June 2021,” 08 2021. [Online]. Available: https://redmonk.com/sogrady/2021/08/05/language-rankings-6-21/

[55] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, “Lessons from building static analysis tools at google,” *Communications of the ACM*, vol. 61, no. 4, pp. 58–66, 2018.

[56] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, “Curating github for engineered software projects,” *Empirical Software Engineering*, vol. 22, no. 6, pp. 3219–3253, 2017.

[57] Y. Jiang and B. Adams, “Co-evolution of infrastructure and source code-an empirical study,” in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 45–55.

[58] J. Saldaña, *The coding manual for qualitative researchers*. sage, 2021.

[59] J. C. Kelly, J. S. Sherif, and J. Hops, “An analysis of defect densities found during software inspections,” *Journal of Systems and Software*, vol. 17, no. 2, pp. 111–117, 1992.

[60] S. Reis, R. Abreu, M. d’Amorim, and D. Fortunato, “Leveraging Practitioners’ Feedback to Improve a Security Linter,” *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022.

[61] X. Jin and F. Servant, “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, 2021, pp. 213–225.

Appendix A

In this appendix, we describe the techniques implemented in GLITCH to detect the nine design & implementation smells described and studied in Chapters 3 and 5. We did not include this information in this work's main body since our focus is on the generalization of smells to multiple IaC technologies. These detection techniques were already studied by Schwarz et al. [3] and we only adapted them to fit into our framework and intermediate representation. Our goal was to compare GLITCH to the state-of-the-art.

Algorithms A.1, A.2, and A.3 describe the detection techniques applied to identify design & implementation smells in unit blocks, atomic units, and comments, respectively. To detect the nine design & implementation smells we selected, we do not need to check the remaining components of our intermediate representation. We consider the global variable config to be defined as an object that contains the configurations for the current run of the algorithm being executed. The configurations vary according to the IaC technology being analyzed. In the pseudo-code presented, we simplify how we return the smells detected by only adding the name of the smell to the list errors. In the implementation of these algorithms, other information, such as the line and a snippet of the source code where the smell occurred, should be provided.

In Algorithm A.1, the function *getVariableNames* returns all the variable names defined in a unit block. The same applies to the function *getStrings*, but instead of variable names, it returns all the string literals. The function *readLines* receives a path for a file, reads it, and returns a list with each line of the file. The variable VAR_REFER_SYMBOL, which is defined in the configuration, refers to the symbol used to interpolate a variable in a string. Chef uses the symbol *#* and Puppet uses the symbol *$*. In Ansible, variables are interpolated by using $\{\{VARIABLE\_NAME\}\}$, which avoids the smell Unguarded Variable. For this reason, the variable VAR_REFER_SYMBOL is not initialized. The function *checkDuplicateBlock* is defined in Algorithm A.8. The functions *checkImproperAlignment* and *checkMisplacedAttribute* are defined in Algorithms A.4 and A.5 and Algorithms A.6 and A.7, respectively. The techniques to detect the smell Improper Alignment and Misplaced Attribute are technology-dependent and are defined on the configuration. These smells do not apply to Ansible scripts.

In Algorithm A.2, the variable EXEC is a list with the types of atomic units that execute shell scripts. The values for the variable EXEC are different for Ansible, Chef, and Puppet. The functions *checkImproperAlignment* and *checkMisplacedAttribute* are defined in the same way as in Algorithm A.1.

The variable FIRST_CODE_LINE used in Algorithm A.3 contains the number of the first line that is not a comment or a white space.

Algorithm A.1 Check implementation & design smells in a Unit Block

```
errors ← []
variableNames ← getVariableNames(x)
strings ← getStrings(x)

linesOfCode ← readLines(x.path)
for each line ∈ linesOfCode do
    if '\t' ∈ line then
        errors.append('improperAlignment')
    end if
    if len(line) > 140 then
        errors.append('longStatement')
    end if
end for

if len(x.variables)/len(linesOfCode) > 0.3 ∧ u.type ≠ VARS then
    errors.append('tooManyVariables')
end if

if config.VAR_REFER_SYMBOL ≠ None then
    for each string ∈ strings do
        for each var ∈ variableNames do
            if config.VAR_REFER_SYMBOL + var ∈ string then
                errors.append('unguardedVariable')
            end if
        end for
    end for
end if

if checkDuplicateBlock(x) then
    errors.append('duplicateBlock')
end if

if config.checkImproperAlignment(x) then
    errors.append('improperAlignment')
end if

if config.checkMisplacedAttribute(x) then
    errors.append('misplacedAttribute')
end if

return errors
```

Algorithm A.2 Check implementation & design smells in an Atomic Unit

```
errors ← []

if x.type ∈ config.EXEC then
    for each attr ∈ x.attributes do
        if '&&' ∈ attr.value ∨ ';' ∈ attr.value ∨ '|' ∈ attr.value then
            errors.append('multifacetedAbstraction')
        end if
    end for
end if

if x.type ∈ config.EXEC ∧ count(x.code, '\n') > 7 then
    errors.append('longResourse')
end if

if config.checkImproperAlignment(x) then
    errors.append('improperAlignment')
end if

if config.checkMisplacedAttribute(x) then
    errors.append('misplacedAttribute')
end if

return errors
```

Algorithm A.3 Check implementation & design smells in a Comment

```
errors ← []
if x.line >= FIRST_CODE_LINE then
    errors.append('avoidComments')
end if
return errors
```

Algorithm A.4 Improper Alignment Detection in Chef scripts

```
1: function CHECKIMPROPERALIGNMENT(x)
2:     if isAtomicUnit(x) then
3:         identation ← None
4:
5:         for each attr ∈ x.attributes do
6:             fst ← attr.code.split('\n')[0]
7:             currIdent ← len(fst) − len(fst.lstrip())
8:             if identation ≠ None then
9:                 identation ← currIdent
10:            else if identation ≠ currIdent then
11:                return true
12:            end if
13:        end for
14:    end if
15:
16:    return false
17: end function
```

Algorithm A.5 Improper Alignment Detection in Puppet scripts

```
1: function CHECKIMPROPERALIGNMENT(x)
2:     lName, lIdent, lSplit ← 0, 0, ”
3:
4:     for each attr ∈ x.attributes do
5:         if len(x.name) > lName ∧ '=>' ∈ x.code then
6:             lName, split ← len(x.name), x.code.split('=>')[0]
7:             lIdent, lSplit ← len(split), split
8:         end if
9:     end for
10:
11:    if lSplit ≠ ”” then
12:        return true
13:    else if len(lSplit) − 1 ≠ len(lSplit.rstrip()) then
14:        return false
15:    end if
16:
17:    for each attr ∈ x.attributes do
18:        fst ← attr.code.split('\n')[0]
19:        curArrow ← len(attr.code.split('=>')[0])
20:        if curArrow ≠ lIdent then
21:            return false
22:        end if
23:    end for
24:
25:    return false
26: end function
```

Algorithm A.6 Misplaced Attribute Detection in Chef scripts

```
1: function CHECKMISPLACEDATTRIBUTE(x)
2:     if isAtomicUnit(x) then
3:         order ← []
4:
5:         for each attr ∈ x.attributes do
6:             if attr.name = 'source' then
7:                 order.append(1)
8:             else if attr.name = 'owner' ∨ attr.name = 'group' then
9:                 order.append(2)
10:            else if attr.name = 'mode' then
11:                order.append(3)
12:            else if attr.name = 'action' then
13:                order.append(4)
14:            end if
15:        end for
16:
17:        if order ≠ sorted(order) then
18:            return true
19:        end if
20:    end if
21:
22:    return false
23: end function
```

Algorithm A.7 Misplaced Attribute Detection in Puppet scripts

```
1: function CHECKMISPLACEDATTRIBUTE(x)
2:     if isAtomicUnit(x) then
3:         i ← 0
4:         for each attr ∈ x.attributes do
5:             if attr.name = 'ensure' ∧ i ≠ 0 then
6:                 return true
7:             end if
8:             i ← i + 1
9:         end for
10:    else if isUnitBlock(x) then
11:        optional ← False
12:        for each attr ∈ x.attributes do
13:            if attr.value ≠ None then
14:                optional ← True
15:            else if optional = True then
16:                return true
17:            end if
18:        end for
19:    end if
20:
21:    return false
22: end function
```

Algorithm A.8 Simplified Duplicate Block Detection

```
1: function CHECKDUPLICATEBLOCK(x)
2:     if isUnitBlock(x) then
3:         code ← read(x.path)
4:         blocks ← set()
5:
6:         for i ← 0 to len(code) − 150 do
7:             h ← hash(code[i : i + 150])
8:             if h ∉ blocks then
9:                 blocks.update(h)
10:            else
11:                return true
12:            end if
13:        end for
14:    end if
15:
16:    return false
17: end function
```

B

Appendix B

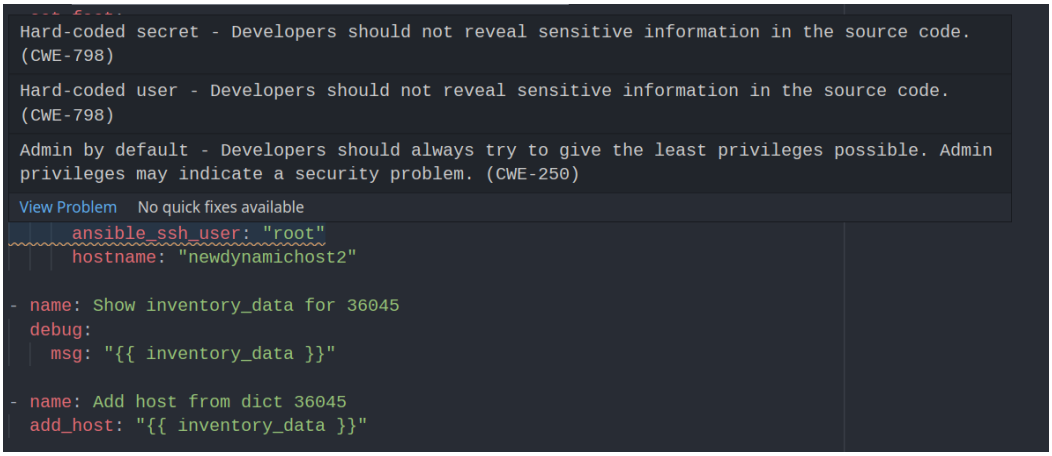

Figure B.1: GLITCH's Visual Studio Code extension warning about the detection of three security smells: Hard-coded secret, Hard-coded user, and Admin by default.